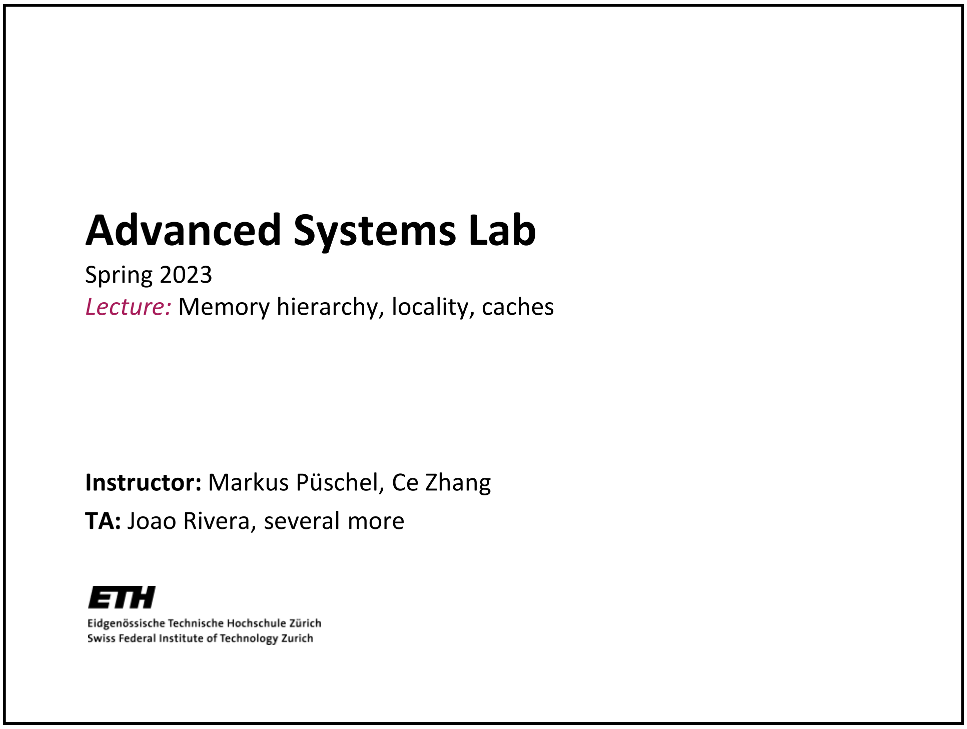


Advanced Systems Lab

Spring 2023

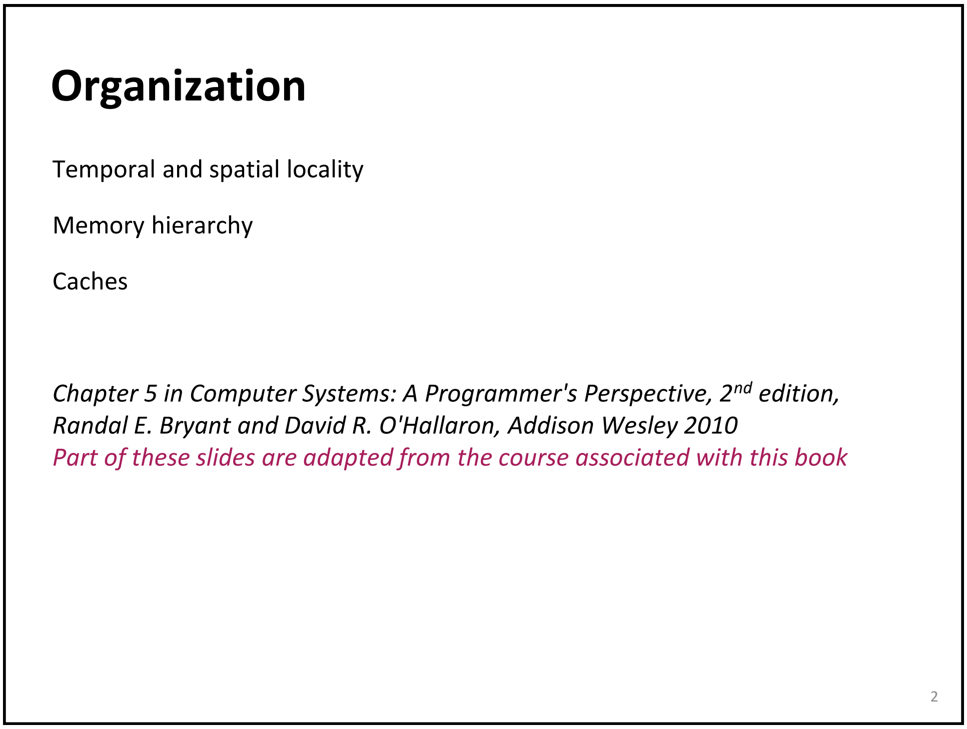
Lecture: Memory hierarchy, locality, caches

Instructor: Markus Püschel, Ce Zhang

TA: Joao Rivera, several more

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

1



Organization

Temporal and spatial locality

Memory hierarchy

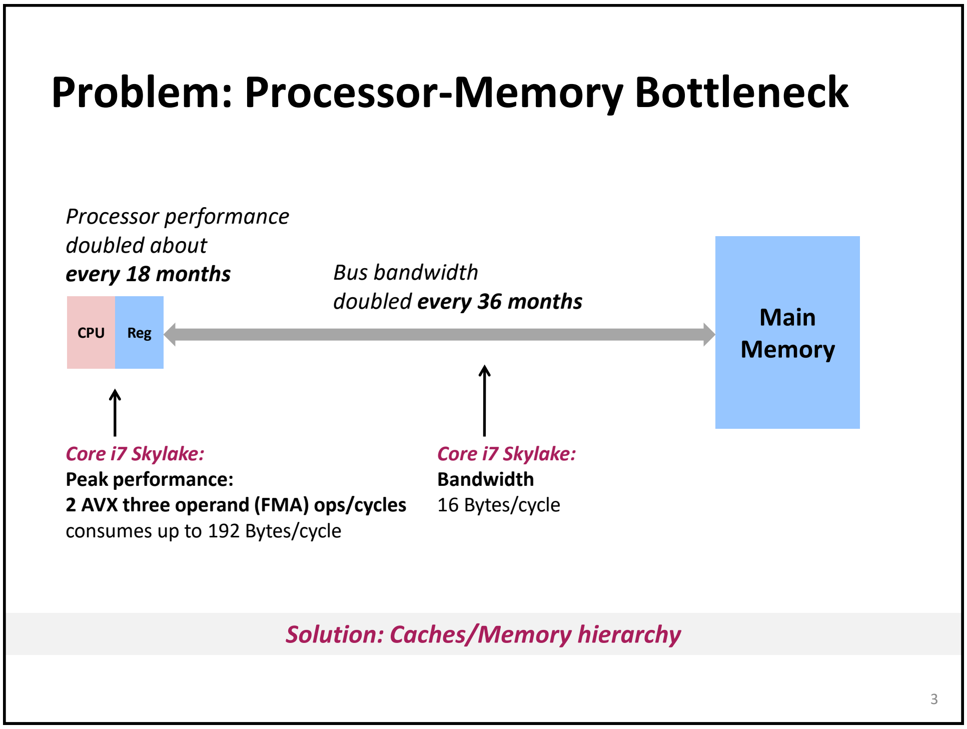
Caches

Chapter 5 in Computer Systems: A Programmer's Perspective, 2nd edition, Randal E. Bryant and David R. O'Hallaron, Addison Wesley 2010

Part of these slides are adapted from the course associated with this book

2

2



Problem: Processor-Memory Bottleneck

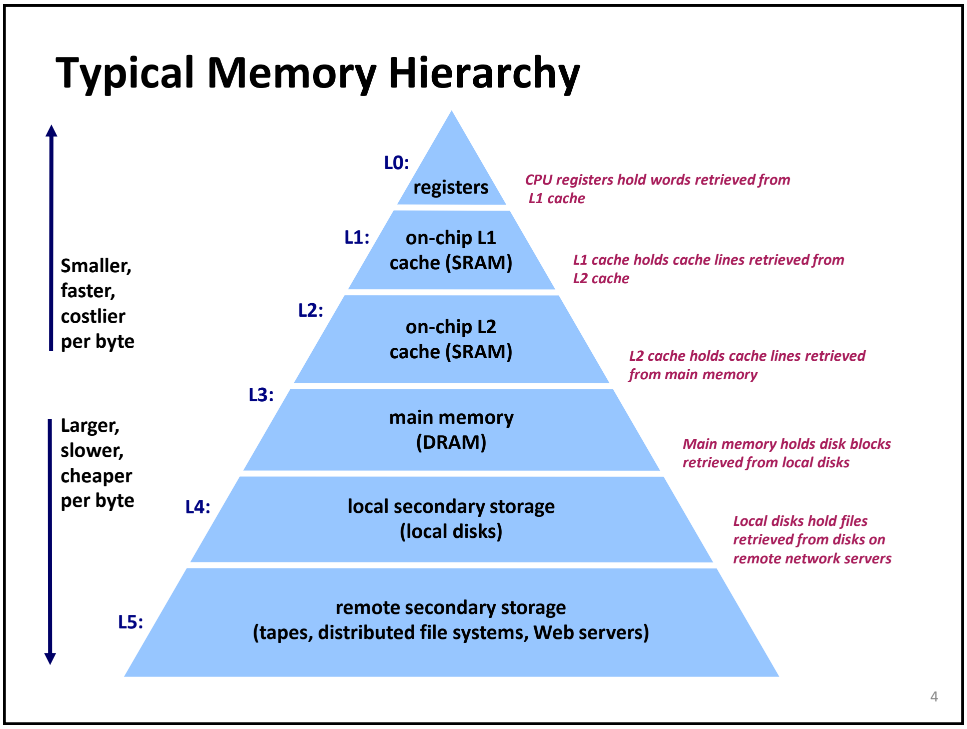
Processor performance doubled about ***every 18 months***

