Performance Engineering of Software Systems

LECTURE 14

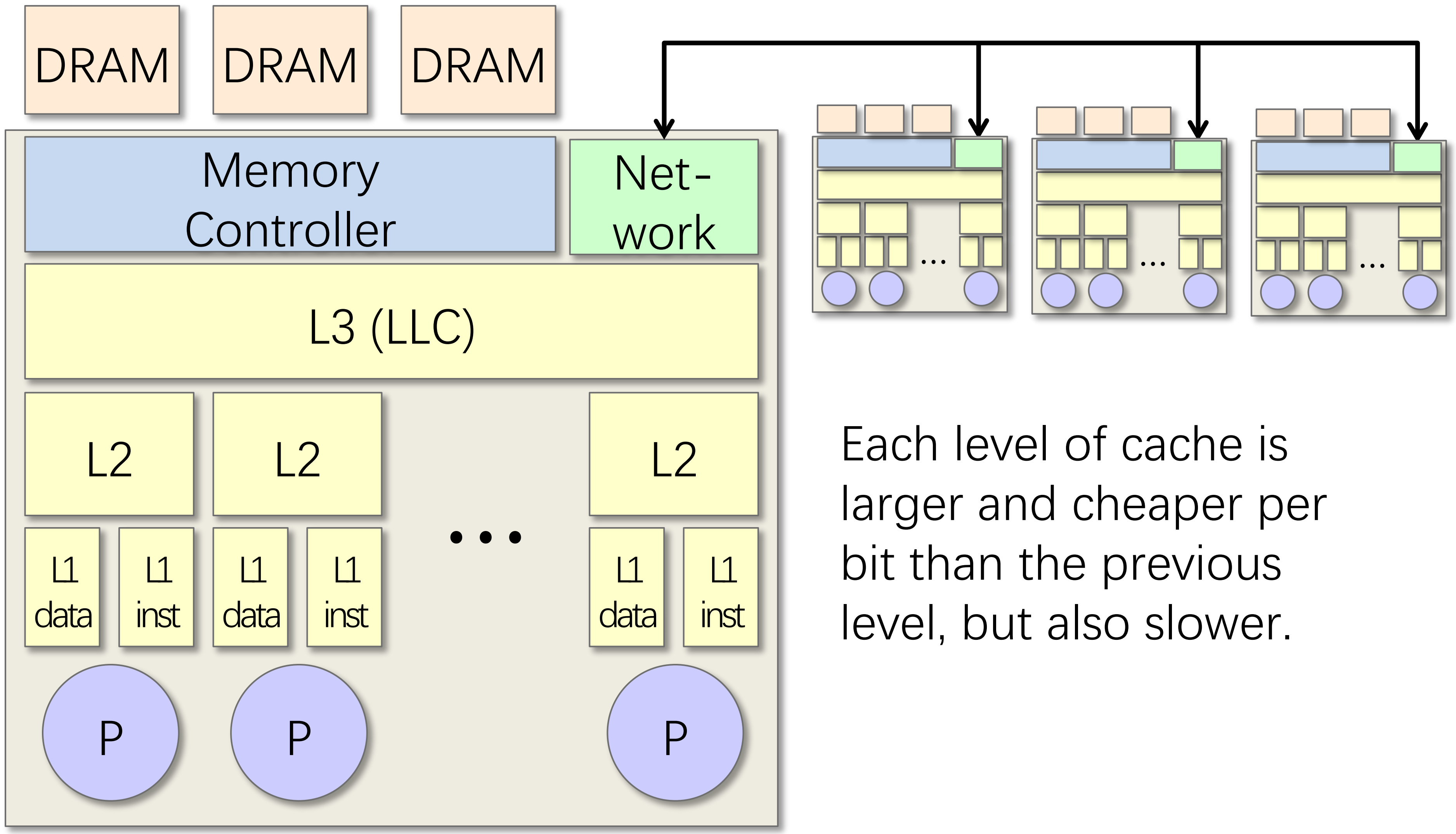
Cache-Efficient Algorithms

Srini Devadas

November 1, 2022

Cache Hardware

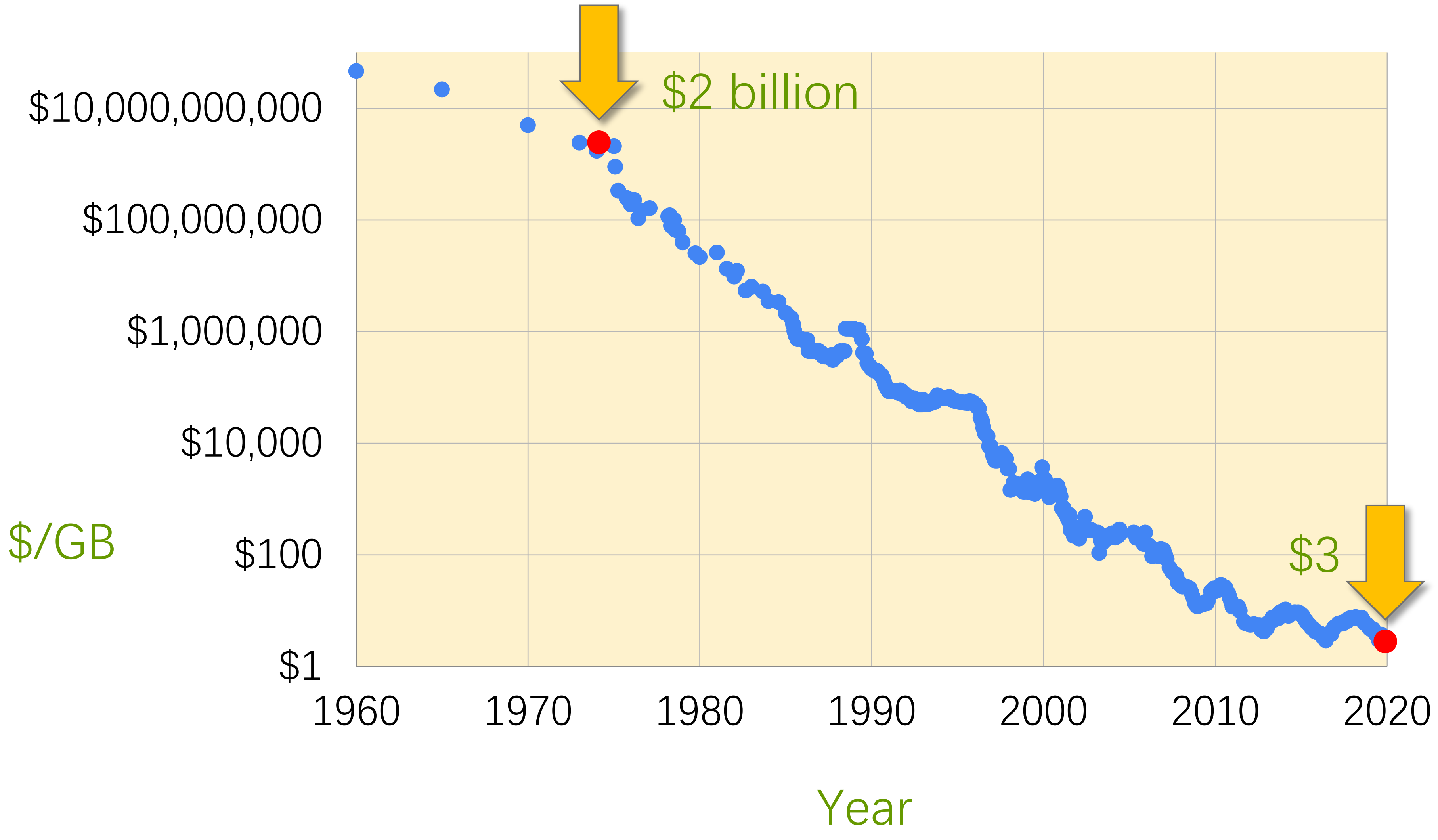
Multicore Cache Hierarchy



Each level of cache is larger and cheaper per bit than the previous level, but also slower.

Memory Prices through History

Source: John C. McCallum, "Memory prices 1957+," available at http://jcmit.net/memoryprice.htm , last updated 2021-10-21.



Cache Specs for Typical High-End Multicore

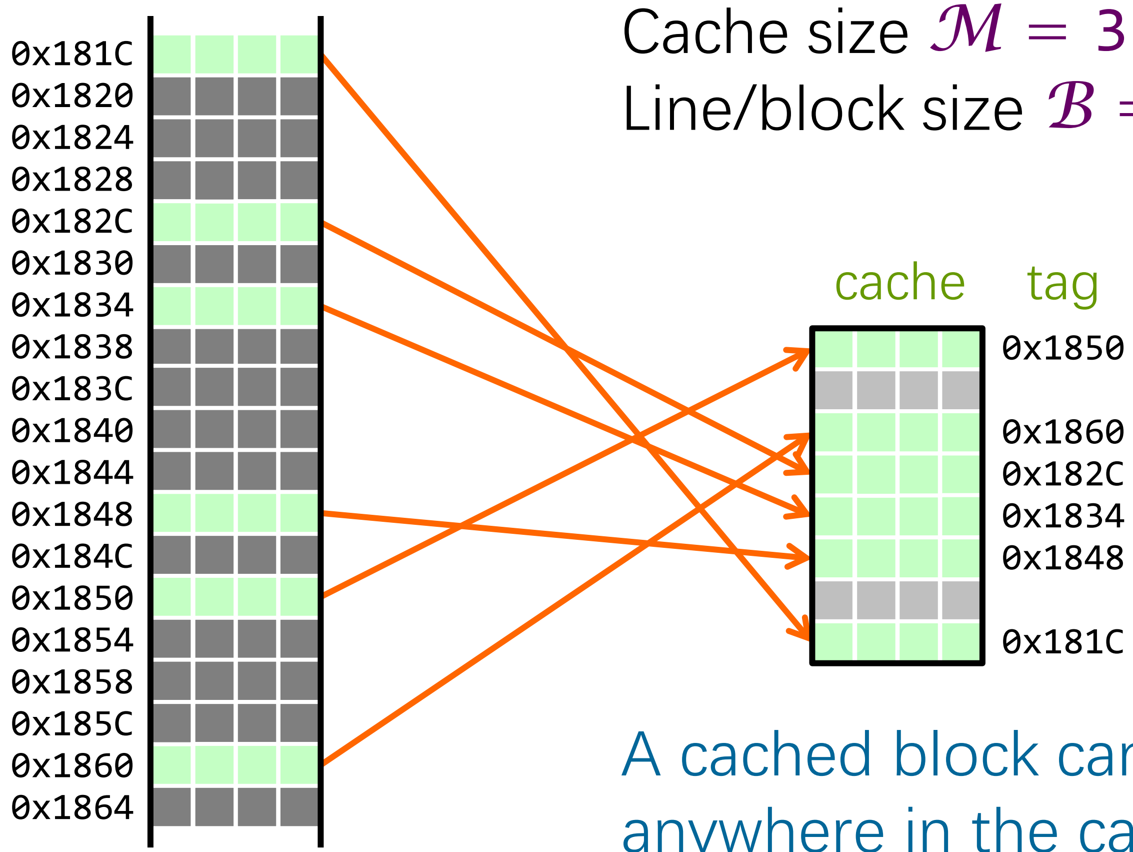
Level	Size/core	Associativity	Latency (cycles)
DRAM	up to 160 GiB		85–240
L3	1.375 MiB	11	50–70
L2	1 MiB	16	14
L1-D	32 KiB	8	4–5
L1-I	32 KiB	8	5

Intel Xeon Platinum 8280L (Cascade Lake)

- Launched April 2019 for $17,906 — cheaper now.
- 2.7 GHz clock, Turbo Boost up to 4 GHz
- 28 cores/chip + 2-way hyperthreading
- 2190 GFLOPS
- 64 B cache lines/blocks
- Up to 8-way multiprocessing

Fully Associative Cache

w-bit virtual-address space



Cache size $\mathcal{M} = 32$.

Line/block size $\mathcal{B} = 4$.

