How to Write Fast Numerical Code

Spring 2011
Lecture 7

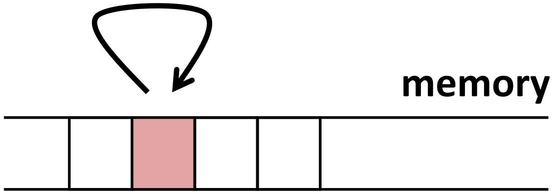
Instructor: Markus Püschel

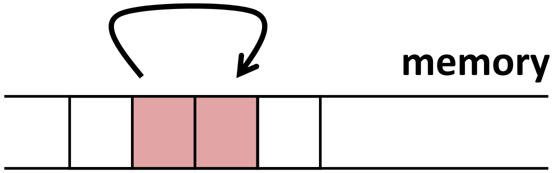
TA: Georg Ofenbeck

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Last Time: Locality

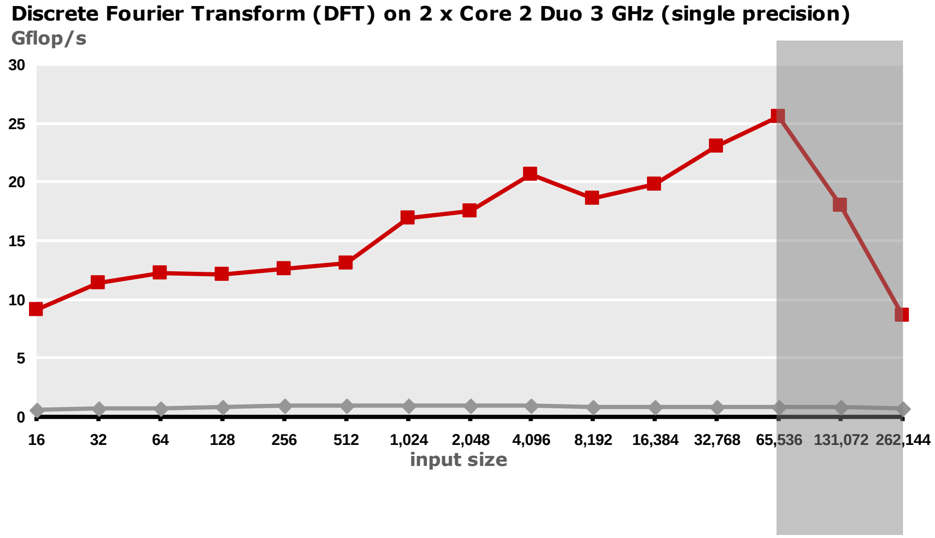
- **Temporal and Spatial**



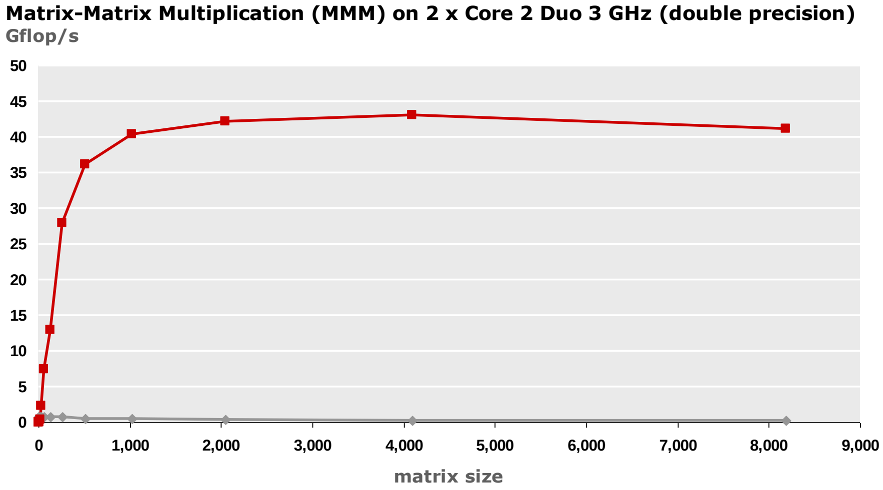


Last Time: Reuse

