Software Performance Engineering

LECTURE 13
Cache-Efficient Algorithms

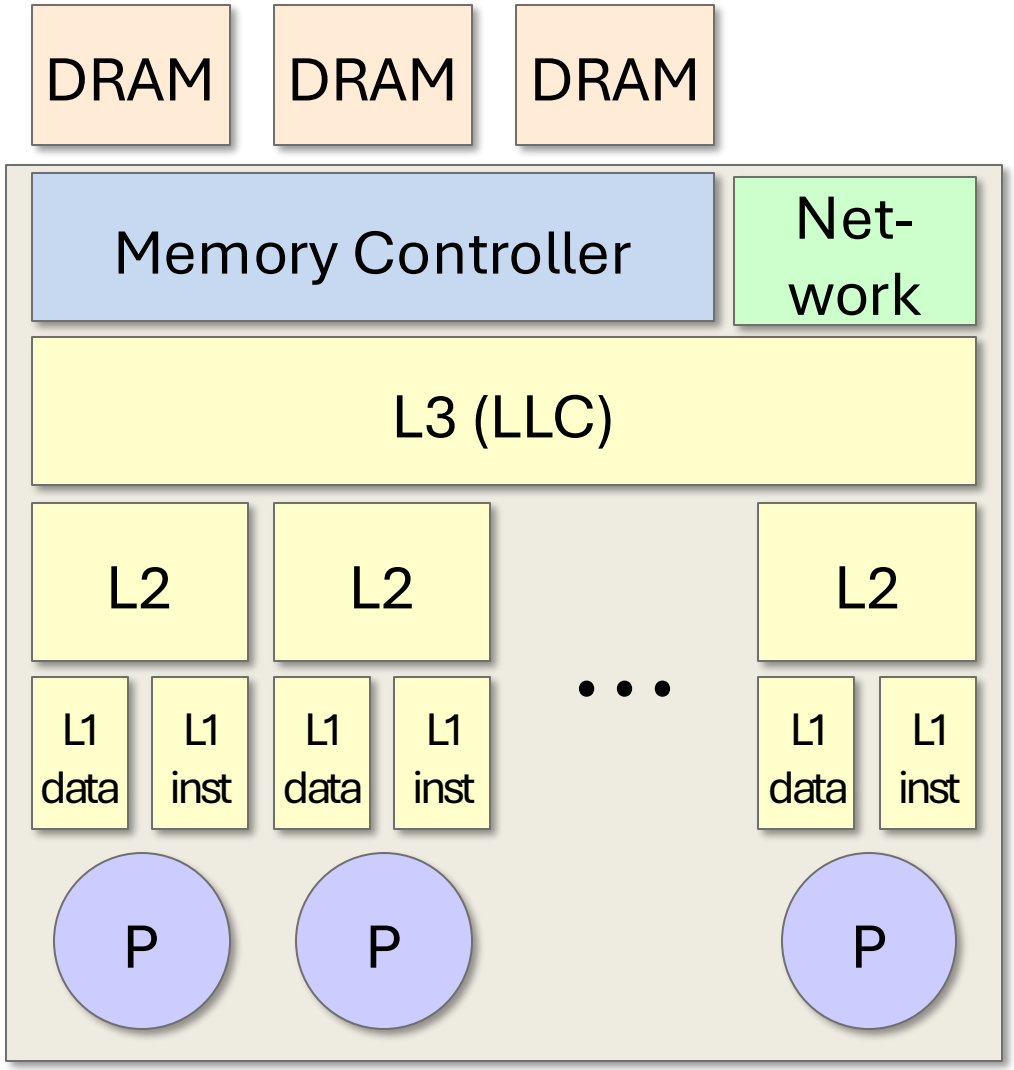
Xuhao Chen

November 11, 2025

Outline

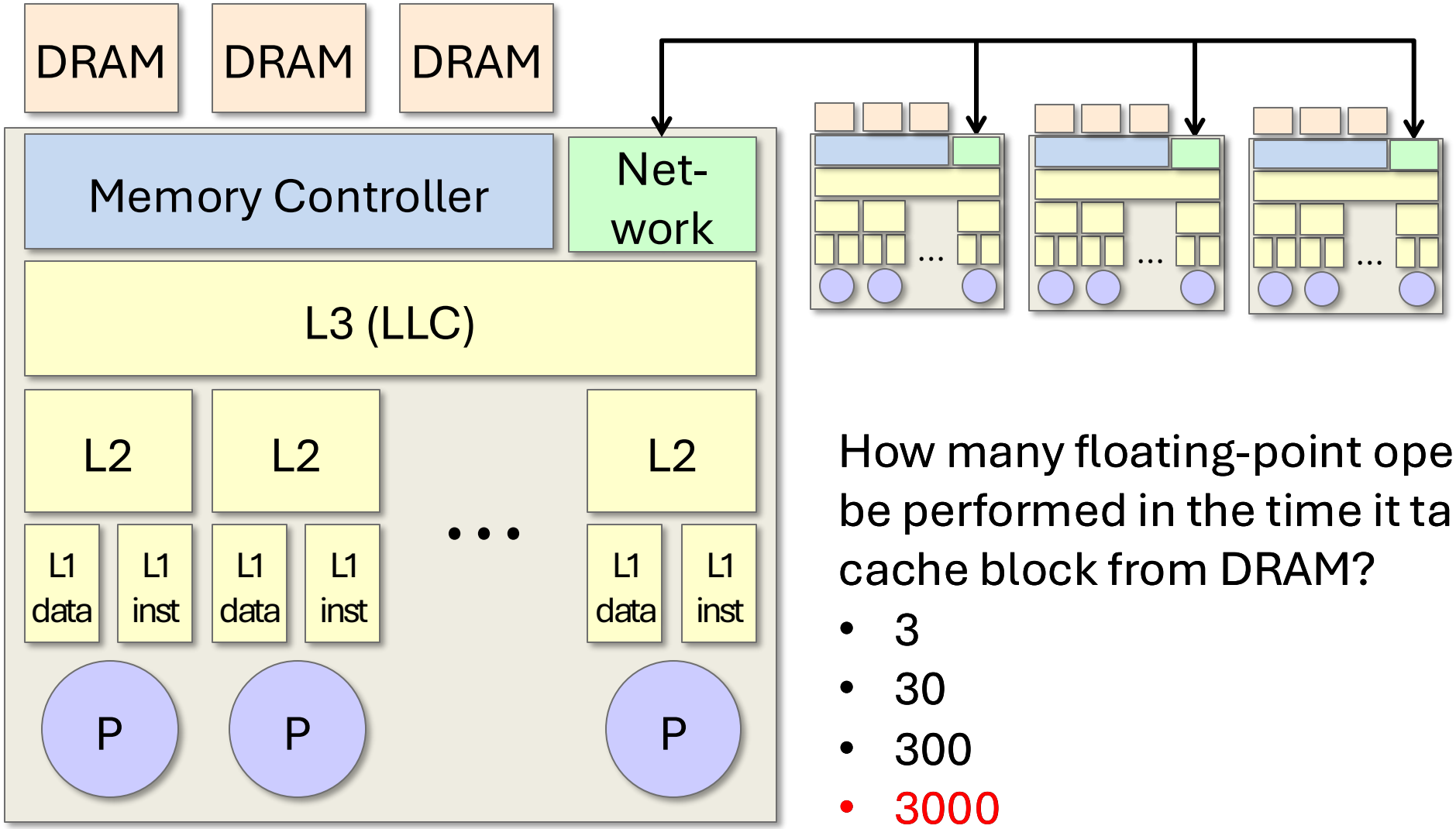
- Cache Hardware
- Ideal Cache Model
- Cache Analysis of Matrix Multiplication
- Tiling
- Divide and Conquer

Multicore Cache Hierarchy



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

Multicore Cache Hierarchy

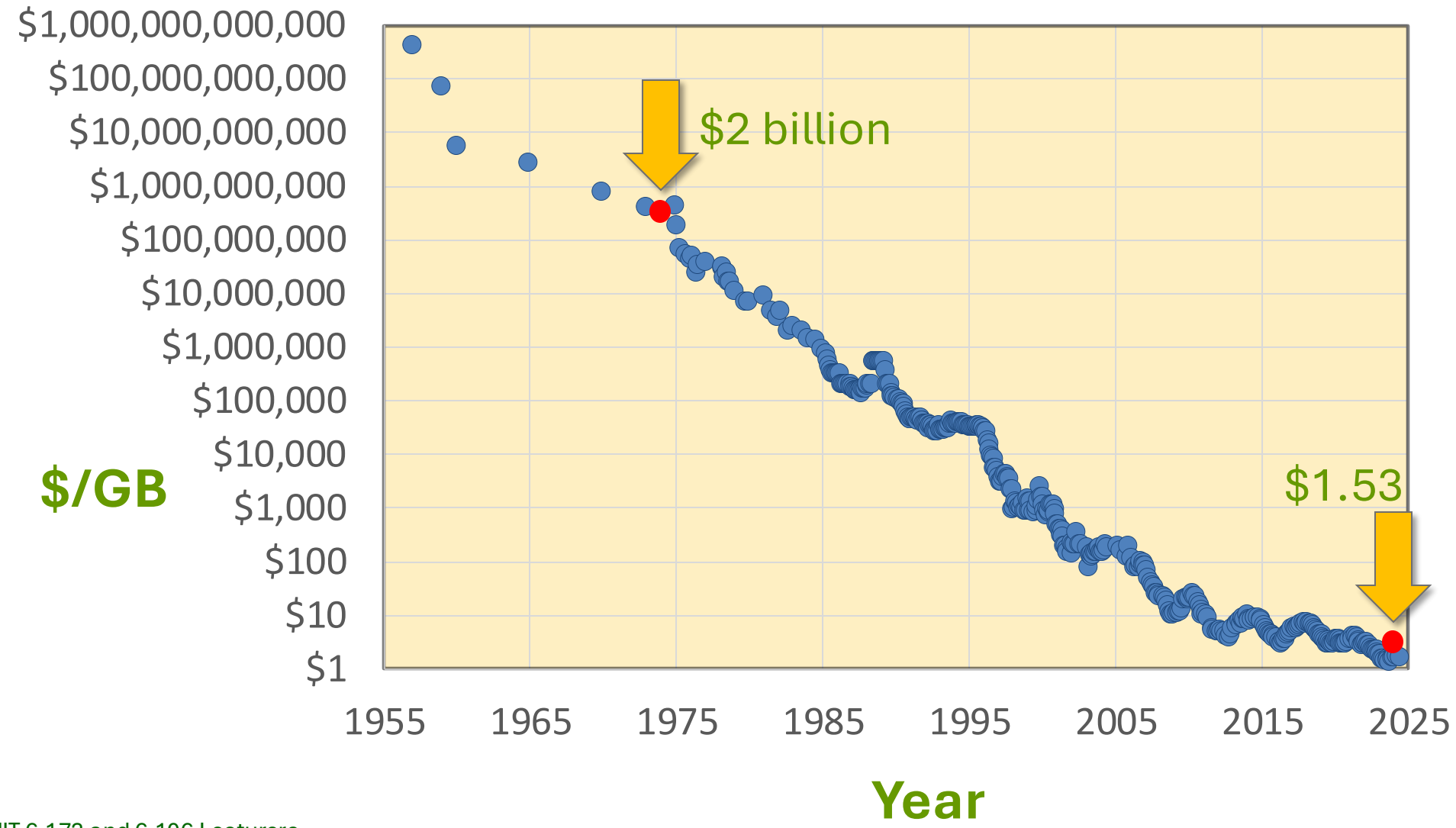


How many floating-point operations can be performed in the time it takes to fetch a cache block from DRAM?