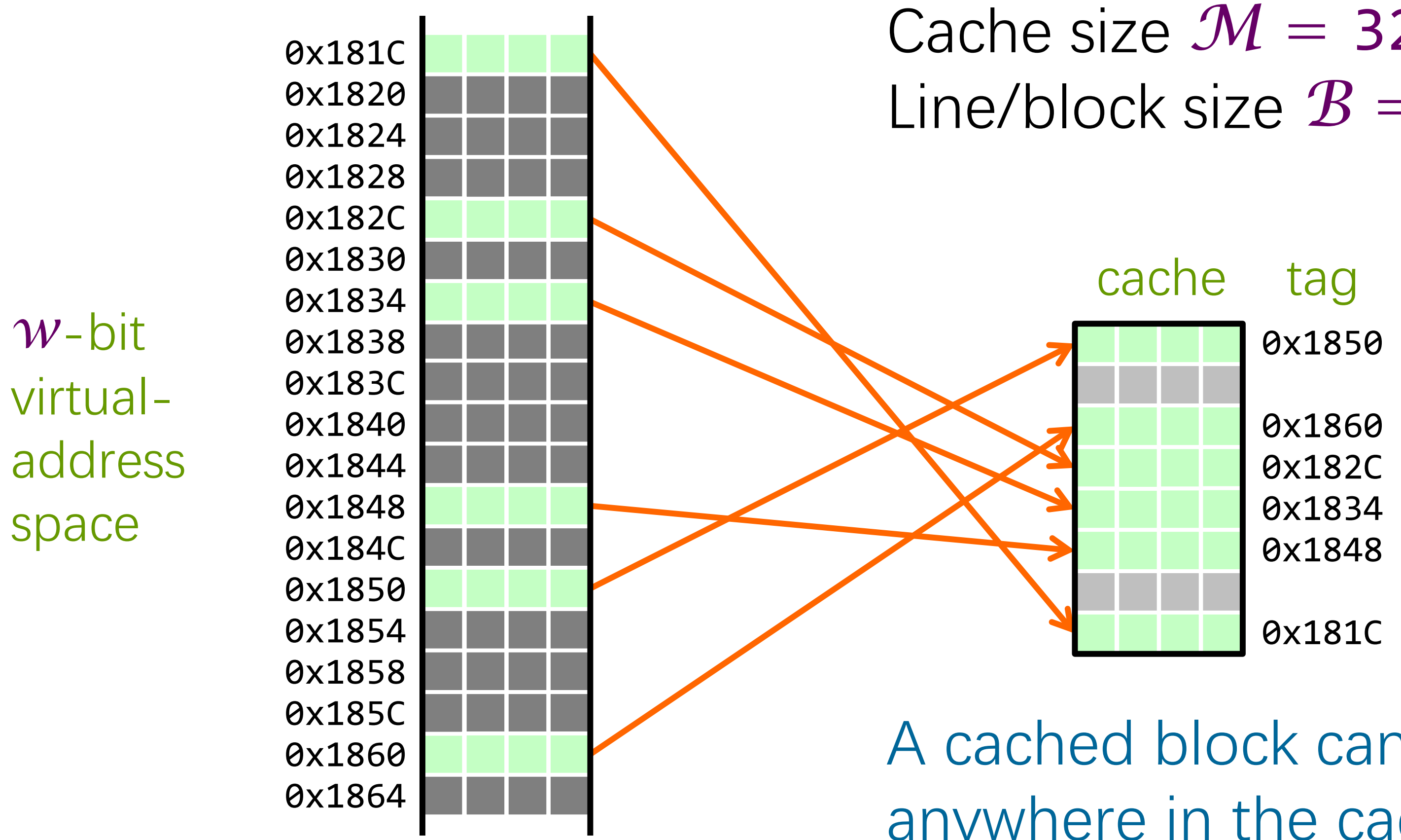
Tag omits low-order $\lg \mathcal{B} = 2$ bits.

A cached block can reside anywhere in the cache.

byte address

	tag	offset
bits	$w - \lg \mathcal{B}$	$\lg \mathcal{B}$

Fully Associative Cache



Cache size $\mathcal{M} = 32$.
Line/block size $\mathcal{B} = 4$.

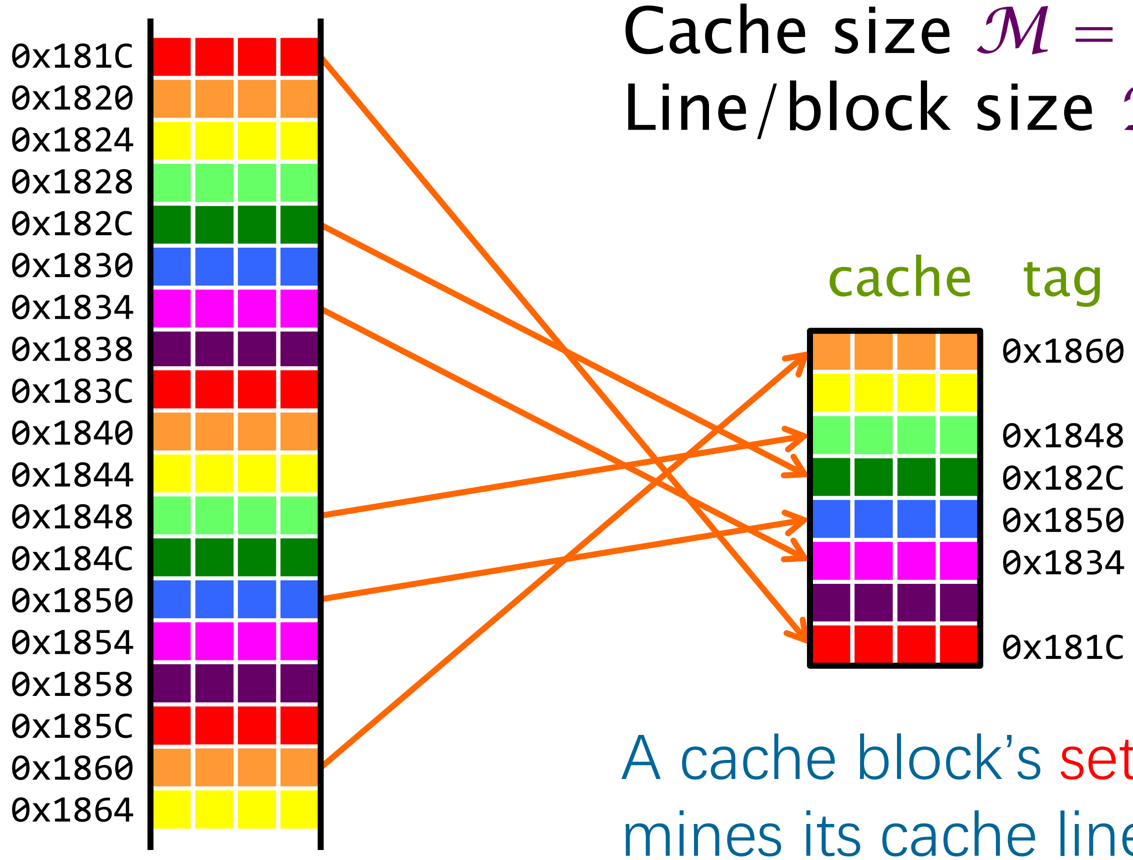
Tag omits low-order $\lg \mathcal{B} = 2$ bits.

A cached block can reside anywhere in the cache.

To find a block in the cache, all the cache lines must be searched for the tag. When the cache becomes full, a replacement policy determines which block to evict to make room for a new block.

Direct-Mapped Cache

w-bit virtual-address space



Cache size $\mathcal{M} = 32$.
Line/block size $\mathcal{B} = 4$.

Tag omits low-order $\lg \mathcal{M} = 5$ bits.

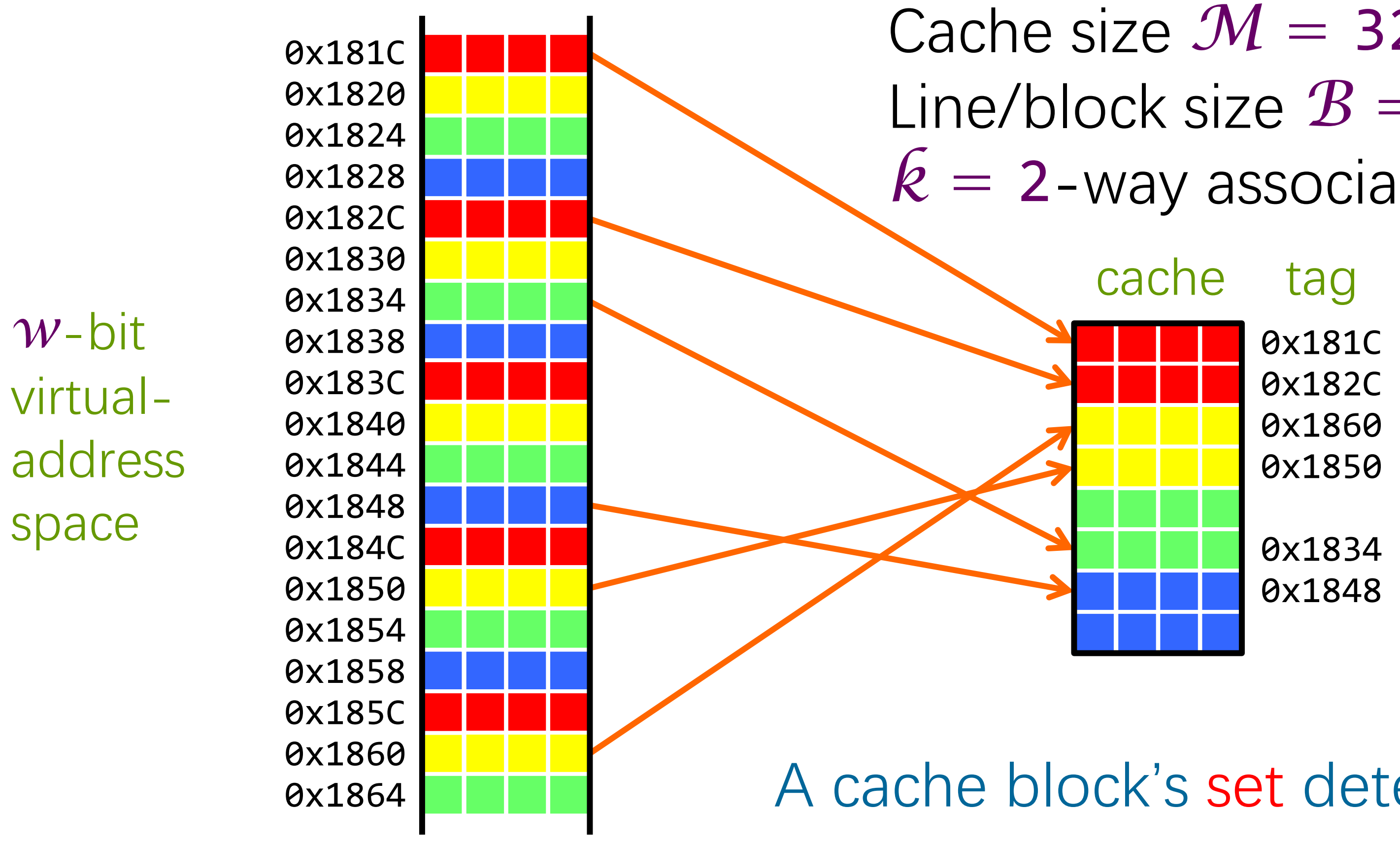
A cache block's set determines its cache line.

byte address

	tag	set	offset
bits	$w - \lg \mathcal{M}$	$\lg(\mathcal{M}/\mathcal{B})$	$\lg \mathcal{B}$

To find a block in the cache, only a single line in the cache needs to be checked.

Set-Associative Cache



Cache size $\mathcal{M} = 32$.
Line/block size $\mathcal{B} = 4$.
$k = 2$-way associativity.

Tag omits low-order $\lg(\mathcal{M}/k) = 4$ bits.

A cache block's set determines k cache lines

byte address

	tag	set	offset
bits	$w - \lg(\mathcal{M}/k)$	$\lg(\mathcal{M}/k\mathcal{B})$	$\lg \mathcal{B}$

To find a block in the cache, the k cache lines in its set need to be searched.

Taxonomy of Cache Misses

Cold miss

- The first time the cache block is accessed.

Capacity miss

- The previous cached copy would have been evicted even with a fully associative cache.

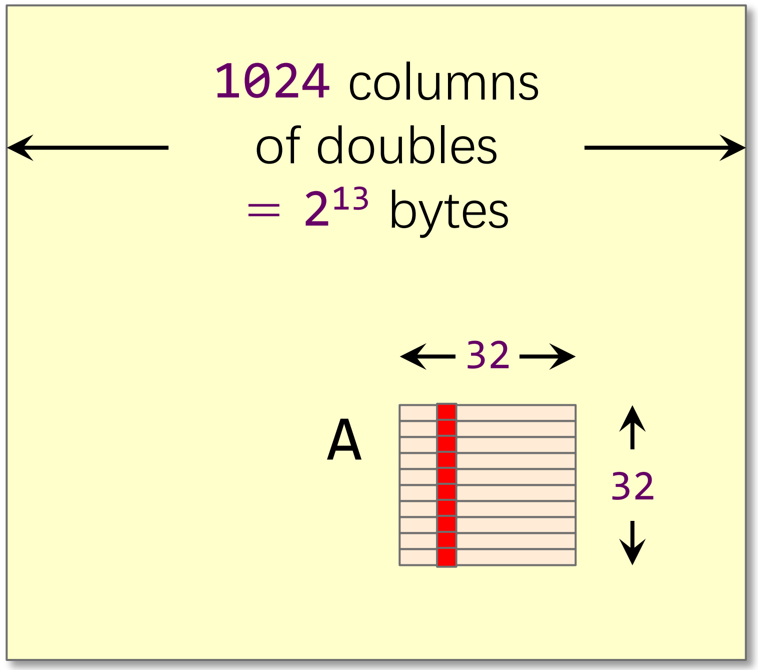
Conflict miss

- Too many blocks from the same set in the cache. The block would not have been evicted with a fully associative cache.

