

# Simulation and Scientific Computing

(Simulation und Wissenschaftliches Rechnen - SiWiR)

Winter Term 2015/16

Florian Schornbaum and Christian Kuschel  
Chair for System Simulation



FRIEDRICH-ALEXANDER  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT



# Assignment 1: Performance Optimization

October 20, 2015 – November 9, 2015



FRIEDRICH-ALEXANDER  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

- **Organizational Matters**
- The Memory Hierarchy: Caches
- Cache-Aware Code Optimization
  - Temporal Locality
  - Spatial Locality
  - Cache Thrashing
- Optimizing Loops
  - Loop Unrolling
  - Loop Fusion

- Don't forget to sign up for the lecture and exercises on [mein campus](#) until **Friday, October 30, 2015!**
- Remember to sign up on the team formation list at our chair until **Friday, October 30, 2015!**
- Team formation is mandatory, you have to divide into groups of three persons.

- Don't forget to activate your **account** at our chair.
- Remote access to our computers (i10cip1.informatik.uni-erlangen.de to i10cip12) is only possible via **ssh** (Windows users: PuTTY & WinSCP).

- Access requires a **public-private key pair**:

```
[user@private-pc-at-home/laptop ~]$ ssh-keygen -t dsa -b 1024
```

Create a default key "~/.ssh/id\_dsa" and copy (e.g., via mail) the public (!) key located at "~/.ssh/id\_dsa.pub" to the "~/.ssh/" directory of your account at LSS. If the directory does not exist at your LSS account, you must create it. Then:

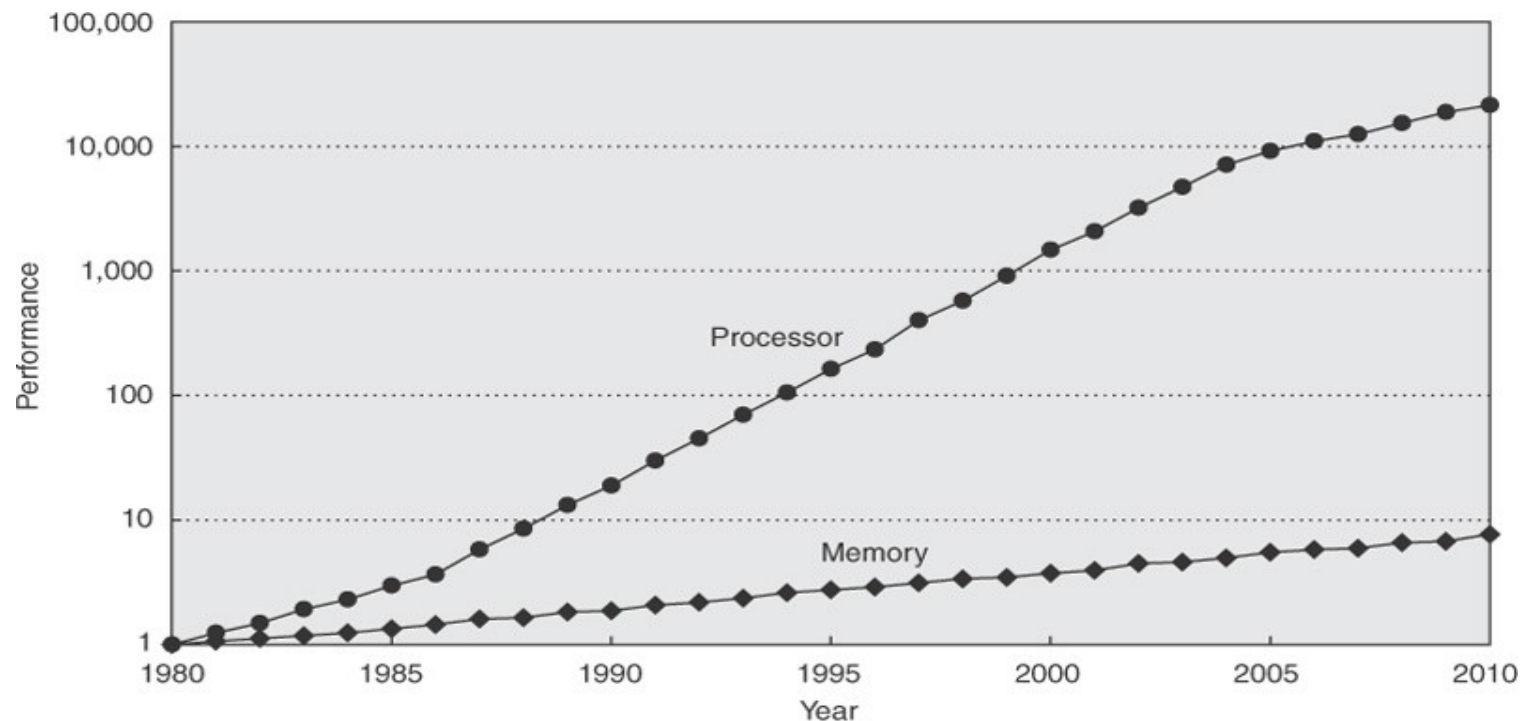
```
[user@i10cip5 ~]$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

- In order to login, you have to either first access a computer from the university network, or log in via VPN:

<http://www.rrze.fau.de/dienste/internet-zugang/vpn/openvpn.shtml>

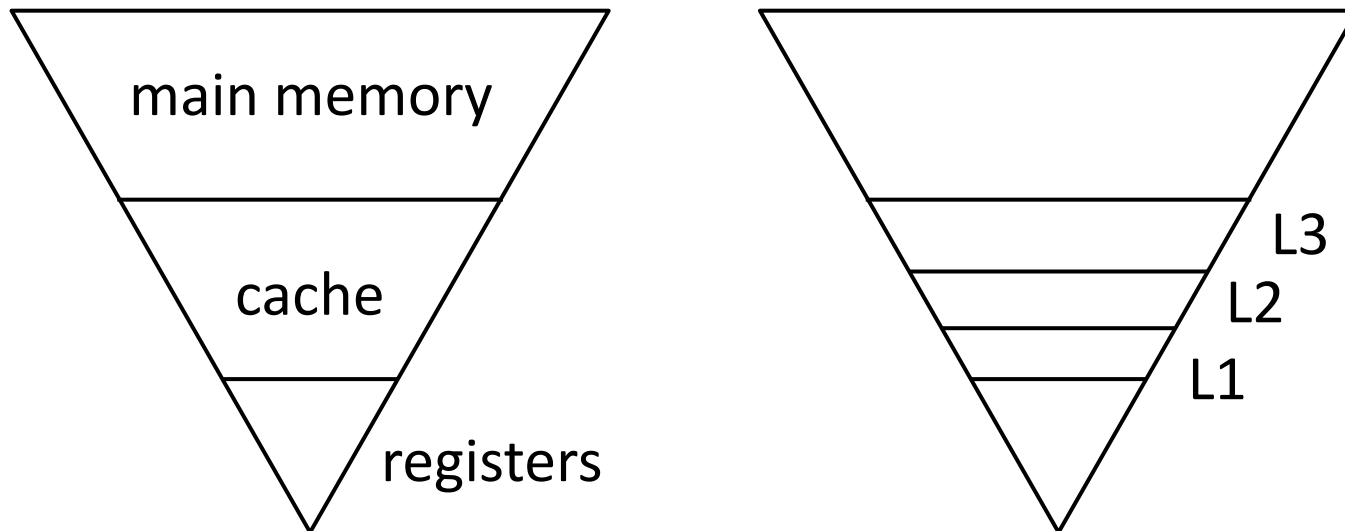
- Organizational Matters
- **The Memory Hierarchy: Caches**
- Cache-Aware Code Optimization
  - Temporal Locality
  - Spatial Locality
  - Cache Thrashing
- Optimizing Loops
  - Loop Unrolling
  - Loop Fusion

- The memory access became one of the mayor bottlenecks in high performance computing (HPC).
- Improvement of CPU speed and memory latency:



© 2007 Elsevier, Inc. All rights reserved.

- Use a memory hierarchy in order to decrease the memory gap!
- Cache(s): small but fast memory between main memory (large & slow) and CPU registers (extremely small & fast)



- The cache, like the main memory, is organized in blocks/lines.



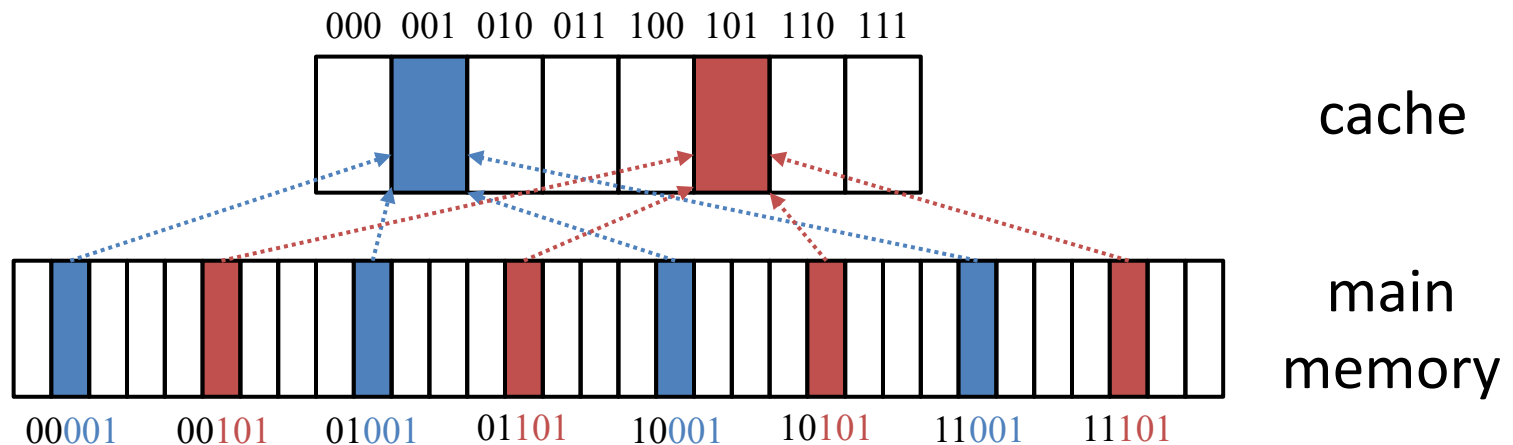
- Primary cache (L1)
  - Split into data and instruction cache
  - Typical size: 32 – 128 KBytes
  - Intel Core i7 3770: data & instruction cache both 4 x 32 KBytes
- Secondary cache (L2)
  - Unified cache (data and instruction together in the same cache)
  - Typical size: 512 KBytes – 2 MBytes
  - Intel Core i7 3770: 4 x 256 KBytes
- Today often also L3 cache
  - Unified cache, size typically in the range of a few MBytes
  - Intel Core i7 3770: 8 Mbytes (shared between all cores)

If a computer program requires access to a certain data item, different actions may get triggered:

- If the data item is already in the cache: **cache hit** → The data is read from cache.
- Otherwise: **cache miss** → The data first must be fetched from a higher cache level or the main memory.
- If all cache entries are occupied, old items are **evicted** from the cache and replaced by the new data.

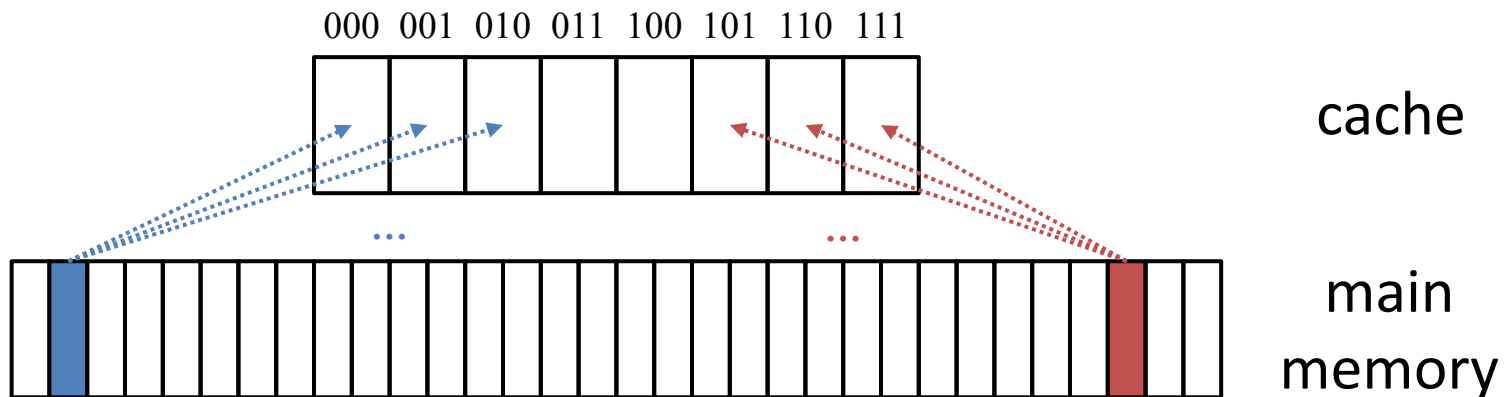
- There is a lot less space in the cache than in main memory!
- How are data items in the main memory **mapped** to the cache?  
Which data item goes where?
- The three most common organizational forms:
  - **Direct mapped** (simple, worst hit rate, fastest access)
  - **Fully associative** (complex, lowest miss rates, slow)
  - **N-way set associative** (compromise between the other two)

- Every memory address MA is directly associated with a certain cache block B out ...
- ... of N cache blocks (for example by using a modulo operation:  $B = MA \bmod N$ ).
- Example: cache capacity  $N = 8$



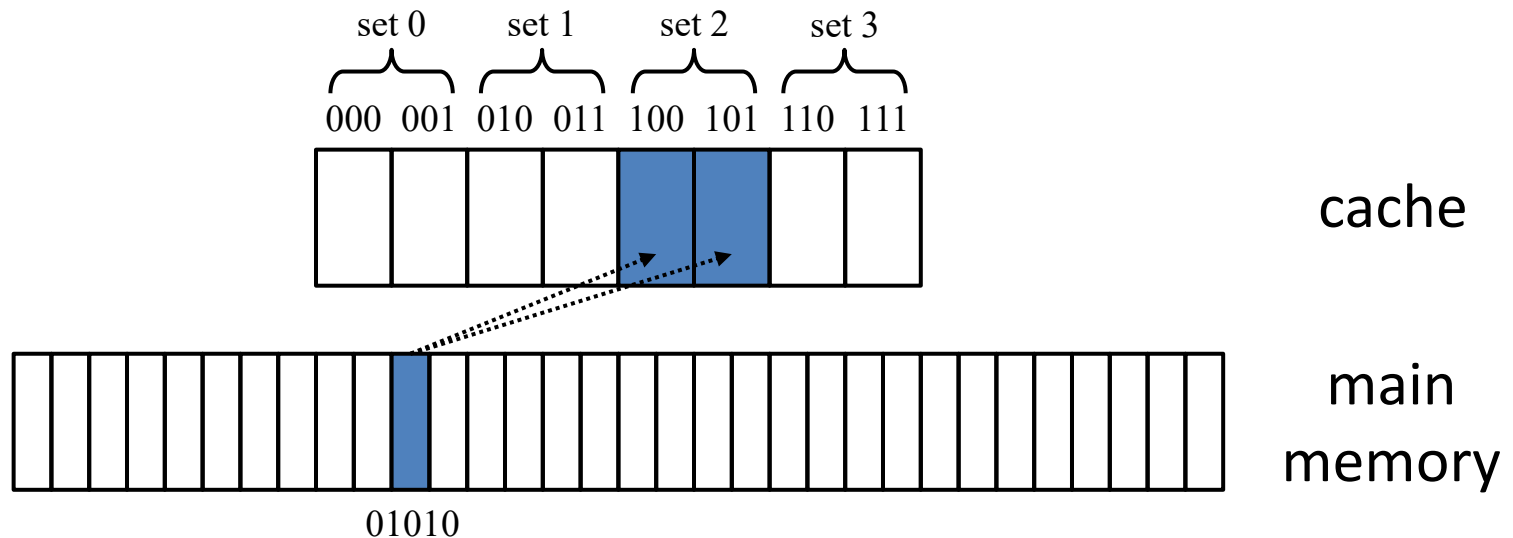
- Fast! But low hit rates & many cache misses ...

- Every data item in main memory can be mapped to an arbitrary cache block.



- Requires complex placement/retrieval strategies.
- High hit rates & few cache misses .... But slow!

- Cache blocks are divided into  $S$  sets of  $n$  blocks each ( $S = N/n$ )
- Two step mapping process:
  - Direct mapping to a specific set
  - Arbitrary (= fully associative) mapping inside each set
- Example: 2-way set associative



Associative caches need an eviction strategy:  
Which block is evicted if the cache or the set is fully occupied?

Strategies:

- Cyclic / first in first out (**FIFO**): The oldest block is evicted.
- Least recently used (**LRU**): The block which was not read for the longest period of time is evicted.
- Least frequently used (**LFU**)
- A **random** block is evicted.
  - Requires minimal hardware effort → fast!
  - In practice only slightly worse than the other strategies!  
(→ predicting the future is difficult ...)

Cache misses are divided into three types:

- **Compulsory misses:** block not yet in cache because it is accessed for the first time
- **Capacity misses:** cache miss due to the small (finite) size of the cache, block had to be evicted
- **Conflict misses:** block was evicted because of an address conflict
- A lot of very good information on CPU caches:

[https://en.wikipedia.org/wiki/CPU\\_cache](https://en.wikipedia.org/wiki/CPU_cache)



- Organizational Matters
- The Memory Hierarchy: Caches
- **Cache-Aware Code Optimization**
  - Temporal Locality
  - Spatial Locality
  - Cache Thrashing
- Optimizing Loops
  - Loop Unrolling
  - Loop Fusion

A word of advice:

- Before starting hardware related performance optimizations, always choose the **best algorithm** for solving your problem!
  - Naïve matrix-matrix multiplication:  $O(N^3) = O(N^{\log_2 8})$
  - Strassen algorithm for matrix multiplication:  $O(N^{\log_2 7})$
- In theory, regardless of its constant overhead, an algorithm with a better asymptotic complexity will always solve a given problem in less time – provided the input is large enough.
  - Small matrices → naïve multiplication
  - Large matrices → Strassen algorithm

Naïve matrix-matrix multiplication:

$$C = A \cdot B \quad ; \quad A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} \in \mathbb{R}^{n \times n} \quad ; \quad B, C \in \mathbb{R}^{n \times n}$$

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$

$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

$$O(n^3)$$

The Strassen algorithm:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{pmatrix}$$

$$M_1 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$$

$$M_2 = (A_{21} + A_{22}) \cdot B_{11}$$

$$M_3 = A_{11} \cdot (B_{12} - B_{22})$$

$$M_4 = A_{22} \cdot (B_{21} - B_{11})$$

$$M_5 = (A_{11} + A_{12}) \cdot B_{22}$$

$$M_6 = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$$

$$M_7 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$$

$$O(n^{\log_2 7}) \approx O(n^{2.8})$$

The underlying idea:

“A cache-aware algorithm is designed to minimize the movement of memory pages in and out of the processor's on-chip memory cache. The idea is to avoid what's called "cache misses," which cause the processor to stall while it loads data from RAM into the processor cache.” – Jim Mischel

(from: <http://stackoverflow.com/questions/473137/a-simple-example-of-a-cache-aware-algorithm>)

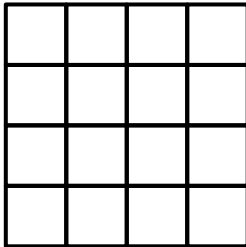
- A common problem:
  - Data is pulled into the cache and not used again for some period of time.
  - It gets evicted.
  - When it is accessed again, it must be reloaded into the cache.
- The solution:
  - Optimize for temporal locality!
  - If some data is accessed multiple times, these accesses should happen within short periods of time – thereby preventing eviction.

- If data is transferred from main memory to the cache, always **cache lines** are transferred, never just single data items.
- Cache line: multiple data items that reside besides each other in the main memory
- Optimizing for spatial locality:
  - If you have to access multiple data items, try to reorder these accesses such that data items that reside in close proximity in main memory are **accessed consecutively** (stride-one / stride-1 memory access).
  - After accessing the first data item, all data items that follow are already in cache!

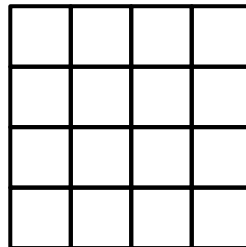
- Naïve, standard implementation of a matrix transposition:

```
for( int i = 0; i < n; ++i )  
    for( int j = 0; j < n; ++j )  
        b[i][j] = a[j][i];
```

$$B = A^T$$



$$A$$

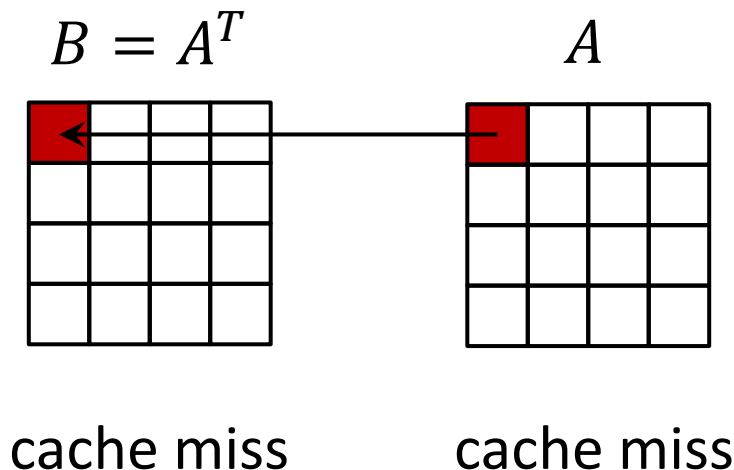


cache line size: 2 elements  
size of cache: 4 cache lines



- Naïve, standard implementation of a matrix transposition:

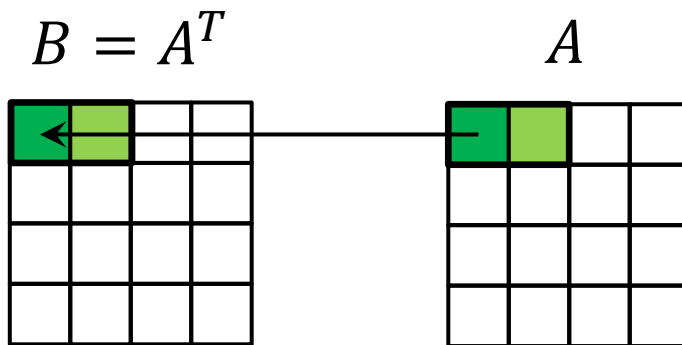
```
for( int i = 0; i < n; ++i )  
  for( int j = 0; j < n; ++j )  
    b[i][j] = a[j][i];
```



cache line size: 2 elements  
size of cache: 4 cache lines

- Naïve, standard implementation of a matrix transposition:

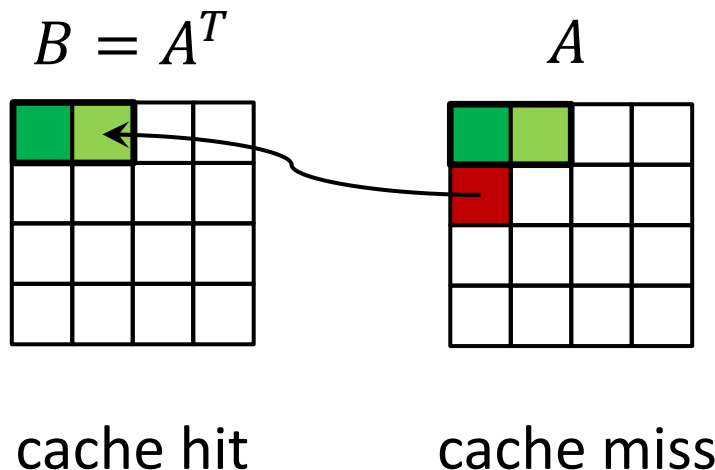
```
for( int i = 0; i < n; ++i )  
  for( int j = 0; j < n; ++j )  
    b[i][j] = a[j][i];
```



cache line size: 2 elements  
size of cache: 4 cache lines

- Naïve, standard implementation of a matrix transposition:

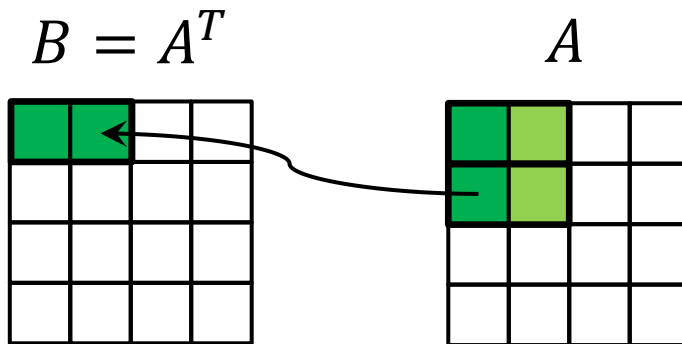
```
for( int i = 0; i < n; ++i )  
    for( int j = 0; j < n; ++j )  
        b[i][j] = a[j][i];
```



cache line size: 2 elements  
size of cache: 4 cache lines

- Naïve, standard implementation of a matrix transposition:

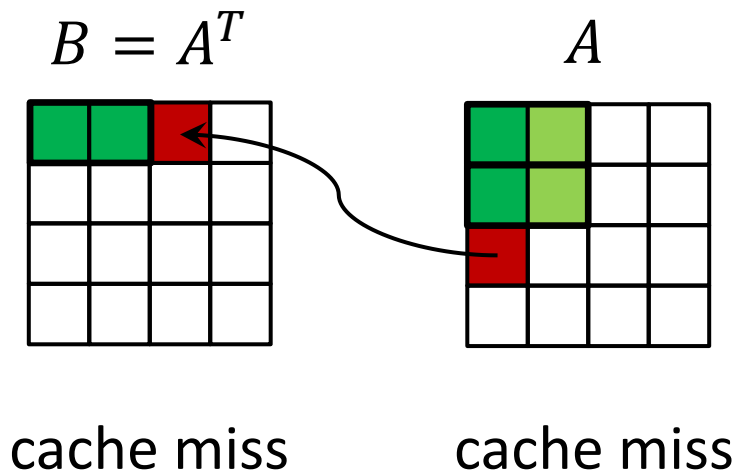
```
for( int i = 0; i < n; ++i )  
    for( int j = 0; j < n; ++j )  
        b[i][j] = a[j][i];
```



cache line size: 2 elements  
size of cache: 4 cache lines

- Naïve, standard implementation of a matrix transposition:

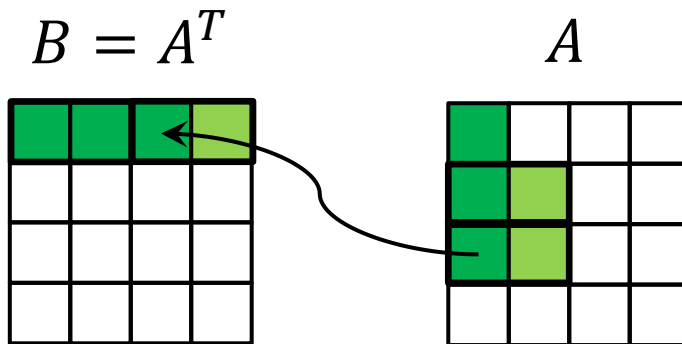
```
for( int i = 0; i < n; ++i )  
  for( int j = 0; j < n; ++j )  
    b[i][j] = a[j][i];
```



cache line size: 2 elements  
size of cache: 4 cache lines

- Naïve, standard implementation of a matrix transposition:

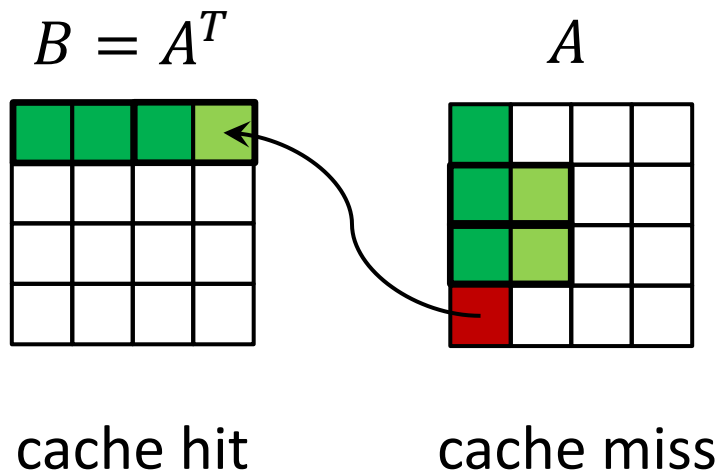
```
for( int i = 0; i < n; ++i )  
    for( int j = 0; j < n; ++j )  
        b[i][j] = a[j][i];
```



cache line size: 2 elements  
size of cache: 4 cache lines

- Naïve, standard implementation of a matrix transposition:

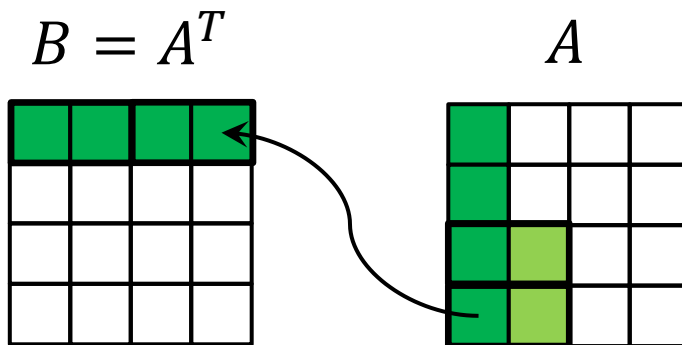
```
for( int i = 0; i < n; ++i )  
  for( int j = 0; j < n; ++j )  
    b[i][j] = a[j][i];
```



cache line size: 2 elements  
size of cache: 4 cache lines

- Naïve, standard implementation of a matrix transposition:

```
for( int i = 0; i < n; ++i )  
    for( int j = 0; j < n; ++j )  
        b[i][j] = a[j][i];
```

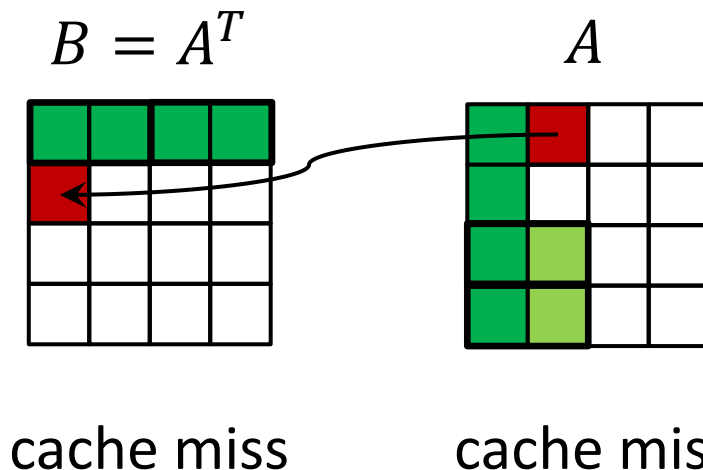


cache line size: 2 elements  
size of cache: 4 cache lines



- Naïve, standard implementation of a matrix transposition:

```
for( int i = 0; i < n; ++i )  
  for( int j = 0; j < n; ++j )  
    b[i][j] = a[j][i];
```

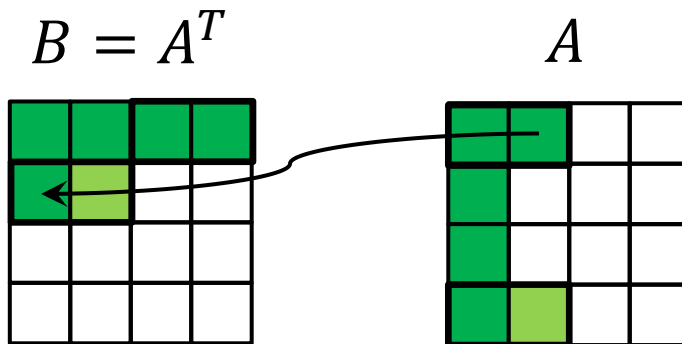


cache line size: 2 elements  
size of cache: 4 cache lines

(a[0][1] was evicted and is not in the cache anymore ...)

- Naïve, standard implementation of a matrix transposition:

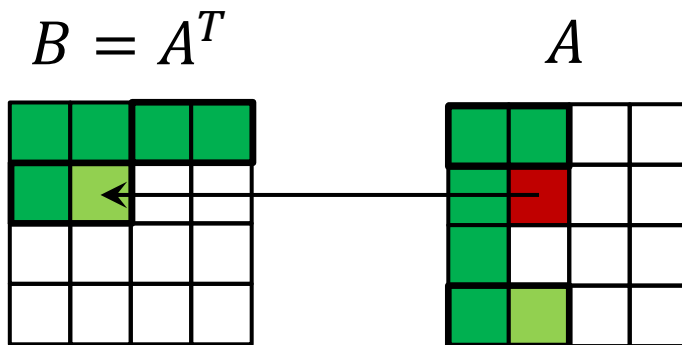
```
for( int i = 0; i < n; ++i )  
    for( int j = 0; j < n; ++j )  
        b[i][j] = a[j][i];
```



cache line size: 2 elements  
size of cache: 4 cache lines

- Naïve, standard implementation of a matrix transposition:

```
for( int i = 0; i < n; ++i )  
    for( int j = 0; j < n; ++j )  
        b[i][j] = a[j][i];
```



cache hit

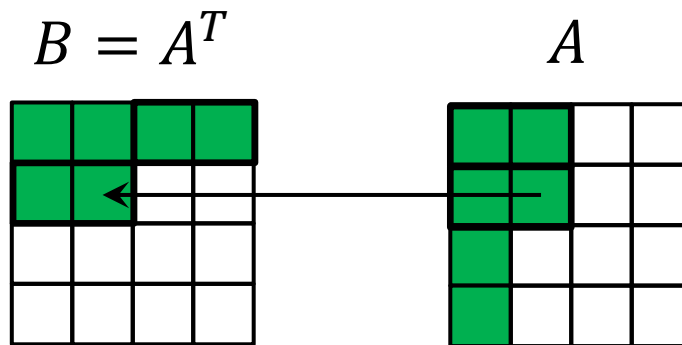
cache miss

cache line size: 2 elements  
size of cache: 4 cache lines

(a[1][1] was evicted and is not in the cache anymore ...)

- Naïve, standard implementation of a matrix transposition:

```
for( int i = 0; i < n; ++i )  
    for( int j = 0; j < n; ++j )  
        b[i][j] = a[j][i];
```



cache line size: 2 elements  
size of cache: 4 cache lines

and so on ...

- Naïve, standard implementation of a matrix transposition:

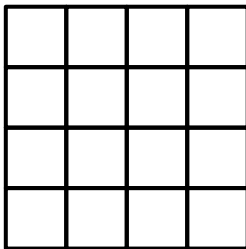
```
for( int i = 0; i < n; ++i )  
    for( int j = 0; j < n; ++j )  
        b[i][j] = a[j][i];
```

- + Stride-1 access for array **b**
- Stride-n access for array **a** → cache lines are rarely/never utilized

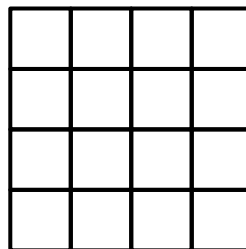
- Square blocking (optimizing temporal locality):

```
for( int ii = 0; ii < n; ii += blocksize )  
    for( int jj = 0; jj < n; jj += blocksize )  
        for( int i = ii; i < ii + blocksize; ++i )  
            for( int j = jj; j < jj + blocksize; ++j )  
                b[i][j] = a[j][i];
```

$$B = A^T$$



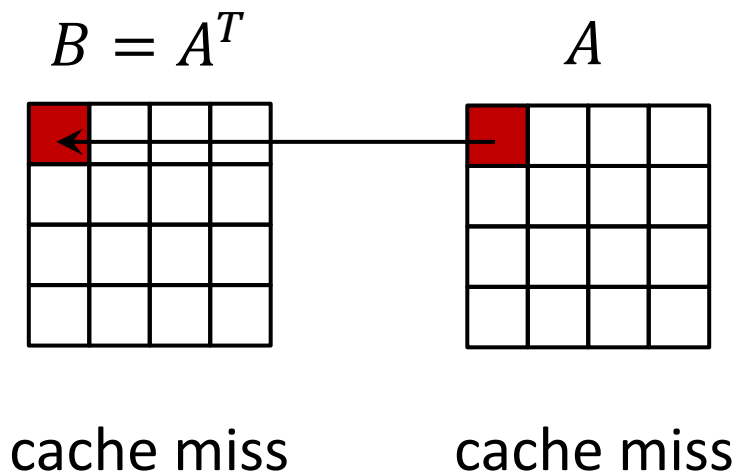
$$A$$



cache line size: 2 elements  
size of cache: 4 cache lines  
block size: 2 x 2

- Square blocking (optimizing temporal locality):

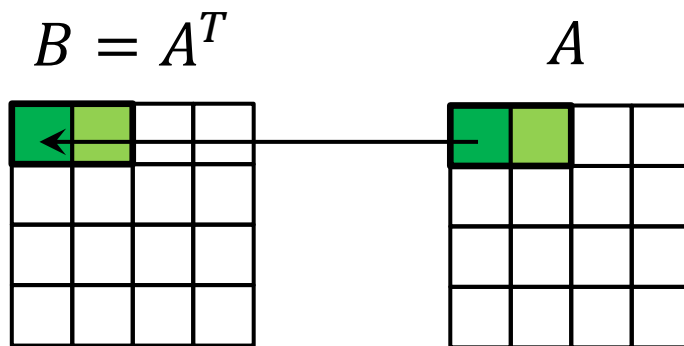
```
for( int ii = 0; ii < n; ii += blocksize )  
  for( int jj = 0; jj < n; jj += blocksize )  
    for( int i = ii; i < ii + blocksize; ++i )  
      for( int j = jj; j < jj + blocksize; ++j )  
        b[i][j] = a[j][i];
```



cache line size: 2 elements  
size of cache: 4 cache lines  
block size: 2 x 2

- Square blocking (optimizing temporal locality):