FFT: O(log(n)) reuse



MMM: O(n) reuse



Today

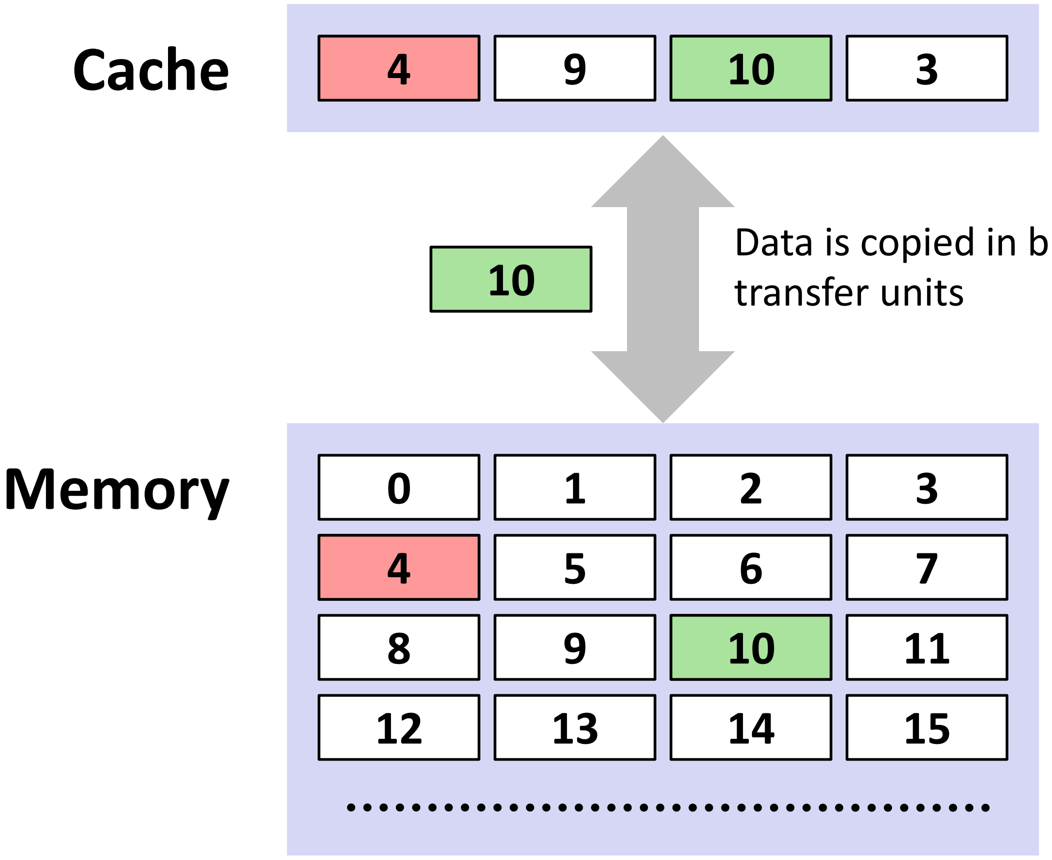
- Caches

Cache

- *Definition:* **Computer memory with short access time used for the storage of frequently or recently used instructions or data**

- **Naturally supports *temporal locality***
- ***Spatial locality* is supported by transferring data in blocks**
  - Core 2: one block = 64 B = 8 doubles