Sharing miss

- Another processor acquired exclusive access to the cache block.
- True-sharing miss: The two processors access to the same data on the cache block.
- False-sharing miss: The two processors access different data on the cache block.

Conflict Misses for Submatrices



Assume:
- word width $w = 64$ bits.
- cache size $\mathcal{M} = 32\text{K}$ bytes.
- line/block size $\mathcal{B} = 64$ bytes.
- $k = 4$-way associativity.

Conflict misses can be problematic for caches with limited associativity.

byte address

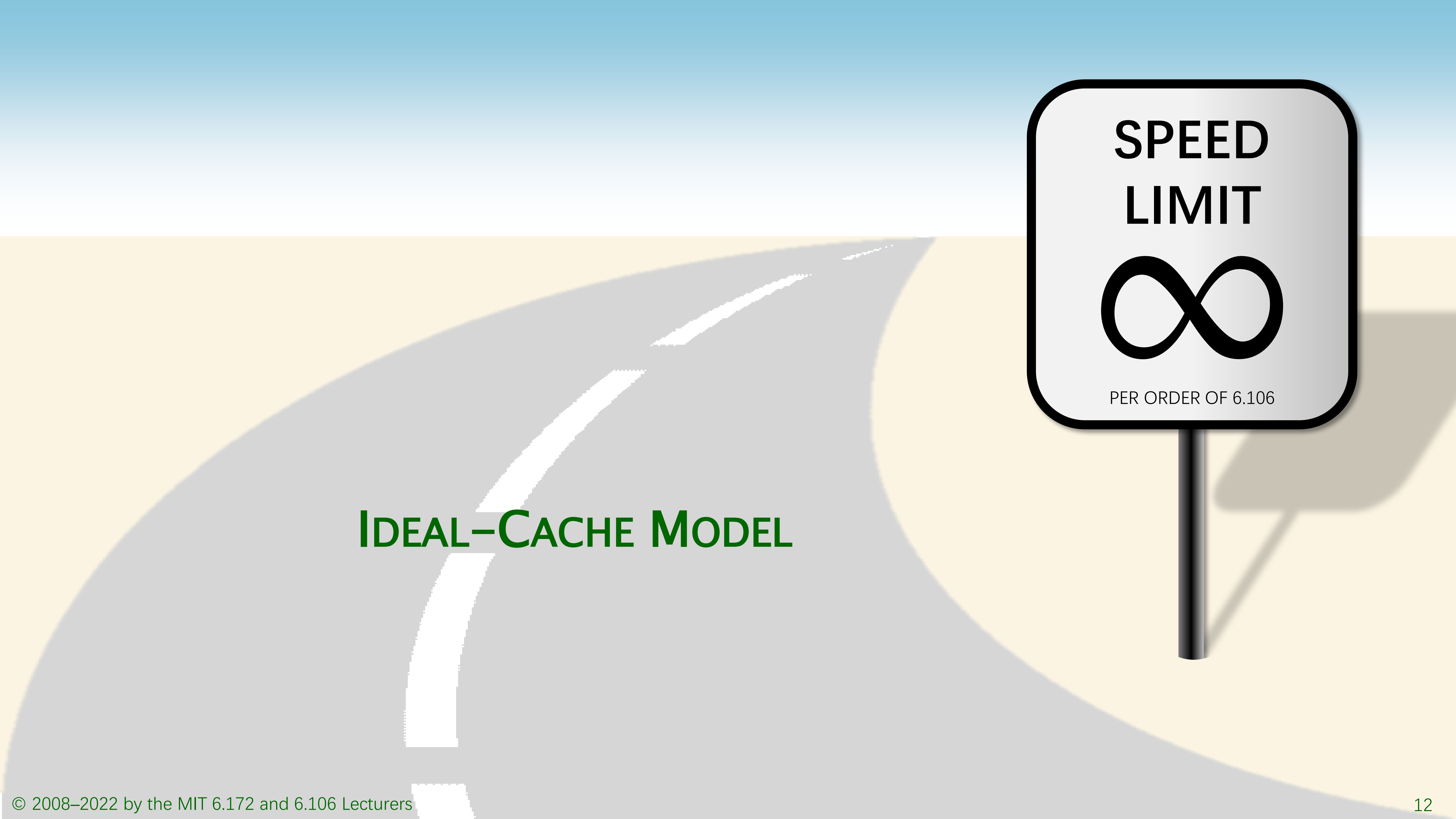
tag	set	offset
$w - \lg(\mathcal{M}/k)$	$\lg(\mathcal{M}/k\mathcal{B})$	$\lg \mathcal{B}$
51	7	6

Analysis

Look at a column of submatrix A.
The addresses of the elements are x, $x+2^{13}$, $x+2 \cdot 2^{13}$, …, $x+31 \cdot 2^{13}$.
They all fall into the same set!

Solutions

Copy A into a temporary 32×32 matrix, or pad rows.

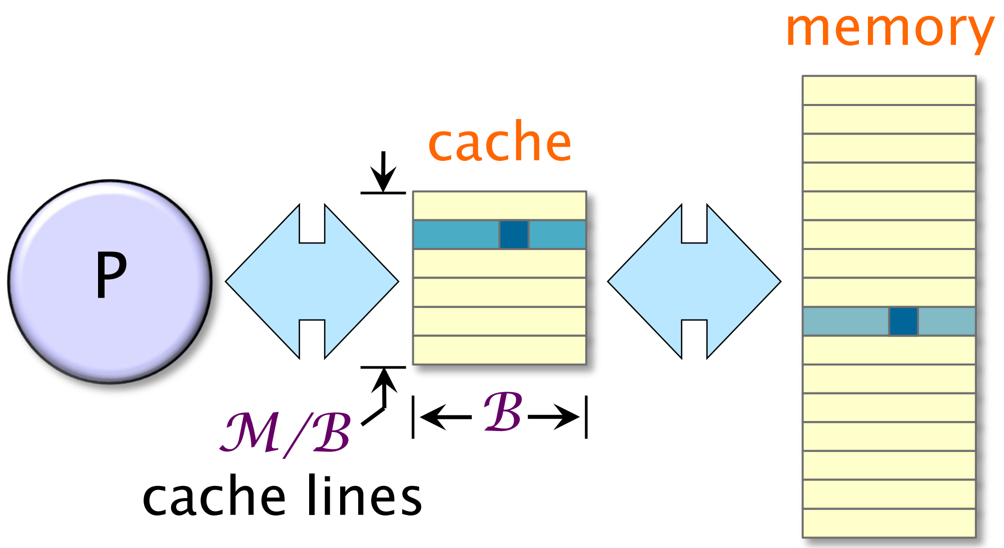


Ideal-Cache Model

Ideal-Cache Model

Parameters

- Two-level hierarchy.
- Cache size of $\mathcal{M}$ bytes.
- Cache-line length of $\mathcal{B}$ bytes.
- Fully associative.
- Optimal, omniscient replacement.



Performance Measures

- work T (ordinary running time)
- cache misses Q

How Reasonable Are Ideal Caches?

"LRU" Lemma [ST85]**.** Suppose that an algorithm incurs Q cache misses on an ideal cache of size $\mathcal{M}$. Then on a fully associative cache of size $2\mathcal{M}$ that uses the least-recently used (LRU) replacement policy, it incurs at most $2Q$ cache misses. ■

Implication: For asymptotic analyses, assume optimal or LRU replacement, whichever is more convenient.

Performance Engineering

- Design a theoretically good algorithm.
- Engineer the details for performance.
  - Real caches are set associative.
  - Loads and stores have different costs with respect to bandwidth and latency.

Segment Caching Lemma

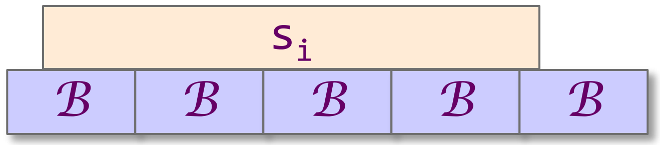
Lemma. Suppose that a program reads a set of r data segments, where the ith segment consists of s_i contiguous bytes in memory, and suppose that

$$\sum_{i=1}^{r} s_i = N < \mathcal{M}/3 \text{ and } N/r \geq \mathcal{B} .$$

Then all the segments fit into cache, and the number of misses to read them all is at most $3N/\mathcal{B}$.

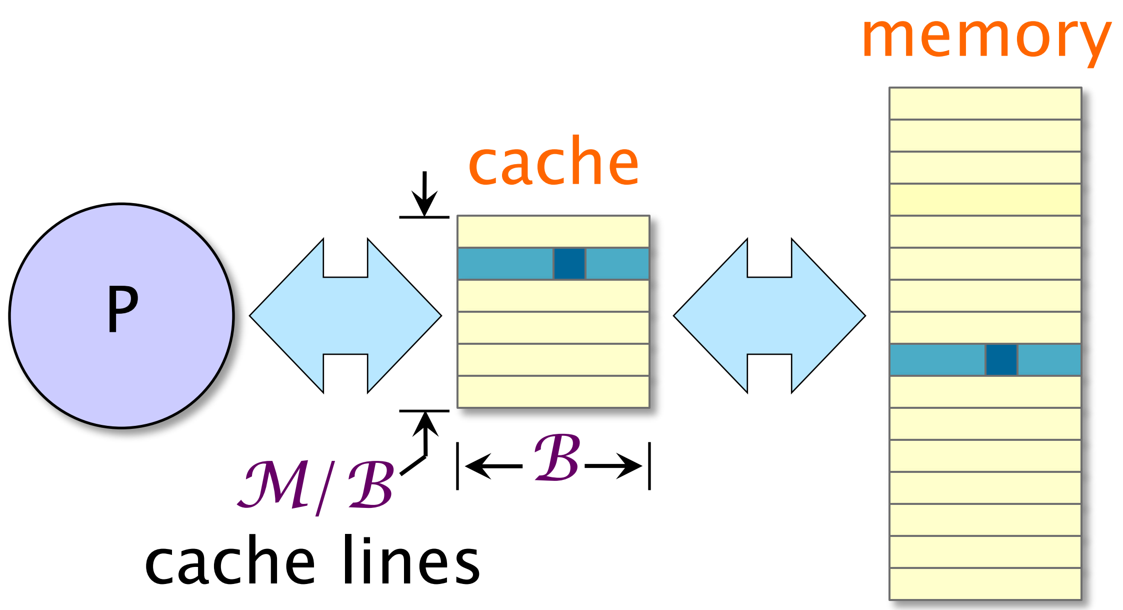
Proof. A single segment s_i incurs at most $s_i/\mathcal{B} + 2$ misses, and hence we have

$$\begin{aligned}\sum_{i=1}^{r}(s_i/\mathcal{B} + 2) &= N/\mathcal{B} + 2r \\ &= N/\mathcal{B} + (2r\mathcal{B})/\mathcal{B} \\ &\leq N/\mathcal{B} + 2N/\mathcal{B} \\ &= 3N/\mathcal{B} . \quad ■\end{aligned}$$