- 3
- 30
- 300
- 3000

Memory Prices through History

Source: John C. McCallum, "Memory prices 1957+," available at http://jcmit.net/memoryprice.htm, last updated 2024-10-27.



Cache Specs for a High-End Multicore

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

Intel Xeon Platinum 8280L (Cascade Lake)

- Launched April 2019 for $17,906 — now ~$13K
- 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

Cache Specs for a High-End Multicore

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

Intel Xeon Platinum 8280L (Cascade Lake)

- Launched April 2019 for $17,906 — now ~$13K
- 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

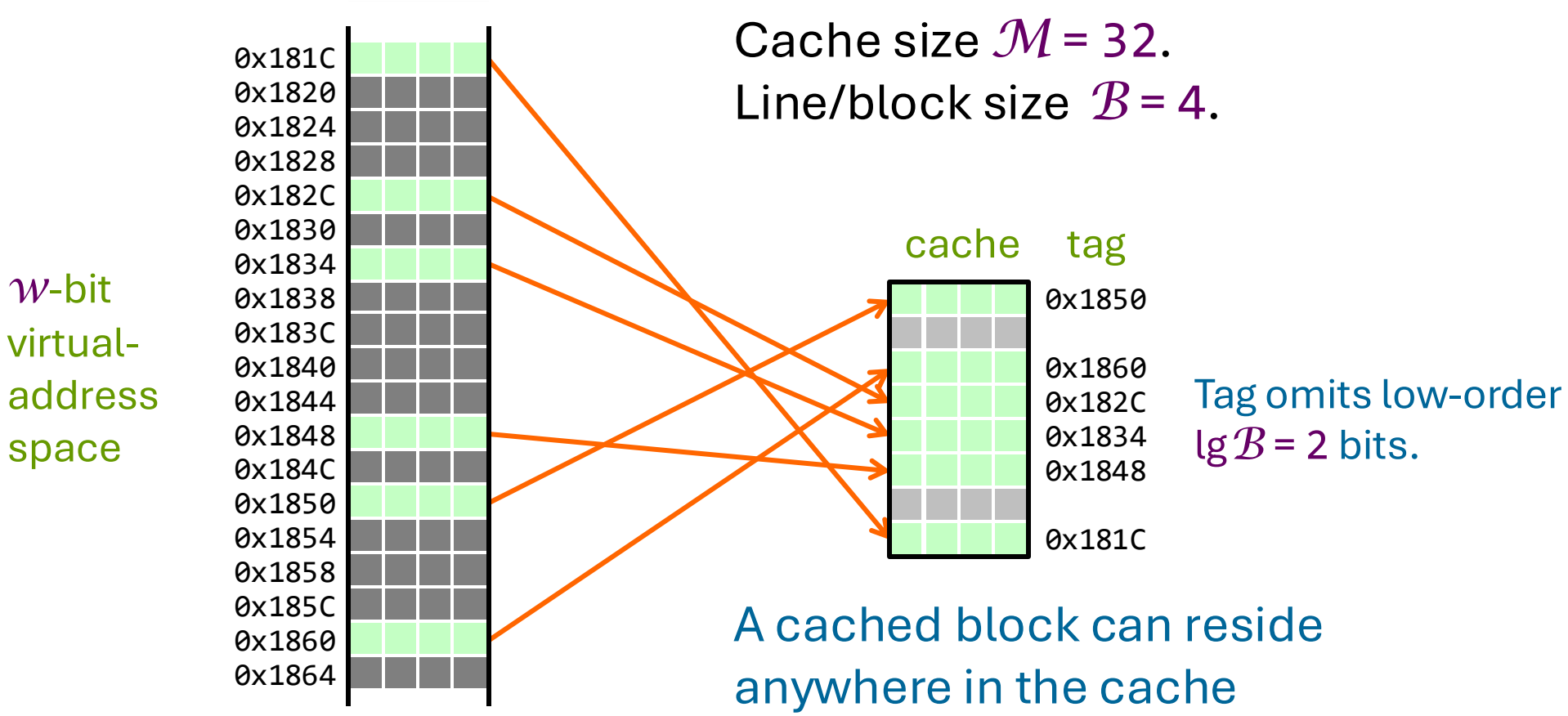
Cache Specs for a High-End Multicore

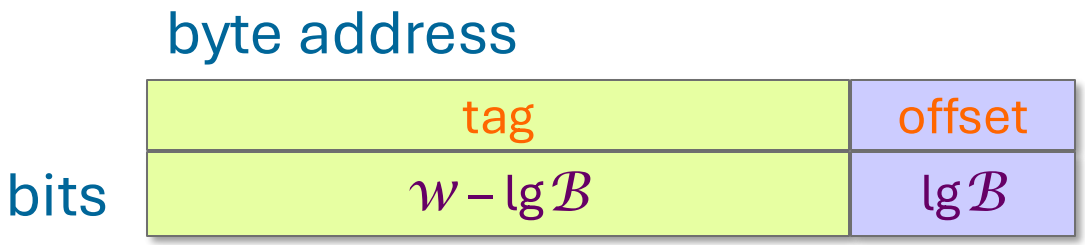
Level	Size/core	Associativity	Latency (cycles)
DRAM	up to **160 GiB**		**85–240**
L3	**1.375MiB**	**11**	**50–70**
L2	**1MiB**	**16**	**14**
L1-D	**32KiB**	**8**	**4–5**
L1-I	**32KiB**	**8**	**5**

Intel Xeon Platinum 8280L (Cascade Lake)

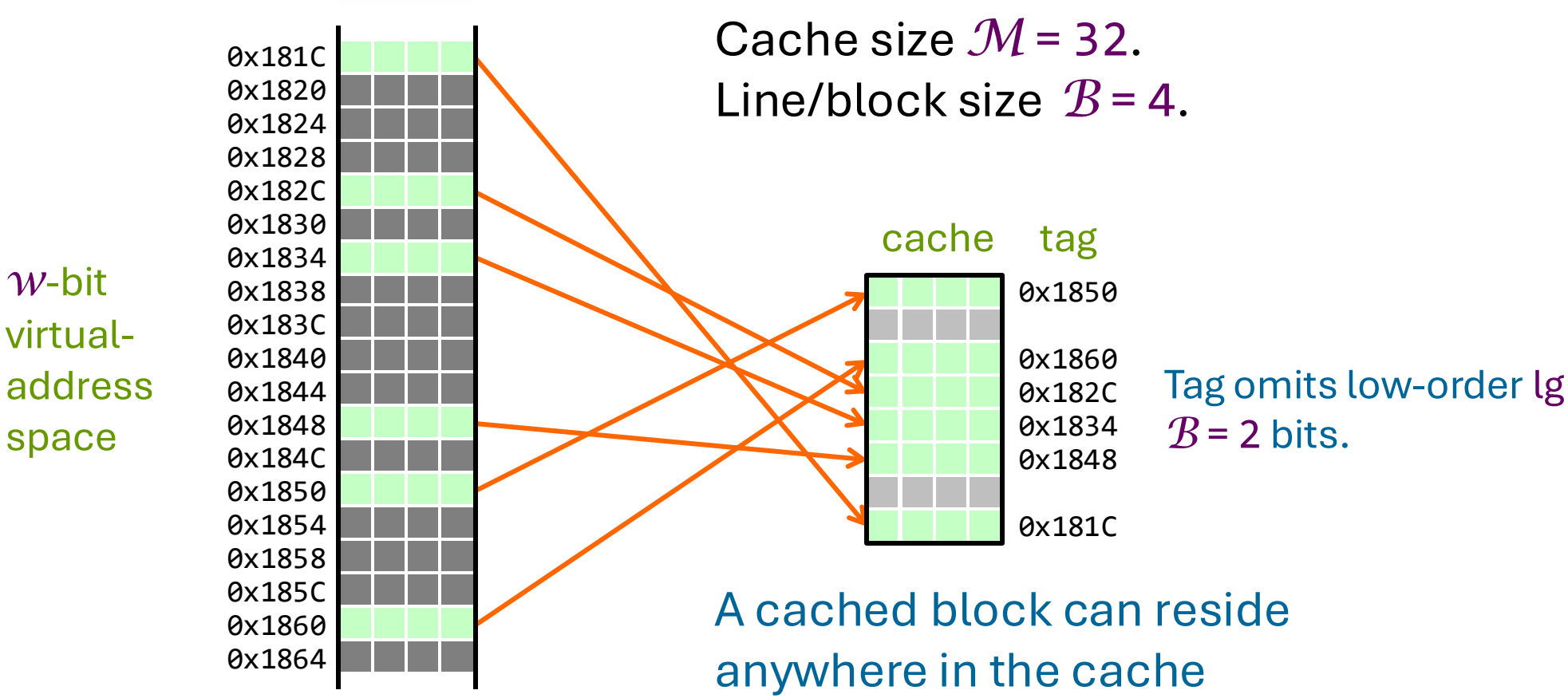
- Launched April 2019 for $17,906 — now ~$13K
- 2.7 GHz clock, Turbo Boost up to 4 GHz
- 28 cores/chip + 2-way hyperthreading
- 2190 GFLOPS
- 64 B cache lines/blocks
- Up to 8-way multiprocessing