```
for( int ii = 0; ii < n; ii += blocksize )  
  for( int jj = 0; jj < n; jj += blocksize )  
    for( int i = ii; i < ii + blocksize; ++i )  
      for( int j = jj; j < jj + blocksize; ++j )  
        b[i][j] = a[j][i];
```

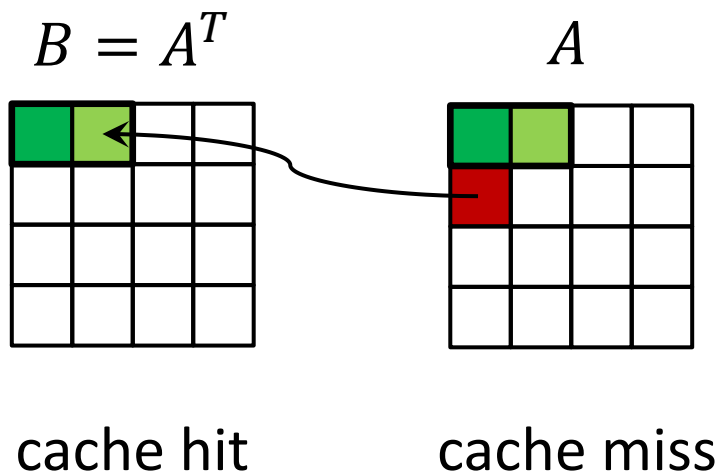


cache line size: 2 elements  
size of cache: 4 cache lines  
block size: 2 x 2



- Square blocking (optimizing temporal locality):

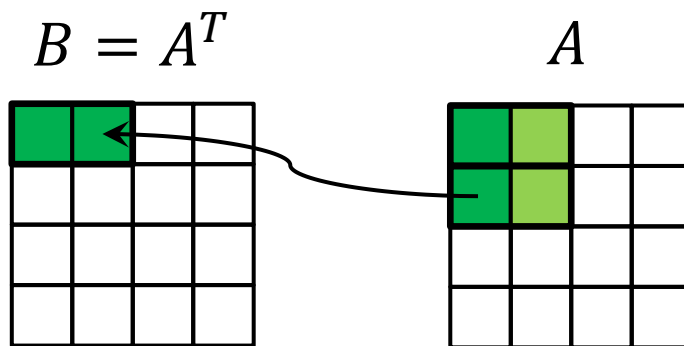
```
for( int ii = 0; ii < n; ii += blocksize )  
  for( int jj = 0; jj < n; jj += blocksize )  
    for( int i = ii; i < ii + blocksize; ++i )  
      for( int j = jj; j < jj + blocksize; ++j )  
        b[i][j] = a[j][i];
```



cache line size: 2 elements  
size of cache: 4 cache lines  
block size: 2 x 2

- Square blocking (optimizing temporal locality):

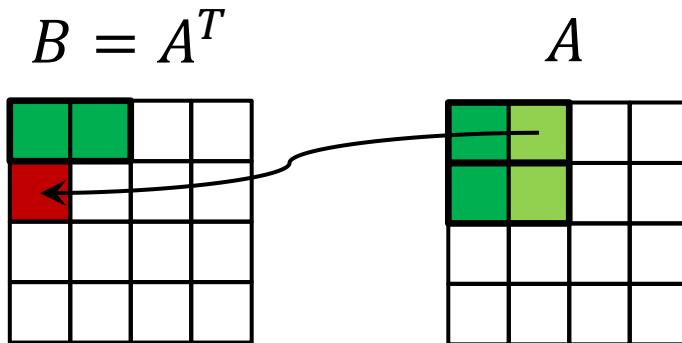
```
for( int ii = 0; ii < n; ii += blocksize )  
  for( int jj = 0; jj < n; jj += blocksize )  
    for( int i = ii; i < ii + blocksize; ++i )  
      for( int j = jj; j < jj + blocksize; ++j )  
        b[i][j] = a[j][i];
```



cache line size: 2 elements  
size of cache: 4 cache lines  
block size: 2 x 2

- Square blocking (optimizing temporal locality):

```
for( int ii = 0; ii < n; ii += blocksize )  
  for( int jj = 0; jj < n; jj += blocksize )  
    for( int i = ii; i < ii + blocksize; ++i )  
      for( int j = jj; j < jj + blocksize; ++j )  
        b[i][j] = a[j][i];
```



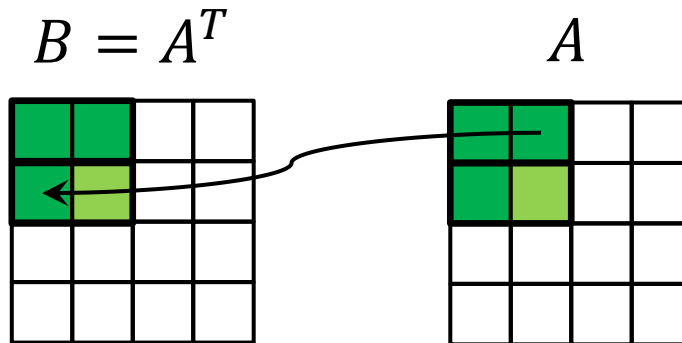
cache line size: 2 elements  
size of cache: 4 cache lines  
block size: 2 x 2

cache miss

cache hit ( $a[0][1]$  is still in cache!)

- Square blocking (optimizing temporal locality):

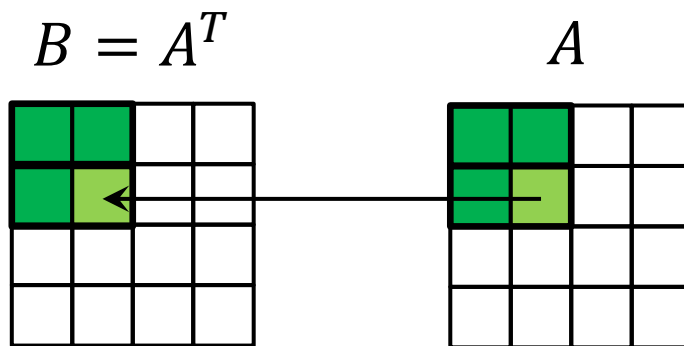
```
for( int ii = 0; ii < n; ii += blocksize )  
  for( int jj = 0; jj < n; jj += blocksize )  
    for( int i = ii; i < ii + blocksize; ++i )  
      for( int j = jj; j < jj + blocksize; ++j )  
        b[i][j] = a[j][i];
```



cache line size: 2 elements  
size of cache: 4 cache lines  
block size: 2 x 2

- Square blocking (optimizing temporal locality):

```
for( int ii = 0; ii < n; ii += blocksize )  
  for( int jj = 0; jj < n; jj += blocksize )  
    for( int i = ii; i < ii + blocksize; ++i )  
      for( int j = jj; j < jj + blocksize; ++j )  
        b[i][j] = a[j][i];
```



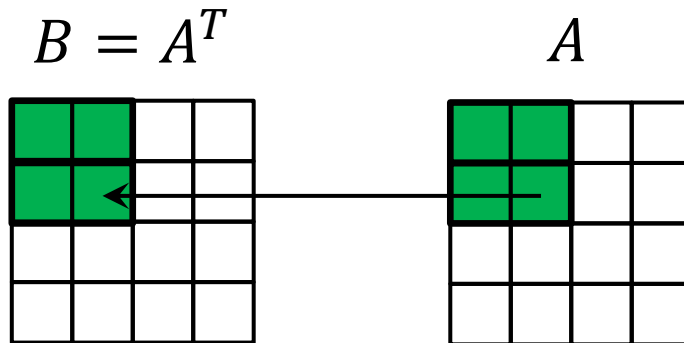
cache line size: 2 elements  
size of cache: 4 cache lines  
block size: 2 x 2

cache hit

cache hit (a[1][1] is still in cache!)

- Square blocking (optimizing temporal locality):

```
for( int ii = 0; ii < n; ii += blocksize )  
  for( int jj = 0; jj < n; jj += blocksize )  
    for( int i = ii; i < ii + blocksize; ++i )  
      for( int j = jj; j < jj + blocksize; ++j )  
        b[i][j] = a[j][i];
```



cache line size: 2 elements  
size of cache: 4 cache lines  
block size: 2 x 2

and so on ...

- Square blocking (optimizing temporal locality):

```
for( int ii = 0; ii < n; ii += blocksize )  
    for( int jj = 0; jj < n; jj += blocksize )  
        for( int i = ii; i < ii + blocksize; ++i )  
            for( int j = jj; j < jj + blocksize; ++j )  
                b[i][j] = a[j][i];
```

- + Stride-1 access for array **b**
  - + Cache lines of **a** are utilized more often.
  - Twice as many loops
- ⇒ Choose the *blocksize* such that  $2 * blocksize^2$  data elements fit into the L2/L3 cache.

- Line blocking instead of square blocking


