High Performance and Parallel Computing (MA 6241)

Lecture 3

Memory Hierarchy

Processor Micro-Architecture

Ulrich Rüde
Lehrstuhl für Simulation
Universität Erlangen-Nürnberg

visiting Professor at
Department of Mathematics
National University of Singapore

Contents of High Performance and Parallel Computing

- Introduction
  - Computational Science and Engineering
  - High Performance Computing
- Computer Architecture
  - memory hierarchy
  - pipeline
  - instruction level parallelism
  - multi-core systems
  - parallel clusters
- Efficient Programming
  - computational complexity
  - efficient coding
  - architecture aware programming
  - shared memory parallel programming (OpenMP)
  - distributed memory parallel programming (MPI)
  - parallel programming with Graphics Cards

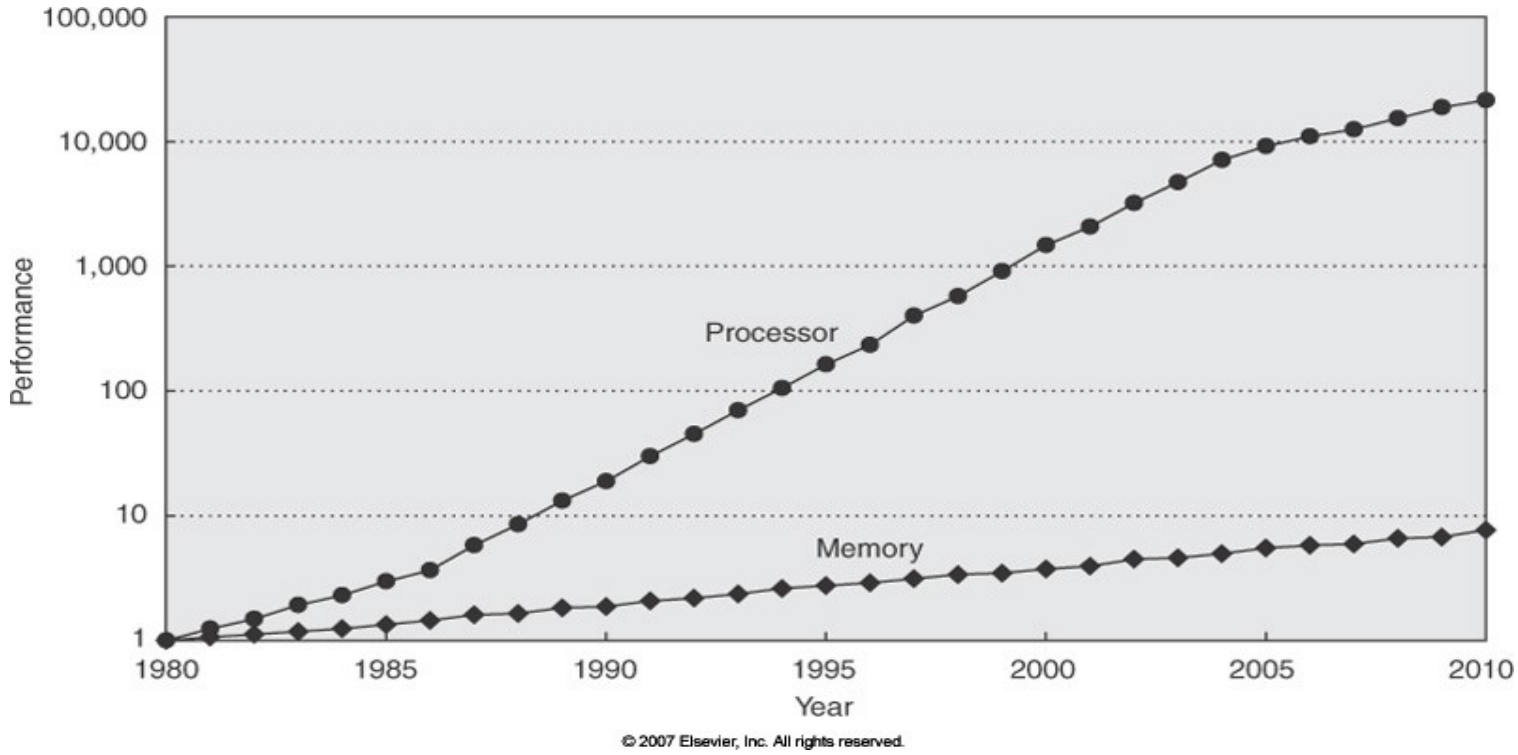
Processors
Memory Architecture
Nodes

Memory Organization (1)

- Registers, typically 32-128 double precision
- Load-Store architecture
- Explicit use of registers: assembly language(?) or register attribute in C, but often ignored by compiler.
- Register allocation algorithms are quite good in state of the art compilers (but not perfect)
- More registers in future CPUs? (128 in Itanium)
- Latency for decoding
- Instruction set (register windows)
- Register allocation more complex, compile times get longer

HPC Bottleneck: Memory Access

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

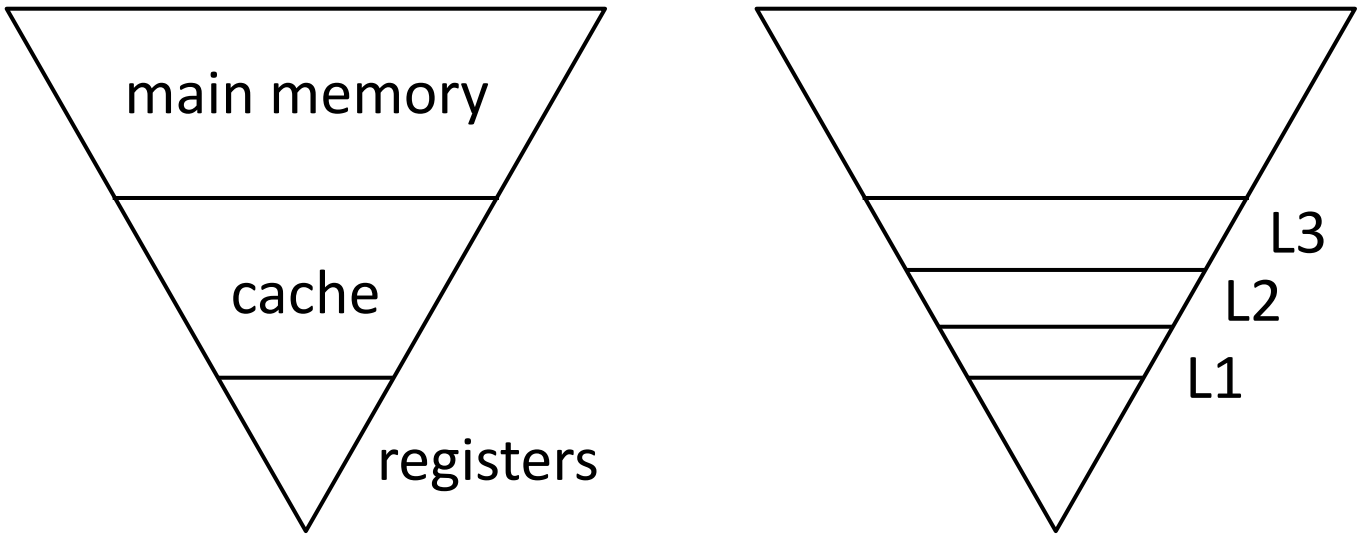


Caches

- Fast (but small) extra memory
  - holding identical copies of main momory
  - lower latency
  - higher bandwidth
  - ususally several levels (2 or 3)
  - same principle as virtual memory
- Memory request satisfied from
  - fast cache, if the data is stored there: ***cache hit***
  - or from slow main memory, if the data is not stored in the cache: ***cache miss***
- Issues:
  - Uniqueness and transparency of addressing
  - Finding a working set
  - Data consistency with main memory

Memory Hierarchy - Caches

- 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.