Bus bandwidth doubled ***every 36 months***

Core i7 Skylake:
Peak performance:
2 AVX three operand (FMA) ops/cycles
consumes up to 192 Bytes/cycle

Core i7 Skylake:
Bandwidth
16 Bytes/cycle

Solution: Caches/Memory hierarchy



Typical Memory Hierarchy

Smaller, faster, costlier per byte

Larger, slower, cheaper per byte

L0:
registers
L1:
on-chip L1 cache (SRAM)
L2:
on-chip L2 cache (SRAM)
L3:
main memory (DRAM)
L4:
local secondary storage (local disks)
L5:
remote secondary storage (tapes, distributed file systems, Web servers)

CPU registers hold words retrieved from L1 cache

L1 cache holds cache lines retrieved from L2 cache

L2 cache holds cache lines retrieved from main memory

Main memory holds disk blocks retrieved from local disks

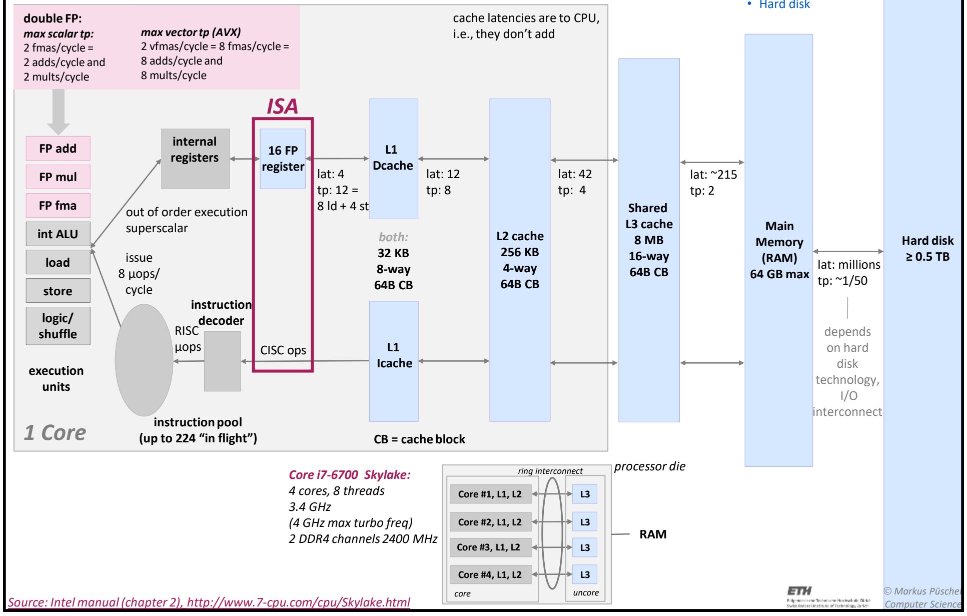
Local disks hold files retrieved from disks on remote network servers

Abstracted Microarchitecture: Example Core i7 Skylake (2015)

Throughput (tp) is measured in doubles/cycle. For example: 4. *Numbers are for loading into registers.*
Latency (lat) is measured in cycles
1 double floating point (FP) = 8 bytes
fma = fused multiply-add
Rectangles not to scale

Memory hierarchy:

- Registers
- L1 cache
- L2 cache
- L3 cache
- Main memory
- Hard disk



Core i7-6700 Skylake:
4 cores, 8 threads
3.4 GHz
(4 GHz max turbo freq)
2 DDR4 channels 2400 MHz

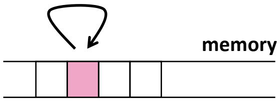
Source: Intel manual (chapter 2), http://www.7-cpu.com/cpu/Skylake.html

Why Caches Work: Locality

Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently
History of locality

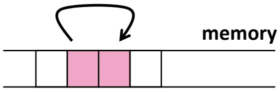
Temporal locality:

Recently referenced items are likely to be referenced again in the near future



Spatial locality:

Items with nearby addresses tend to be referenced close together in time



Example: Locality?

```
sum = 0;
for (i = 0; i < n; i++)
  sum += a[i];
return sum;
```

Data:

- *Temporal:* ***sum*** *referenced in each iteration*
- *Spatial: array* ***a[]*** *accessed consecutively*

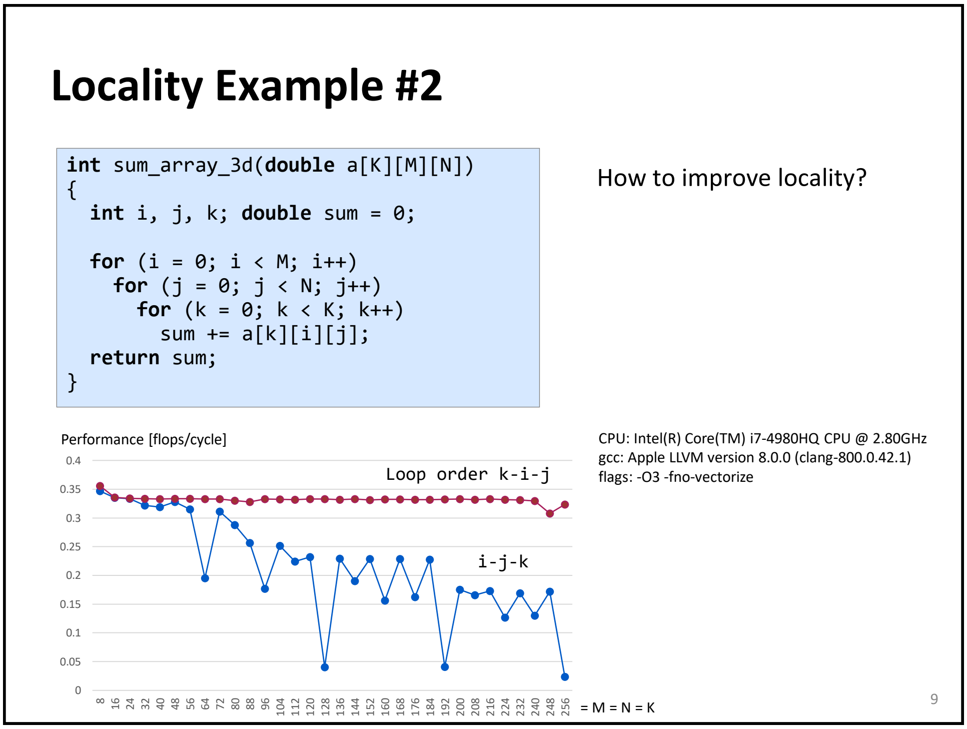
Instructions:

- *Temporal: loops cycle through the same instructions*
- *Spatial: instructions referenced in sequence*

Being able to assess the locality of code is a crucial skill for a performance programmer

Locality Example #1

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

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

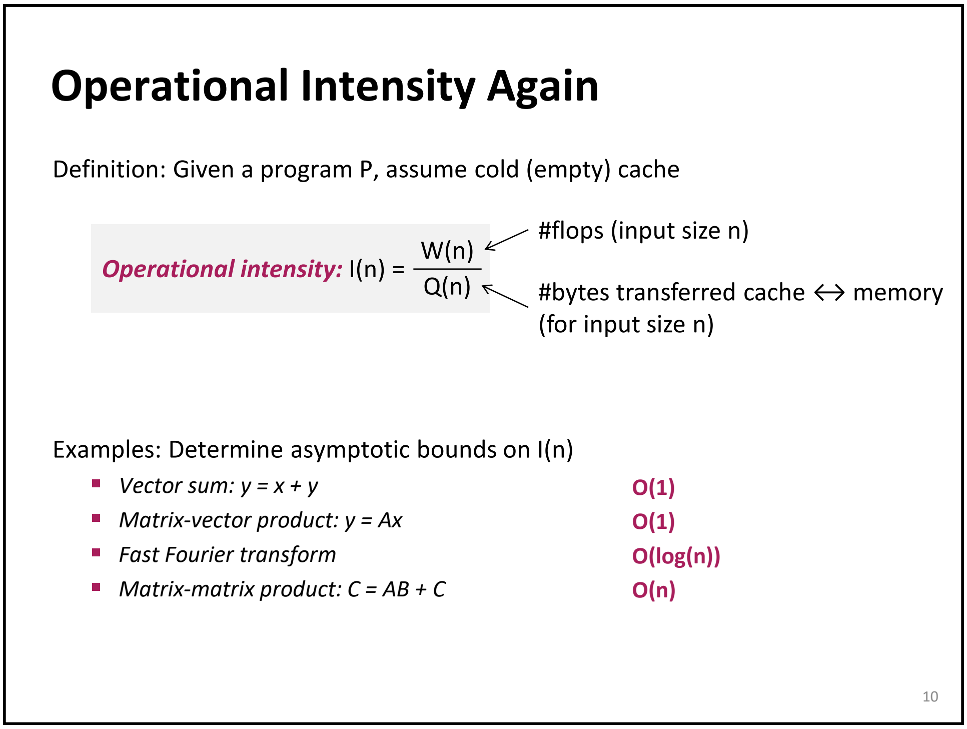


Locality Example #2

```
int sum_array_3d(double a[K][M][N])
{
  int i, j, k; double sum = 0;

  for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
      for (k = 0; k < K; k++)
        sum += a[k][i][j];
  return sum;
}
```

How to improve locality?