General Cache Mechanics

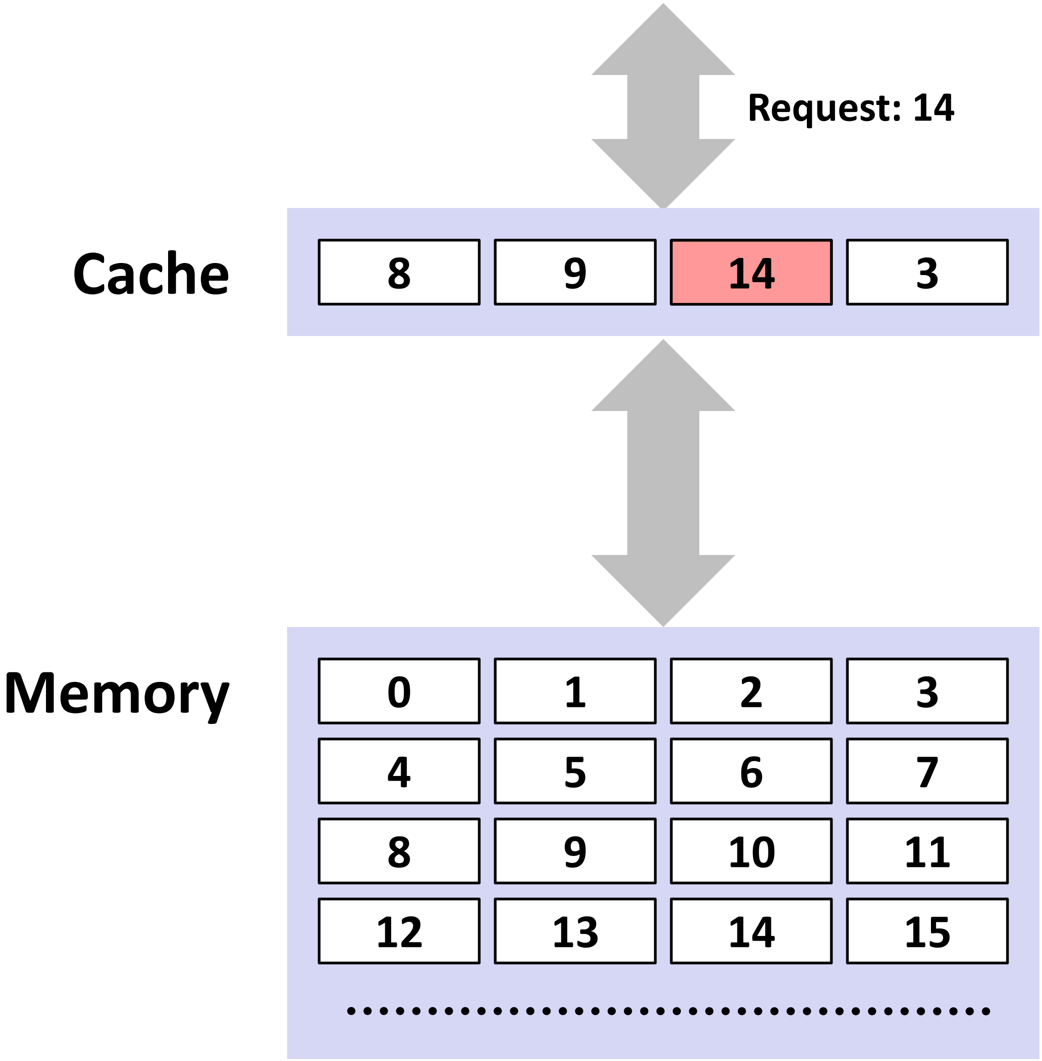


Smaller, faster, more expensive memory caches a subset of the blocks

Data is copied in block-sized transfer units

Larger, slower, cheaper memory viewed as partitioned into “blocks”