Cache Types

- 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)

Memory Access

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.

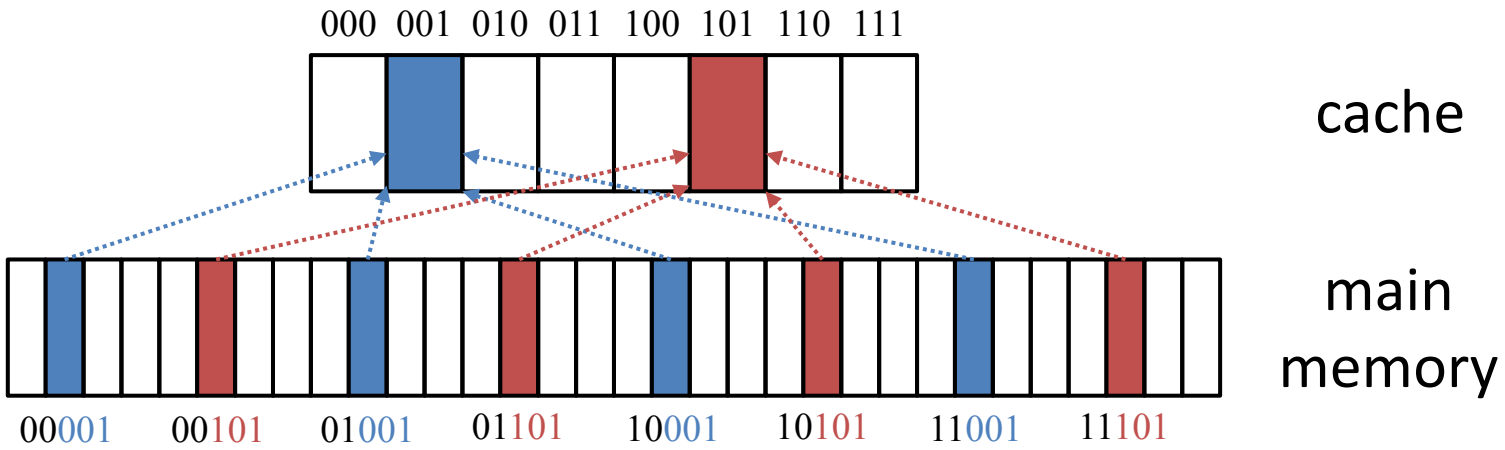
Associativity

1. direct mapped (associativity 1): Each main memoy word can be stored in one (and only) one location in the cache.
2. (fully) associative: A main memory word can be stored in any location in the cache
3. set associative (associativity k, typically k=2,4,8,...): Each main memory word can be stored in one of k places in the cache.

- 1 and 3 give rises to conflict misses.
- Direct mapped caches are faster, fully associative caches too expensive and slow (if reasonably large). Set-associative caches are a compromise.

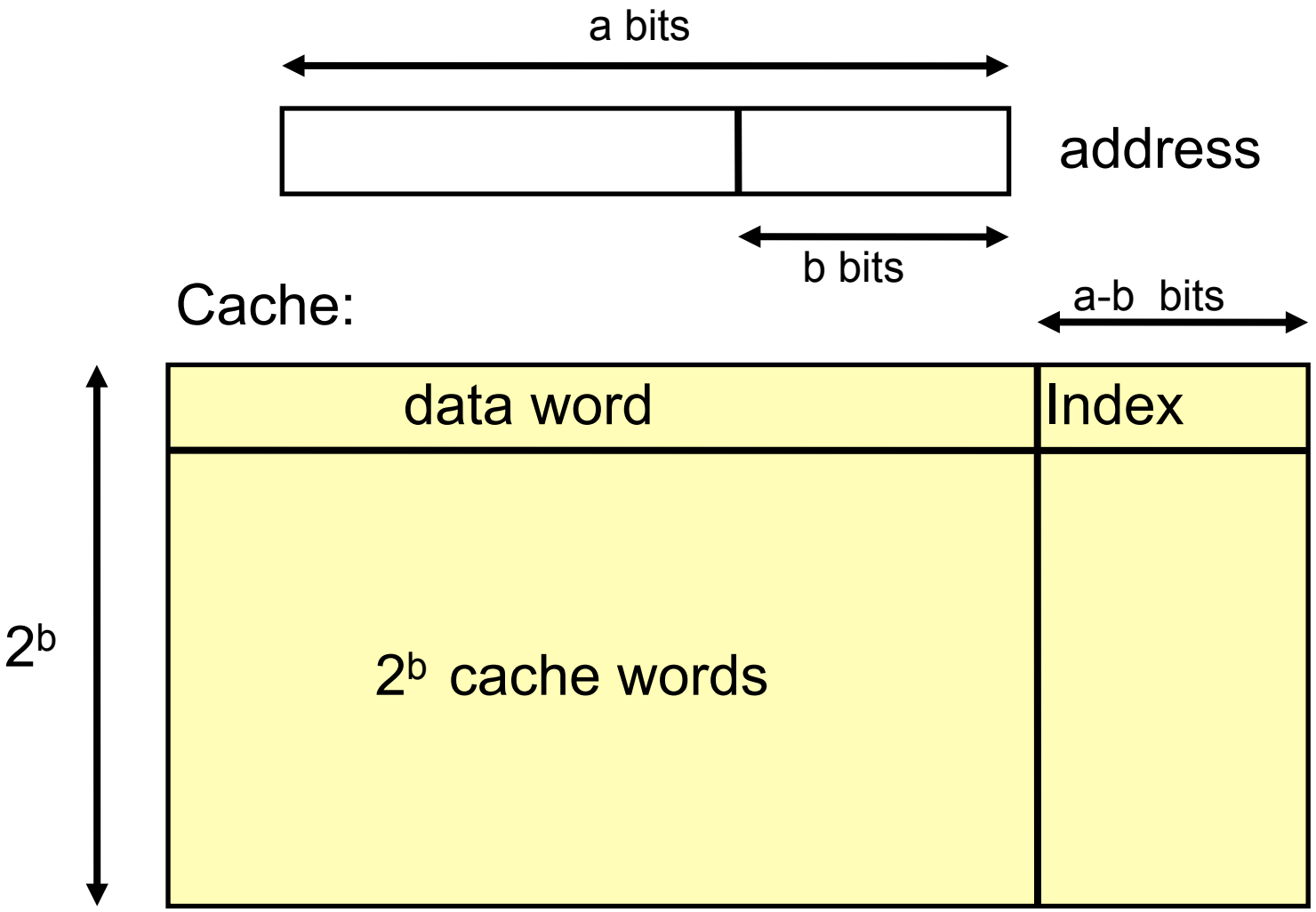
Direct Mapped Cache

- 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 mod N).
- Example: cache capacity N = 8