CPU: Intel(R) Core(TM) i7-4980HQ CPU @ 2.80GHz
gcc: Apple LLVM version 8.0.0 (clang-800.0.42.1)
flags: -O3 -fno-vectorize



Operational Intensity Again

Definition: Given a program P, assume cold (empty) cache

Operational intensity: $I(n) = \frac{W(n)}{Q(n)}$

#flops (input size n)

#bytes transferred cache ↔ memory (for input size n)

Examples: Determine asymptotic bounds on I(n)

Vector sum: y = x + y	**O(1)**
Matrix-vector product: y = Ax	**O(1)**
Fast Fourier transform	**O(log(n))**
Matrix-matrix product: C = AB + C	**O(n)**

Compute/Memory Bound

A function/piece of code is:

- ***Compute bound*** *if it has high operational intensity*
- ***Memory bound*** *if it has low operational intensity*

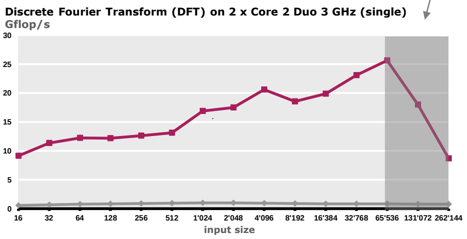
Relationship between operational intensity and locality?

- *They are closely related*
- *Operational intensity only describes the boundary last level cache/memory*

Effects

dark gray = outside LLC

FFT: *I(n) = O(log(n))*

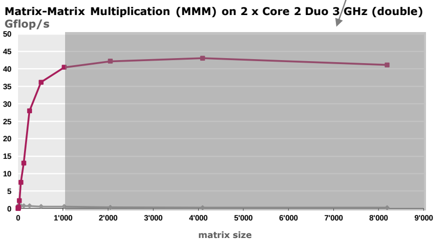


Up to 40-50% peak

Performance drop outside last level cache (LLC)

Most time spent transferring data

MMM: *I(n) = O(n)*



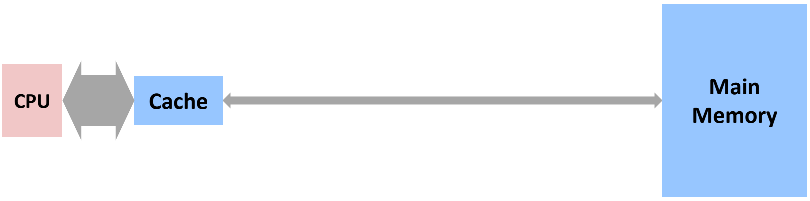
Up to 80-90% peak

Performance can be maintained outside LLC

Cache miss time compensated/hidden by computation

Cache

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

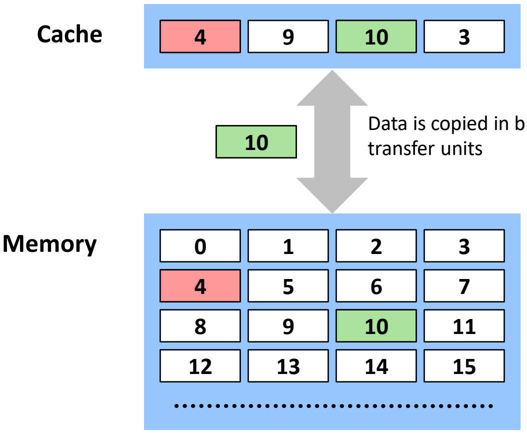


Naturally supports *temporal locality*

Spatial locality is supported by transferring data in blocks

- *Core family: one block = 64 B = 8 doubles*

General Cache Mechanics

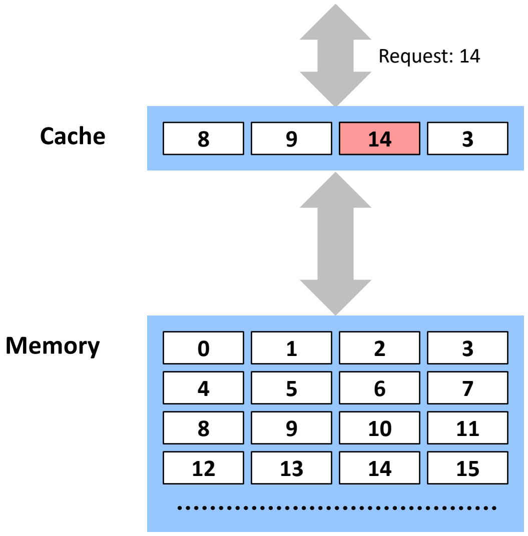


Smaller, faster, more expensive memory caches a subset of the blocks

Data is copied in block-sized transfer units

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

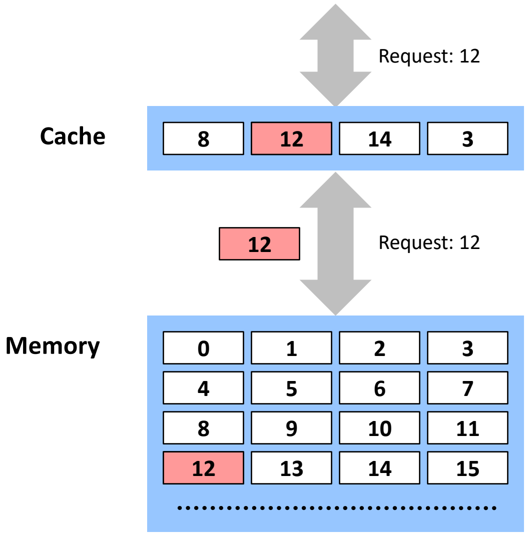
General Cache Concepts: Hit



Data in block b is needed

Block b is in cache:
Hit!

General Cache Concepts: Miss



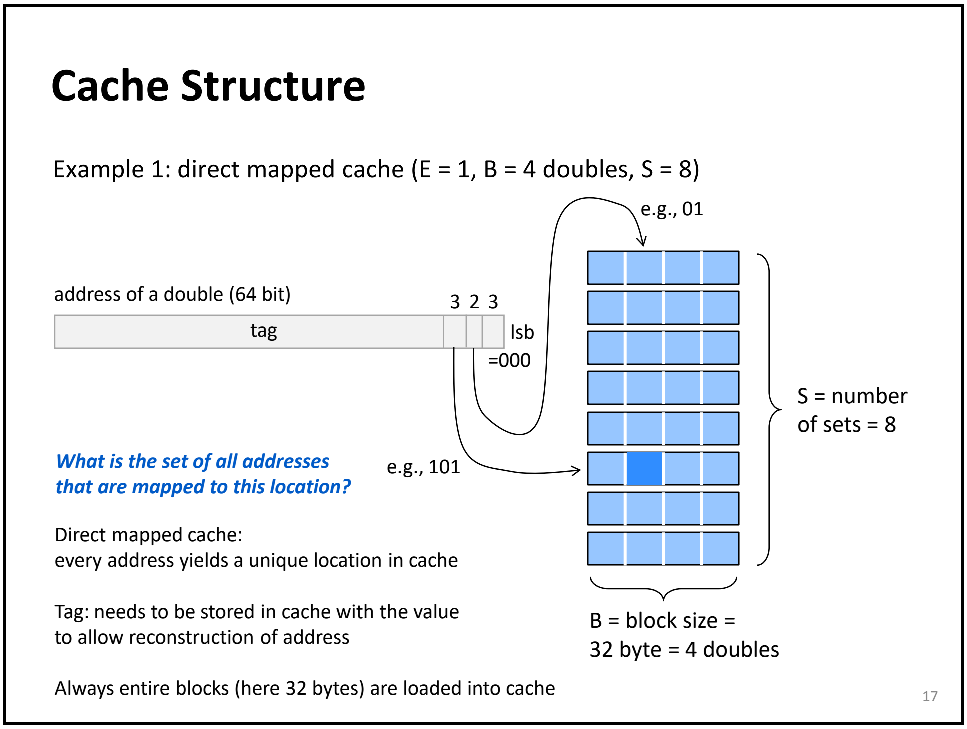
Data in block b is needed

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)



Cache Structure

Example 1: direct mapped cache (E = 1, B = 4 doubles, S = 8)