General Cache Concepts: Hit



Request: 14

Cache

8	9	14	3

Memory

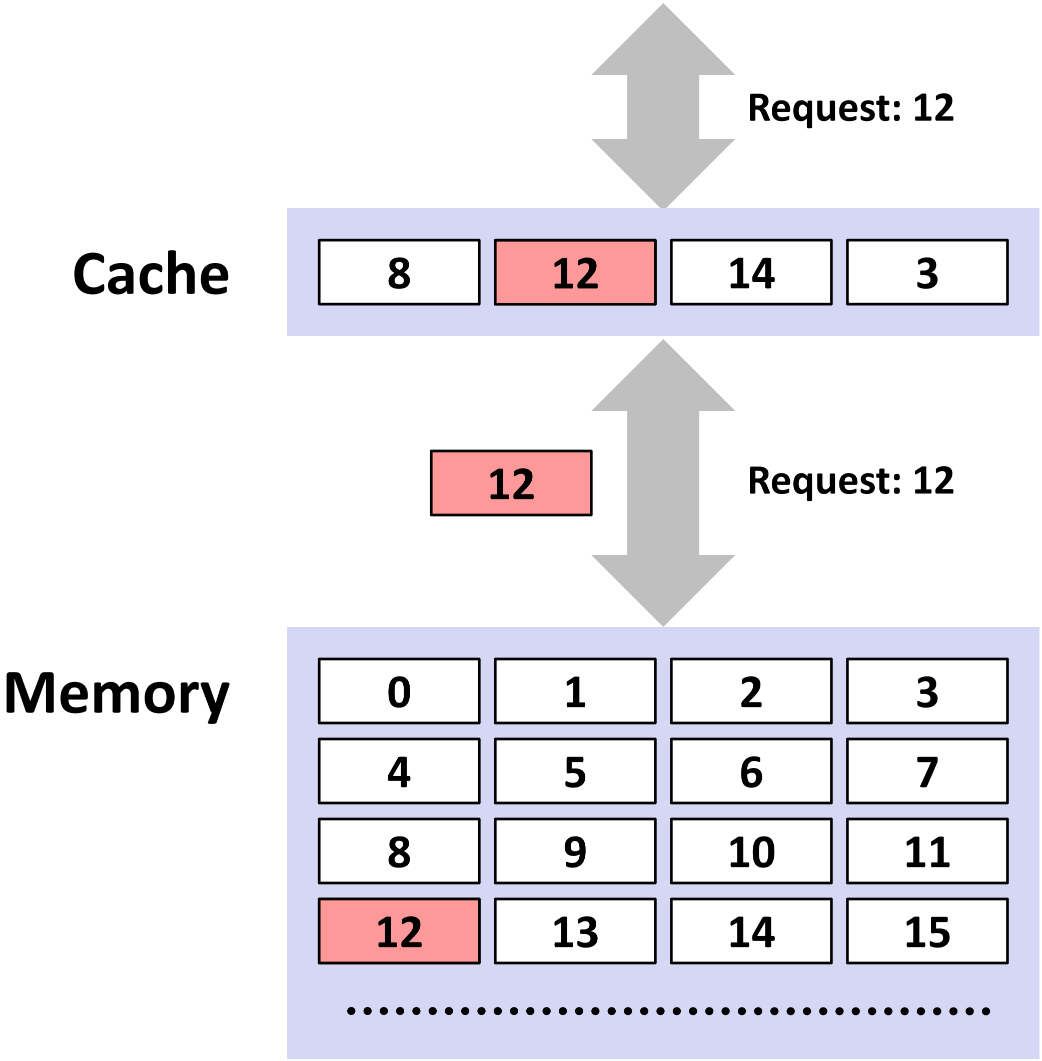
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

……

Data in block b is needed

Block b is in cache:
Hit!

General Cache Concepts: Miss



Data in block b is needed

Block b is not in cache:
Miss!

Block b is fetched from memory

Block b is stored in cache

- *Placement policy:* determines where b goes
- *Replacement policy:* determines which block gets evicted (victim)

Types of Cache Misses (The 3 C’s)