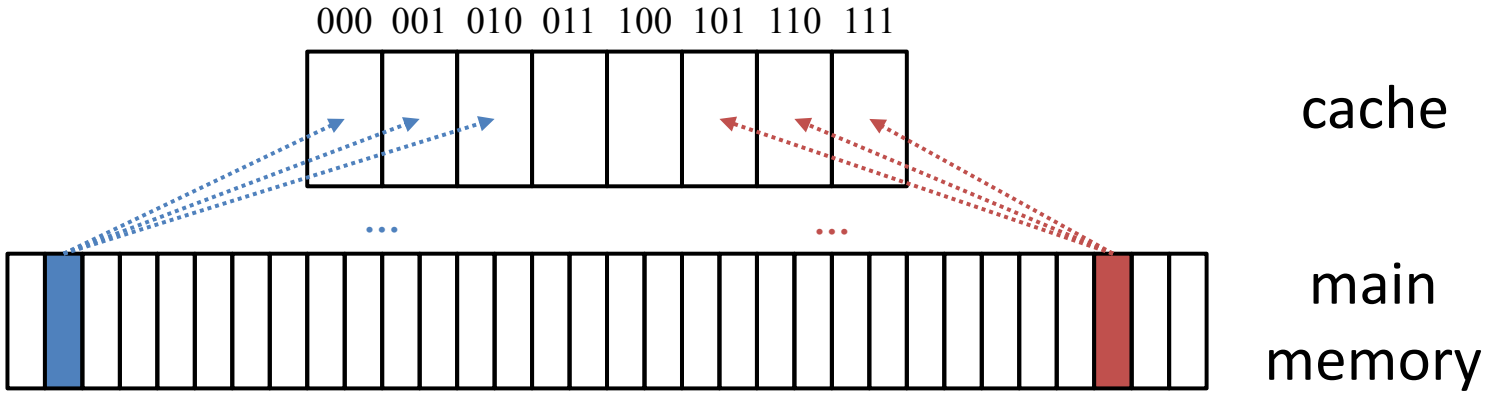
- Fast! But low hit rates & many cache misses …

Direct mapped cache



Fully Associative Cache

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



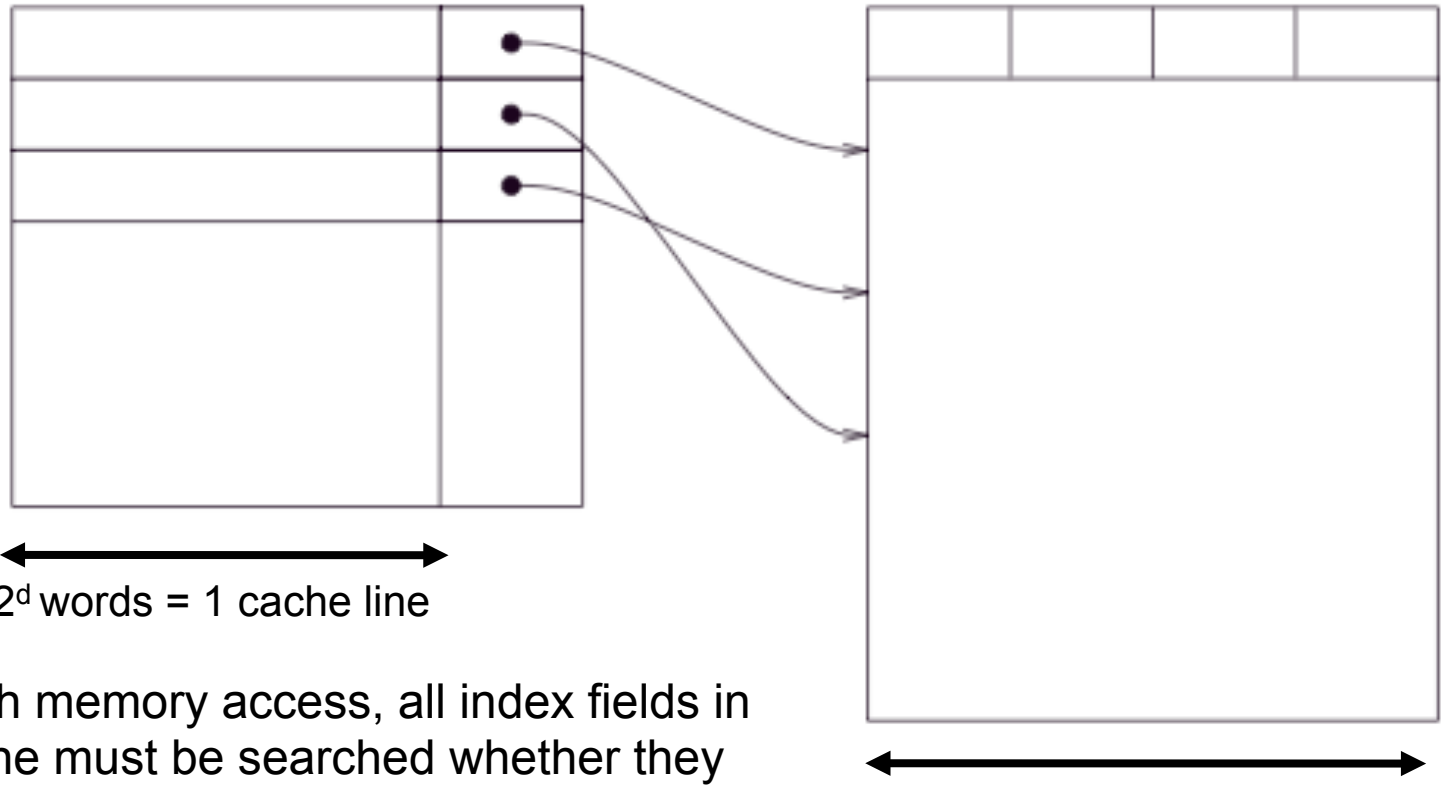
- Requires complex placement/retrieval strategies.
- High hit rates & few cache misses …. But slow!