Fully Associative Cache





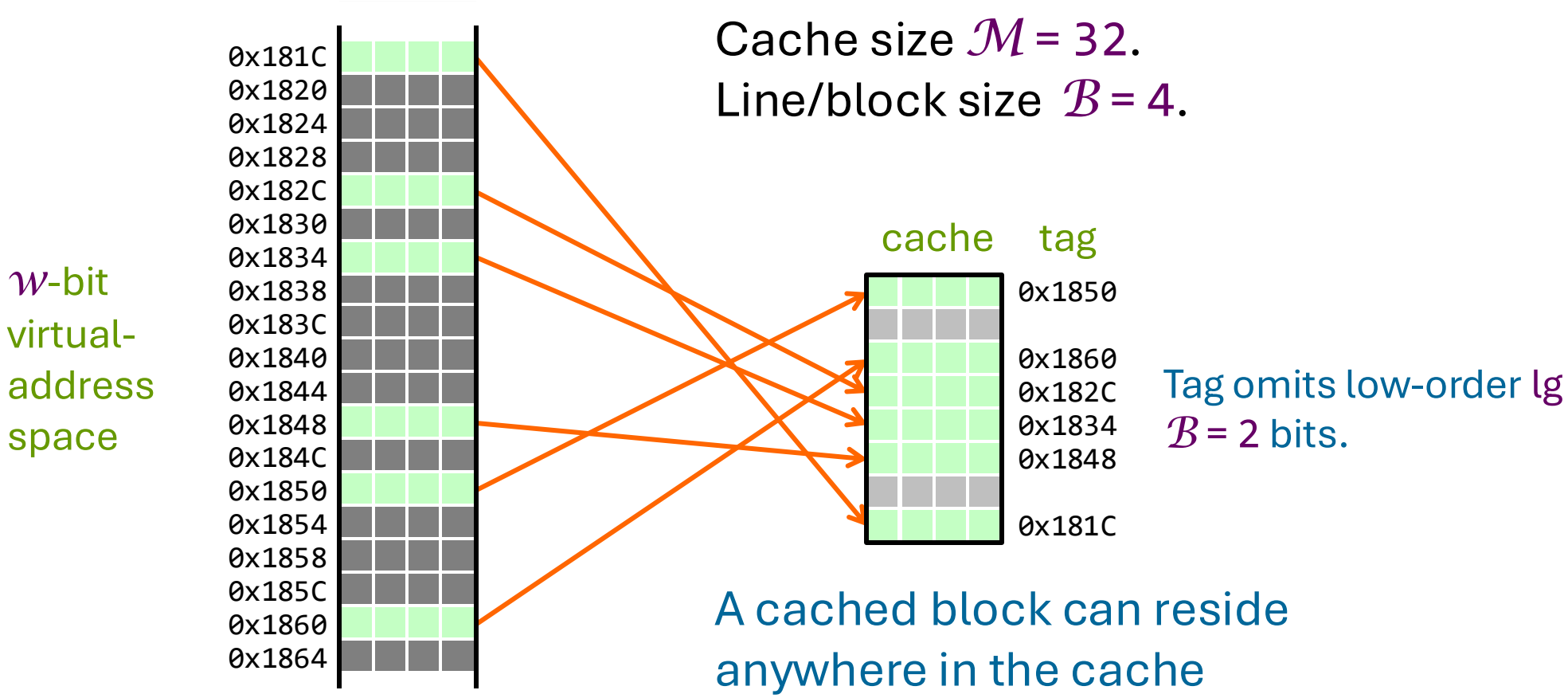
Fully Associative Cache



- To find a block in the cache, all the cache lines must be searched for the tag

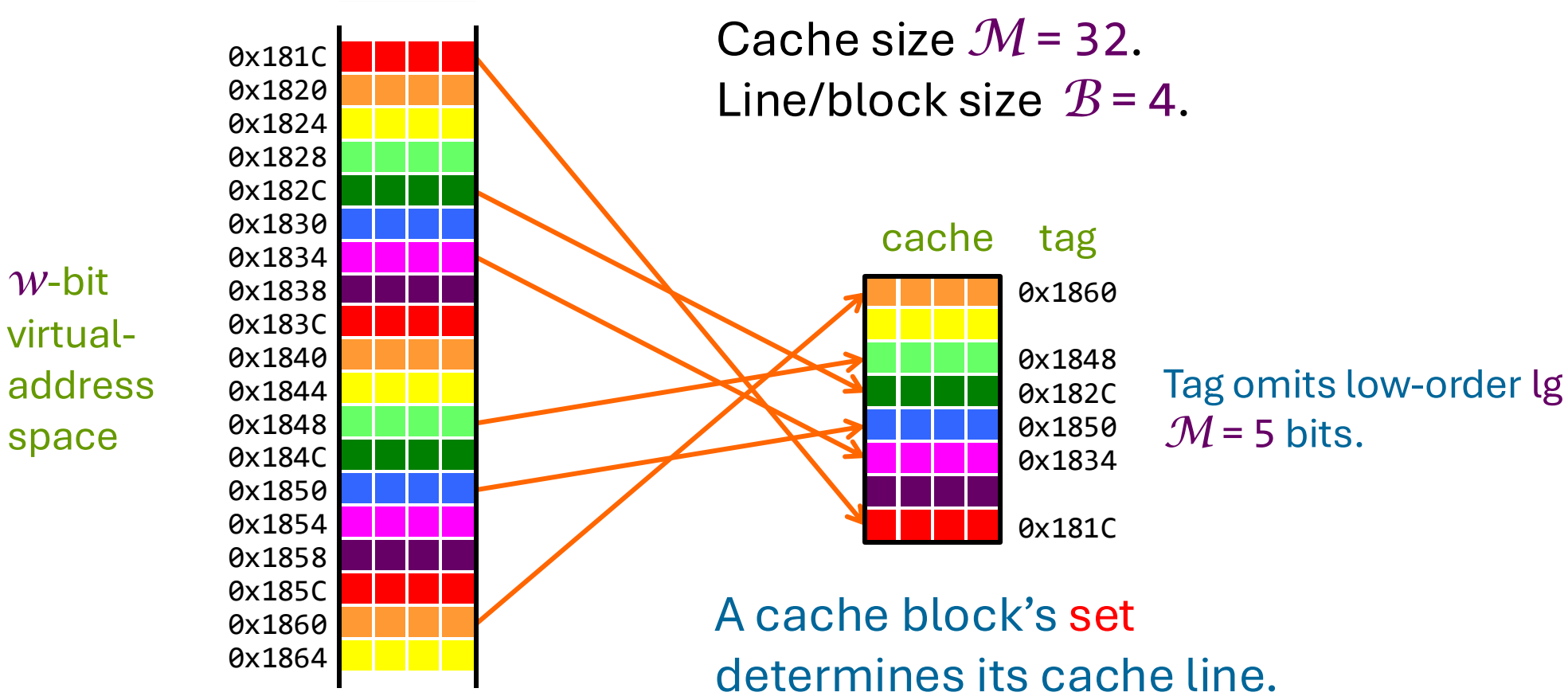
  HW use content-addressable memory for fast, parallel Tag search

Fully Associative 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, e.g. **LRU**

Direct-Mapped Cache

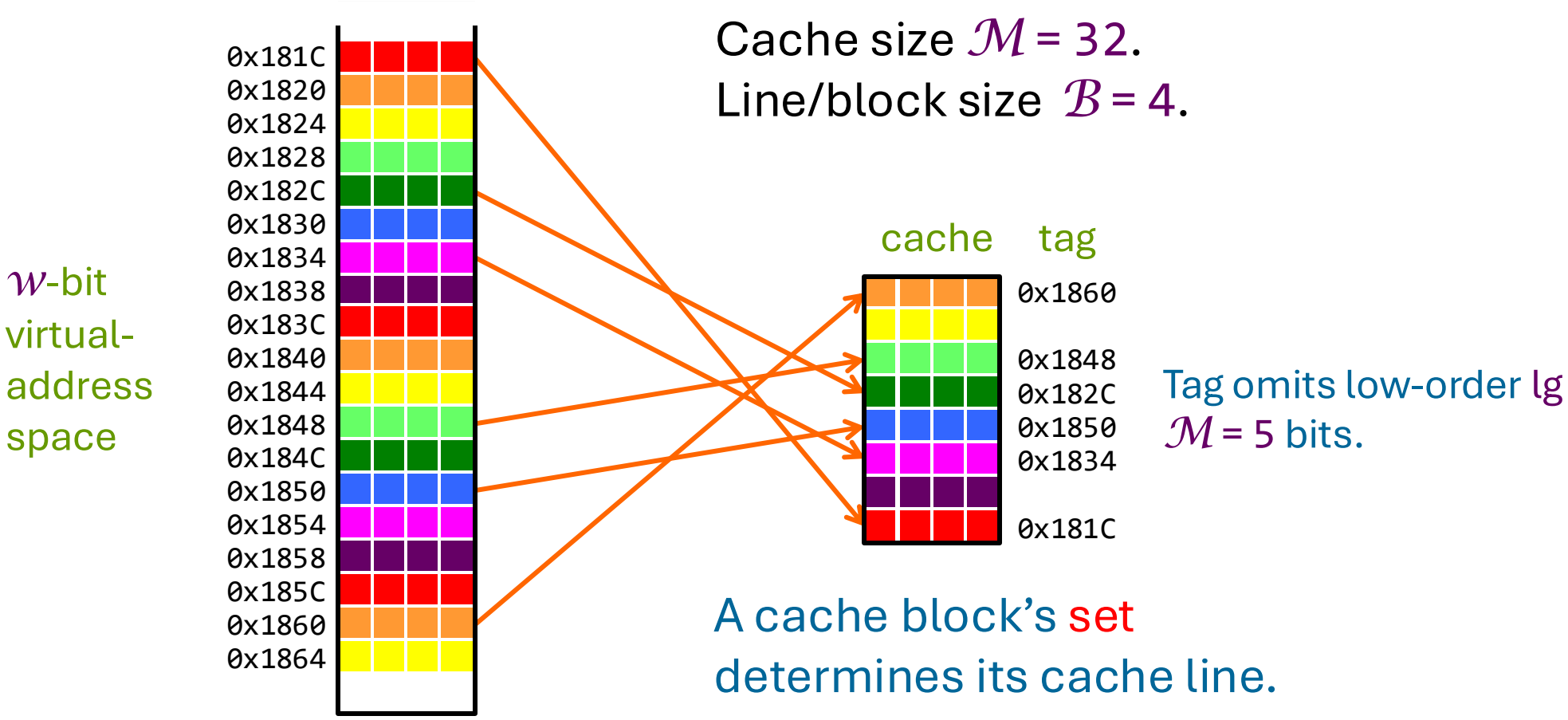


byte address

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

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

Direct-Mapped Cache

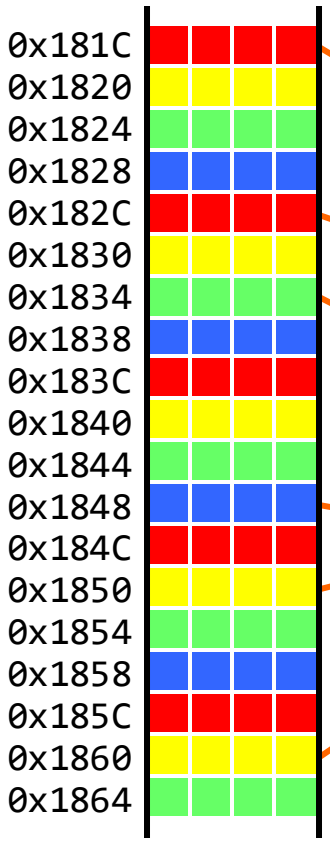


- To find a block in the cache, only a single line in the cache needs to be checked
- This scheme is good for spatial locality: e.g. stack frame or fields of small structs

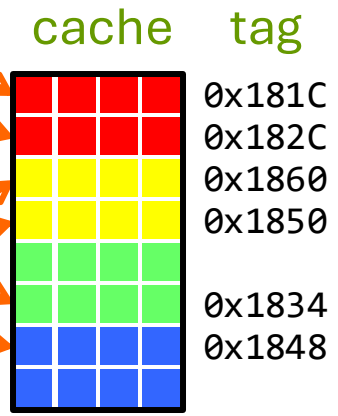
Set-Associative Cache

Multiple location for each set

w-bit virtual-address space



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



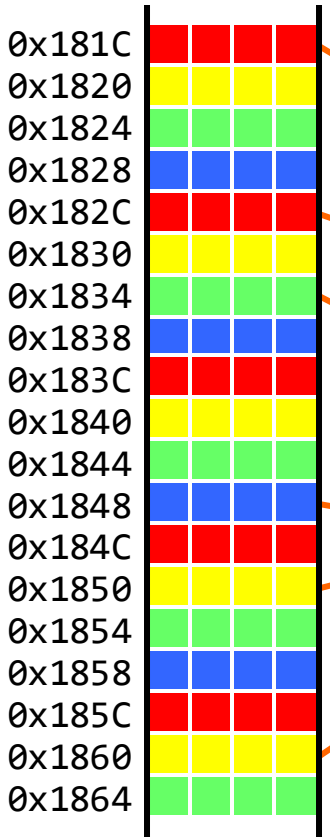
A cache block's set determines k cache lines.