- ***Compulsory (cold)*** **miss**
  - Occurs on first access to a block

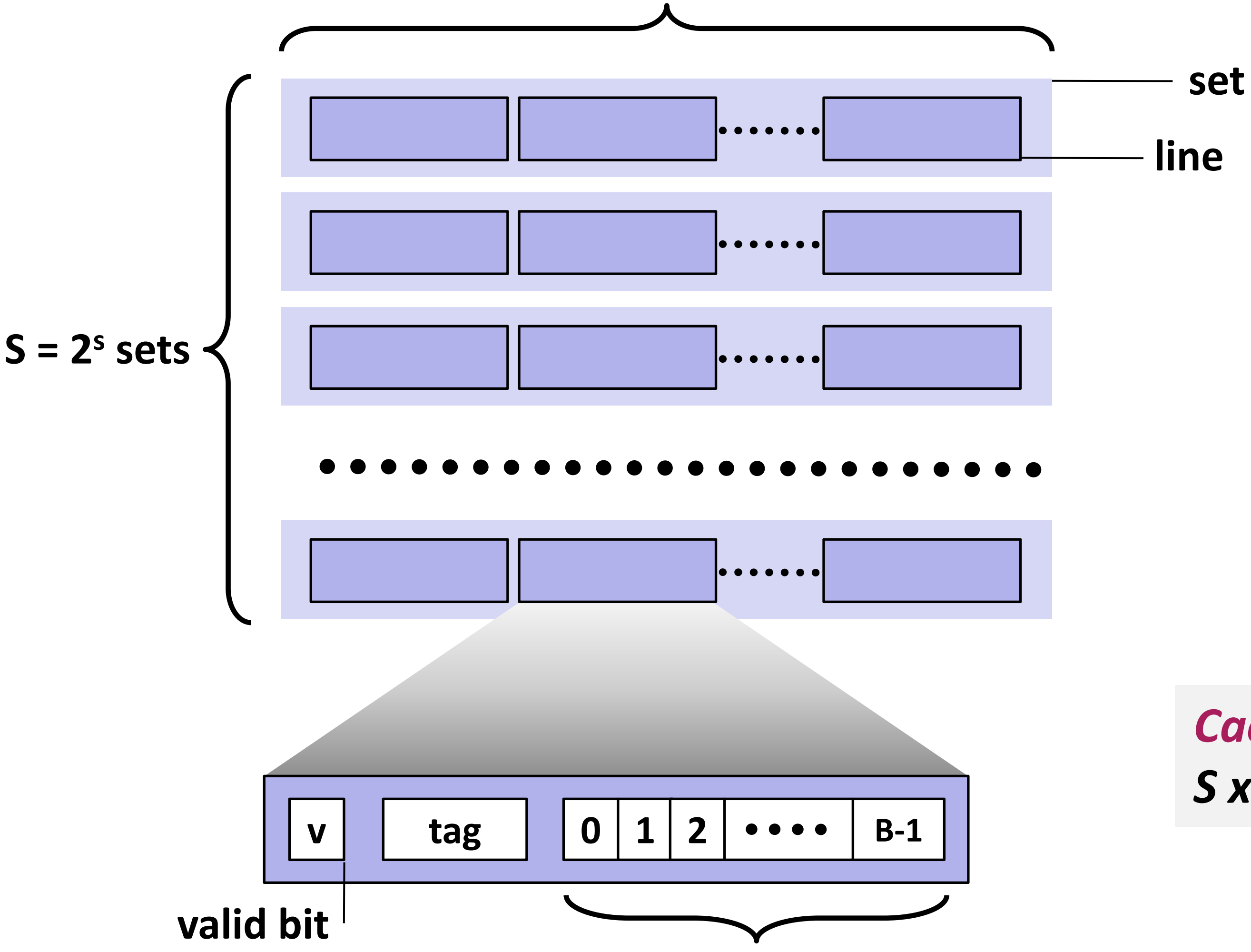
- ***Capacity*** **miss**
  - Occurs when working set is larger than the cache

- ***Conflict*** **miss**
  - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot

Cache Performance Metrics

- **Miss Rate**
  - Fraction of memory references not found in cache: misses / accesses
= 1 – hit rate
- **Hit Time**
  - Time to deliver a block in the cache to the processor
  - Core 2:
3 clock cycles for L1
14 clock cycles for L2
- **Miss Penalty**
  - Additional time required because of a miss
  - Core 2: about 100 cycles for L2 miss

General Cache Organization (S, E, B)