Fully associative cache (with cache lines)

Cache with 2^b words

Index
a-d bits

Main memory with 2^a words,
a>>b



organized into 2^{b-d} lines

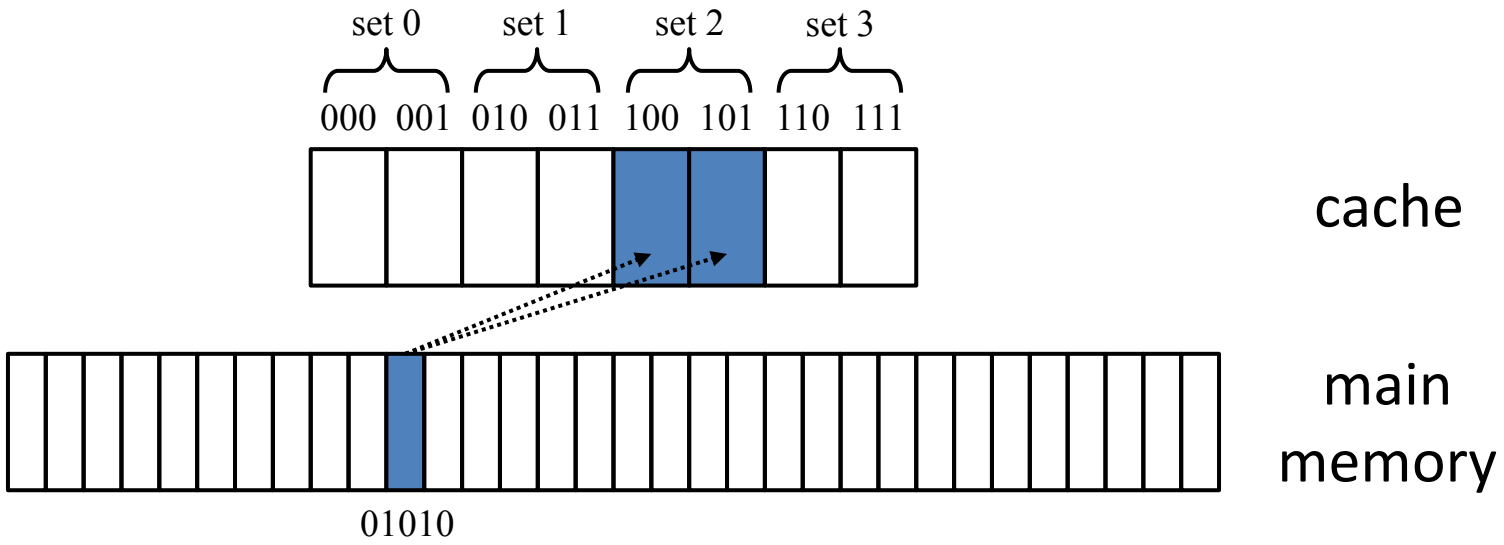
2^d words = 1 cache line

2^d words = 1 cache line

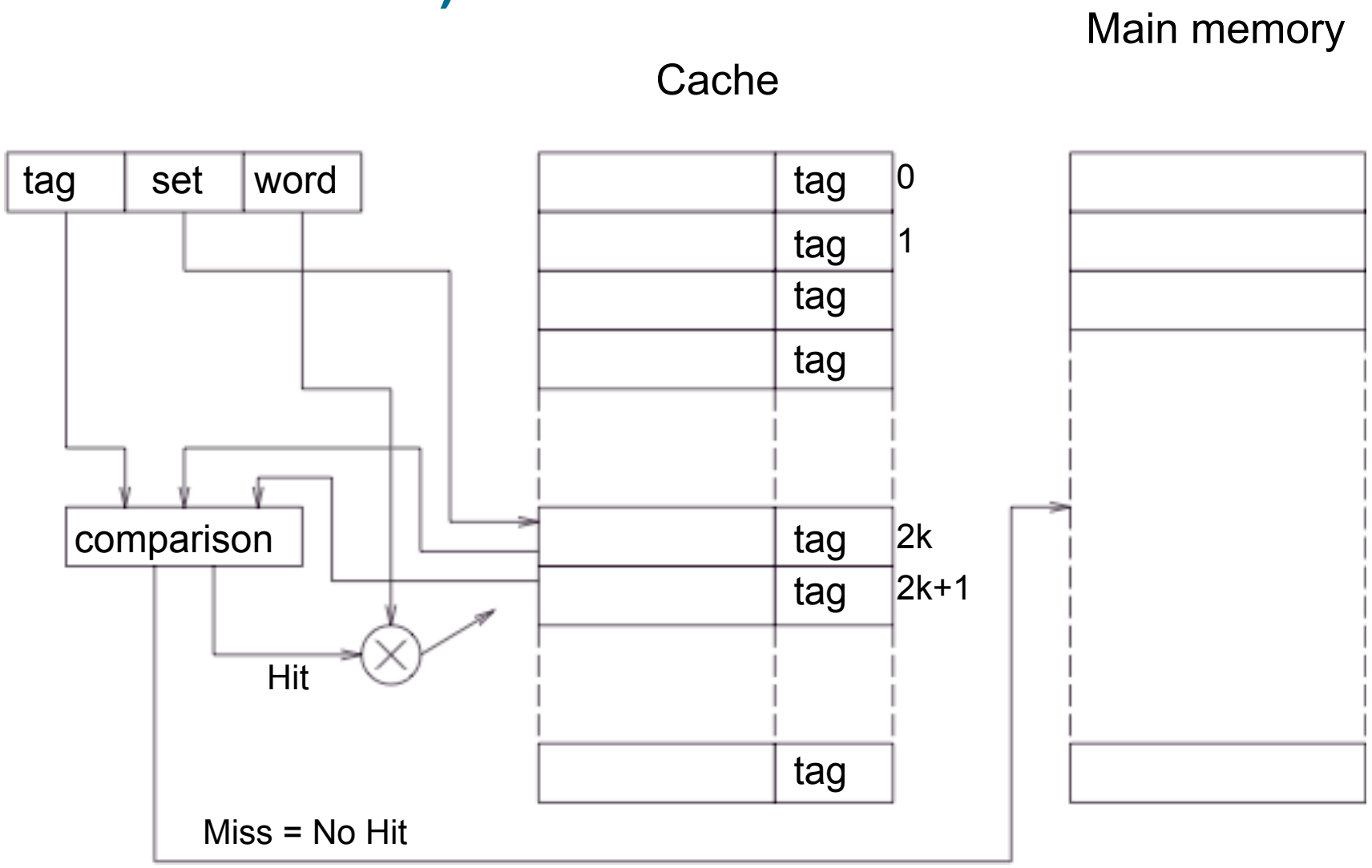
For each memory access, all index fields in the cache must be searched whether they contain the bit pattern that indicates that the requested word is in cache

k-way set associative cache

- 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



Two way set associative cache (with cache lines)



Eviction Strategies

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 …)

Other memory and cache aspects

- typically today three levels of cache
- caches favor temporal locality of memory access
- physical or virtual addresses
- write back or write through
  - „dirty bit“
- replacement strategy: (e.g. LRU or random)
- cache line size 32 B, 128 B, ...
  - Large cache lines help exploit spatial locality
  - But increase the danger of conflict misses
- TLB = translation lookaside buffer: CPU internal cache for holding base registers for virtual memory system: possible effect: TLB thrashing - has similar effects as another level of memory hierarchy
- cache coherence in shared memory systems

Classification of cache misses

- Compulsory (also called „cold start“ misses): For the first access to a memory location, there is an unavoidable cache miss
- Capacity: The cache is too small to hold all data items. data lines are overwritten according to the replacement policy.
- Conflict: In case of a direct mapped cache or a set-associative cache, it may not be possible to use the full cache size when too many items are mapped to the same set of slots in the cache.