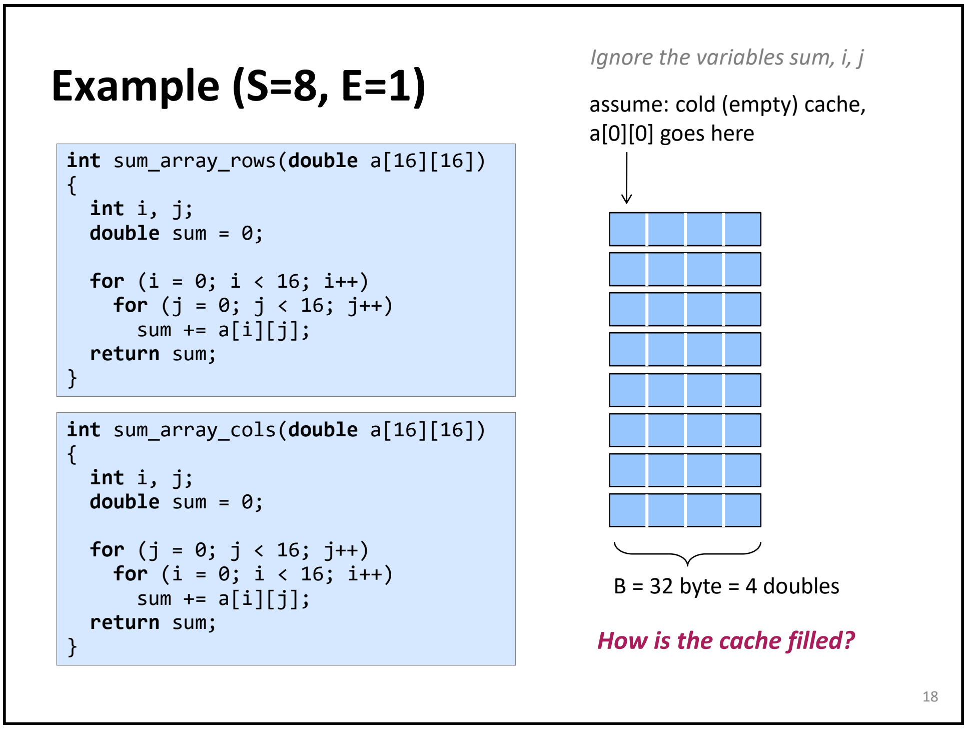
address of a double (64 bit)
tag
3 2 3
lsb
=000
e.g., 01
e.g., 101
S = number of sets = 8
B = block size = 32 byte = 4 doubles

What is the set of all addresses that are mapped to this location?

Direct mapped cache:
every address yields a unique location in cache

Tag: needs to be stored in cache with the value to allow reconstruction of address

Always entire blocks (here 32 bytes) are loaded into cache



Example (S=8, E=1)

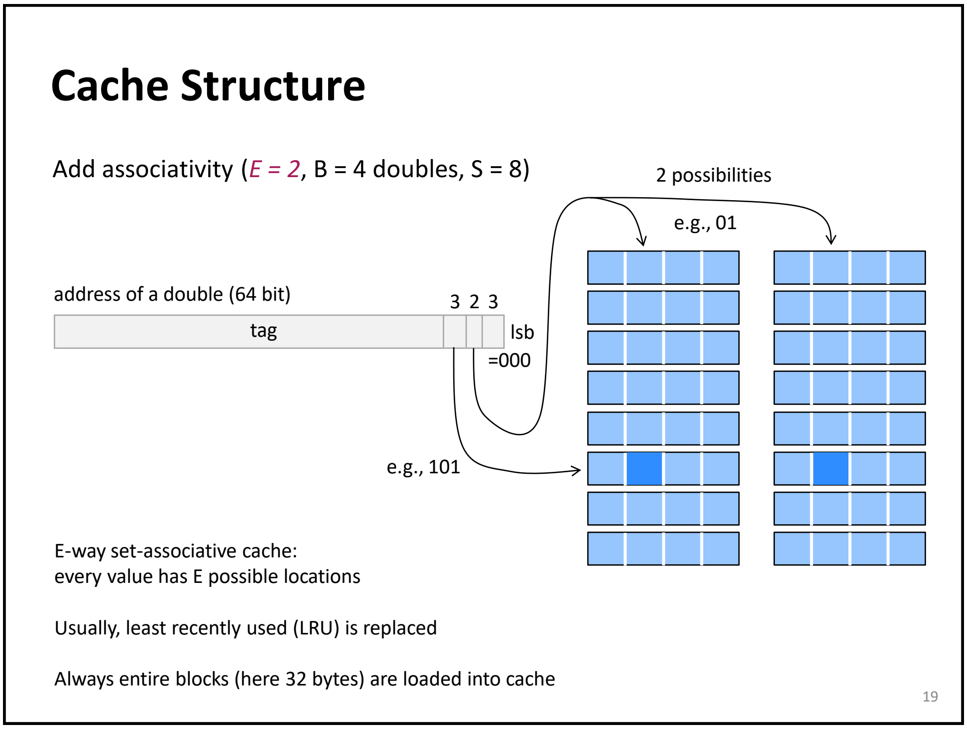
Ignore the variables sum, i, j

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

assume: cold (empty) cache, a[0][0] goes here
B = 32 byte = 4 doubles

How is the cache filled?



Cache Structure

Add associativity (*E = 2*, B = 4 doubles, S = 8)

2 possibilities

e.g., 01

address of a double (64 bit)

3 2 3

tag | | | lsb

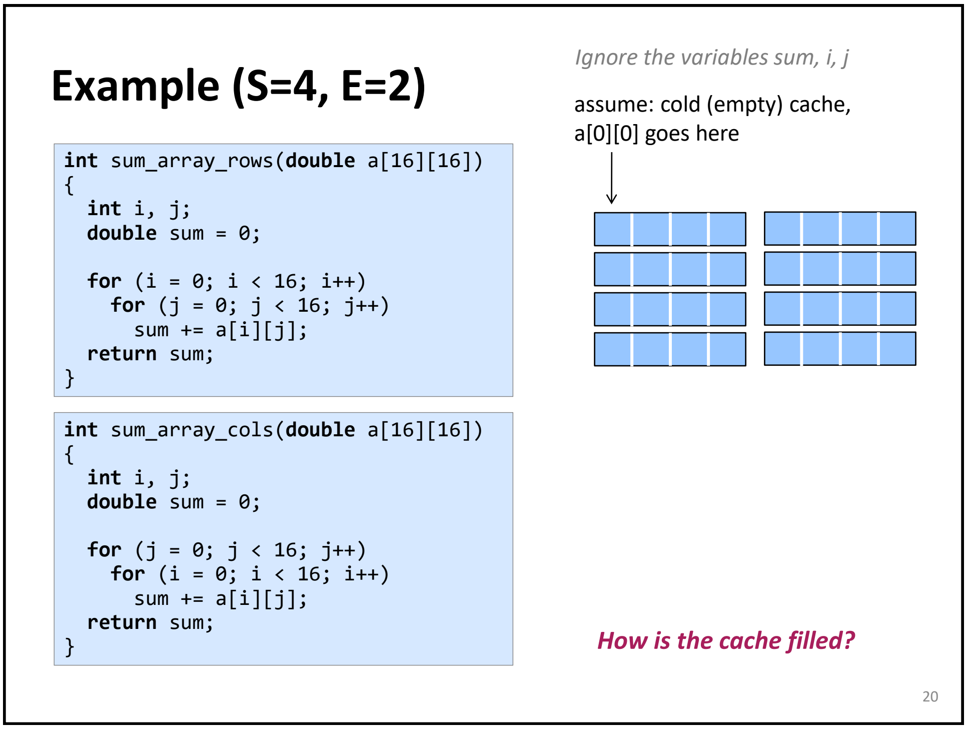
=000

e.g., 101

E-way set-associative cache:
every value has E possible locations

Usually, least recently used (LRU) is replaced

Always entire blocks (here 32 bytes) are loaded into cache



Example (S=4, E=2)

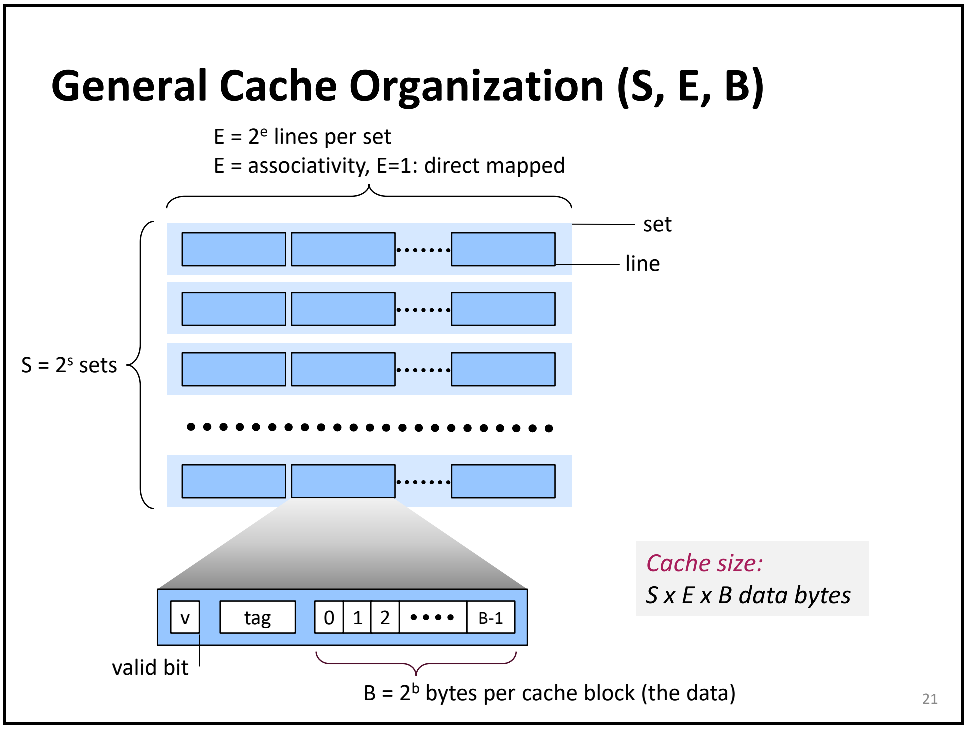
Ignore the variables sum, i, j

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

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

How is the cache filled?