R.I.P
Happy
Halloween

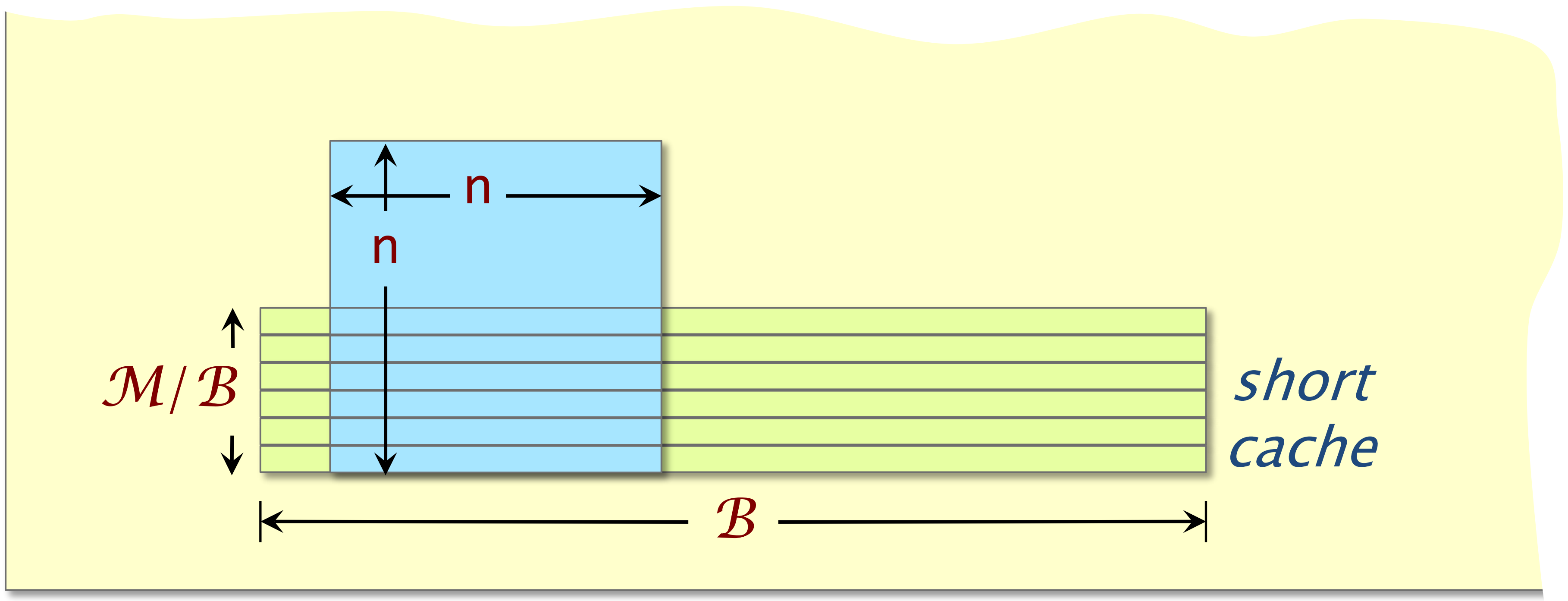
Tall Caches



Tall-cache assumption
$\mathcal{B}^2 < c\,\mathcal{M}$ for some sufficiently small constant $c \le 1$.

Example: Intel Xeon Platinum 8280L
- Cache-line length $\mathcal{B} = 64$ bytes.
- L1-cache size $\mathcal{M} = 32$ kibibytes.

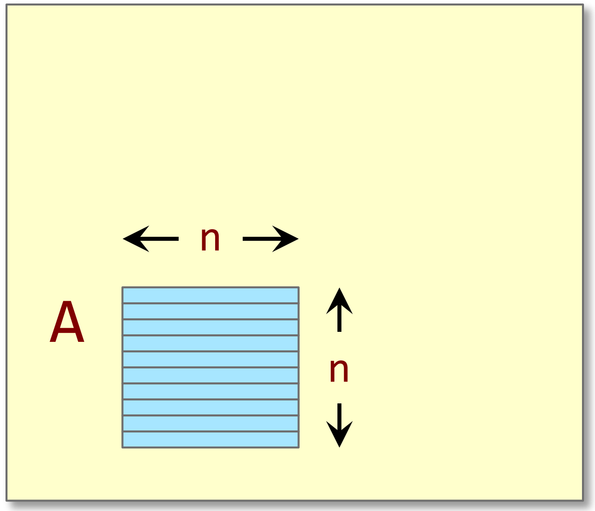
What's Wrong with Short Caches?



Tall-cache assumption
$\mathcal{B}^2 < c\mathcal{M}$ for some sufficiently small constant $c \le 1$.

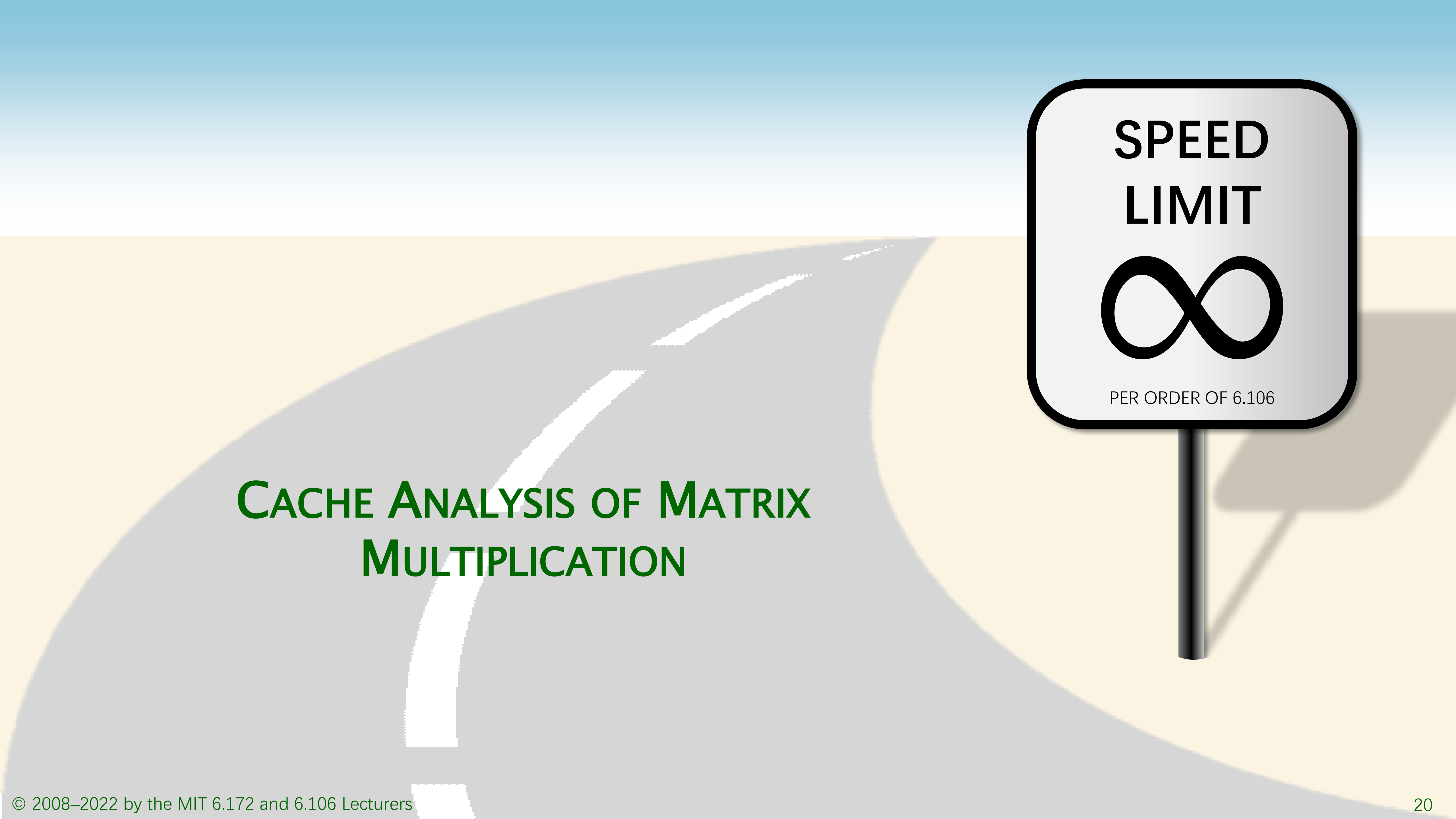
An $n \times n$ submatrix stored in row-major order may not fit in a short cache even if $n^2 < c\mathcal{M}$!

Submatrix Caching Lemma



Lemma. Suppose that an $n \times n$ submatrix A is read into a tall cache satisfying $\mathcal{B}^2 < c\mathcal{M}$, where $c < 1/3$ is constant, and suppose that $c\mathcal{M} \leq n^2 < \mathcal{M}/3$. Then A fits into the cache, and the number of misses to read all of A's elements is at most $3n^2/\mathcal{B}$.

Proof. We have $r = n$, $s_i = n$, $N = n^2$. Since $\mathcal{B}^2 < c\mathcal{M} \leq n^2$, we have $\mathcal{B} \leq n = N/r$. And since $N < \mathcal{M}/3$, the segment caching lemma applies. ■



Cache Analysis of Matrix Multiplication

Multiply Square Matrices

```
void Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i=0; i < n; i++)
    for (int64_t j=0; j < n; j++)
      for (int64_t k=0; k < n; k++)
        C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```

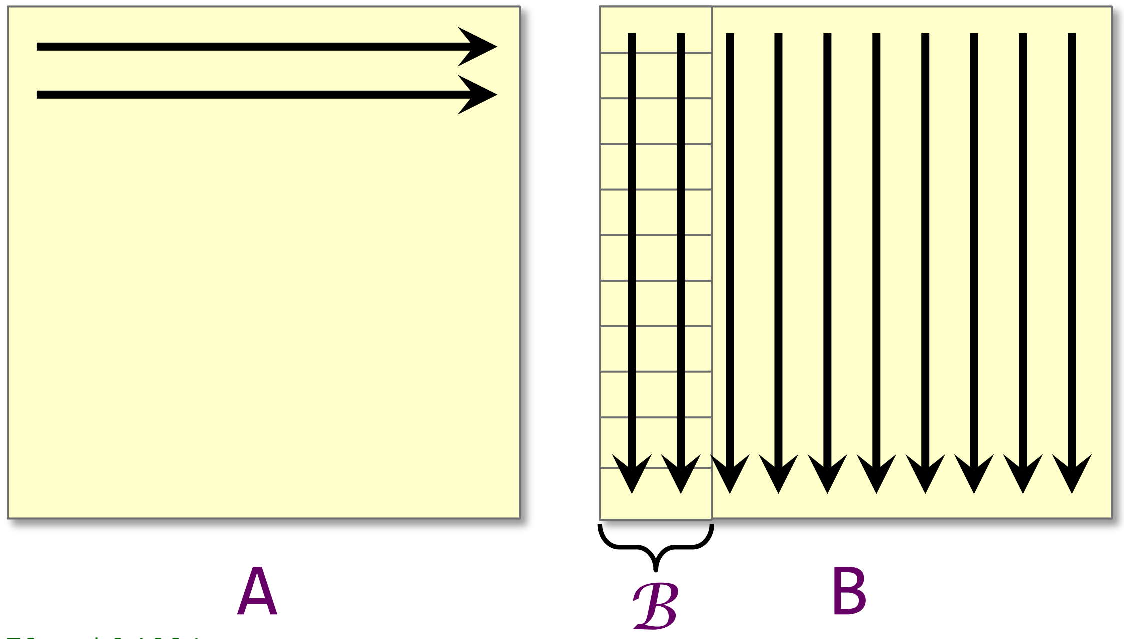
Analysis of work

$T(n) = \Theta(n^3)$.

Analysis of Cache Misses

```
void Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i=0; i < n; i++)
    for (int64_t j=0; j < n; j++)
      for (int64_t k=0; k < n; k++)
        C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```

Assume row major and tall cache



Case 1

$n \geq \mathcal{M}/\mathcal{B}$.

Analyze matrix B.

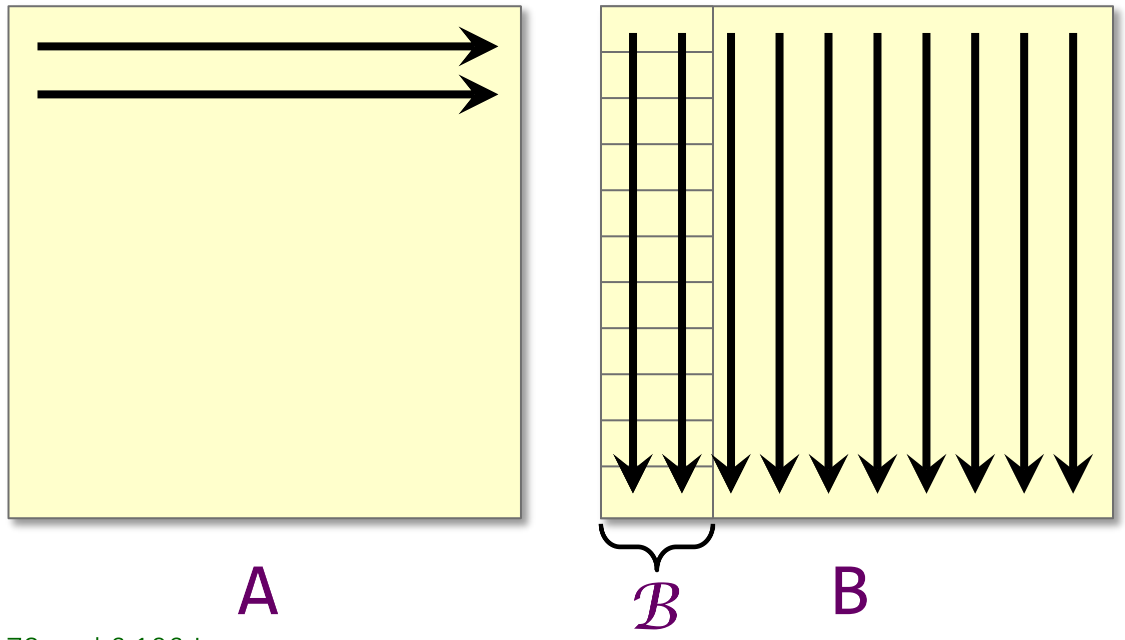
Assume LRU.