- Good Info on Caches in Wipipedia:
  https://en.wikipedia.org/wiki/CPU_cache

Pipeline Techniques

- a form of on-chip parallelism
- break operation in several steps
  --> high clock rate
- dependencies: result is not finished but already needed for other operations
- (conditional) branches --> branch prediction
  - and/or speculative execution
- pipeline stalls

Example Pipeline Architecture

F	Load Instruction (from cache)
D	decode and identify resources (registers)
I	reserve registers
E_1	execute
E_2	execute
E_3	execute
W	write back results

- In sequential execution, each operation requires 7 clock cycles
- cycles/instruction (CPI) = 7

Optimization of a machine program

```
                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
                ===============================================
R1 = Mem(Y)     F D I E E E W
R2 = Mem(Z)       F D I E E E W
R3 = R1+R2          F D X X X I E E E W
Mem(X) = R3           F X X X D X X X I E E E W
R4 = Mem(B)                 F X X X D I E E E W
R5 = Mem(C)                       F D I E E E W
R6 = R4*R5                          F D X X X I E E E W
Mem(A) = R6                           F X X X D X X X I E E E W
```

- 8 instr. × 7 cycles = 56 cycles
- Pipelining: 26 cycles
- From start to start: 14 cycles
- From end to end: 20 cycles

Re-ordering operations

```
                    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7
                    ================================
R1 = Mem(Y)         F D I E E E W
R2 = Mem(Z)           F D I E E E W
R4 = Mem(B)             F D I E E E W
R5 = Mem(C)               F D I E E E W
R3 = R1+R2                  F D X I E E E W
R6 = R4*R5                    F D X X I E E E W
Mem(X) = R3                     F X X D X I E E E W
Mem(A) = R6                       F X X D X X I E E E W
```

- Instruction sequence: 18 cycles
- From end to end: 11 cycles (5 times faster than non-pipelined)

Further Aspects

- out-of-order execution
- prefetching of instructions (several sequences in the case of jumps)
- data forwarding
- prefetching of data
- (overlapping) register windows
- dynamic instruction scheduling (out of order execution)
- jump optimization
- superscalar pipelines
- very long instruction word (aka EPIC)

Basic Efficiency Guidelines

- choose best algorithm
- use efficient libraries
  - for Matrix-Matrix multiply, use DGEMM, see e.g.:
  - http://www.lrz-muenchen.de/services/software/mathematik/atlas/
- find good compiler options
- use suitable data layout

Use effective libraries

- Good libraries often outperform own software.
  - clever, sophisticated algorithms
  - optimized for special machine

Sources for libraries

- vendor independent
  - commercial: e.g. NAG, IMSL: only available as binary
  - then often optimized for specific platform
  - free codes, usually available as source code: e.g. netlib (lapack, odepack, and many more), usually not specifically optimized.
- vendor dependent:
  - often highly optimized
  - often re-implementation of free code
- Many libraries are quasi-standards (BLAS, LAPACK).

Compiler options

- Choose the right compiler: gcc may not be the best!
  - And C++/Java/Your-Favorite may not be the best language in the first place
- Compiler options (RTFM)
- Modern compilers have numerous flags to select individual code optimization options. Options are vendor- and compiler-specific ...
- -On (n=1...5) successively more aggressive optimization.
- -fast (may change roundoff behavior)
- -unroll
- -arch
- ....

Hints:

- read the manuals: man cc (man gcc, man f90)
- look up compiler options documented in www.specbench.org
- experiment and compare performance

Homework 2 (20%)

1. Your task is to implement a matrix-matrix multiplication
   $C = A \cdot B$, where A, B, and C are square matrices.
2. Feel free to use every known algorithm and programming technique to decrease the single-core runtime of your program as long as you adhere to the following guidelines. All three matrices are
   a) represented as a linearized one-dimensional array with adjacent elements.
   b) passed to your multiplication routine in a row-major format. Meaning: it is not allowed to store one of the input matrices in a transposed layout. However, the computation of the transpose is allowed to be a part of the multiplication routine itself such that the matrix is transposed after the start of the time measurement.
   c) Make sure you use double precision floating-point operations for your multiplication. The use of threads is prohibited.
3. Measure run times for matrix sizes of 1000×1000, 1500x1500, 2000×2000, 3000×3000, 4000×4000, 5000×5000,
4. Compare your run times with that of a highly optimized professional routine (eg. dgemm from BLAS level 3)
5. Summarize your findings on p pages with $p \leqq 2$
   1. due date Feb 16, 2015.
   2. submit as pdf file by e-mail to ulrich.ruede@fau.de

Data layout

- Access memory in order. For a two dimensional matrix

```
double a[n][m];
```

- the loops should be such that

```
for(i ...)
for(j....)
        a[i][j]
```

(For FORTRAN it must be the other way round)

Other data layout example:

Three vectors accessed together

```
double a[n], b[n], c[n];
```

can often be handled more efficiently by using

```
double abc[n][3];
```

(In FORTRAN again indices permuted)

What compilers can do

Common subexpression elimination

```
s1= a+b+c
s2= a+b-c
```

can be converted to

```
t=  a+b
s1= t+c
s2= t-c
```

What compilers can do

Problem: Trick does not work (without compiler option „-fast“ or equivalent) if e.g. written as

```
s1= a+c+b
s2= a-c+b
```

since associativity does not hold for floating point arithmetic and so the compiler cannot exchange

```
a+c+b
```

with

```
a+b+c
```

unless explicitly permitted (this is unclear in C?).

In a laguage with clean floating point handling the optimization should be prohibited (unless explictly enabled) since the results may differ significantly.

Common subexpression elimination

```
r= 2.0*x[i];
s= x[i]+5.0
```

`x[i]` is a common subexpression (needs only be loaded once).

```
r=f(i) + 2.0;
s=f(i) - 5.0;
```

For function calls, side effects prohibit optimization. Most compilers can optimize only in special cases when global information about the libraries is available to them.

Strength Reduction. Replace arithmetic expression by equivalent which can be executed faster. Examples:

- `2*i` → `i+i` (doubtful for float/double)
- `x**2` → `x*x`

Loop invariant code motion. Example:

```
for i = 1, ..., n
      a[i]= r*s*a[i];
```

Optimization: Compute r*s outside loop, code effectively becomes:

```
t= r*s;
for i = 1, ..., n
      a[i]= t*a[i];
```

Again a potential loss of accuracy may prevent this optimization when it requires law of associativity.

Compile Time Evaluation of Constants.

Simplification of induction variables

```
double a[4][100];
for i = 0...99
      x= x+ a[i][1];
```

The address of the array element `a[i][1]` is given by `4*i+1`, requiring an (expensive) integer multiplication. After optimization, the code is equivalent to

```
double a[4*100]
iadd= 1
for i= 0...99
      x= x+a[iadd]
      iadd= iadd+4
```

The index calculation is now done by a simple increment by 4 in the loop. Typically this rearragnement for induction variables is done automatically by the compiler, but it may fail in more complex constructs.