$E = 2^e$ lines per set

E = associativity, E=1: direct mapped

set

line

$S = 2^s$ sets

v

tag

0 1 2 •••• B-1

valid bit

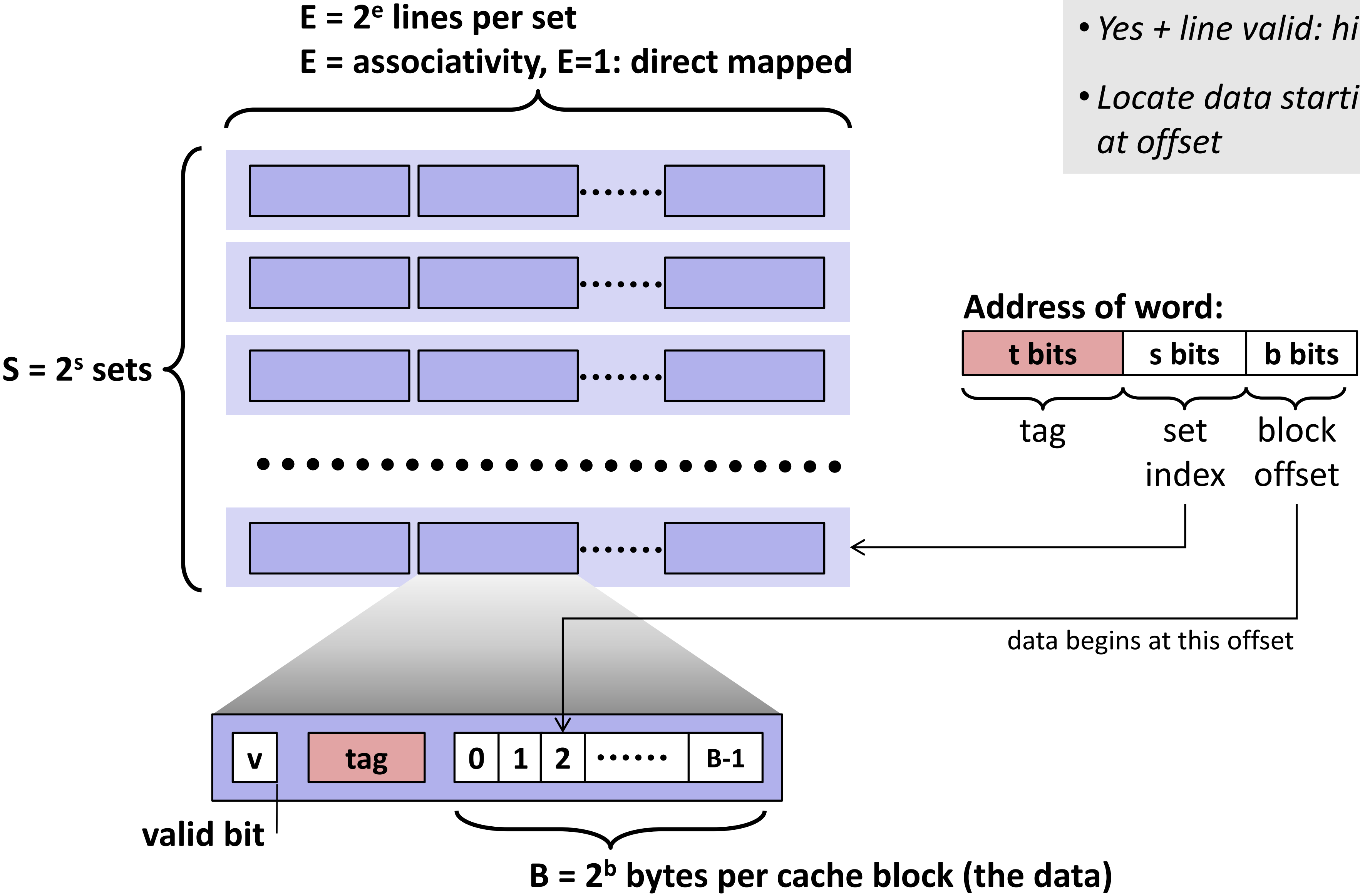
$B = 2^b$ bytes per cache block (the data)

Cache size:

S x E x B data bytes

Cache Read

- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*



$E = 2^e$ lines per set

E = associativity, E=1: direct mapped

$S = 2^s$ sets

Address of word:

t bits	**s bits**	**b bits**
tag	set index	block offset

data begins at this offset

v	tag	0	1	2	••••••	B-1

valid bit

$B = 2^b$ bytes per cache block (the data)

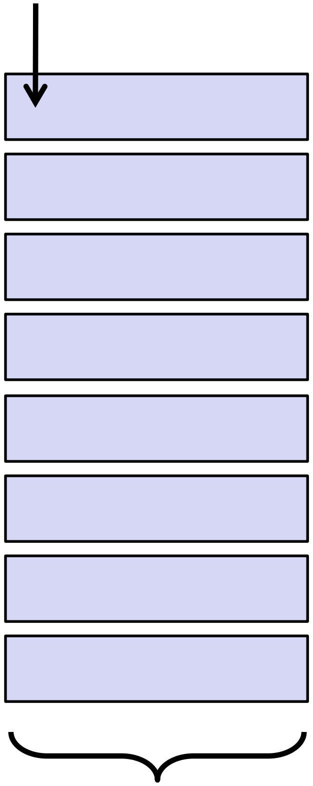
Example (S=8, E=1)

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; i < 16; i++)
        for (i = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

Ignore the variables sum, i, j

assume: cold (empty) cache,
a[0][0] goes here



B = 32 byte = 4 doubles

blackboard

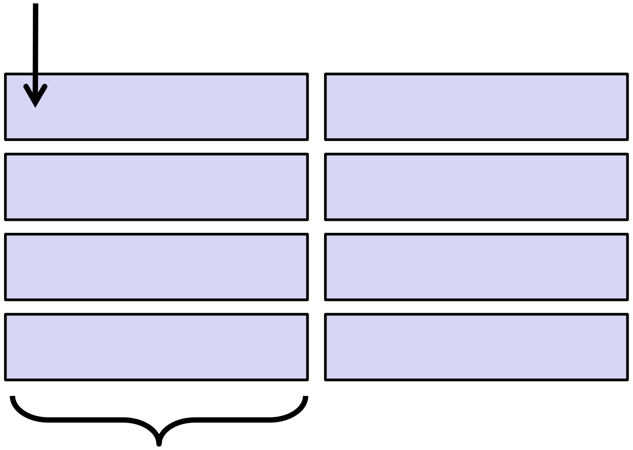
Example (S=4, E=2)