$Q(n) = \Theta(n^3)$, since matrix B misses on every access.

Analysis of Cache Misses

```
void Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i=0; i < n; i++)
    for (int64_t j=0; j < n; j++)
      for (int64_t k=0; k < n; k++)
        C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```

Assume row major and tall cache



Case 2

$\mathcal{M}^{1/2} \le n < \mathcal{M}/\mathcal{B}$.

Analyze matrix B.

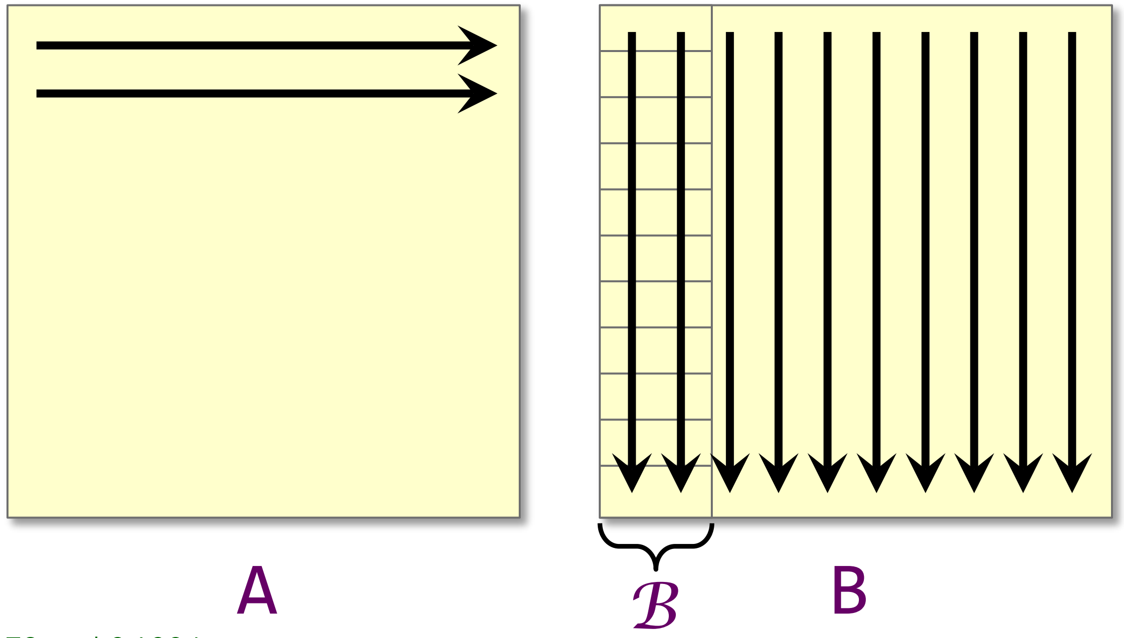
Assume LRU.

$Q(n) = n \cdot \Theta(n^2/\mathcal{B}) = \Theta(n^3/\mathcal{B})$, since matrix B can exploit spatial locality.

Analysis of Cache Misses

```
void Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i=0; i < n; i++)
    for (int64_t j=0; j < n; j++)
      for (int64_t k=0; k < n; k++)
        C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```

Assume row major and tall cache



Case 3

$n < c\mathcal{M}^{1/2}$.

Analyze matrix B.

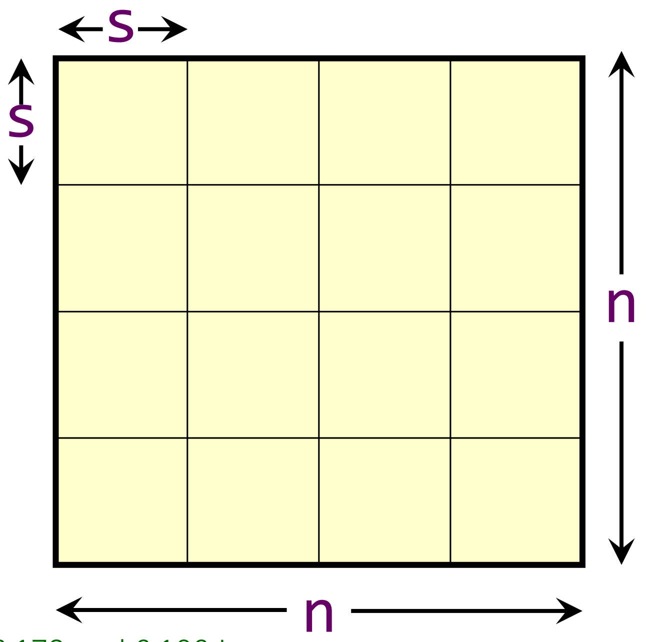
Assume LRU.

$Q(n) = \Theta(n^2/\mathcal{B})$, by the submatrix caching lemma.

Tiling

Tiled Matrix Multiplication

```
void Tiled_Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i1=0; i1<n/s; i1+=s)                          } outer nest
    for (int64_t j1=0; j1<n/s; j1+=s)
      for (int64_t k1=0; k1<n/s; k1+=s)
        for (int64_t i=i1; i<i1+s && i<n; i++)               } inner nest
          for (int64_t j=j1; j<j1+s && j<n; j++)
            for (int64_t k=k1; k<k1+s && k<n; k++)
              C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```

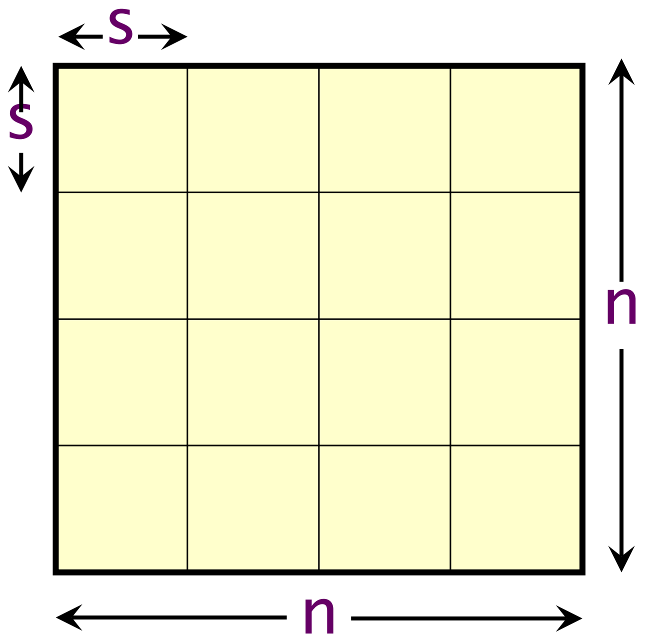


Analysis of work

- Work $T(n) = \Theta((n/s)^3(s^3))$
  $= \Theta(n^3)$.

Tiled Matrix Multiplication

```
void Tiled_Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i1=0; i1<n/s; i1+=s)                          } outer nest
    for (int64_t j1=0; j1<n/s; j1+=s)
      for (int64_t k1=0; k1<n/s; k1+=s)
        for (int64_t i=i1; i<i1+s && i<n; i++)               } inner nest
          for (int64_t j=j1; j<j1+s && j<n; j++)
            for (int64_t k=k1; k<k1+s && k<n; k++)
              C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```



Analysis of cache misses

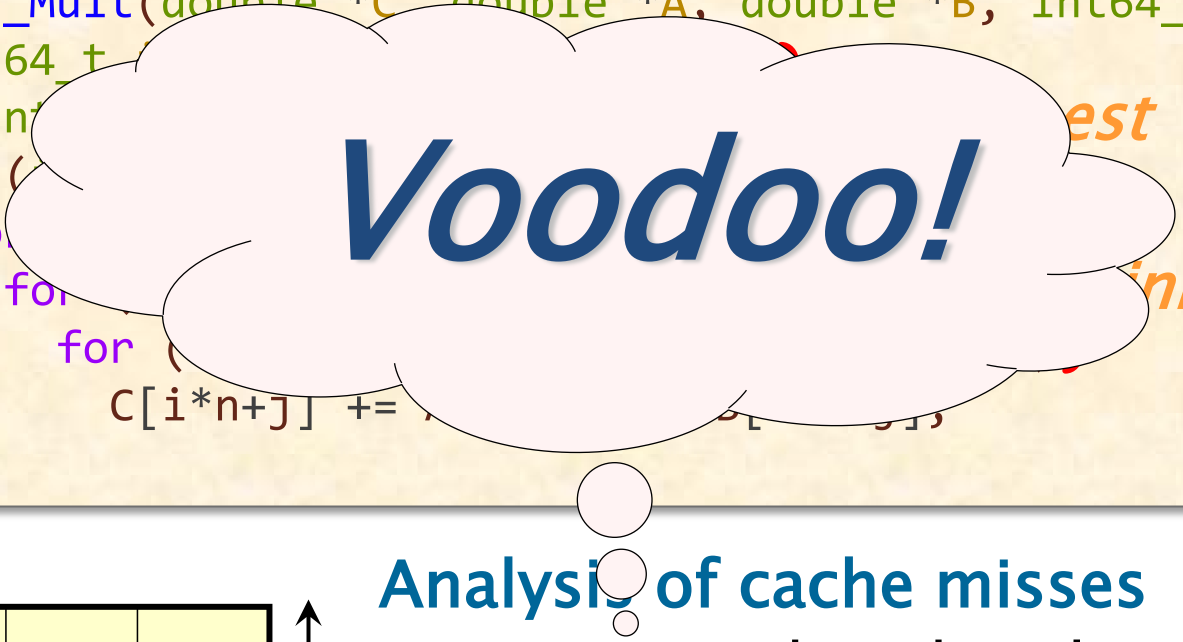
- Tune s so that the tiles just fit into cache ⇒ $s = \Theta(\mathcal{M}^{1/2})$.
- Submatrix caching lemma implies $\Theta(s^2/\mathcal{B})$ misses per tile.
- $Q(n) = \Theta((n/s)^3(s^2/\mathcal{B}))$
  $= \Theta(n^3/\mathcal{B}\mathcal{M}^{1/2})$.
- Optimal [HK81].