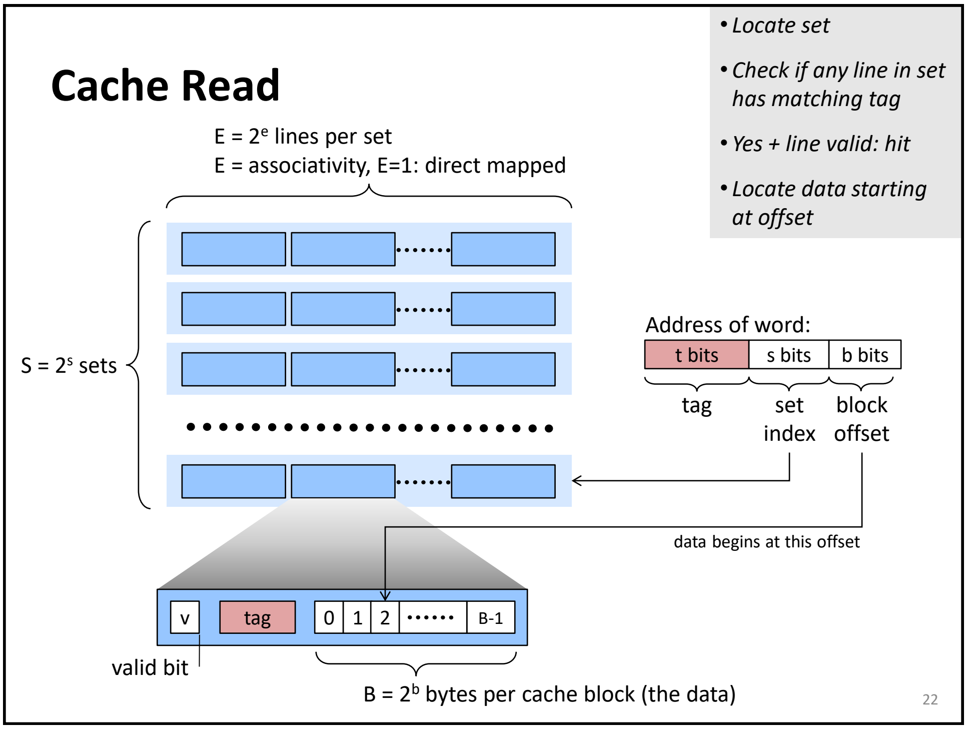
General Cache Organization (S, E, B)

E = 2^e lines per set

E = associativity, E=1: direct mapped

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

Cache size:
S x E x B data bytes



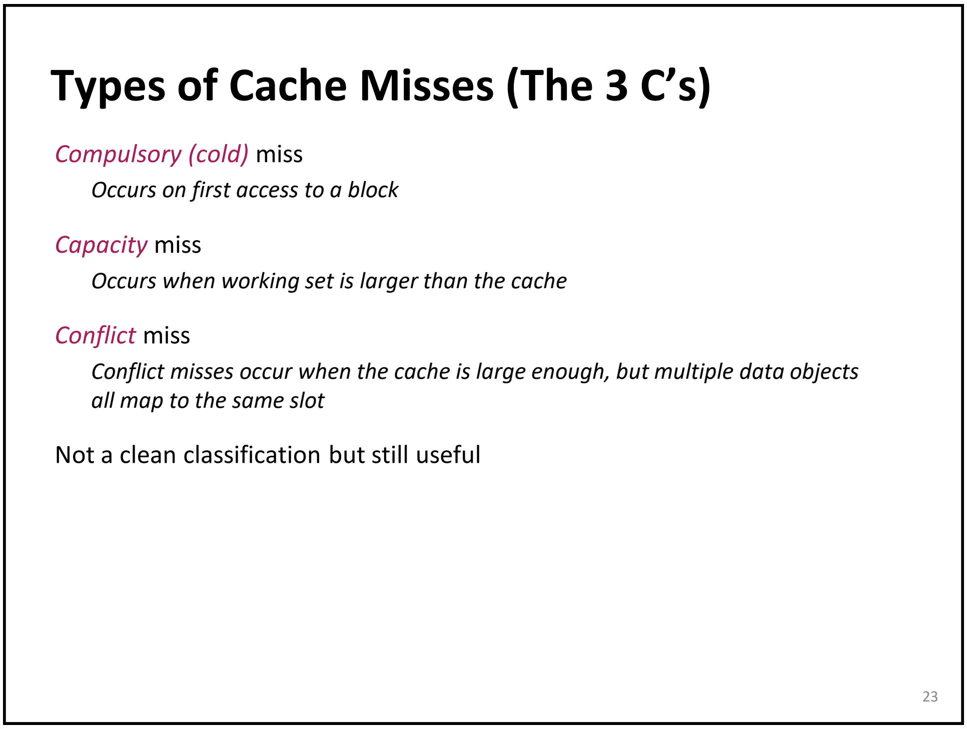
Cache Read

- *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

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



Types of Cache Misses (The 3 C's)

Compulsory (cold) miss

Occurs on first access to a block

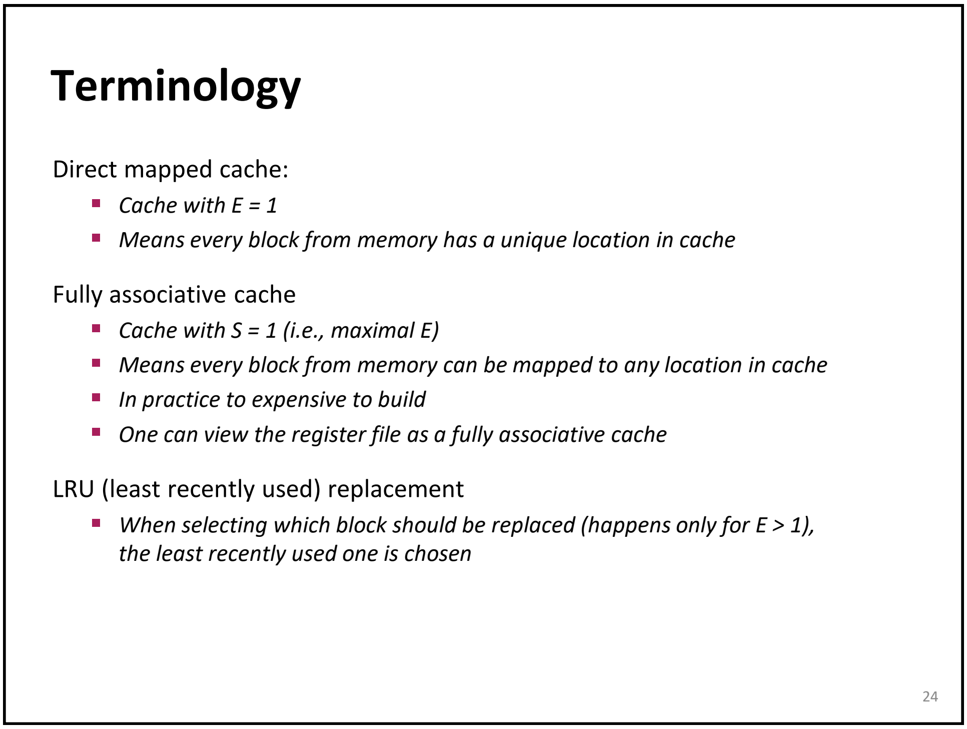
Capacity miss

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

Not a clean classification but still useful



Terminology

Direct mapped cache:

- *Cache with E = 1*
- *Means every block from memory has a unique location in cache*

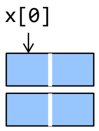
Fully associative cache

- *Cache with S = 1 (i.e., maximal E)*
- *Means every block from memory can be mapped to any location in cache*
- *In practice to expensive to build*
- *One can view the register file as a fully associative cache*

LRU (least recently used) replacement

- *When selecting which block should be replaced (happens only for E > 1), the least recently used one is chosen*