[1] J. R. Diamond, D. S. Fussell and S. W. Keckler, "Arbitrary Modulus Indexing," *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Cambridge, 2014, pp. 140-152, doi: 10.1109/MICRO.2014.13.

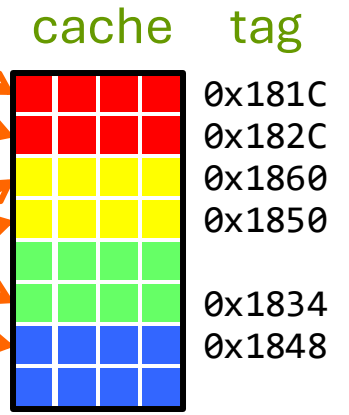
Set-Associative Cache

Multiple location for each set

w-bit virtual-address space



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

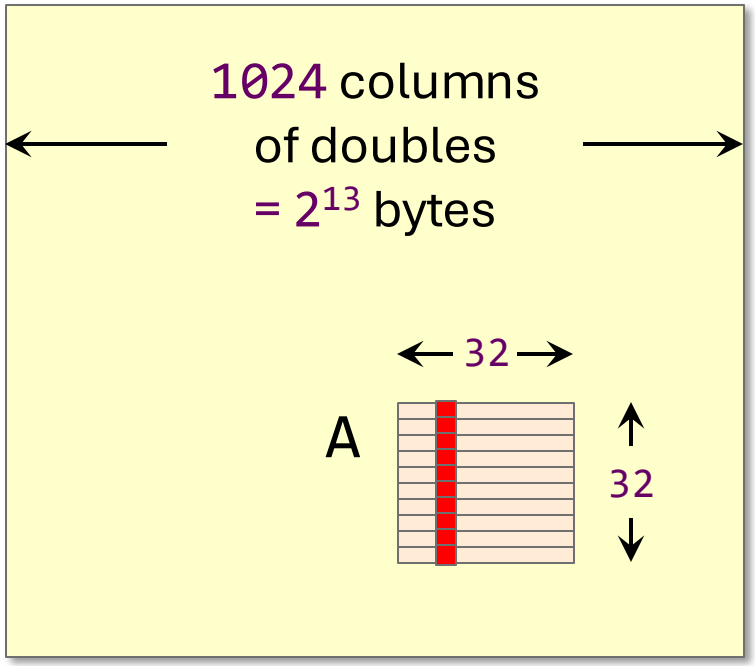
bits	tag	set	offset
	$w - \lg(\mathcal{M}/k)$	$\lg(\mathcal{M}/k\mathcal{B})$	$\lg \mathcal{B}$

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 $M = 32\text{K}$ bytes.
- line/block size $B = 64$ bytes.
- $k = 4$-way associativity.

Conflict misses can be problematic for caches with limited associativity.

byte address

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

Analysis

Look at a column of submatrix A.

The addresses of the elements are x, $x+2^{13}$, $x+2\cdot2^{13}$, …, $x+31\cdot2^{13}$.

They all fall into the same set!

Solutions

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

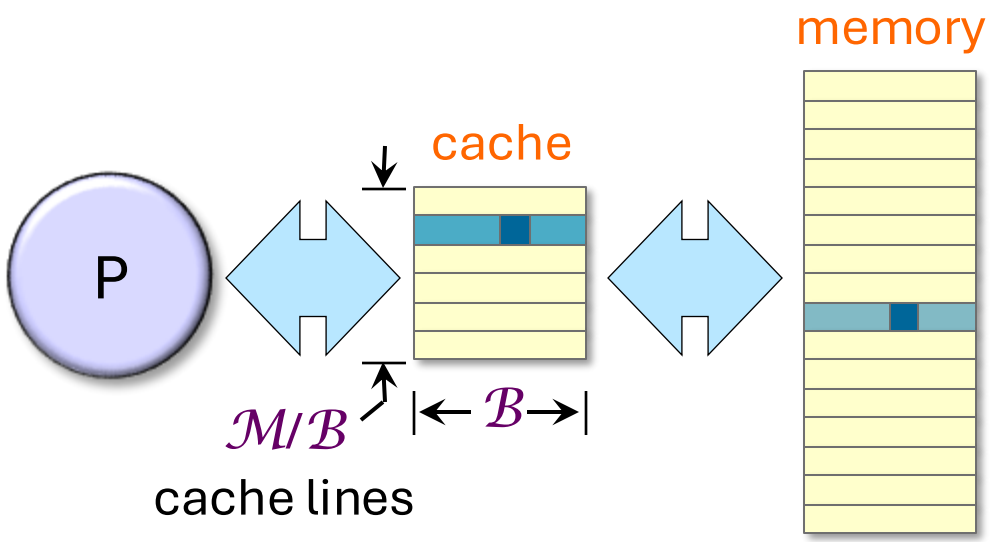
Outline

- Cache Hardware
- **Ideal Cache Model**
- Cache Analysis of Matrix Multiplication
- Tiling
- Divide and Conquer

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 & stores have different costs w.r.t bandwidth and latency

Segment Caching Lemma

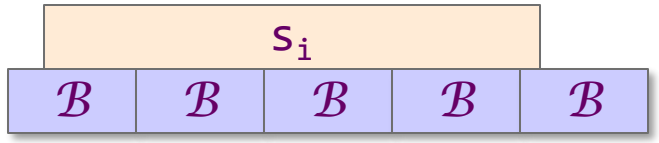
Lemma. Suppose that a program reads a set of r data segments, where the ith segment consists of s_i contiguous bytes in memory, and suppose that

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

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

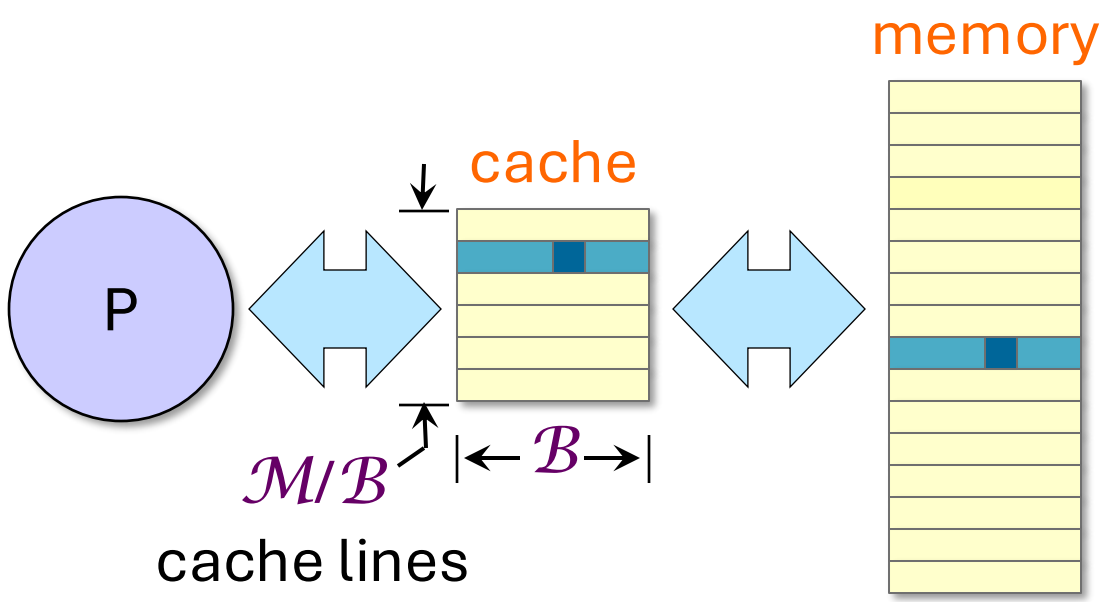
Proof. A single segment s_i incurs at most $s_i/\mathcal{B} + 2$ misses, and hence we have

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



Spatial Locality

Tall Caches



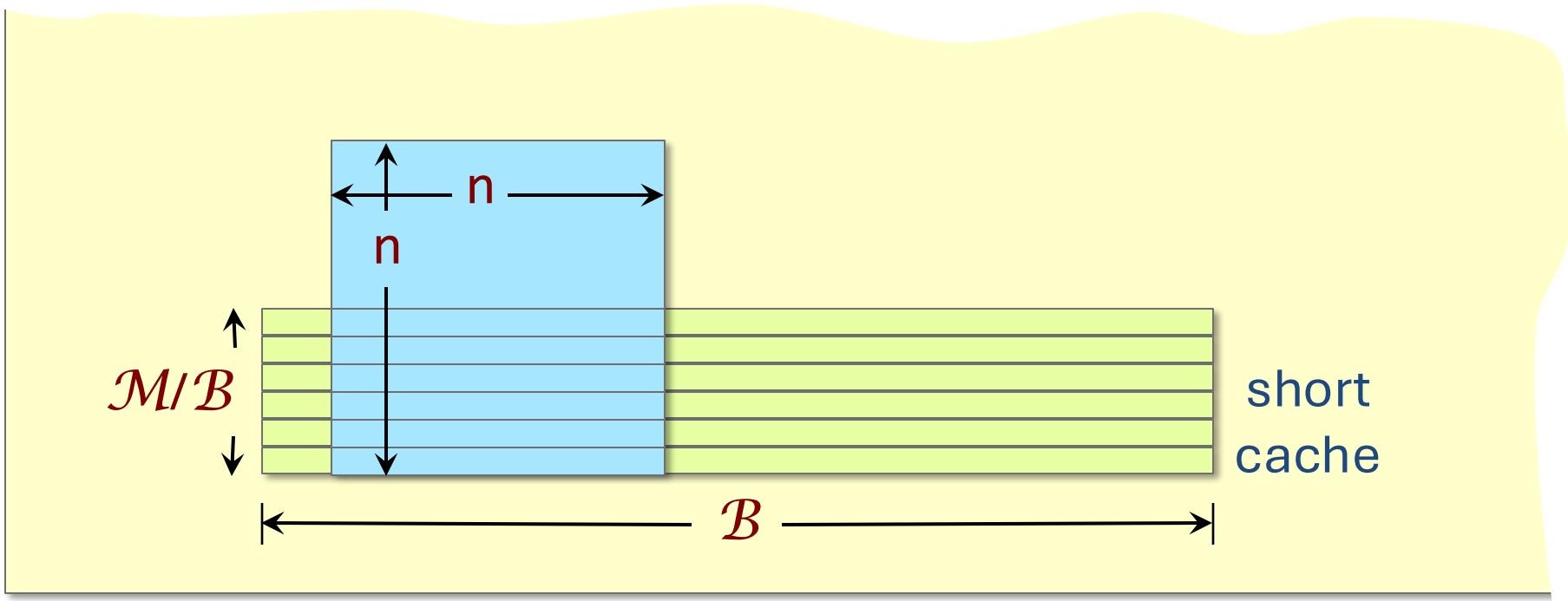
Tall-cache assumption
$\mathcal{B}^2 < c\,\mathcal{M}$ for some sufficiently small constant $c \leq 1$.