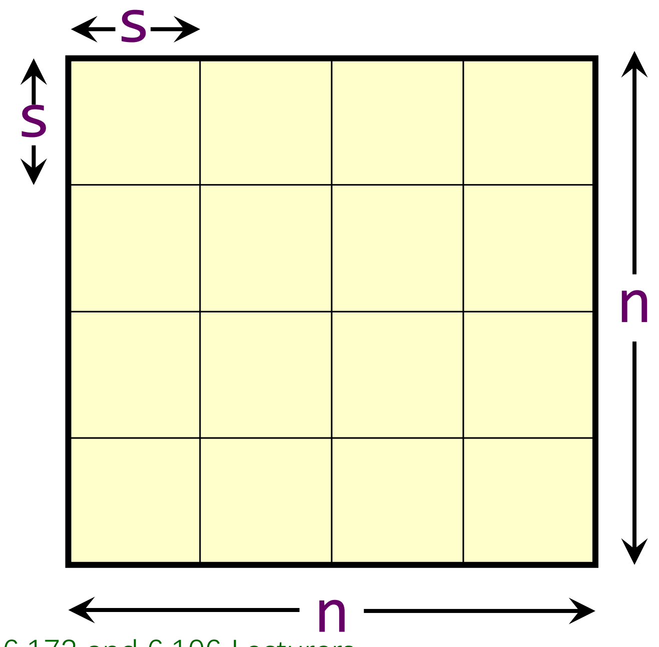
Tiled Matrix Multiplication

```
void Tiled_Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t
    for (in
      for (
        fo
          fo
            for (
              C[i*n+j] +=                    B[      j]];
}
```

est

inner nest

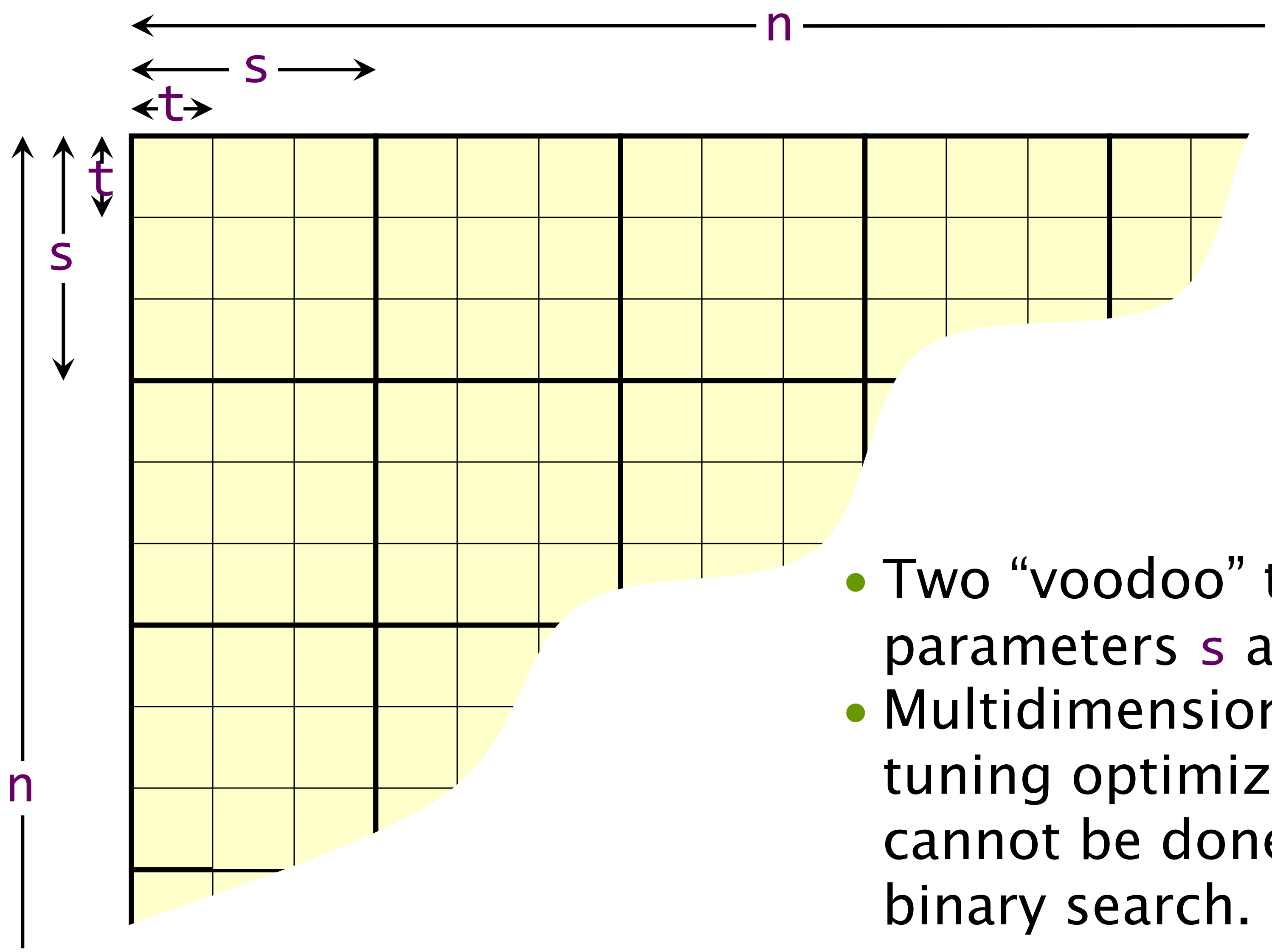




Analysis of cache misses

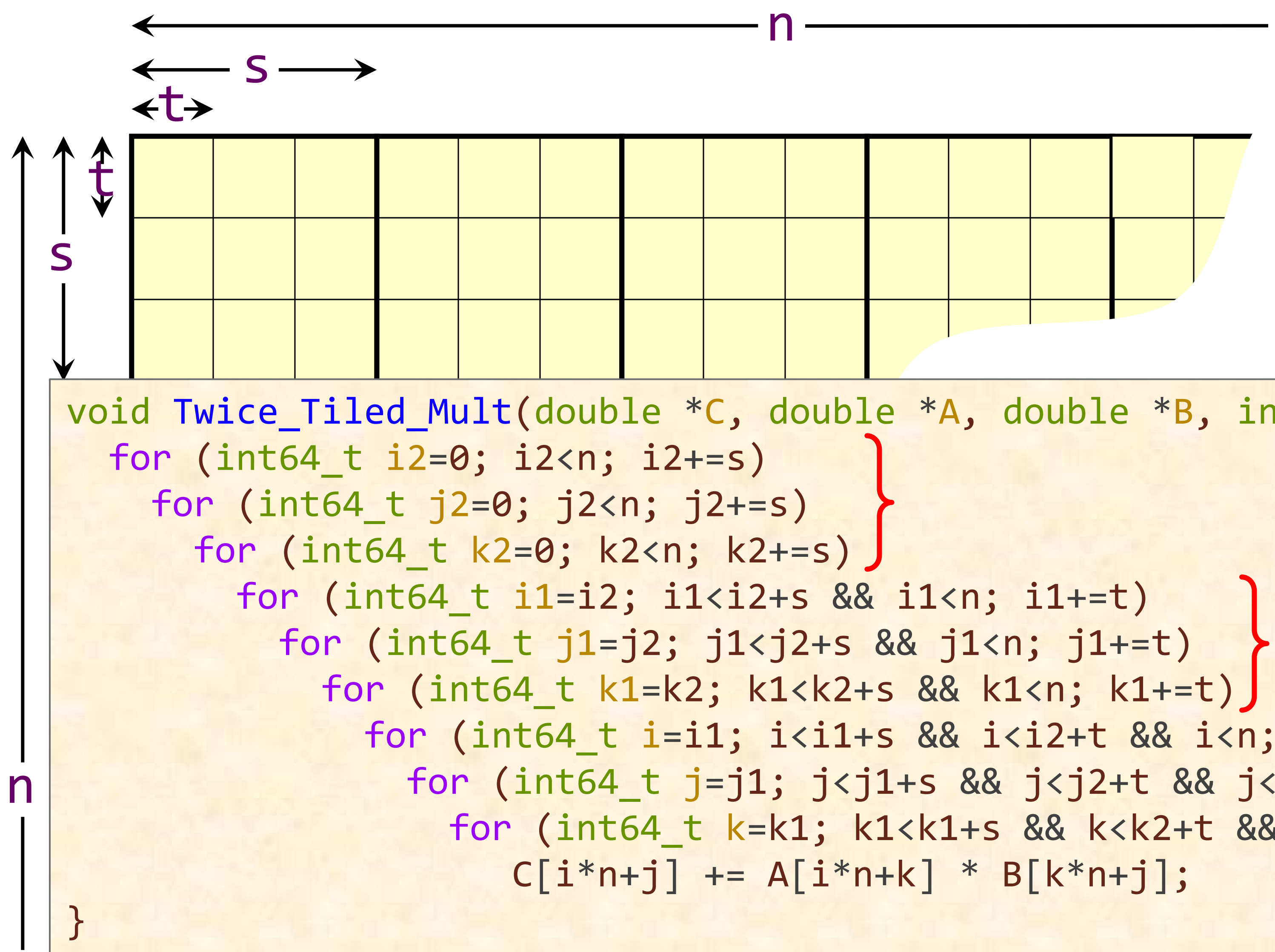
- Tune s so that the tiles just fit into cache ⇒ $s = \Theta(\mathcal{M}^{1/2})$.
- Submatrix caching lemma implies $\Theta(s^2/\mathcal{B})$ misses per tile.
- $Q(n) = \Theta((n/s)^3(s^2/\mathcal{B}))$
  $= \Theta(n^3/\mathcal{B}\mathcal{M}^{1/2})$.
- Optimal [HK81].

Two-Level Cache



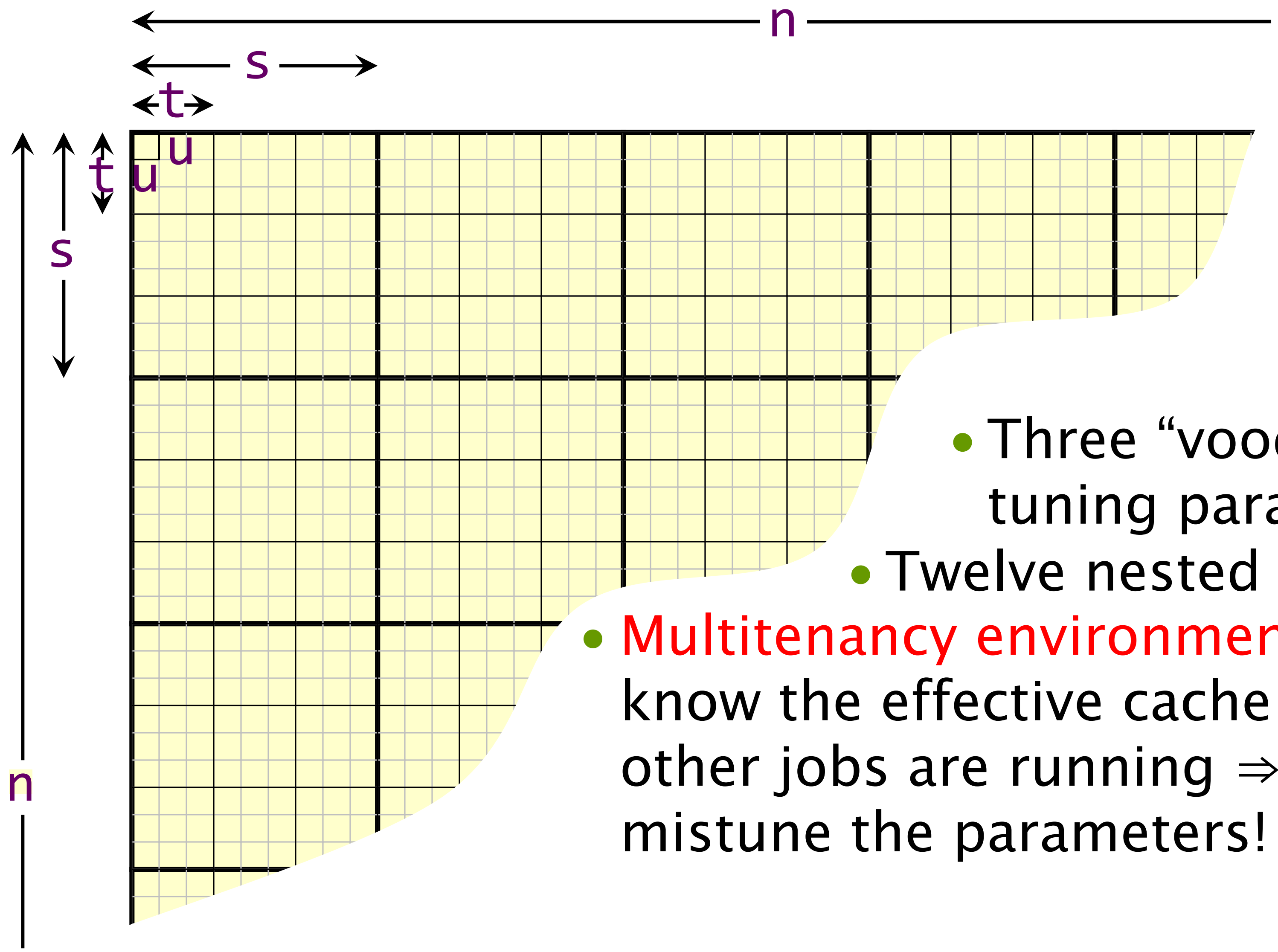
- Two “voodoo” tuning parameters `s` and `t`.
- Multidimensional tuning optimization cannot be done with binary search.