Register allocation and instruction scheduling

difficult and important

- Register allocation is NP complete, therefore heuristics are needed.
- Register allocation algorithms (in the compiler) often yield less than optimal results, but still likely better than a too naive allocation by the programmer
- control by programmer (register in C) rarely better than compiler or ignored anyway.

Elimination of dependencies

- true dependence
- anti dependence

```
x= y + c
y= 10
```

- output dependence

```
x= 10
x= 20
```

Dependent instructions cannot be executed simultaneously nor executed in consecutive cycles.

Fused Multiply-Add Instruction

On many modern processors there is an instruction which multiplies two operands and adds the result to a third.

Consider code

```
a= b + c*d + f*g
```

versus

```
a= c*d + f*g + b
```

Can reordering be done automatically?

Performance Optimization

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

Standard Algorithm

Naïve matrix-matrix multiplication:

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

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

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned} \qquad O(n^3)$$

Strassen Algorithm

The Strassen algorithm:

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

$$\begin{aligned} M_1 &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\ M_2 &= (A_{21} + A_{22}) \cdot B_{11} \\ M_3 &= A_{11} \cdot (B_{12} - B_{22}) \\ M_4 &= A_{22} \cdot (B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12}) \cdot B_{22} \\ M_6 &= (A_{21} - A_{11}) \cdot (B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \end{aligned}$$

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

Cache Aware Program Optimization

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)

Temporal Locality

- 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.

Spatial Locality

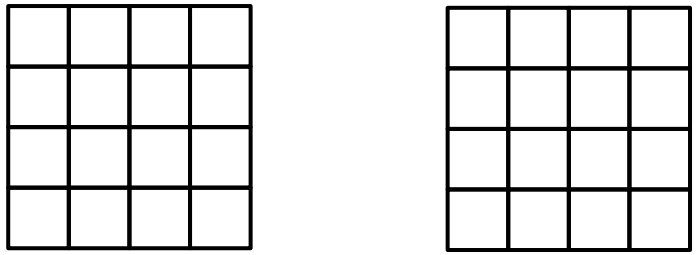
- If data is transferred from main memory to the cache, always **cache lines** are transferred, never just single data items.
- Cache line: multiple data items that reside besides each other in the main memory
- Optimizing for spatial locality:
  - If you have to access multiple data items, try to reorder 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!

Matrix Transposition

- 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

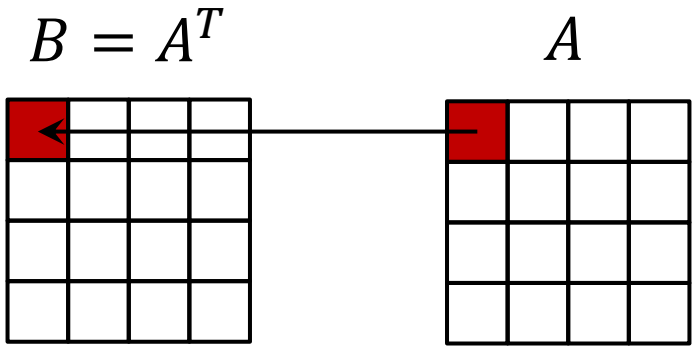


cache line size: 2 elements
size of cache: 4 cache lines

Matrix Transposition

- 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];
```



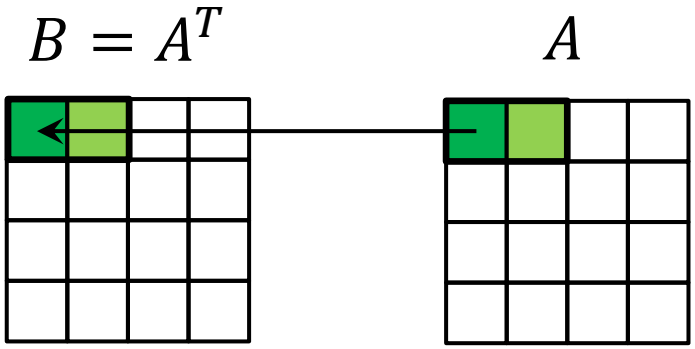
cache miss cache miss

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

Matrix Transposition

- 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];
```

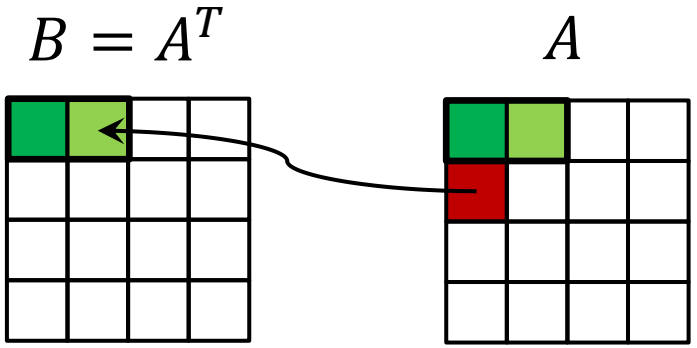


cache line size: 2 elements
size of cache: 4 cache lines

Matrix Transposition

- 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];
```



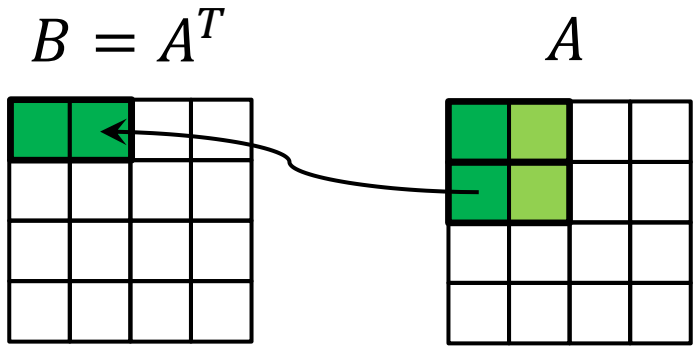
cache hit cache miss

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

Matrix Transposition

- 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];
```