Ignore the variables sum, i, j

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; i < 16; i++)
        for (i = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

assume: cold (empty) cache, a[0][0] goes here



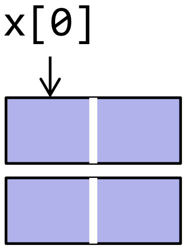
B = 32 byte = 4 doubles

blackboard

What about writes?

- **What to do on a write-hit?**
  - ***Write-through:*** write immediately to memory
  - ***Write-back:*** defer write to memory until replacement of line (*needs a valid bit)*
- **What to do on a write-miss?**
  - ***Write-allocate:*** load into cache, update line in cache
  - ***No-write-allocate:*** writes immediately to memory
- **Core 2:**
  - Write-back + Write-allocate

Small Example, Part 1



Cache:

E = 1 (direct mapped)
S = 2
B = 16 (2 doubles)

Array (accessed twice in example)

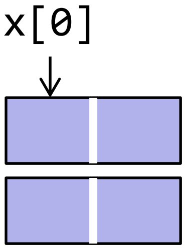
`x = x[0], …, x[7]`

```
% Matlab style code
for j = 0:1
   for i = 0:7
      access(x[i])
```

Access pattern: 0123456701234567
Hit/Miss: MHMHMHMHMHMHMHMH

Result: 8 misses, 8 hits
Spatial locality: yes
Temporal locality: no

Small Example, Part 2



Cache:

E = 1 (direct mapped)

S = 2

B = 16 (2 doubles)

Array (accessed twice in example)

`x = x[0], …, x[7]`

```
% Matlab style code
for j = 0:1
   for i = 0:2:7
      access(x[i])
   for i = 1:2:7
      access(x[i])
```

Access pattern: 0246135702461357

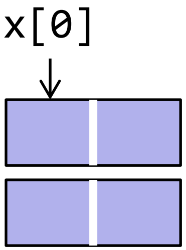
Hit/Miss: MMMMMMMMMMMMMMMM

Result: 16 misses

Spatial locality: no

Temporal locality: no

Small Example, Part 3



Cache:

E = 1 (direct mapped)

S = 2

B = 16 (2 doubles)

Array (accessed twice in example)

`x = x[0], …, x[7]`

```
% Matlab style code
for j = 0:1
    for k = 0:1
        for i = 0:3
            access(x[i+4j])
```

Access pattern: 0123012345674567

Hit/Miss: MHMHHHHHMHMHHHHH

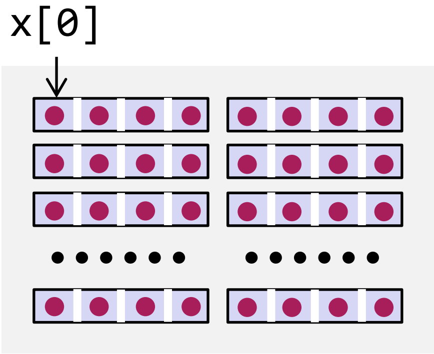
Result: 4 misses, 8 hits (is optimal, why?)