Two-Level Cache

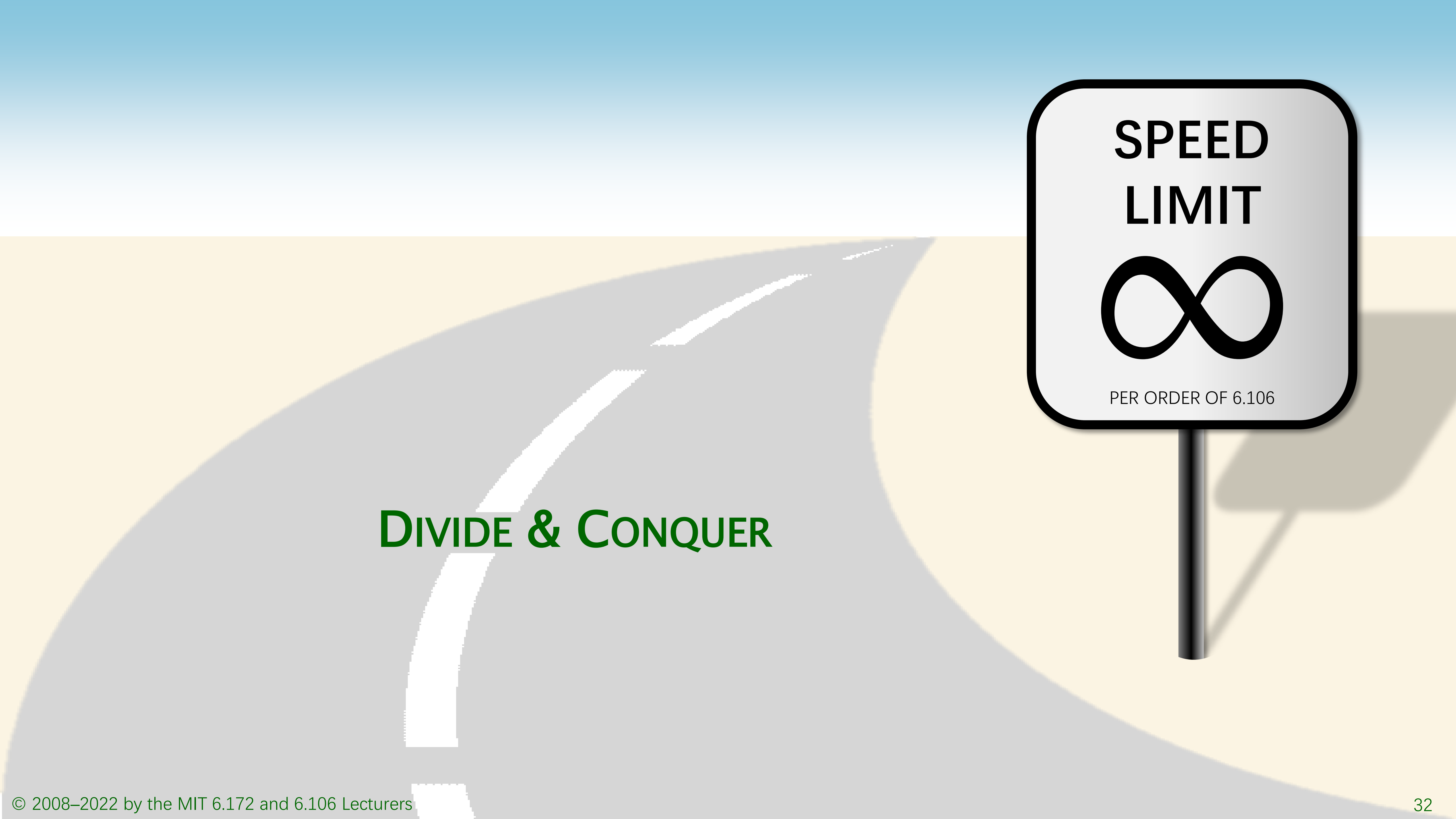


```
void Twice_Tiled_Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i2=0; i2<n; i2+=s)
    for (int64_t j2=0; j2<n; j2+=s)
      for (int64_t k2=0; k2<n; k2+=s)
        for (int64_t i1=i2; i1<i2+s && i1<n; i1+=t)
          for (int64_t j1=j2; j1<j2+s && j1<n; j1+=t)
            for (int64_t k1=k2; k1<k2+s && k1<n; k1+=t)
              for (int64_t i=i1; i<i1+s && i<i2+t && i<n; i++)
                for (int64_t j=j1; j<j1+s && j<j2+t && j<n; j++)
                  for (int64_t k=k1; k1<k1+s && k<k2+t && k<n; k++)
                     C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```

Three-Level Cache



- Three “voodoo” tuning parameters.
- Twelve nested `for` loops.
- Multitenancy environment: Don’t know the effective cache size when other jobs are running ⇒ easy to mistune the parameters!



Divide & Conquer

Recursive Matrix Multiplication

Divide-and-conquer on $n \times n$ matrices.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$= \begin{bmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{bmatrix} + \begin{bmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{bmatrix}$$

8 multiply-adds of $(n/2) \times (n/2)$ matrices.

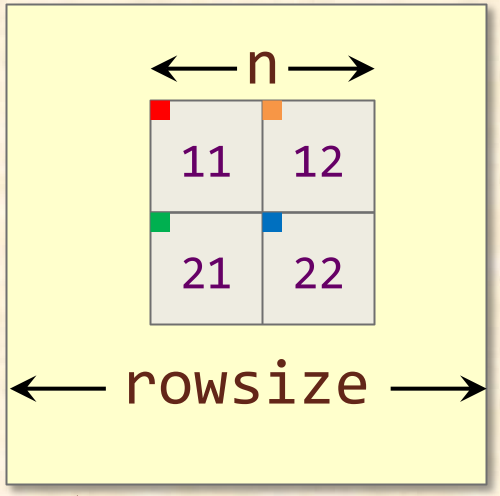
Recursive Code

```
// Assume that n is an exact power of 2.
void Rec_Mult(double *C, double *A, double *B,
              int64_t n, int64_t rowsize) {
  if (n == 1)
    C[0] += A[0] * B[0];
  else {
    int64_t d11 = 0;
    int64_t d12 = n/2;
    int64_t d21 = (n/2) * rowsize;
    int64_t d22 = (n/2) * (rowsize+1);

    Rec_Mult(C+d11, A+d11, B+d11, n/2, rowsize);
    Rec_Mult(C+d11, A+d12, B+d21, n/2, rowsize);
    Rec_Mult(C+d12, A+d11, B+d12, n/2, rowsize);
    Rec_Mult(C+d12, A+d12, B+d22, n/2, rowsize);
    Rec_Mult(C+d21, A+d21, B+d11, n/2, rowsize);
    Rec_Mult(C+d21, A+d22, B+d21, n/2, rowsize);
    Rec_Mult(C+d22, A+d21, B+d12, n/2, rowsize);
    Rec_Mult(C+d22, A+d22, B+d22, n/2, rowsize);
  }
}
```

Coarsen base case to overcome function-call overheads.

Recursive Code

```
// Assume that n is an exact power of 2.
void Rec_Mult(double *C, double *A, double *B,
              int64_t n, int64_t rowsize) {
  if (n == 1)
    C[0] += A[0] * B[0];
  else {
    int64_t d11 = 0;
    int64_t d12 = n/2;
    int64_t d21 = (n/2) * rowsize;
    int64_t d22 = (n/2) * (rowsize+1);

    Rec_Mult(C+d11, A+d11, B+d11, n/2, rowsize);
    Rec_Mult(C+d11, A+d12, B+d21, n/2, rowsize);
    Rec_Mult(C+d12, A+d11, B+d12, n/2, rowsize);
    Rec_Mult(C+d12, A+d12, B+d22, n/2, rowsize);
    Rec_Mult(C+d21, A+d21, B+d11, n/2, rowsize);
    Rec_Mult(C+d21, A+d22, B+d21, n/2, rowsize);
    Rec_Mult(C+d22, A+d21, B+d12, n/2, rowsize);
    Rec_Mult(C+d22, A+d22, B+d22, n/2, rowsize);
  }
}
```



Analysis of Work

```
// Assume that n is an exact power of 2.
void Rec_Mult(double *C, double *A, double *B,
              int64_t n, int64_t rowsize) {
  if (n == 1)
    C[0] += A[0] * B[0];
  else {
    int64_t d11 = 0;
    int64_t d12 = n/2;
    int64_t d21 = (n/2) * rowsize;
    int64_t d22 = (n/2) * (rowsize+1);

    Rec_Mult(C+d11, A+d11, B+d11, n/2, rowsize);
    Rec_Mult(C+d11, A+d12, B+d21, n/2, rowsize);
    Rec_Mult(C+d12, A+d11, B+d12, n/2, rowsize);
    Rec_Mult(C+d12, A+d12, B+d22, n/2, rowsize);
    Rec_Mult(C+d21, A+d21, B+d11, n/2, rowsize);
    Rec_Mult(C+d21, A+d22, B+d21, n/2, rowsize);
    Rec_Mult(C+d22, A+d21, B+d12, n/2, rowsize);
    Rec_Mult(C+d22, A+d22, B+d22, n/2, rowsize);
  }
}
```

$$\begin{aligned} T(n) &= 8\,T(n/2) + \Theta(1) \\ &= \Theta(n^3) \end{aligned}$$

Analysis of Work

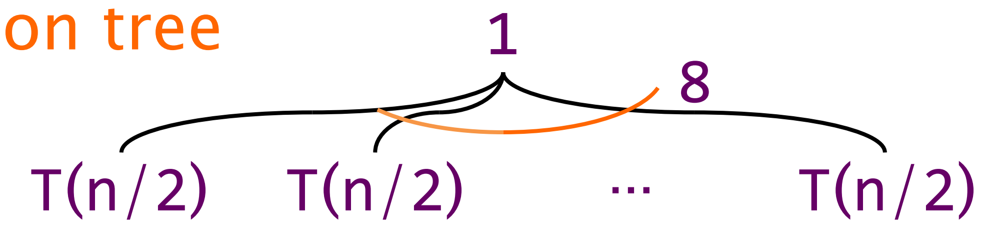
$$\boxed{T(n) = 8T(n/2) + 1}$$

recursion tree $T(n)$

Analysis of Work

$$\boxed{T(n) = 8T(n/2) + 1}$$

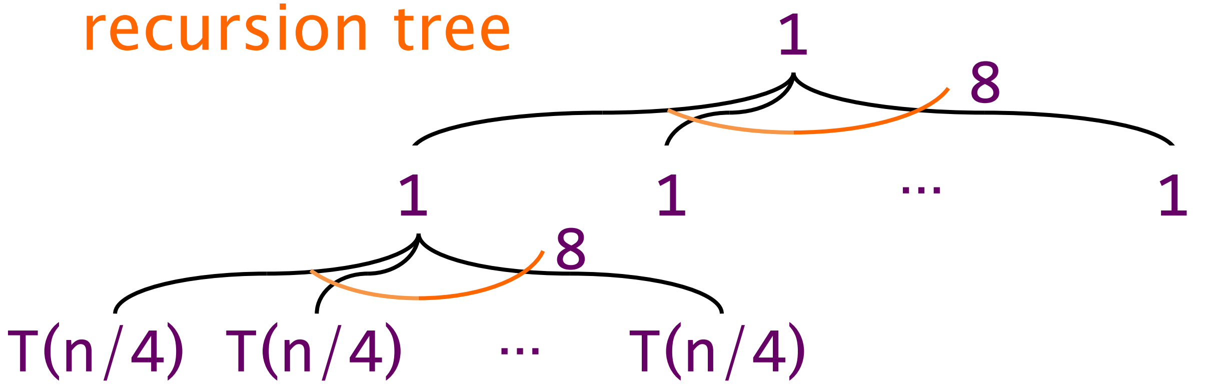
recursion tree



Analysis of Work

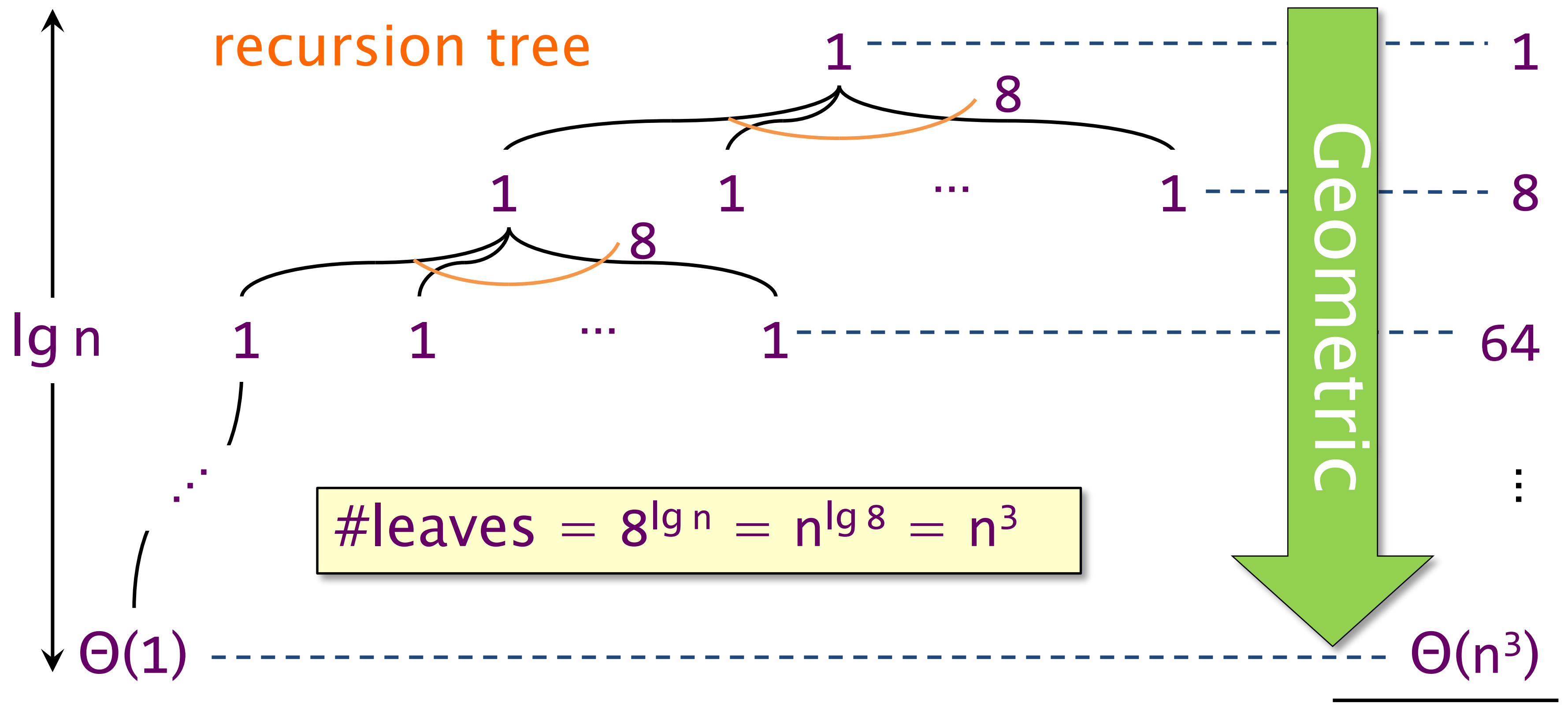
$$\boxed{T(n) = 8T(n/2) + 1}$$

recursion tree



Analysis of Work

$$\boxed{T(n) = 8\,T(n/2) + 1}$$



$\#\text{leaves} = 8^{\lg n} = n^{\lg 8} = n^3$

Note: Same work as looping versions.

$T(n) = \Theta(n^3)$

Analysis of Cache Misses