Example: Intel Xeon Platinum 8280L

- Cache-line length $\mathcal{B}$ = 64 bytes. 2^6
- L1-cache size $\mathcal{M}$ = 32 kibibytes. 2^{15}

$$\mathcal{B}^2 = \frac{1}{8} \times \mathcal{M}$$

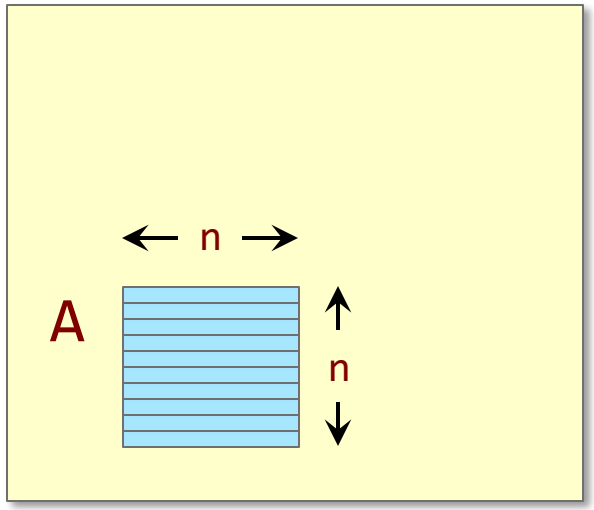
What's Wrong with Short Caches?



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

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

Outline

- Cache Hardware
- Ideal Cache Model
- **Cache Analysis of Matrix Multiplication**
- Tiling
- Divide and Conquer

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

$T(n) = ?$

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];
}
```

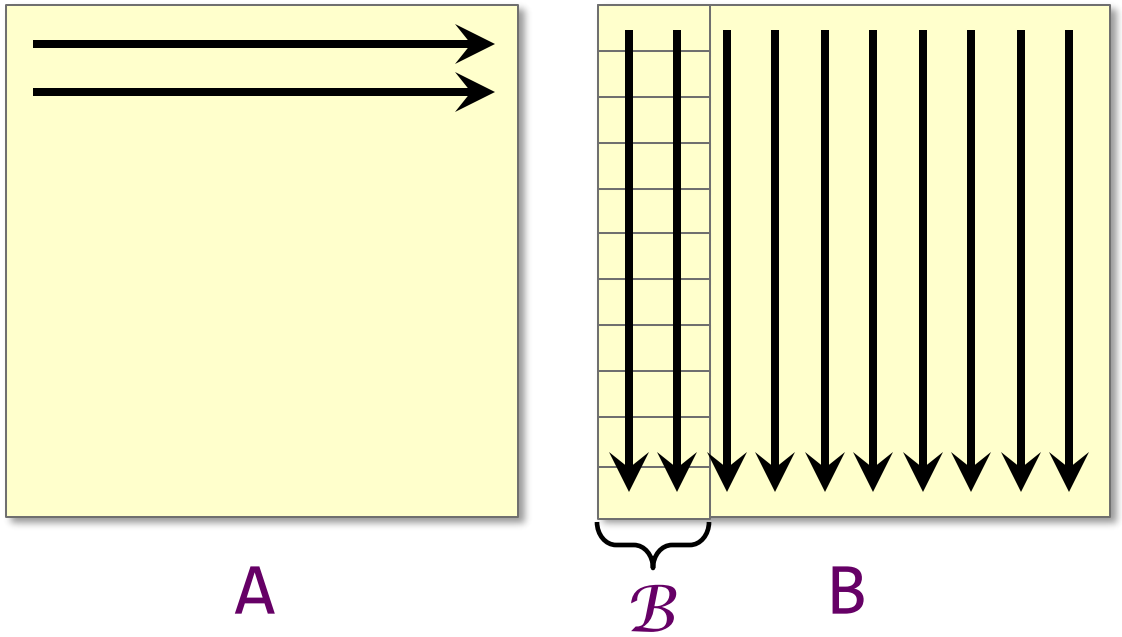
Analysis of work

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

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.

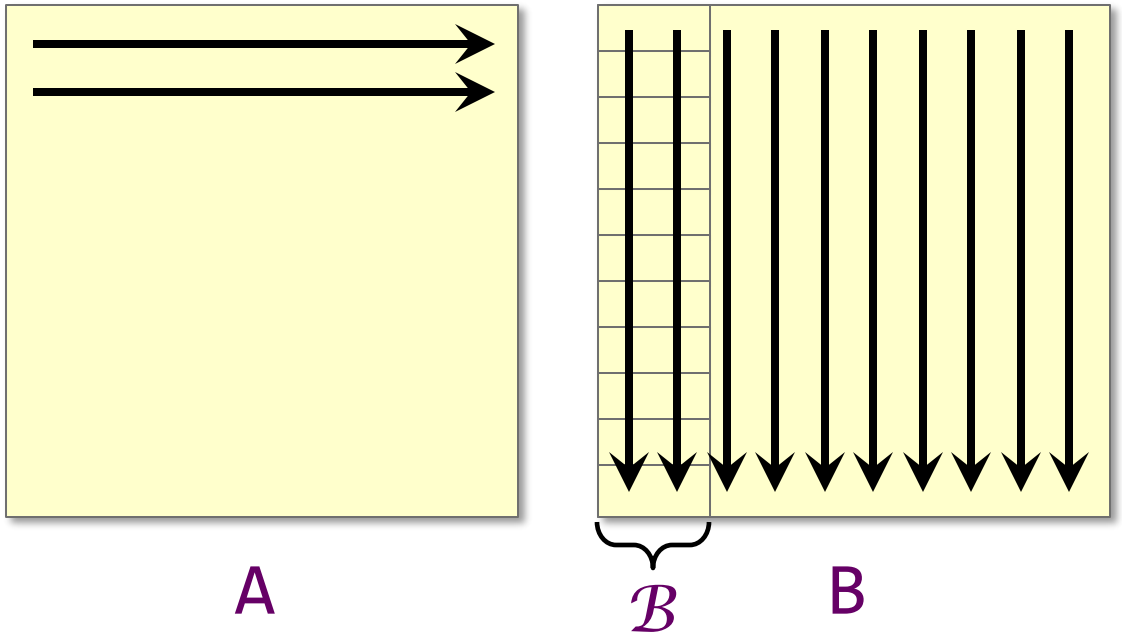
Assume LRU.

$Q(n) = \Theta(n^3)$, 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.

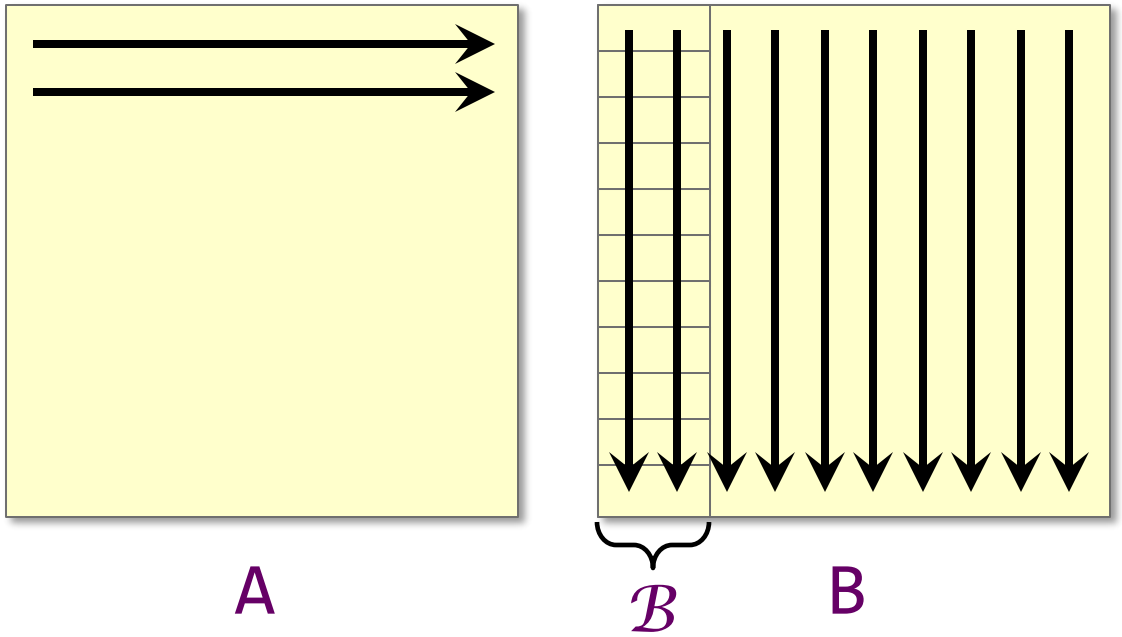
$Q(n) = n \cdot \Theta(n^2/\mathcal{B}) = \Theta(n^3/\mathcal{B})$, since matrix B can exploit spatial locality.

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

$n < c\mathcal{M}^{1/2}$, $c \le 1/3$.

Analyze matrix B.

Assume LRU.

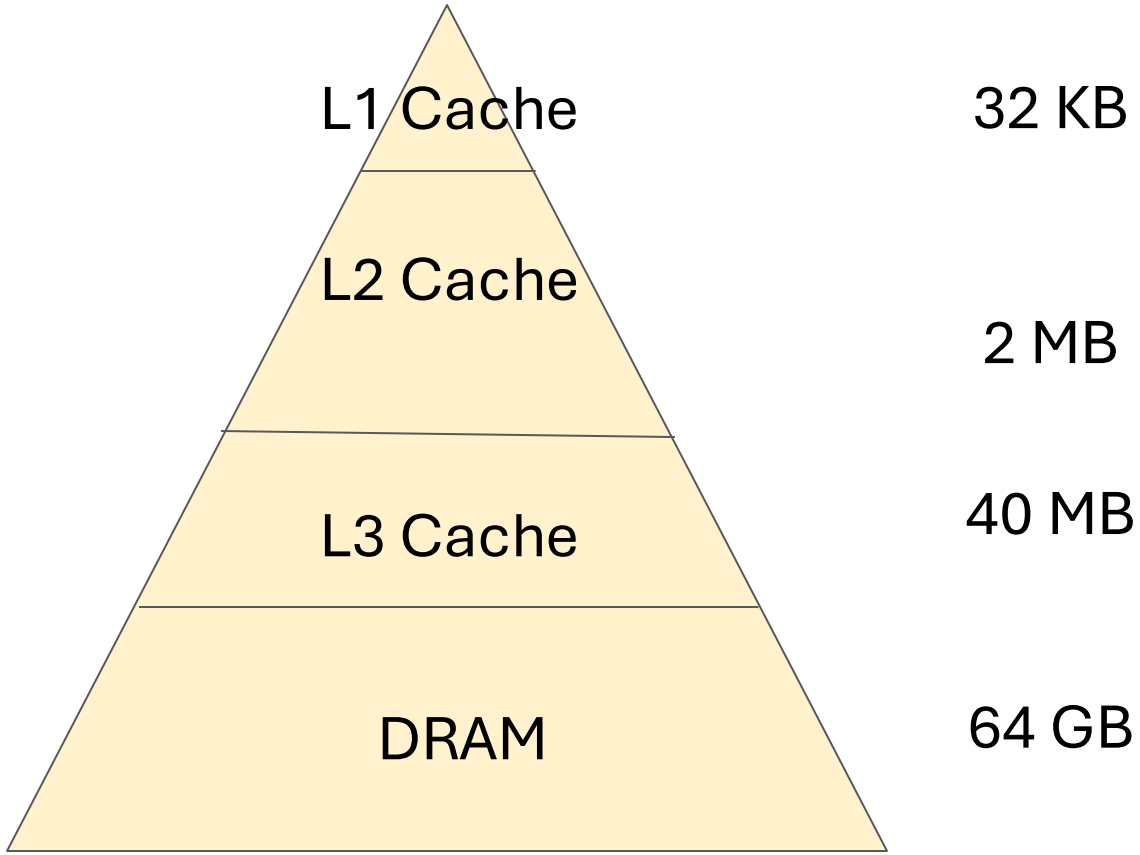
$Q(n) = \Theta(n^2/\mathcal{B})$, by the submatrix caching lemma.

Outline

- Cache Hardware
- Ideal Cache Model
- Cache Analysis of Matrix Multiplication
- **Tiling**
- Divide and Conquer

Tiling

- Split work/data into **tiles** to fit into closer memory to exploit data reuse, i.e., **locality**

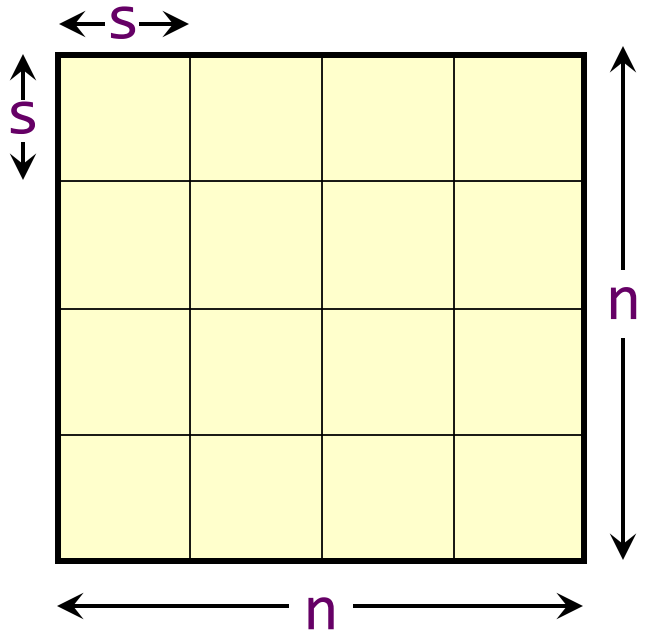


Tiled Matrix Multiplication