Spatial locality: yes

Temporal locality: yes

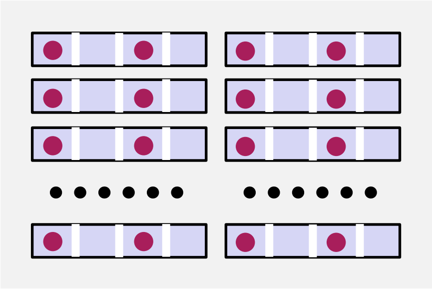
The Killer: Two-Power Strided Access

`x = x[0], …, x[n-1], n >> cache size`



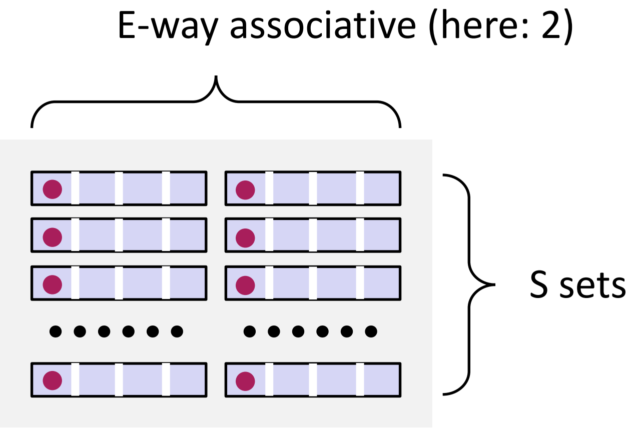
Stride 1: 0123…

Spatial locality
Full cache used



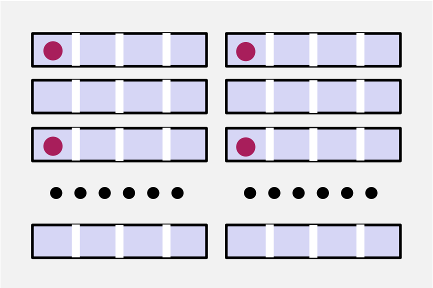
Stride 2: 0 2 4 6 …

Some spatial locality
1/2 cache used



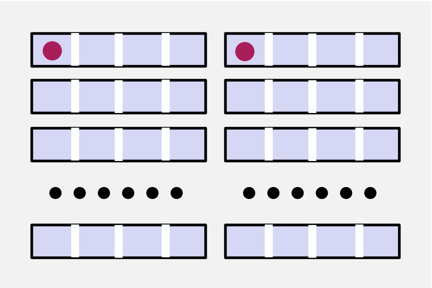
Stride 4: 0 4 8 12 …

No spatial locality
1/4 cache used



Stride 8: 0 8 16 24 …

No spatial locality
1/8 cache used



Same for larger stride

Stride 4S: 0 4S 8S 16S …

No spatial locality
1/(4S) cache used

The Killer: Where Does It Occur?

- **Accessing two-power size 2D arrays (e.g., images) columnwise**
  - 2d Transforms
  - Stencil computations
  - Correlations
- **Various transform algorithms**
  - Fast Fourier transform
  - Wavelet transforms
  - Filter banks

Today

- Linear algebra software: history, LAPACK and BLAS
- Blocking: key to performance
- MMM
- ATLAS: MMM program generator

Linear Algebra Algorithms: Examples

- Solving systems of linear equations
- Eigenvalue problems
- Singular value decomposition
- LU/Cholesky/QR/… decompositions
- … and many others

- Make up most of the numerical computation across disciplines (sciences, computer science, engineering)
- Efficient software is extremely relevant

The Path to LAPACK

- **EISPACK and LINPACK**
  - Libraries for linear algebra algorithms
  - Developed in the early 70s
  - Jack Dongarra, Jim Bunch, Cleve Moler, Pete Stewart, …
  - LINPACK still used as benchmark for the TOP500 (Wiki) list of most powerful supercomputers
- **Problem:**
  - Implementation “vector-based,” i.e., little locality in data access
  - Low performance on computers with deep memory hierarchy
  - Became apparent in the 80s
- **Solution: LAPACK**
  - Reimplement the algorithms “block-based,” i.e., with locality
  - Developed late 1980s, early 1990s
  - Jim Demmel, Jack Dongarra et al.

LAPACK and BLAS

- **Basic Idea:**

- **Basic Linear Algebra Subroutines (BLAS, list)**
  - BLAS 1: vector-vector operations (e.g., vector sum) *Reuse: O(1)*
  - BLAS 2: matrix-vector operations (e.g., matrix-vector product) *Reuse: O(1)*
  - BLAS 3: matrix-matrix operations (e.g., MMM) ***Reuse: O(n)***
- **LAPACK implemented on top of BLAS**
  - Using BLAS 3 as much as possible

Why is BLAS3 so important?

- Using BLAS3 = blocking = enabling reuse
- Cache analysis for blocking MMM (blackboard)

- *Blocking* (for the memory hierarchy) is the single most important optimization for dense linear algebra algorithms

- *Unfortunately:* The introduction of multicore processors requires a reimplementation of LAPACK
  just multithreading BLAS is not good enough