```c
// Assume that n is an exact power of 2.
void Rec_Mult(double *C, double *A, double *B,
              int64_t n, int64_t rowsize) {
  if (n == 1)
    C[0] += A[0] * B[0];
  else {
    int64_t d11 = 0;
    int64_t d12 = n/2;
    int64_t d21 = (n/2) * rowsize;
    int64_t d22 = (n/2) * (rowsize+1);

    Rec_Mult(C+d11, A+d11, B+d11, n/2, rowsize);
    Rec_Mult(C+d11, A+d12, B+d21, n/2, rowsize);
    Rec_Mult(C+d12, A+d11, B+d12, n/2, rowsize);
    Rec_Mult(C+d12, A+d12, B+d22, n/2, rowsize);
    Rec_Mult(C+d21, A+d21, B+d11, n/2, rowsize);
    Rec_Mult(C+d21, A+d22, B+d21, n/2, rowsize);
    Rec_Mult(C+d22, A+d21, B+d12, n/2, rowsize);
    Rec_Mult(C+d22, A+d22, B+d22, n/2, rowsize);
  }
}
```

Submatrix caching lemma

$$Q(n) = \begin{cases} \Theta(n^2/\mathcal{B}) \text{ if } n^2 < c\mathcal{M} \text{ for suff. small const } c \le 1, \\ 8\,Q(n/2) + \Theta(1) \text{ otherwise.} \end{cases}$$

Analysis of Cache Misses

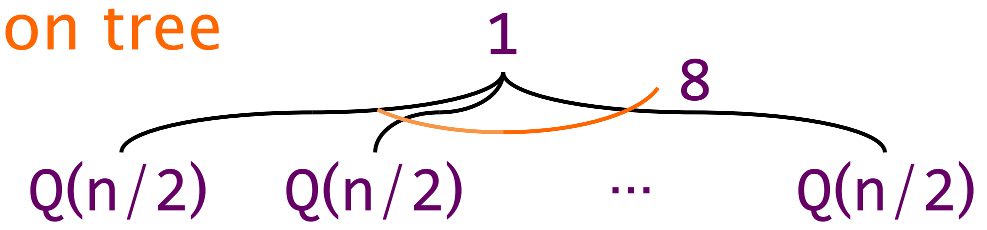
$$Q(n) = \begin{cases} \Theta(n^2/\mathcal{B}) & \text{if } n^2 < c\mathcal{M} \text{ for suff. small const } c \le 1, \\ 8\,Q(n/2) + 1 & \text{otherwise.} \end{cases}$$

recursion tree $Q(n)$

Analysis of Cache Misses

$$Q(n) = \begin{cases} \Theta(n^2/\mathcal{B}) \text{ if } n^2 < c\mathcal{M} \text{ for suff. small const } c \le 1, \\ 8\,Q(n/2) + 1 \text{ otherwise.} \end{cases}$$

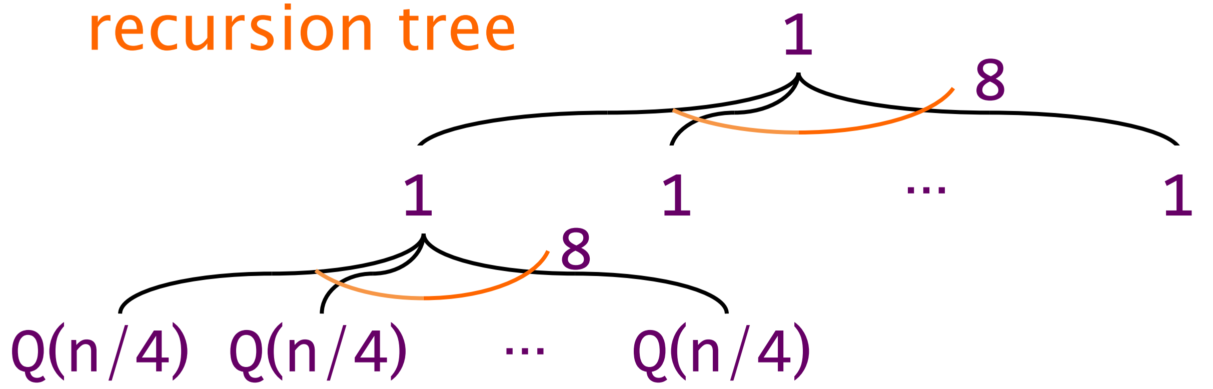
recursion tree



Analysis of Cache Misses

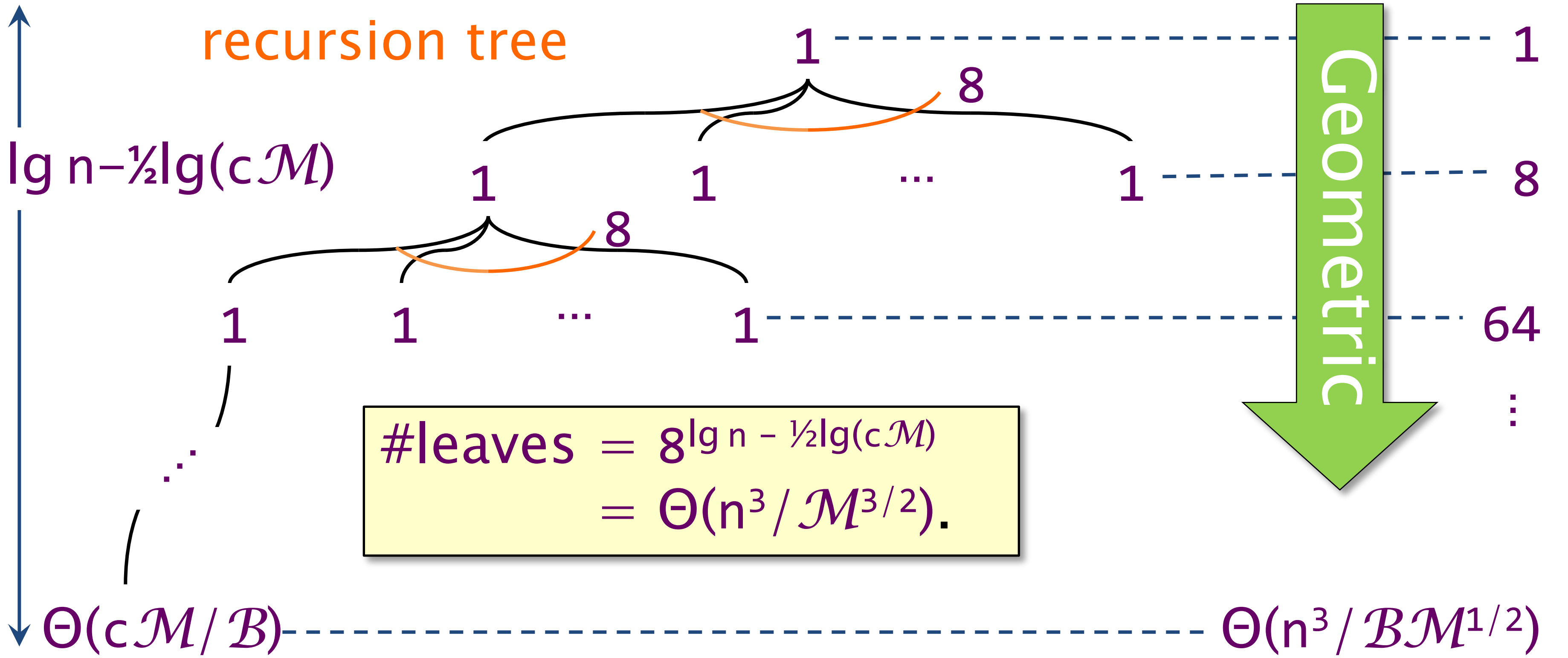
$$Q(n) = \begin{cases} \Theta(n^2/\mathcal{B}) & \text{if } n^2 < c\mathcal{M} \text{ for suff. small const } c \le 1, \\ 8\,Q(n/2) + 1 & \text{otherwise.} \end{cases}$$

recursion tree



Analysis of Cache Misses

$$Q(n) = \begin{cases} \Theta(n^2/\mathcal{B}) & \text{if } n^2 < c\mathcal{M} \text{ for suff. small const } c \le 1, \\ 8\,Q(n/2) + 1 & \text{otherwise.} \end{cases}$$


recursion tree
1
8
1
1
…
1
8
1
1
…
1
$\lg n - \frac{1}{2}\lg(c\mathcal{M})$
Geometric
1
8
64
⋮
$\Theta(c\mathcal{M}/\mathcal{B})$
$\Theta(n^3/\mathcal{B}\mathcal{M}^{1/2})$

$$\#\text{leaves} = 8^{\lg n - \frac{1}{2}\lg(c\mathcal{M})} = \Theta(n^3/\mathcal{M}^{3/2}).$$

Same cache misses as with tiling!

$$Q(n) = \Theta(n^3/\mathcal{B}\mathcal{M}^{1/2})$$

Cache-Oblivious Algorithms

Cache-oblivious algorithms [FLPR99]

- No voodoo tuning parameters.
- No explicit knowledge of caches.
- Passively autotune.
- Handle multilevel caches automatically.
- Good in multitenancy environments.

Recursive Parallel Matrix Multiply

```
// Assume that n is an exact power of 2.
void Rec_Mult(double *C, double *A, double *B,
              int64_t n, int64_t rowsize) {
  if (n == 1)
    C[0] += A[0] * B[0];
  else {
    int64_t d11 = 0;
    int64_t d12 = n/2;
    int64_t d21 = (n/2) * rowsize;
    int64_t d22 = (n/2) * (rowsize+1);

    cilk_scope {
      cilk_spawn Rec_Mult(C+d11, A+d11, B+d11, n/2, rowsize);
      cilk_spawn Rec_Mult(C+d21, A+d22, B+d21, n/2, rowsize);
      cilk_spawn Rec_Mult(C+d12, A+d11, B+d12, n/2, rowsize);
```

Minimizing cache misses in serial projection tends to minimize them in (Cilk) parallel executions.

```
      cilk_spawn Rec_Mult(C+d11, A+d12, B+d21, n/2, rowsize);
      cilk_spawn Rec_Mult(C+d21, A+d21, B+d11, n/2, rowsize);
      cilk_spawn Rec_Mult(C+d12, A+d12, B+d22, n/2, rowsize);
      cilk_spawn Rec_Mult(C+d22, A+d21, B+d12, n/2, rowsize);
    }
  }
}
```