```
void Tiled_Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i1=0; i1<n/s; i1+=s)
    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++)
          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];
}
```

outer nest

inner nest



Analysis of work

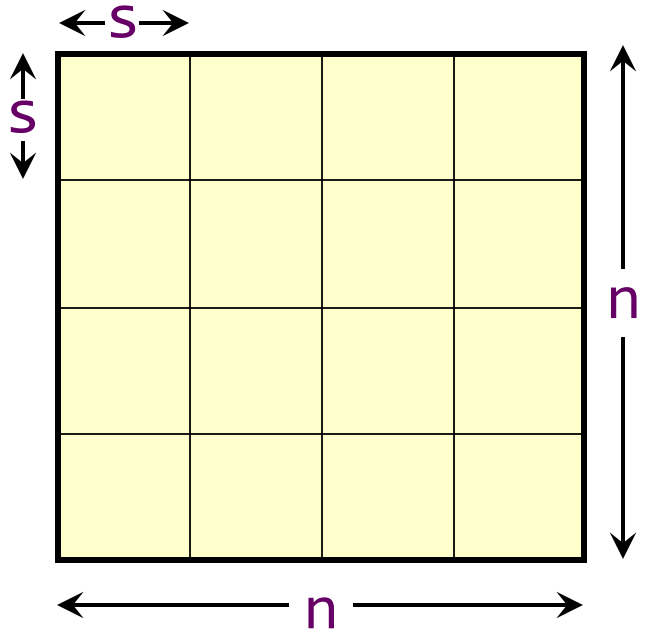
- Work $T(n) = \Theta((n/s)^3(s^3))$
  $= \Theta(n^3)$.

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)
    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++)
          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];
}
```

outer nest

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].

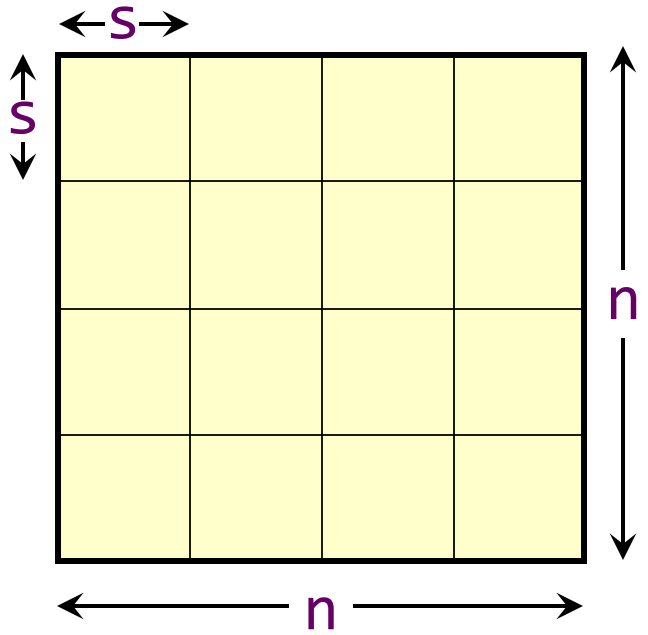
Spatial Locality

Temporal Locality

Tiled Matrix Multiplication

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

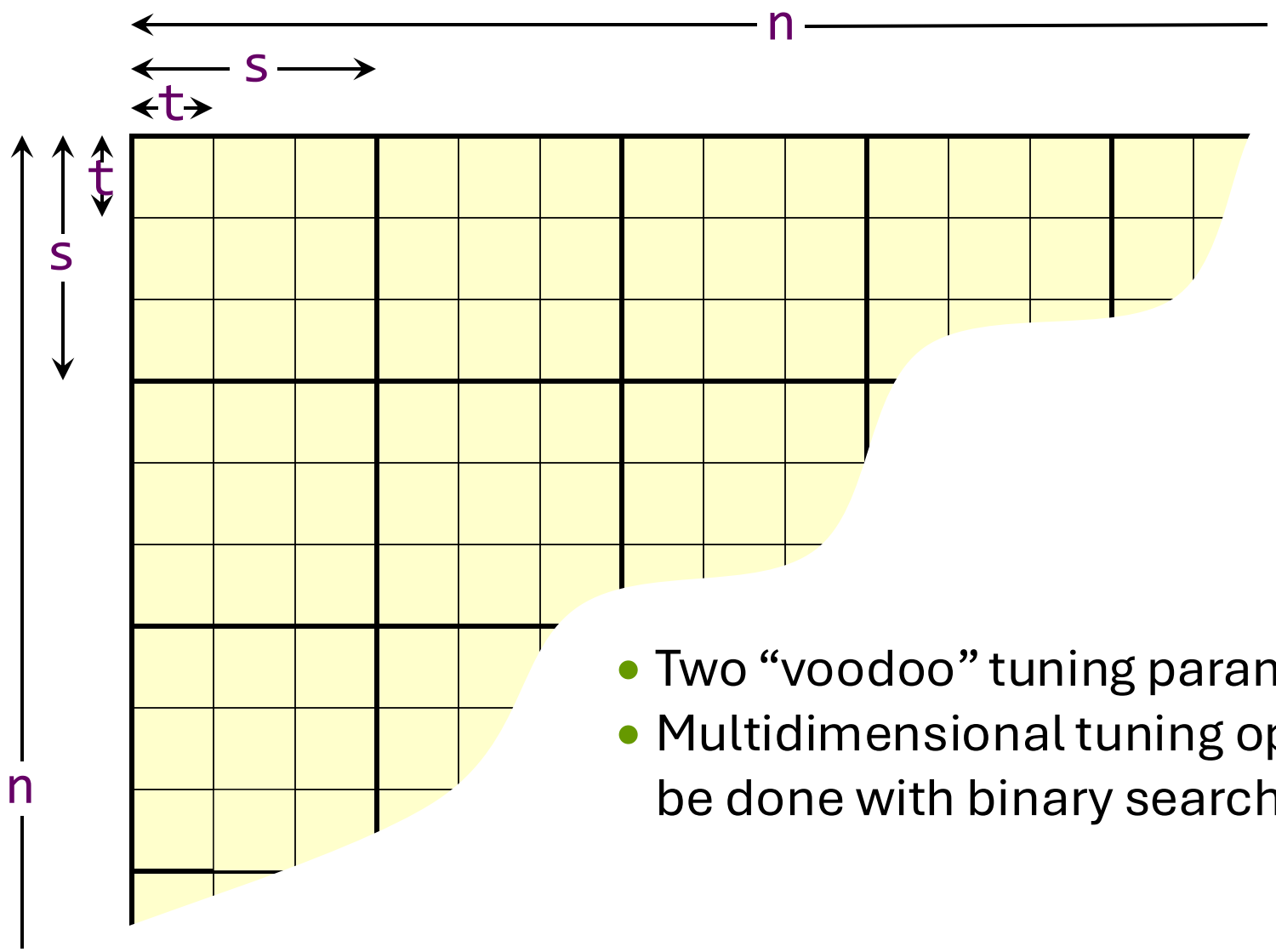
Voodoo!