Small Example, Part 1



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

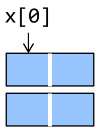
Array (accessed twice in example)
`x = x[0], …, x[7]`

```
% 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



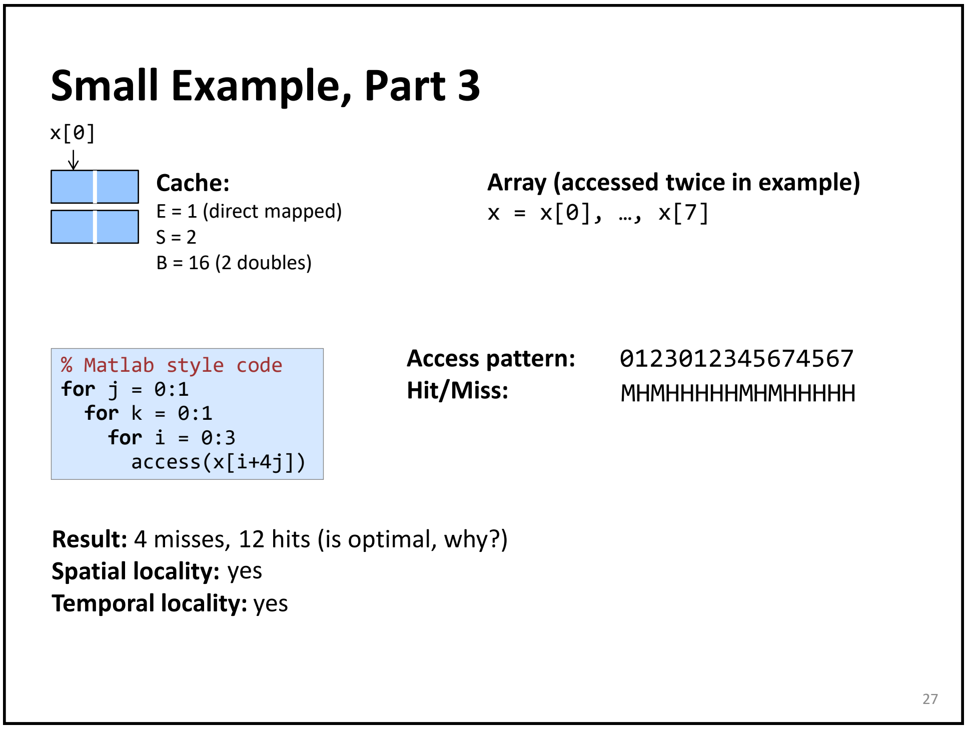
Cache:
E = 1 (direct mapped)
S = 2
B = 16 (2 doubles)

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

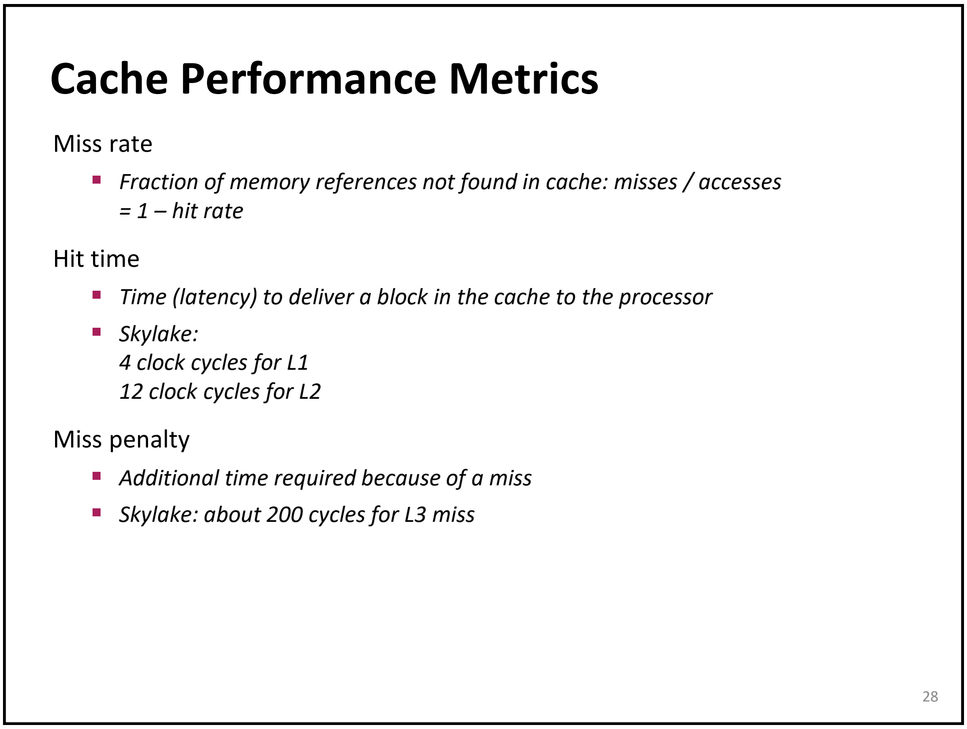
Cache:
E = 1 (direct mapped)
S = 2
B = 16 (2 doubles)

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, 12 hits (is optimal, why?)
Spatial locality: yes
Temporal locality: yes



Cache Performance Metrics

Miss rate

- *Fraction of memory references not found in cache: misses / accesses*
  = 1 – hit rate

Hit time

- *Time (latency) to deliver a block in the cache to the processor*
- *Skylake:*
  4 clock cycles for L1
  12 clock cycles for L2

Miss penalty

- *Additional time required because of a miss*
- *Skylake: about 200 cycles for L3 miss*

What about writes?

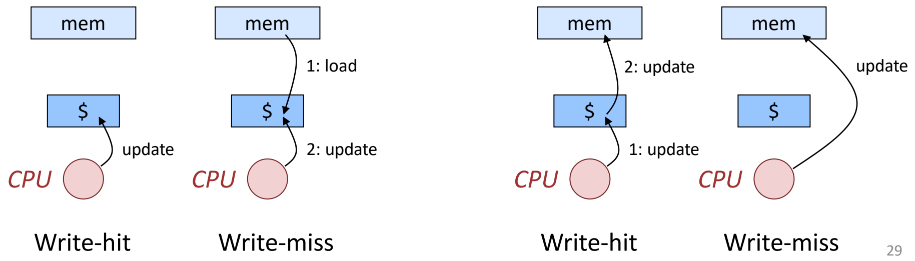
What to do on a write-hit?

- *Write-through:* *write immediately to memory*
- *Write-back:* *defer write to memory until replacement of line*

What to do on a write-miss?

- *Write-allocate:* *load into cache, update line in cache*
- *No-write-allocate:* *writes immediately to memory*

Write-back/write-allocate (Core) *Write-through/no-write-allocate*



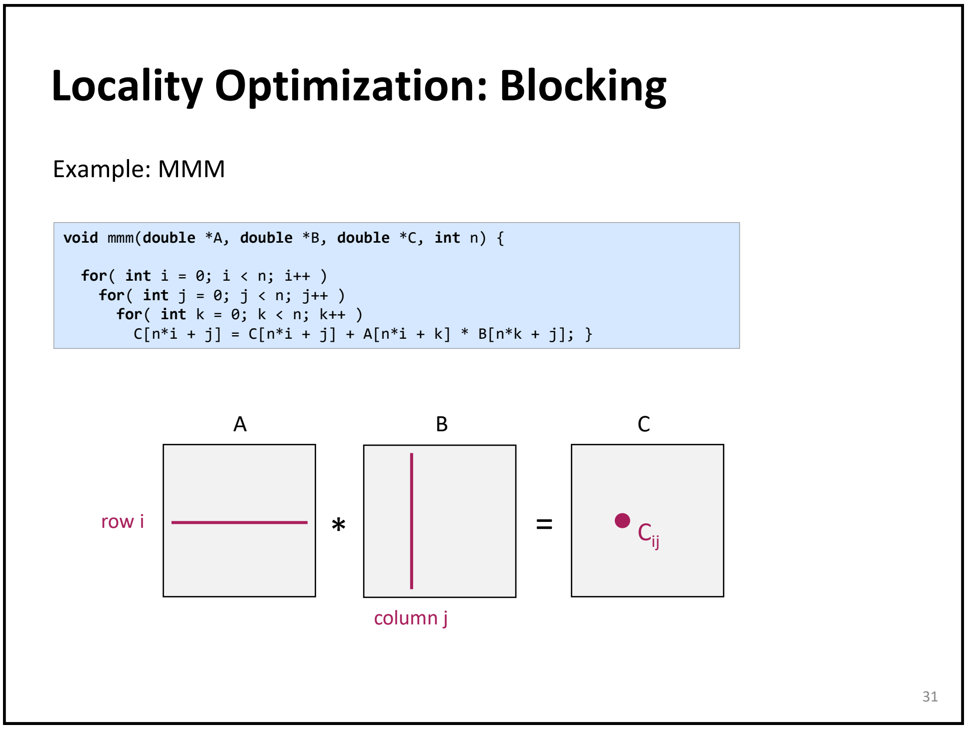
Example:

z = x + y, x, y, z vector of doubles of length n

assume they fit jointly in cache + cold cache

memory traffic Q(n): 4n doubles = 32n bytes

operational intensity I(n)? W(n) = n flops, so
I(n) = W(n)/Q(n) = 1/32

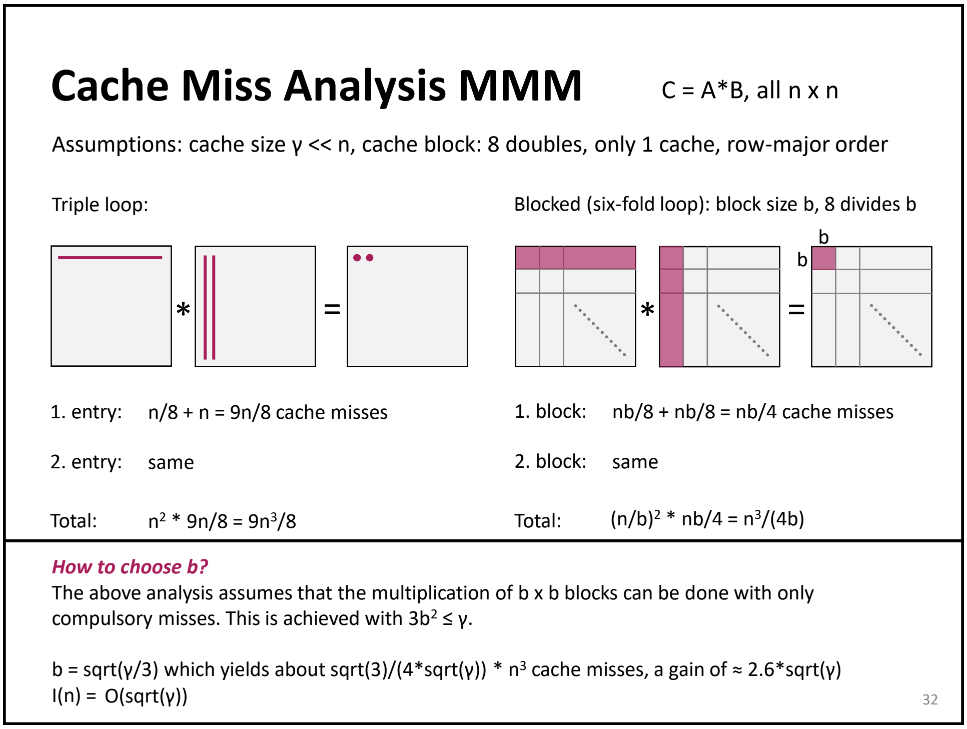


Locality Optimization: Blocking

Example: MMM