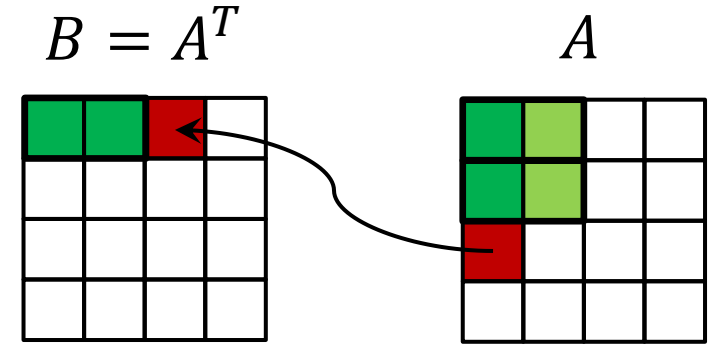


cache line size: 2 elements
size of cache: 4 cache lines

Matrix Transposition

- 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];
```



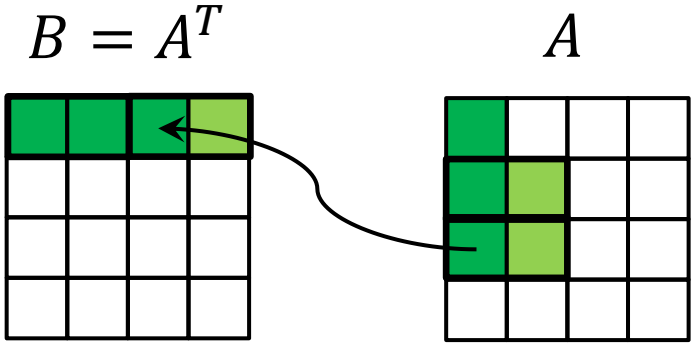
cache miss cache miss

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

Matrix Transposition

- Naïve, standard implementation of a matrix transposition:

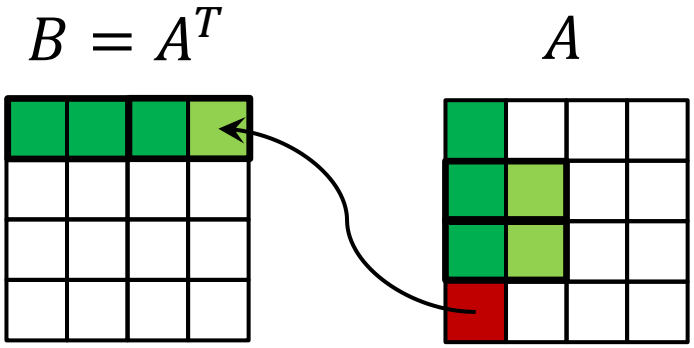
```
for( int i = 0; i < n; ++i )
  for( int j = 0; j < n; ++j )
    b[i][j] = a[j][i];
```



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

Matrix Transposition

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

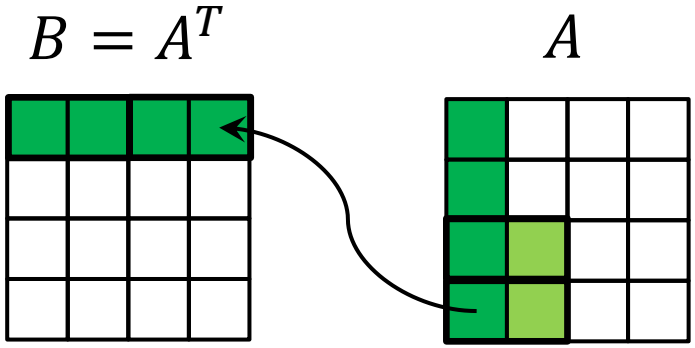


cache hit cache miss

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

Matrix Transposition

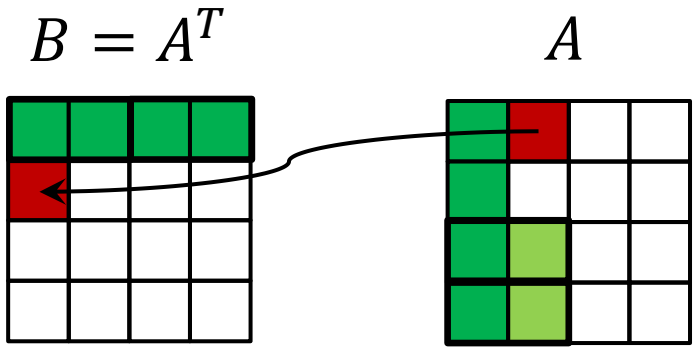
```
for( int i = 0; i < n; ++i )
  for( int j = 0; j < n; ++j )
    b[i][j] = a[j][i];
```



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

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

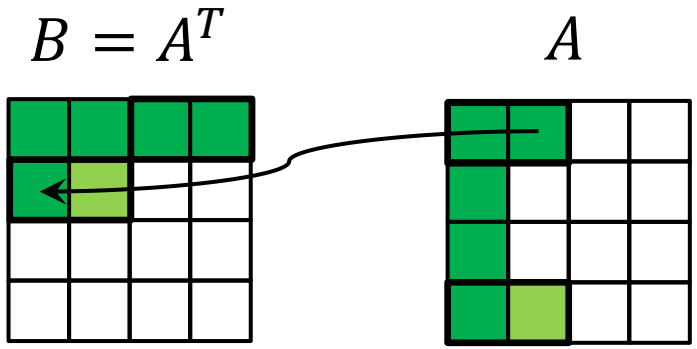
cache miss cache miss

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

Matrix Transposition

- 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];
```

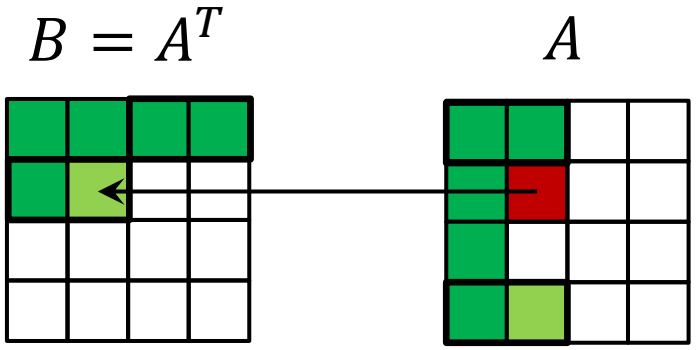


cache line size: 2 elements
size of cache: 4 cache lines

Matrix Transposition

- 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

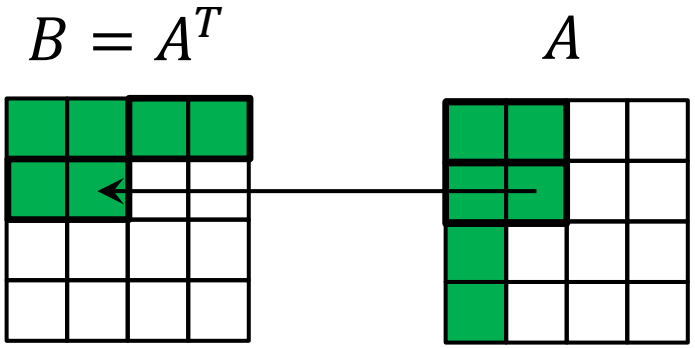
cache hit cache miss

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

Matrix Transposition

- 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 …

Matrix Transposition

- 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];
```

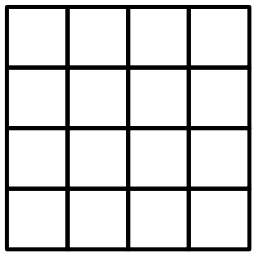
+ Stride-1 access for array **b**
– Stride-n access for array **a** → cache lines are rarely/never utilized

Blocking

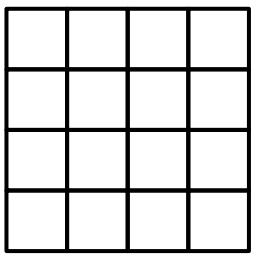
- Square blocking (optimizing temporal locality):