Analysis of cache misses

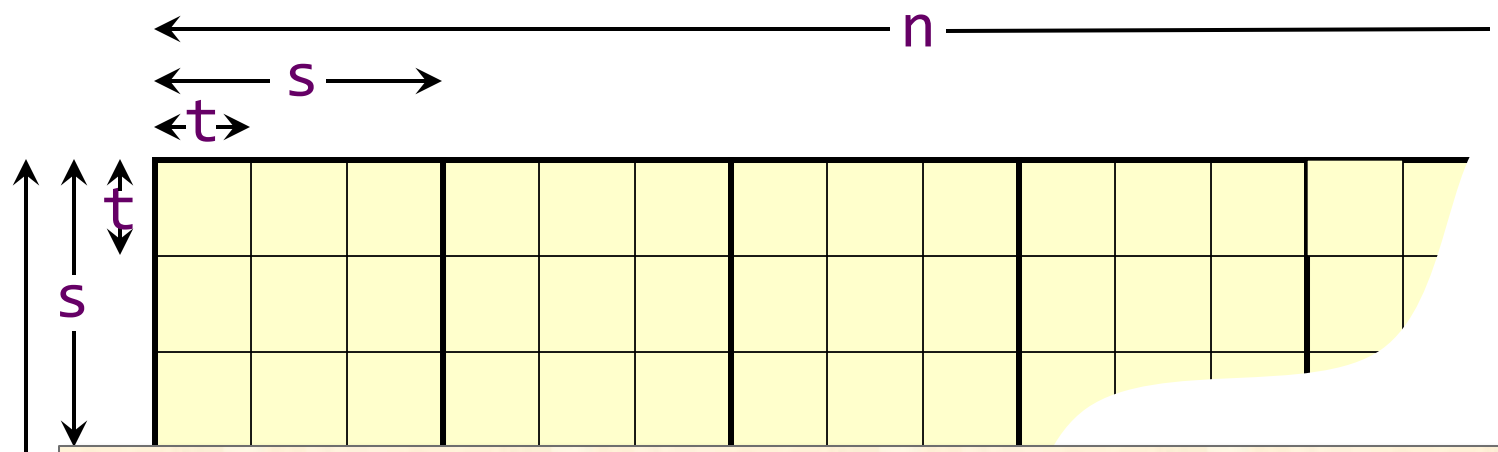
- Tune s so that the tiles just fit into cache $\Rightarrow s = \Theta(\mathcal{M}^{1/2})$.
- Submatrix caching lemma implies $\Theta(s^2/\mathcal{B})$ misses per tile.
- $Q(n) = \Theta((n/s)^3(s^2/\mathcal{B}))$
  $= \Theta(n^3/\mathcal{B}\mathcal{M}^{1/2})$.
- Optimal [HK81].

Two-Level Cache



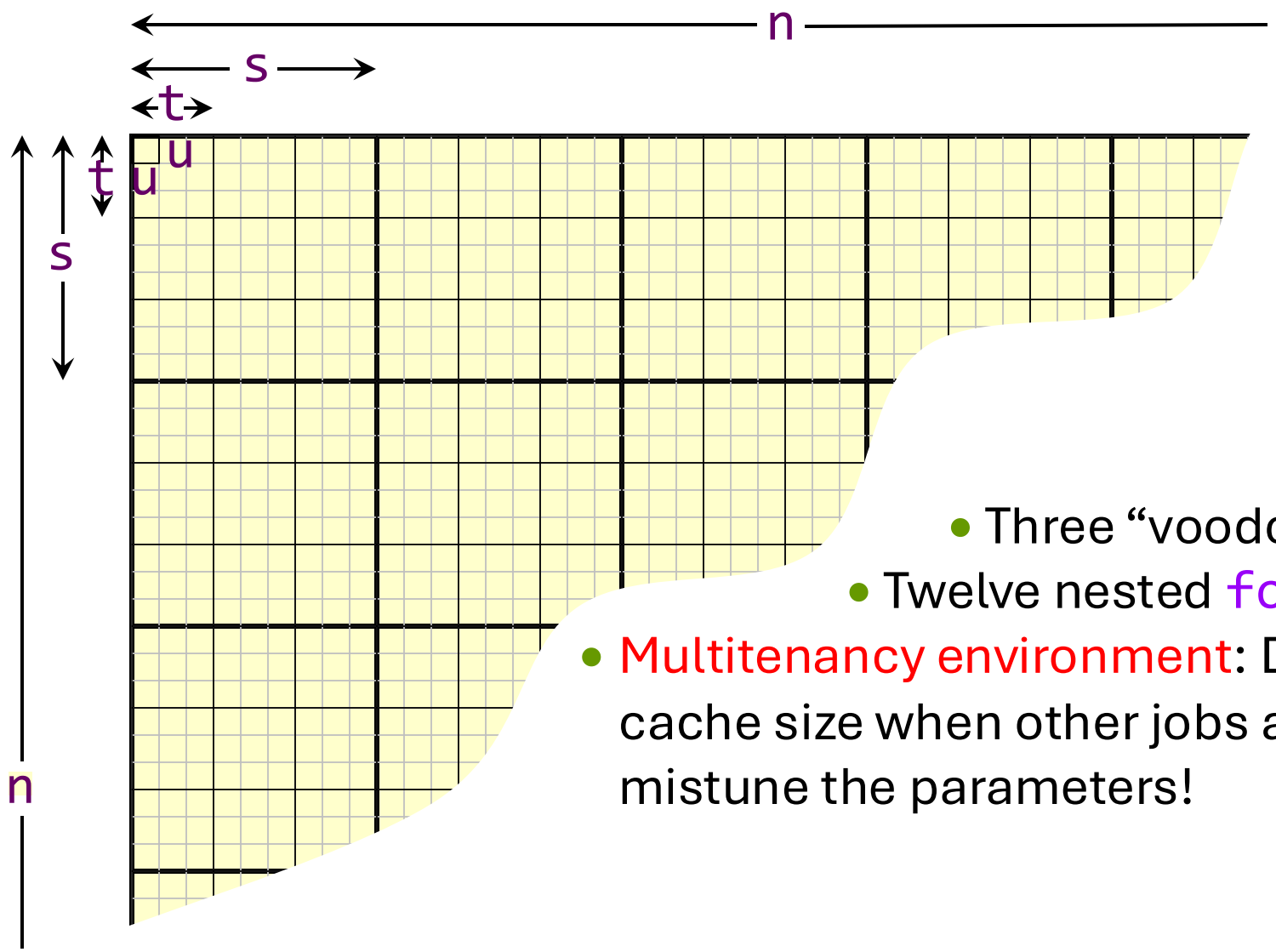
- Two “voodoo” tuning parameters `s` and `t`.
- Multidimensional tuning optimization cannot be done with binary search.

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!

Outline

- Cache Hardware
- Ideal Cache Model
- Cache Analysis of Matrix Multiplication
- Tiling
- **Divide and 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.

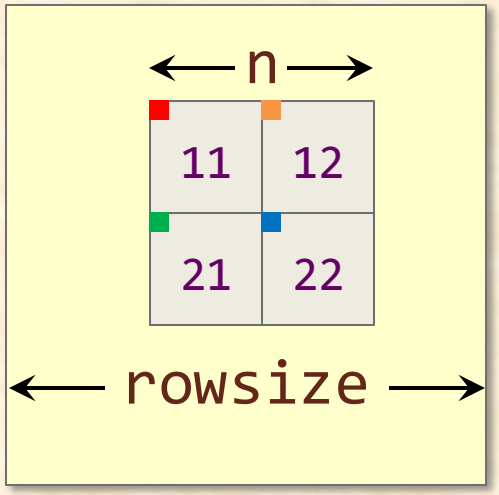
Recursive Code

```
// 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);
  }
}
```