```
void mmm(double *A, double *B, double *C, int n) {

  for( int i = 0; i < n; i++ )
    for( int j = 0; j < n; j++ )
      for( int k = 0; k < n; k++ )
        C[n*i + j] = C[n*i + j] + A[n*i + k] * B[n*k + j]; }
```

31



Cache Miss Analysis MMM

C = A*B, all n x n

Assumptions: cache size γ << n, cache block: 8 doubles, only 1 cache, row-major order

Triple loop:

Blocked (six-fold loop): block size b, 8 divides b

Triple loop		Blocked	
1. entry:	n/8 + n = 9n/8 cache misses	1. block:	nb/8 + nb/8 = nb/4 cache misses
2. entry:	same	2. block:	same
Total:	$n^2 * 9n/8 = 9n^3/8$	Total:	$(n/b)^2 * nb/4 = n^3/(4b)$

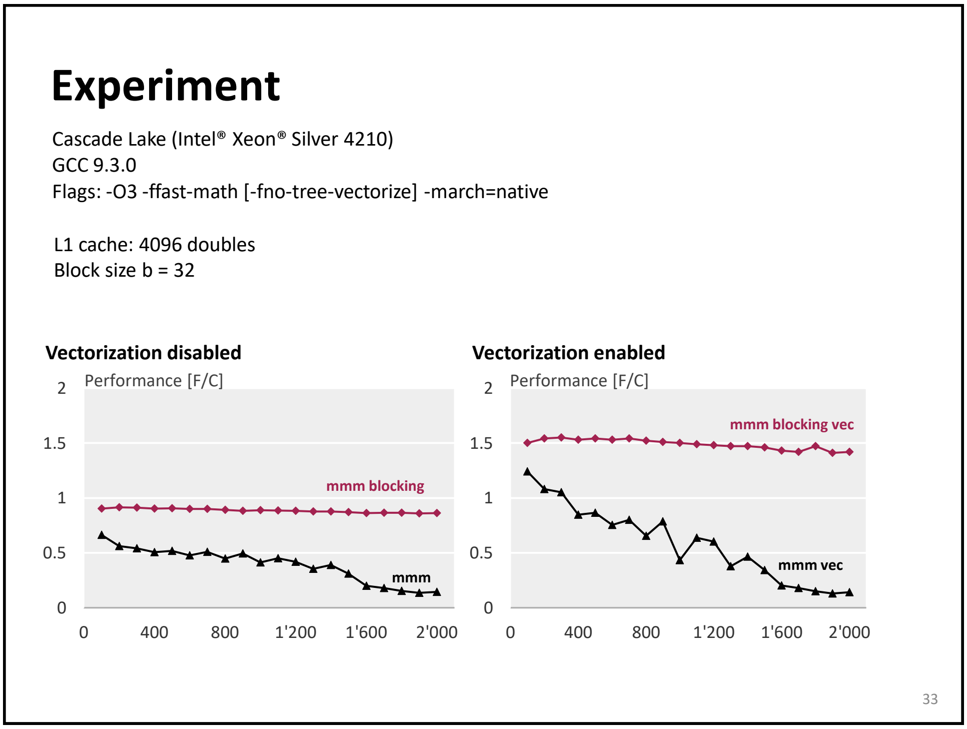
How to choose b?

The above analysis assumes that the multiplication of b x b blocks can be done with only compulsory misses. This is achieved with $3b^2 \leq \gamma$.

b = sqrt(γ/3) which yields about sqrt(3)/(4*sqrt(γ)) * n^3 cache misses, a gain of ≈ 2.6*sqrt(γ)

I(n) = O(sqrt(γ))

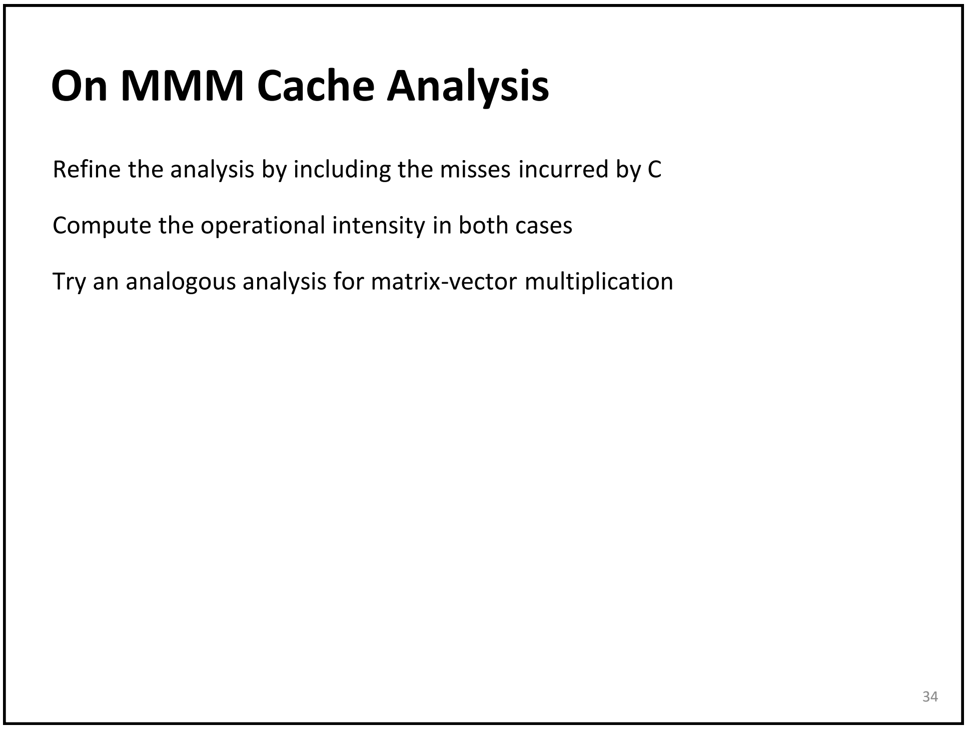
32



Experiment

Cascade Lake (Intel® Xeon® Silver 4210)
GCC 9.3.0
Flags: -O3 -ffast-math [-fno-tree-vectorize] -march=native

L1 cache: 4096 doubles
Block size b = 32

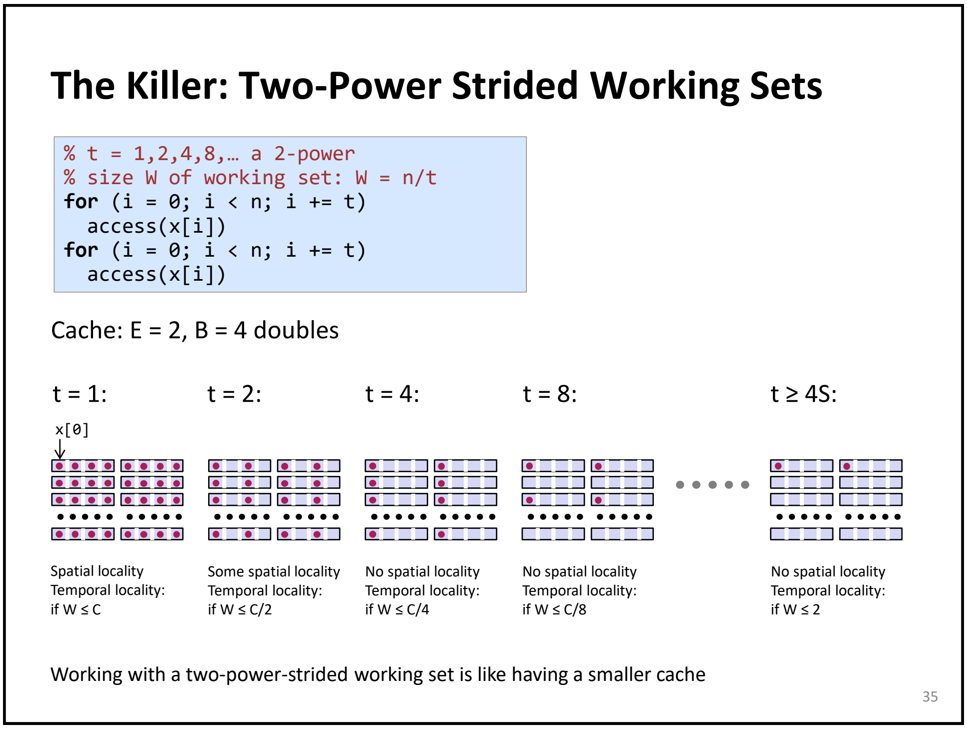


On MMM Cache Analysis

Refine the analysis by including the misses incurred by C

Compute the operational intensity in both cases

Try an analogous analysis for matrix-vector multiplication



The Killer: Two-Power Strided Working Sets