```
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

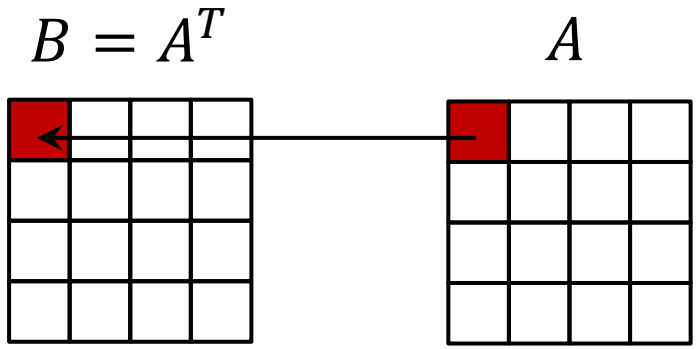


cache line size: 2 elements
size of cache: 4 cache lines
block size: 2 x 2

Blocking

- 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];
```



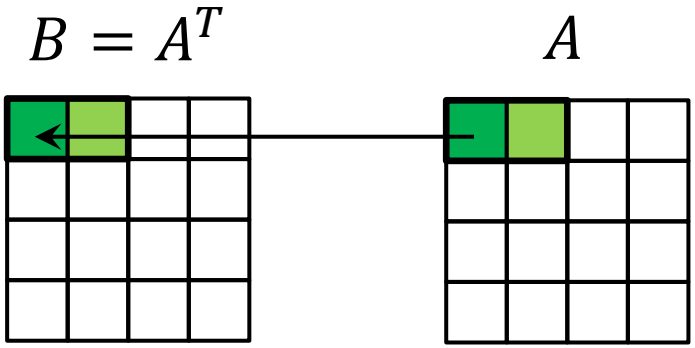
cache miss cache miss

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

Blocking

- 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];
```

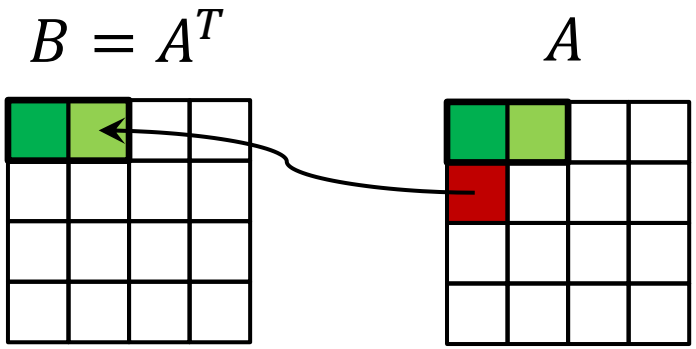


cache line size: 2 elements
size of cache: 4 cache lines
block size: 2 x 2

Blocking

- 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];
```



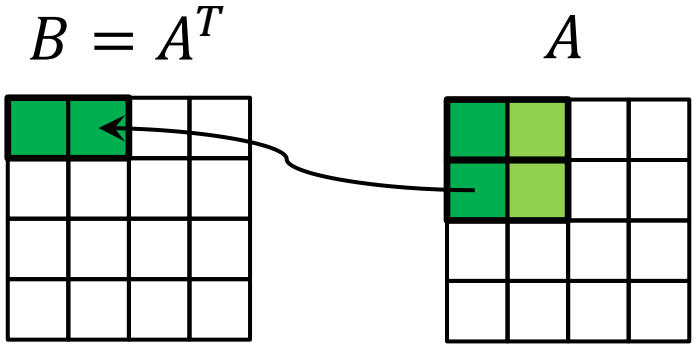
cache hit cache miss

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

Blocking

- 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];
```

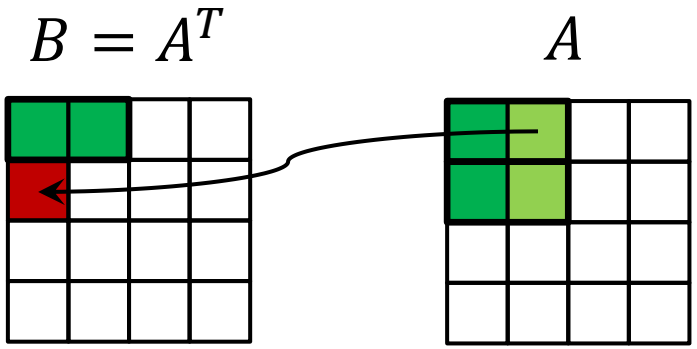


cache line size: 2 elements
size of cache: 4 cache lines
block size: 2 x 2

Blocking

- 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];
```



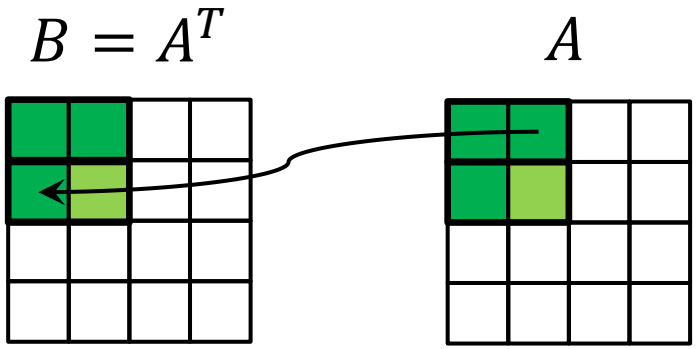
cache line size: 2 elements
size of cache: 4 cache lines
block size: 2 x 2

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

Blocking

- 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];
```

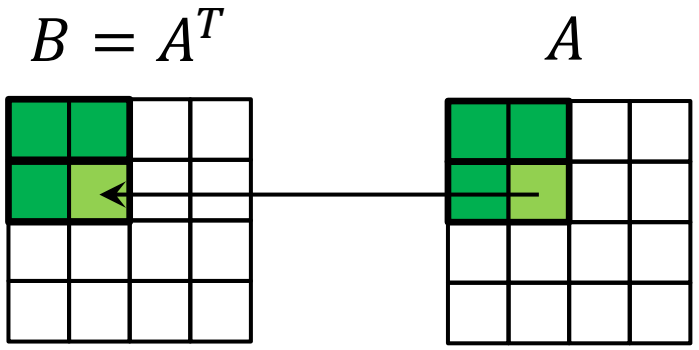


cache line size: 2 elements
size of cache: 4 cache lines
block size: 2 x 2

Blocking

- 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];
```



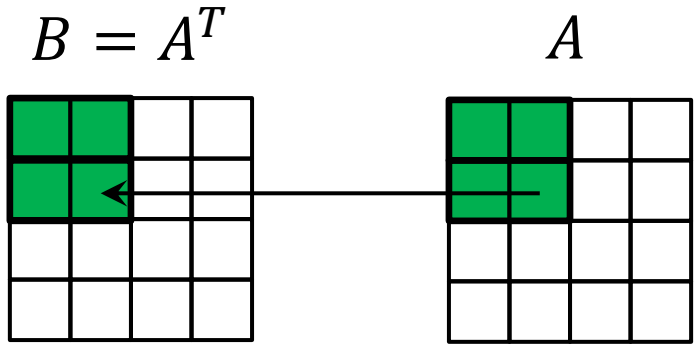
cache line size: 2 elements
size of cache: 4 cache lines
block size: 2 x 2

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

Blocking

- 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 …

Blocking

- 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

- 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

Cache Thrashing

- 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.

Cache Thrashing

- 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.