$$T(n) = 8\,T(n/2) + \Theta(1) = \Theta(n^3)$$

Analysis of Work

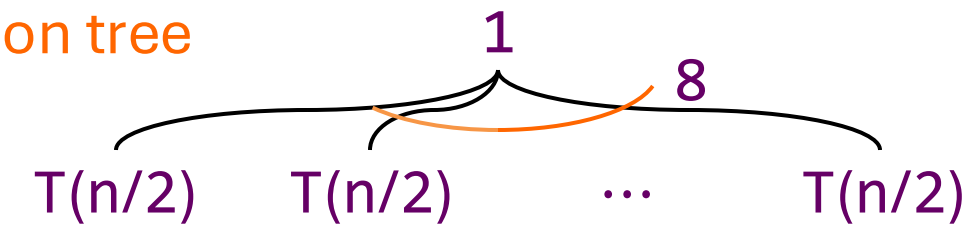
$$T(n) = 8\,T(n/2) + 1$$

recursion tree $T(n)$

Analysis of Work

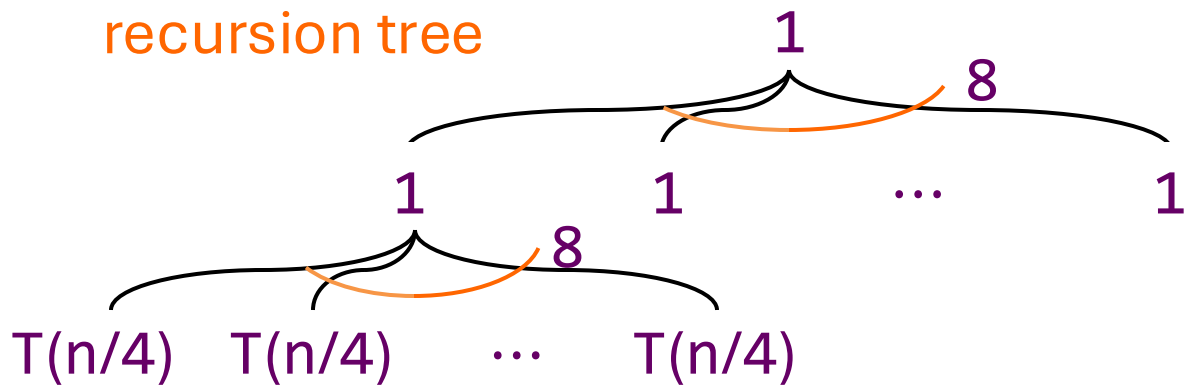
$$T(n) = 8\,T(n/2) + 1$$

recursion tree



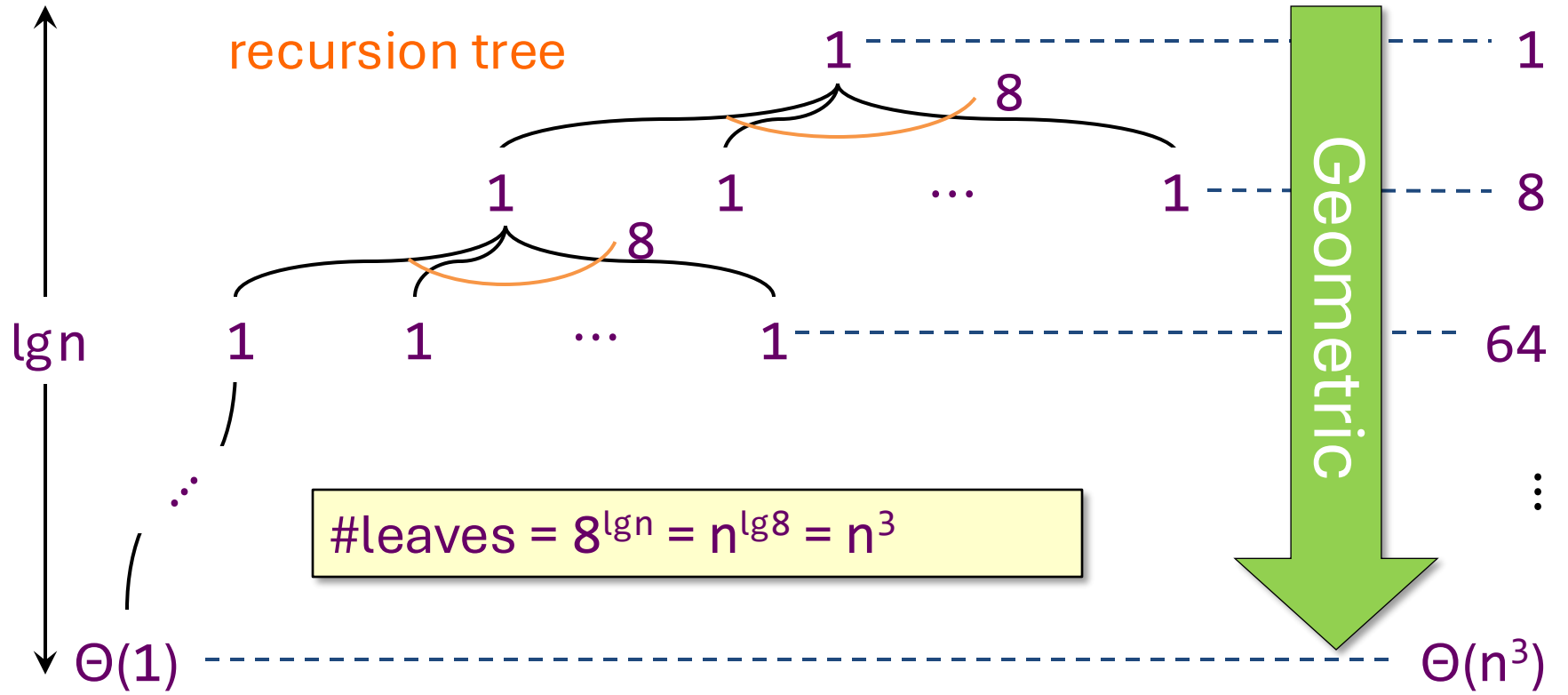
Analysis of Work

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



Analysis of Work

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



Note: Same work as looping versions.

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

Analysis of Cache Misses

```
// 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

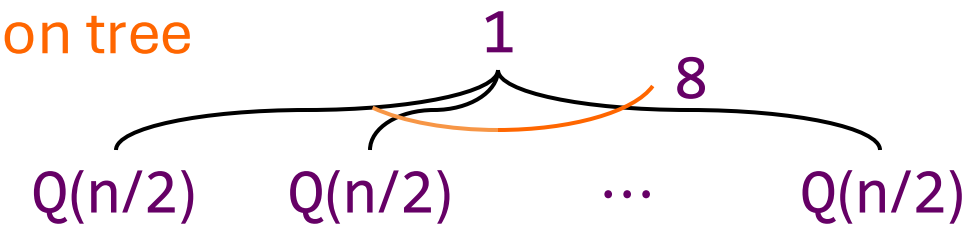
$$Q(n) = \begin{cases} \Theta(n^2/\mathcal{B}) & \text{if } n^2 < c\mathcal{M} \text{ for suff. small const } c \le 1, \\ 8\,Q(n/2) + 1 & \text{otherwise.} \end{cases}$$

recursion tree Q(n)

Analysis of Cache Misses

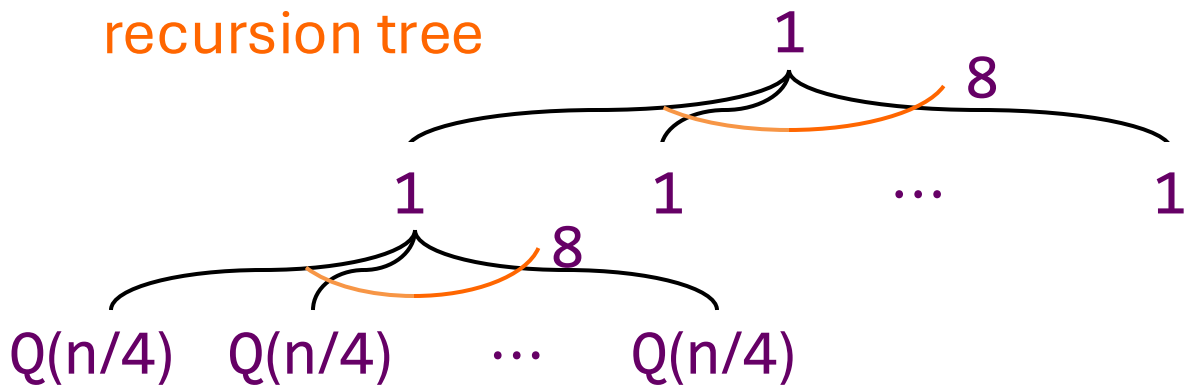
$$Q(n) = \begin{cases} \Theta(n^2/\mathcal{B}) & \text{if } n^2 < c\mathcal{M} \text{ for suff. small const } c \le 1, \\ 8\,Q(n/2) + 1 & \text{otherwise.} \end{cases}$$

recursion tree



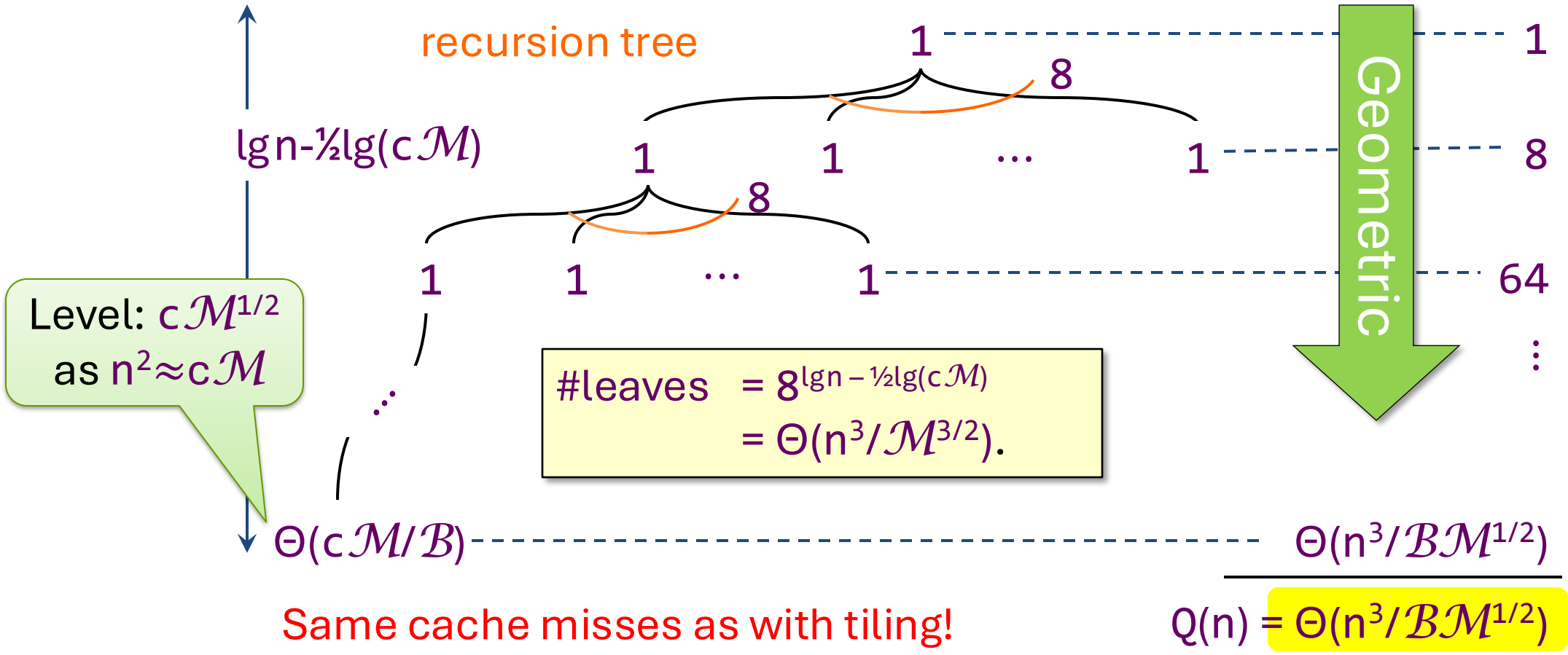
Analysis of Cache Misses

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



Analysis of Cache Misses

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



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.

Matrix multiplication

The best cache-oblivious matrix-multiplication codes to date work on arbitrary rectangular matrices in parallel and perform binary splitting on the largest dimension of A and B

Recursive Parallel Matrix Multiply

```
void mm_dac2(double *restrict C, int n_C,
             double *restrict A, int n_A,
             double *restrict B, int n_B,
             int n)
{ // C += A * B
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
#define X(M,row,col) (M + (row*(n_ ## M) + col)*(n/2))
    cilk_scope {
      cilk_spawn mm_dac2(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
      cilk_spawn mm_dac2(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
      cilk_spawn mm_dac2(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
      cilk_spawn mm_dac2(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    }
    cilk_scope {
      cilk_spawn mm_dac2(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
      cilk_spawn mm_dac2(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
      cilk_spawn mm_dac2(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
      cilk_spawn mm_dac2(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    }
  }
}
```

Cilk and Caching

Theorem. Let Q_P be the number of cache misses in a deterministic Cilk computation when run on P processors, each with a private cache of size $\mathcal{M}$. In the ideal-cache model, we have

$$Q_P = Q_1 + O(S_P \mathcal{M}/\mathcal{B}) ,$$

where $\mathcal{M}$ is the cache size, $\mathcal{B}$ is the cache-block size, and S_P is the number of processor-processor migrations (steals) during the computation

Proof. After a worker steals a continuation, its cache is completely cold in the worst case. But after $\mathcal{M}/\mathcal{B}$ (cold) cache misses, its cache is identical to that in the serial execution. The same is true when a worker resumes a stolen subcomputation after exiting a `cilk_scope`. The number of times these two situations can occur is at most $2S_P$. ■

IN PRACTICE: Minimizing cache misses in the serial projection tends to minimize them in parallel executions.

Take-Aways

- Use the ideal cache model to analyze overall cache behavior
  - further engineering may be required to handle set-associative caches
- The segment caching lemma can simplify the cache analysis
- Cache-oblivious algorithms can use cache as well as cache-aware algorithms without resorting to voodoo parameters
- Truncated recursion trees help to analyze the cache behavior of divide-and-conquer algorithms
- To minimize cache misses in Cilk programs, it usually suffices to minimize them in the serial projection