```
% t = 1,2,4,8,… a 2-power
% size W of working set: W = n/t
for (i = 0; i < n; i += t)
  access(x[i])
for (i = 0; i < n; i += t)
  access(x[i])
```

Cache: E = 2, B = 4 doubles

t = 1: Spatial locality
Temporal locality:
if W ≤ C

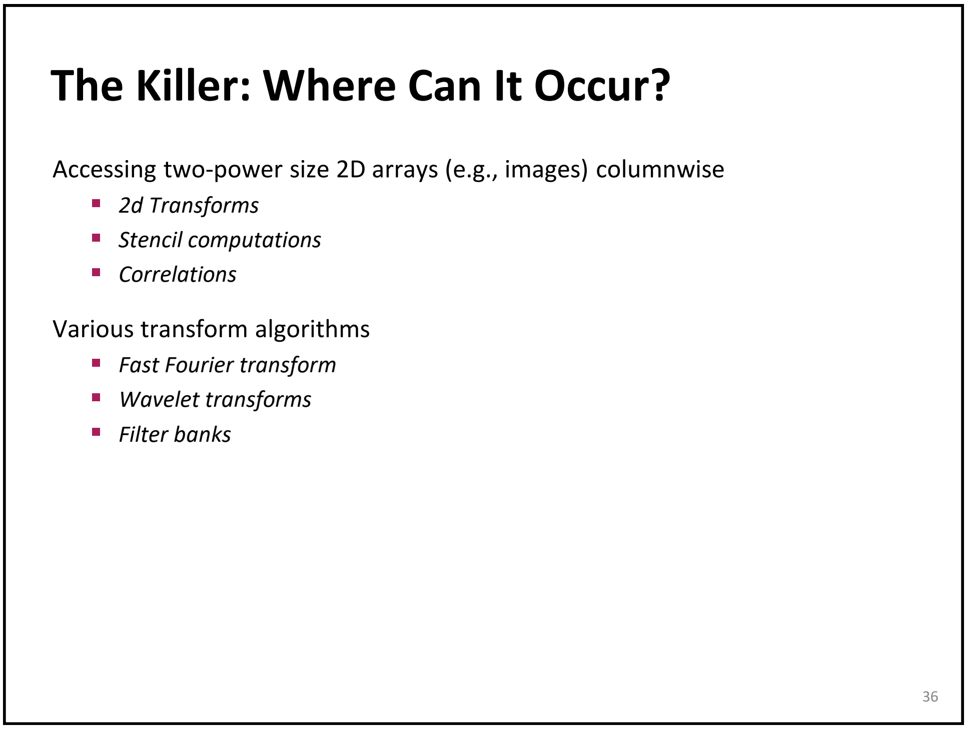
t = 2: Some spatial locality
Temporal locality:
if W ≤ C/2

t = 4: No spatial locality
Temporal locality:
if W ≤ C/4

t = 8: No spatial locality
Temporal locality:
if W ≤ C/8

t ≥ 4S: No spatial locality
Temporal locality:
if W ≤ 2

Working with a two-power-strided working set is like having a smaller cache



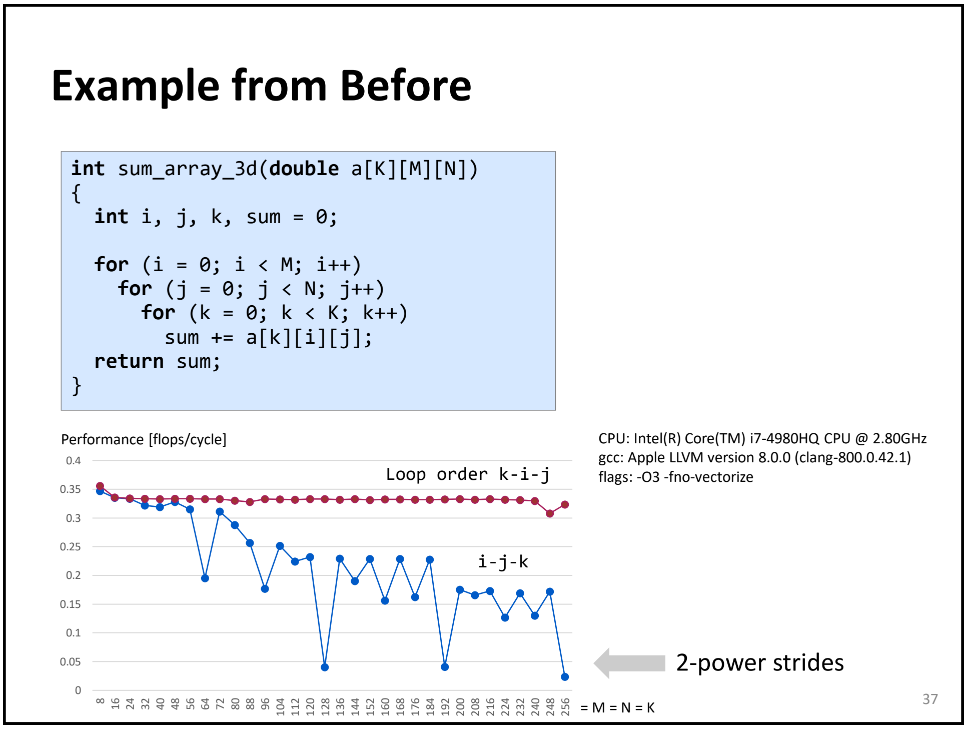
The Killer: Where Can It Occur?

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*

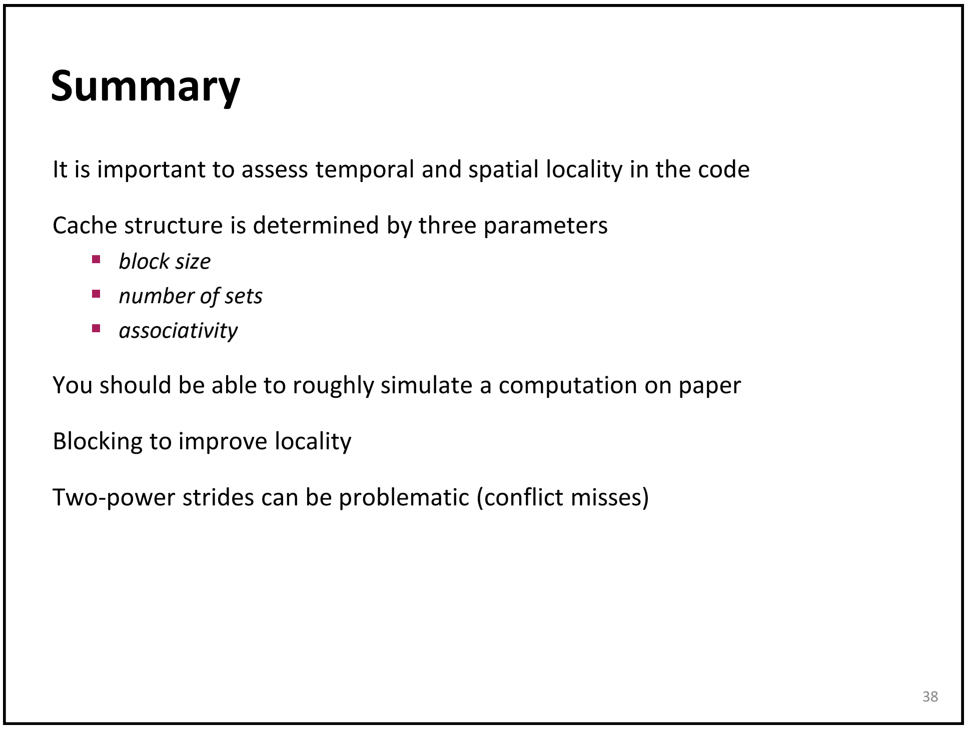


Example from Before

```
int sum_array_3d(double a[K][M][N])
{
  int i, j, k, sum = 0;

  for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
      for (k = 0; k < K; k++)
        sum += a[k][i][j];
  return sum;
}
```

CPU: Intel(R) Core(TM) i7-4980HQ CPU @ 2.80GHz
gcc: Apple LLVM version 8.0.0 (clang-800.0.42.1)
flags: -O3 -fno-vectorize

2-power strides



Summary

It is important to assess temporal and spatial locality in the code

Cache structure is determined by three parameters

- *block size*
- *number of sets*
- *associativity*

You should be able to roughly simulate a computation on paper

Blocking to improve locality

Two-power strides can be problematic (conflict misses)