```
for( int jj = 0; jj < n; jj += blocksize )  
    for( int i = 0; i < n; ++i )  
        for( int j = jj; j < jj + blocksize; ++j )  
            b[i][j] = a[j][i];
```

- + Stride-1 access for array **b**
- + Cache lines of **a** are utilized more often.
- + Fewer and longer loops than with square blocking



- Assume we have a 2-way set associative cache with 1024 cache lines (each cache line holds 4 double precision values):

```
double a[N][4096];  
for( int i = 0; i < N; ++i )  
    for( int j = 0; j < 4096; ++j )  
        a[i][j] = a[i+1][j] + a[i+2][j] + a[i+3][j];
```




All four values are mapped to the same set!  
(But only two can be in the cache at the same time ...)

- Consequence: Data items are constantly loaded into and removed from the cache.

- Assume we have a 2-way set associative cache with 1024 cache lines (each cache line holds 4 double precision values):

```
double a[N][4096];  
for( int i = 0; i < N; ++i )  
    for( int j = 0; j < 4096; ++j )  
        a[i][j] = a[i+1][j] + a[i+2][j] + a[i+3][j];
```



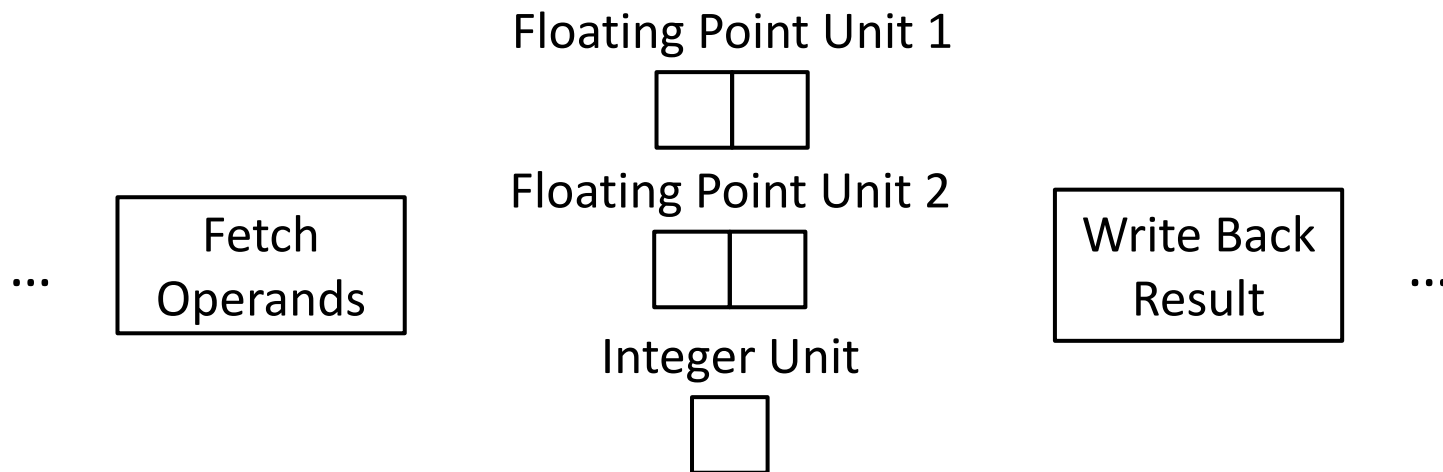
All four values are mapped to the same set!  
(But only two can be in the cache at the same time ...)

- Can be prevented with padding: `double a[N][4096+81];`
- Rule of thumb: Use odd multiples of 16 for the leading array dimension.

- Organizational Matters
- The Memory Hierarchy: Caches
- Cache-Aware Code Optimization
  - Temporal Locality
  - Spatial Locality
  - Cache Thrashing
- **Optimizing Loops**
  - Loop Unrolling
  - Loop Fusion

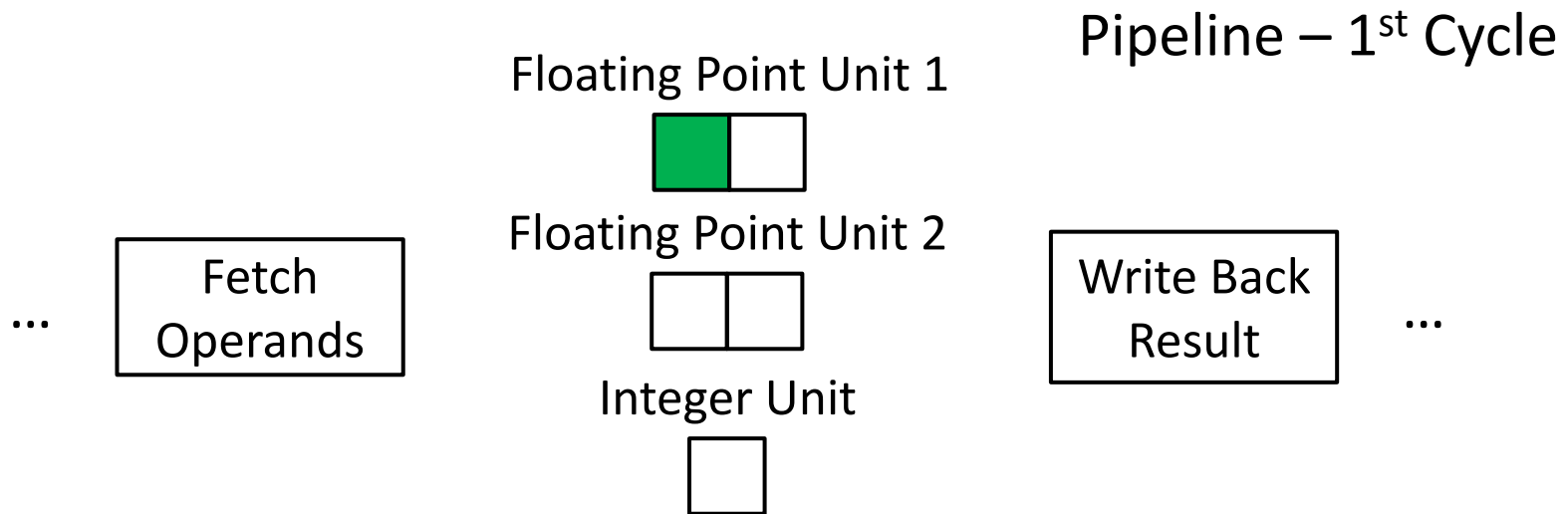
- Naïve, standard implementation of calculating a vector norm:

```
double r = 0.0;  
for( int i = 0; i < n; ++i )  
    r = r + a[i] * a[i];
```



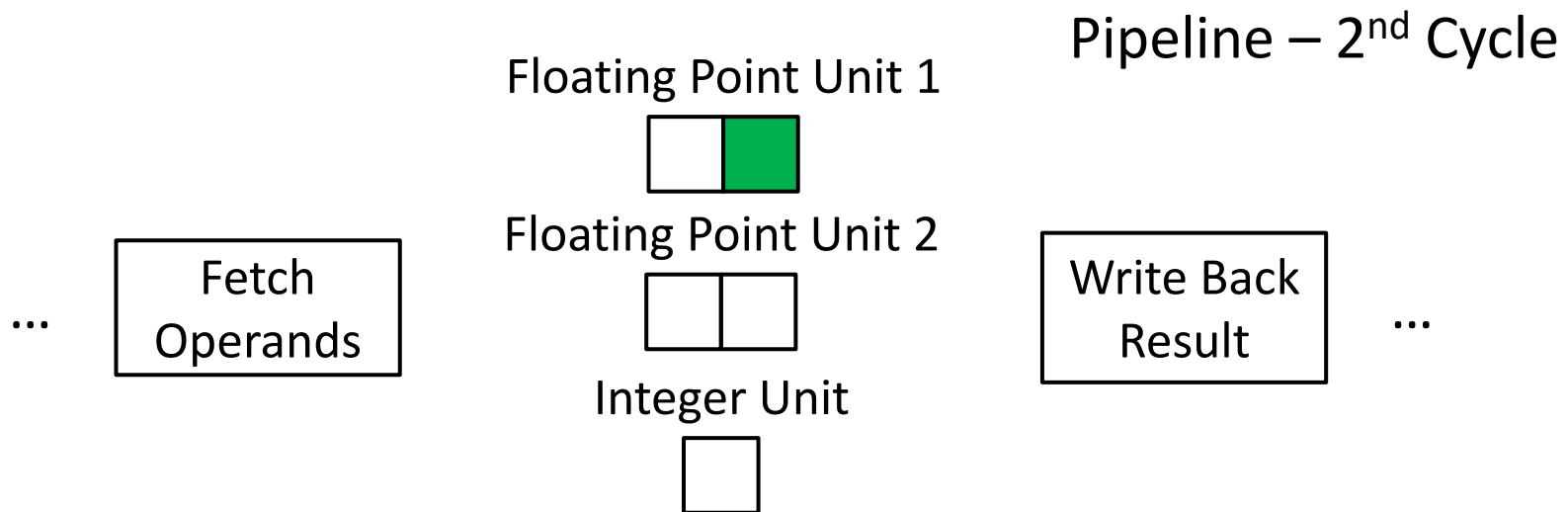
- Naïve, standard implementation of calculating a vector norm:

```
double r = 0.0;  
for( int i = 0; i < n; ++i )  
    r = r + a[i] * a[i];
```



- Naïve, standard implementation of calculating a vector norm:

```
double r = 0.0;  
for( int i = 0; i < n; ++i )  
    r = r + result_of( a[i] * a[i] );
```



- Naïve, standard implementation of calculating a vector norm:

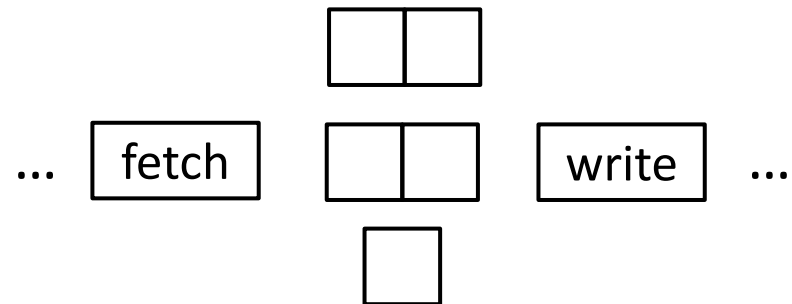
```
double r = 0.0;  
for( int i = 0; i < n; ++i )  
    r = r + a[i] * a[i];
```

⇒ 2 cycles per fused multiply add operation

- Alternative implementation using 4-way loop unrolling:

```
double r = 0.0;
double r0 = 0.0, r1 = 0.0, r2 = 0.0, r3 = 0.0;
for( int i = 0; i < n - 3; i += 4 ) {
    r0 = r0 + a[ i ] * a[ i ];
    r1 = r1 + a[i+1] * a[i+1];
    r2 = r2 + a[i+2] * a[i+2];
    r3 = r3 + a[i+3] * a[i+3];
}
r = r0 + r1 + r2 + r3;
```

Only works if n is multiple of 4!  
Otherwise more computation is  
needed after the loop.

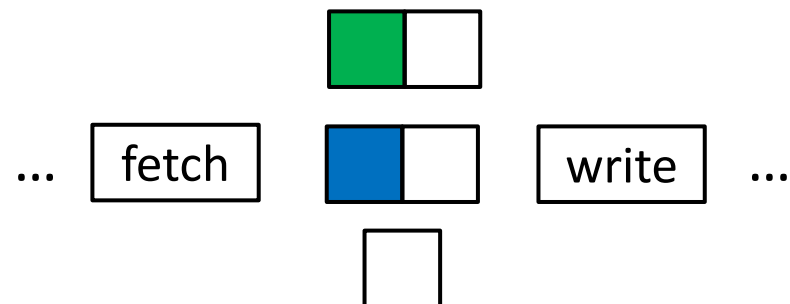




- Alternative implementation using 4-way loop unrolling:

```
double r = 0.0;
double r0 = 0.0, r1 = 0.0, r2 = 0.0, r3 = 0.0;
for( int i = 0; i < n - 3; i += 4 ) {
    r0 = r0 + a[ i ] * a[ i ];
    r1 = r1 + a[i+1] * a[i+1];
    r2 = r2 + a[i+2] * a[i+2];
    r3 = r3 + a[i+3] * a[i+3];
}
r = r0 + r1 + r2 + r3;
```

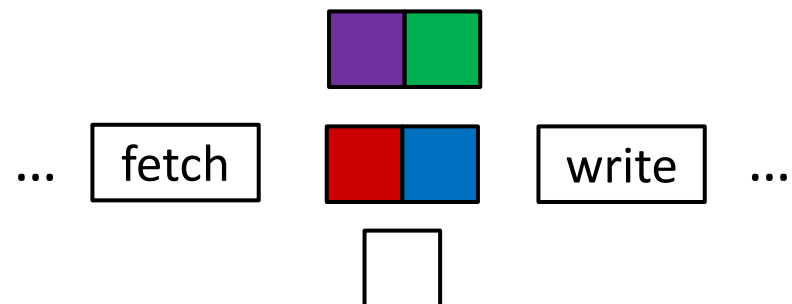
Pipeline – 1<sup>st</sup> Cycle



- Alternative implementation using 4-way loop unrolling:

```
double r = 0.0;
double r0 = 0.0, r1 = 0.0, r2 = 0.0, r3 = 0.0;
for( int i = 0; i < n - 3; i += 4 ) {
    r0 = r0 + result_of( a[ i ] * a[ i ] );
    r1 = r1 + result_of( a[i+1] * a[i+1] );
    r2 = r2 + a[i+2] * a[i+2];
    r3 = r3 + a[i+3] * a[i+3];
}
r = r0 + r1 + r2 + r3;
```

Pipeline – 2<sup>nd</sup> Cycle



- Almost 4 times faster for long loops!

- Loading the operands takes time (e.g., 2 loads per cycle). These loads can be hidden by prefetching (loading data in advance):

```
double r0 = 0.0, r1 = 0.0, r2 = 0.0, r3 = 0.0;
double a0 = 0.0, a1 = 0.0, a2 = 0.0, a3 = 0.0;
for( int i = 0; i < n - 3; i += 4 ) {
    a2 = a[i+2];
    r0 = r0 + a0 * a0;
    a3 = a[i+3];
    r1 = r1 + a1 * a1;

    r2 = r2 + a2 * a2;
    a0 = a[ i ];
    r3 = r3 + a3 * a3;
    a1 = a[i+1];
}
r0 = r0 + a0 * a0;
r1 = r1 + a1 * a1;
double r = r0 + r1 + r2 + r3 ;
```

- Compilers have gotten extremely good at automatic loop unrolling & prefetching.
- Loop unrolling comes with a price: CPU registers!  
All the temporary loop variables must be stored in registers for optimal performance, but the number of registers is limited.  
Running out of registers will decrease the performance!
- For more information on loop unrolling:  
[https://en.wikipedia.org/wiki/Loop\\_unwinding](https://en.wikipedia.org/wiki/Loop_unwinding)

- Very simple basic principle (better utilization of registers):

```
for( int i = 0; i < n; ++i )  
    b[i] = a[i] + a[i];  
  
for( int i = 0; i < n; ++i )  
    c[i] = a[i] * a[i];
```

should be transformed into:

```
for( int i = 0; i < n; ++i ) {  
    b[i] = a[i] + a[i];  
    c[i] = a[i] * a[i];  
}
```

# THE END QUESTIONS ?



FRIEDRICH-ALEXANDER  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT