CS260 – Lecture 1

Yan Gu

Algorithm Engineering (aka. How to Write Fast Code)

I/O (Cache) Efficiency

Many slides in this lecture are borrowed from Lecture 14 in 6.172 Performance Engineering of Software Systems at MIT. The credit is to Prof. Charles E. Leiserson, and the instructor appreciates the permission to use them in this course.

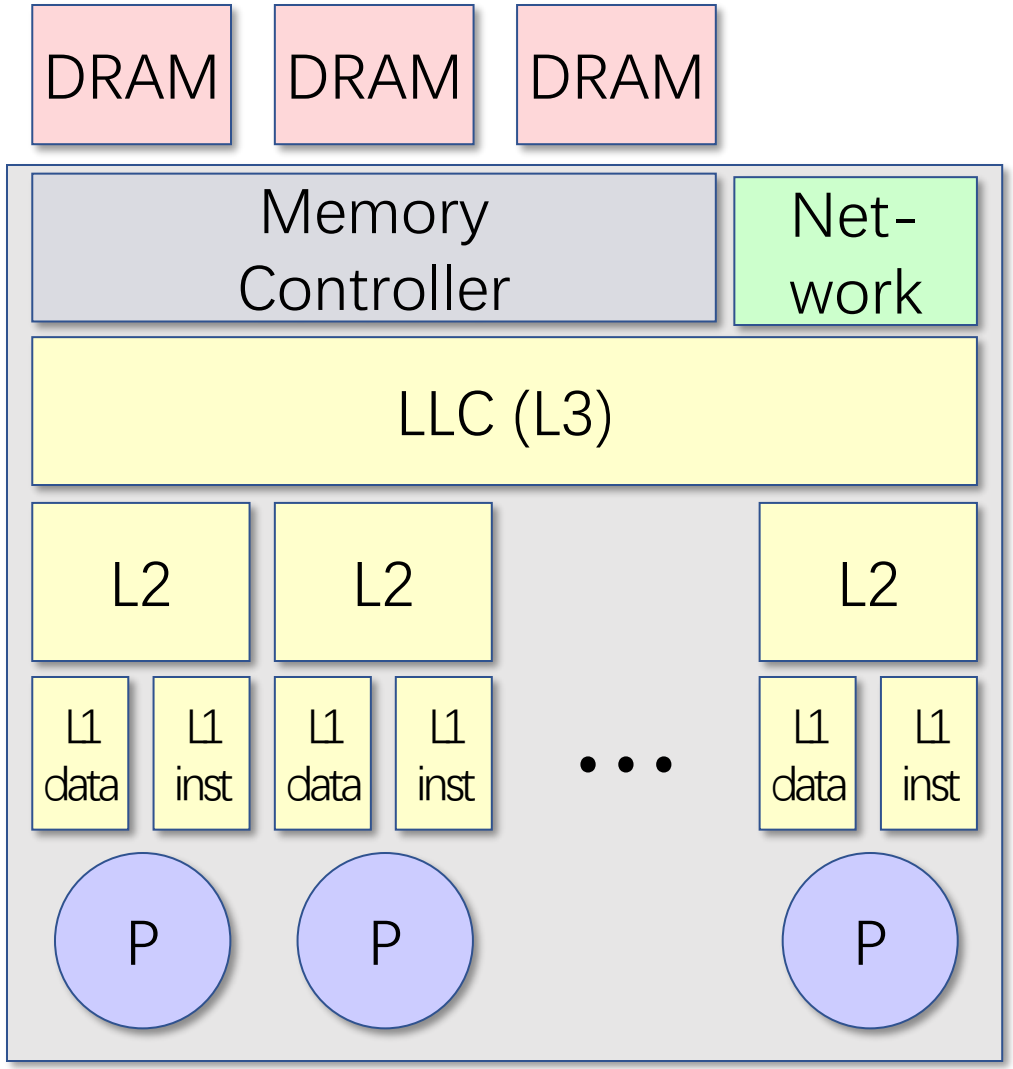
CS260: Algorithm Engineering Lecture 1

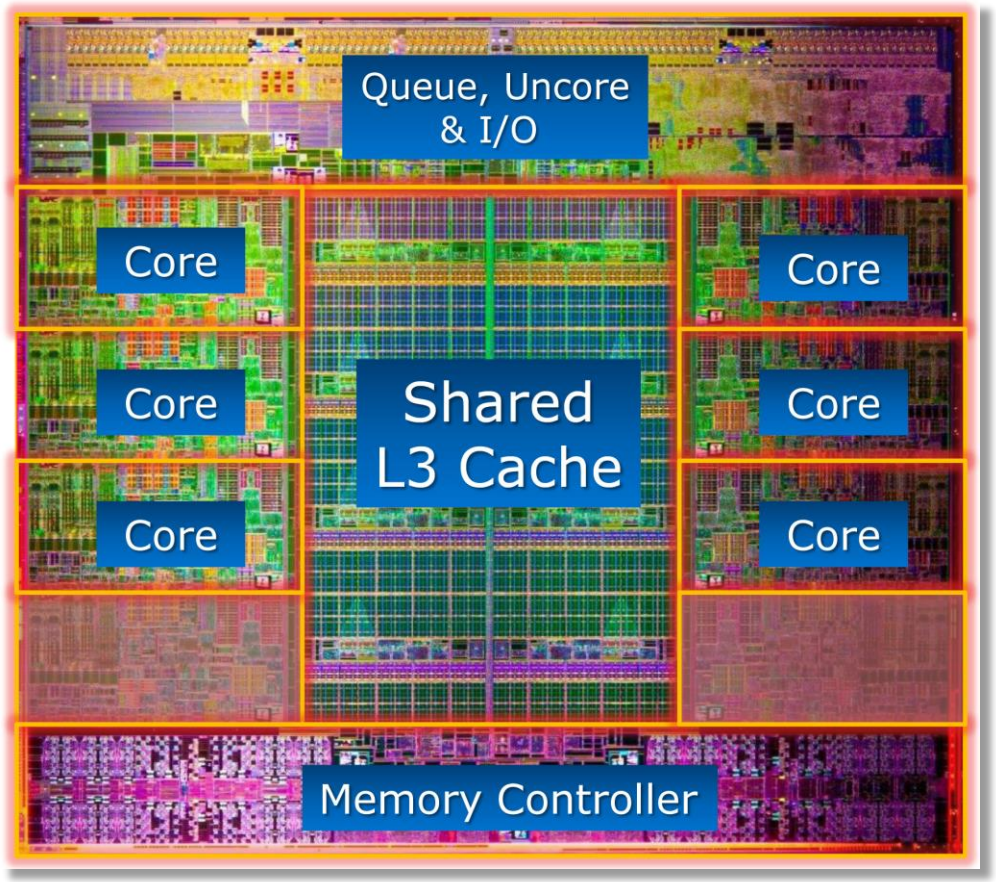
Cache Hardware

The I/O model

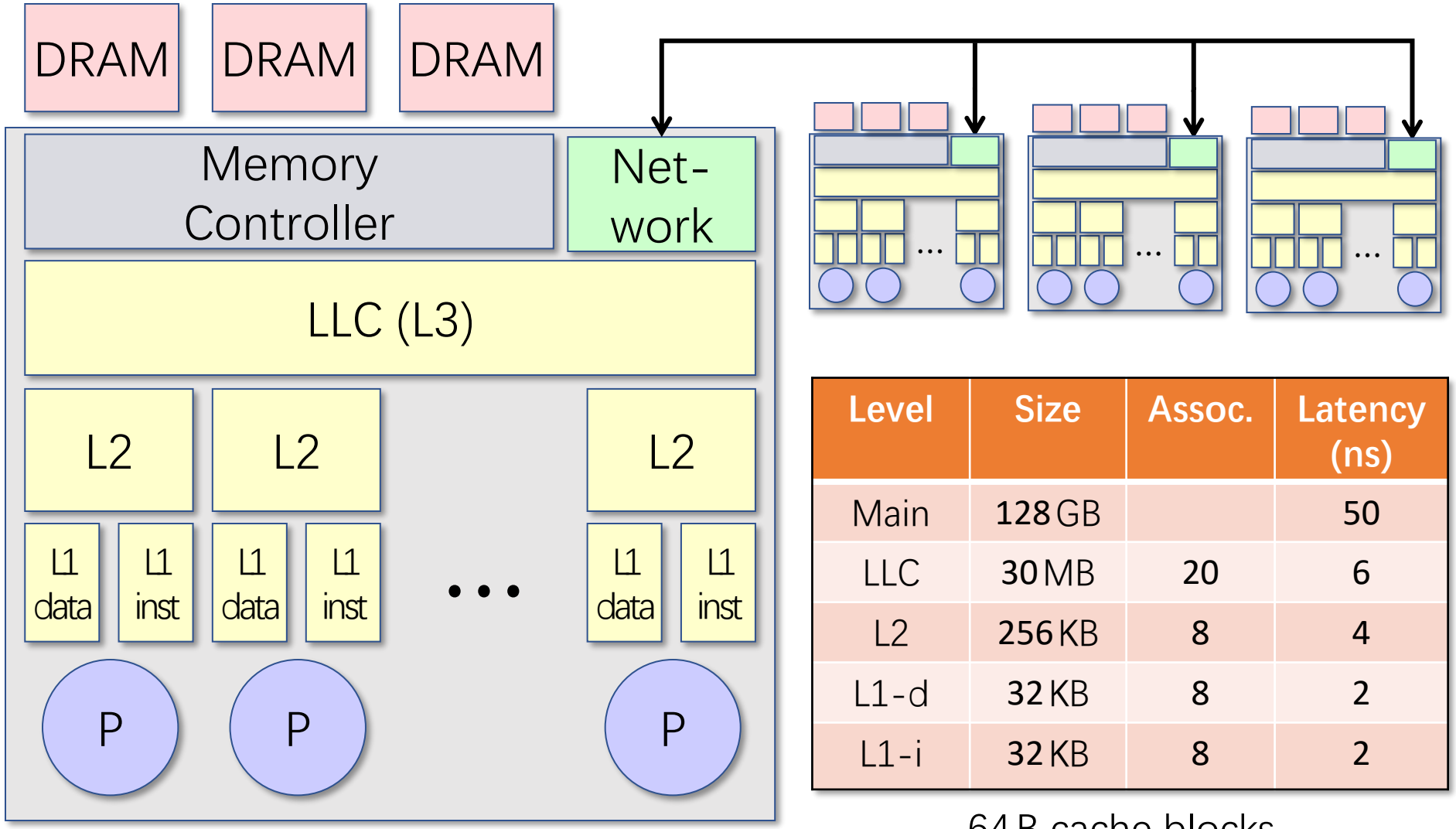
Revisit of matrix multiplication and I/O analysis

Multicore Cache Hierarchy





Multicore Cache Hierarchy

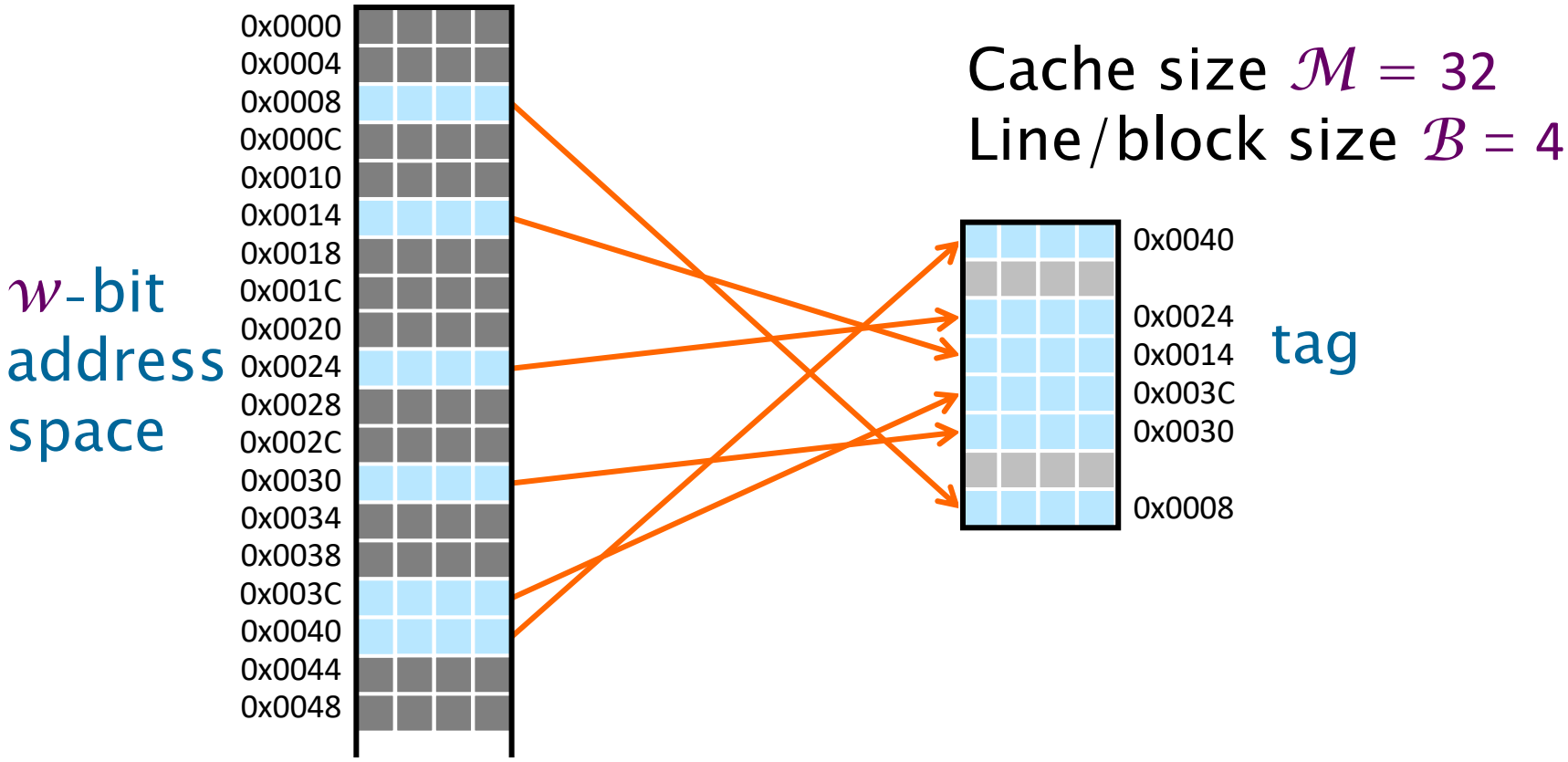


Level	Size	Assoc.	Latency (ns)
Main	128 GB		50
LLC	30 MB	20	6
L2	256 KB	8	4
L1-d	32 KB	8	2
L1-i	32 KB	8	2

64 B cache blocks

Fully Associative Cache

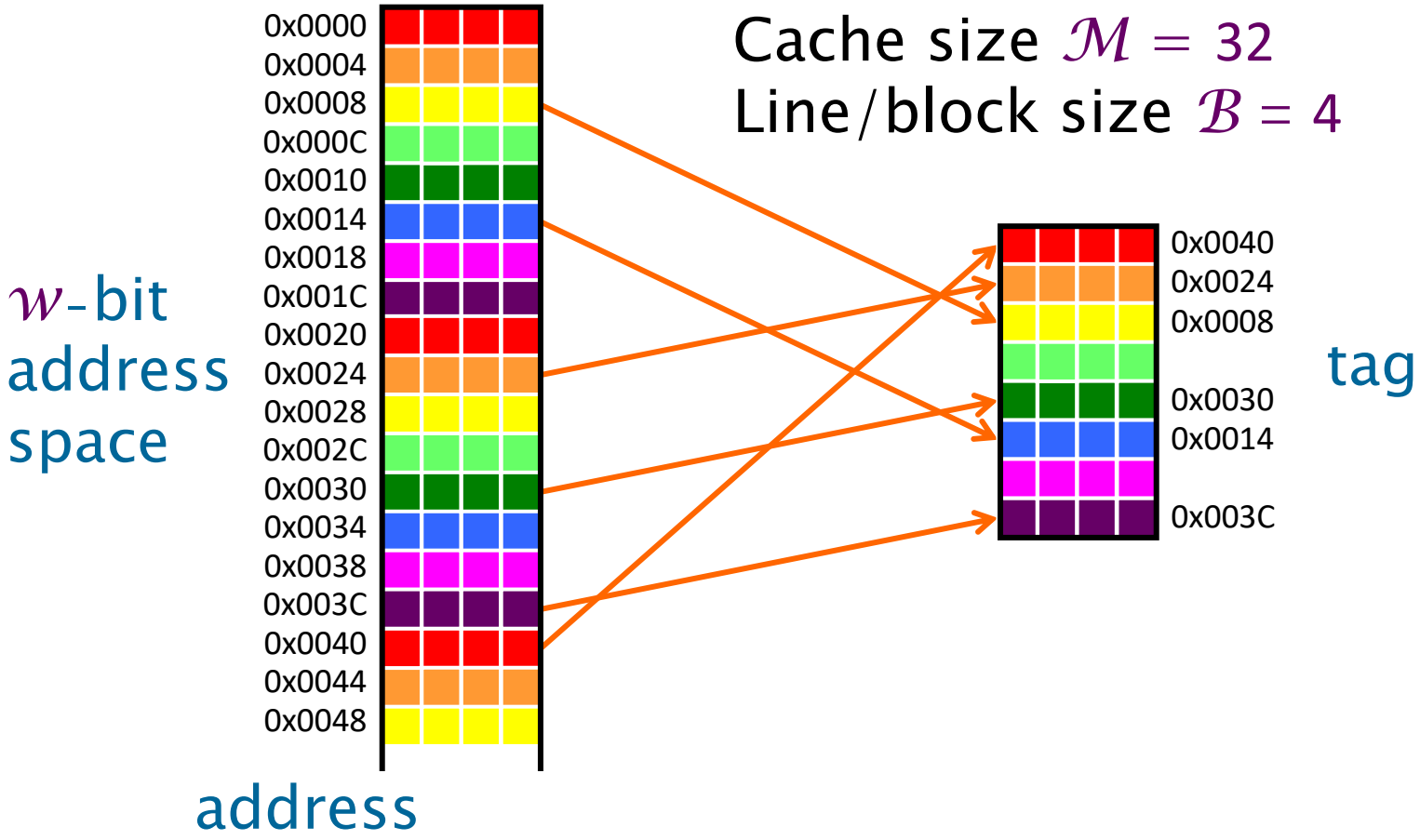
A cache block can reside anywhere in the cache



- To find a block in the cache, the entire cache must be searched for the tag
- When the cache becomes full, a block must be evicted for a new block
- The replacement policy determines which block to evict

Direct-Mapped Cache

A cache block's set determines its location in the cache

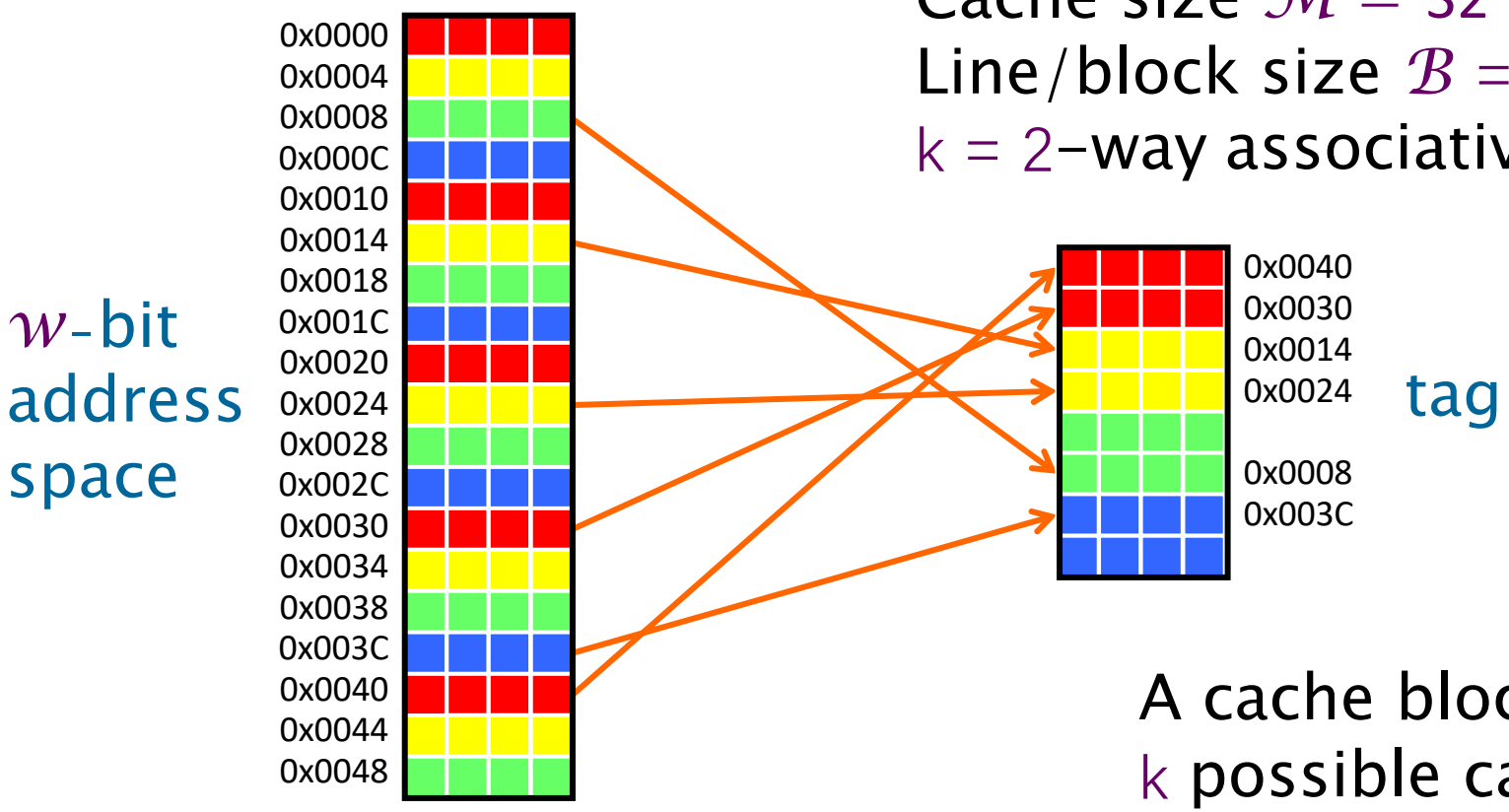


address

bits	tag	set	offset
	$w - \lg \mathcal{M}$	$\lg(\mathcal{M}/\mathcal{B})$	$\lg \mathcal{B}$
	61	3	2

To find a block in the cache, only a single location in the cache need be searched

Set-Associative Cache



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

A cache block's set determines k possible cache locations

address

	tag	set	offset
bits	$w - \lg(\mathcal{M}/k)$	$\lg(\mathcal{M}/k\mathcal{B})$	$\lg \mathcal{B}$
	62	2	2

To find a block in the cache, only the k locations of its set must 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**
  - Another p ock
  - True-shar me data on the cache line
  - False-sharing miss: The two processors are accessing different data that happen to reside on the same cache line

```
int x, y;
in-parallel:
   for (int i=0; i<10000; i++) x++;
   for (int j=0; j<10000; j++) y++;
```

CS260: Algorithm Engineering Lecture 1

Cache Hardware

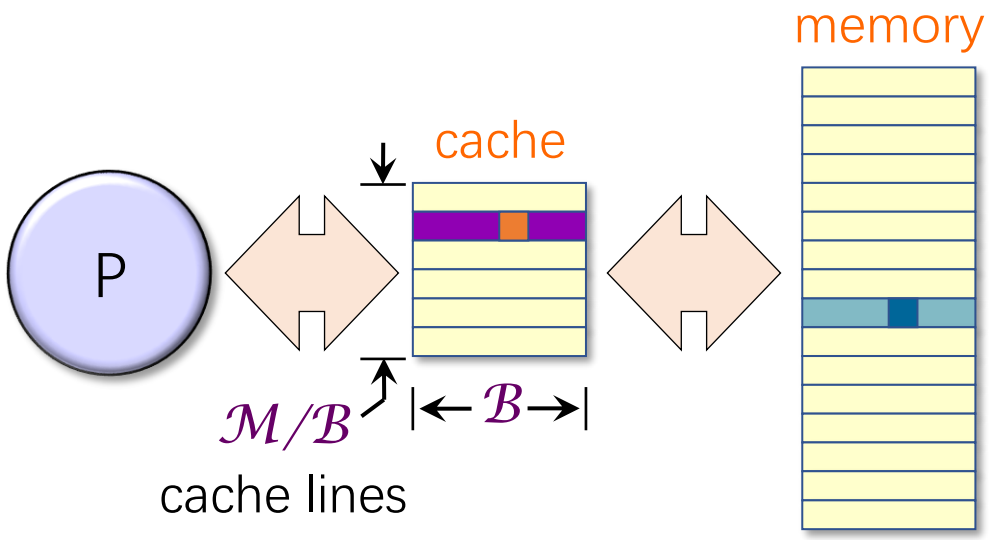
The I/O model

Revisit of matrix multiplication and I/O analysis

I/O Model (External Memory-, Ideal Cache-)

Parameters

- Two-level hierarchy
- Cache size of M bytes
- Cache-line length of B bytes
- Fully associative
- Optimal, omniscient replacement



Performance Measures

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

How Reasonable to Assume Optimal Replacement?

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

Implication

For asymptotic analyses, one can assume optimal or LRU replacement, as convenient

Algorithm Engineering

- Design a theoretically good algorithm.
- Engineer for detailed performance.
  - Real caches are not fully associative.
  - Loads and stores have different costs with respect to bandwidth and latency.

CS260: Algorithm Engineering Lecture 1

Cache Hardware

The I/O model

Revisit of matrix multiplication and I/O analysis

Multiply Square Matrices

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

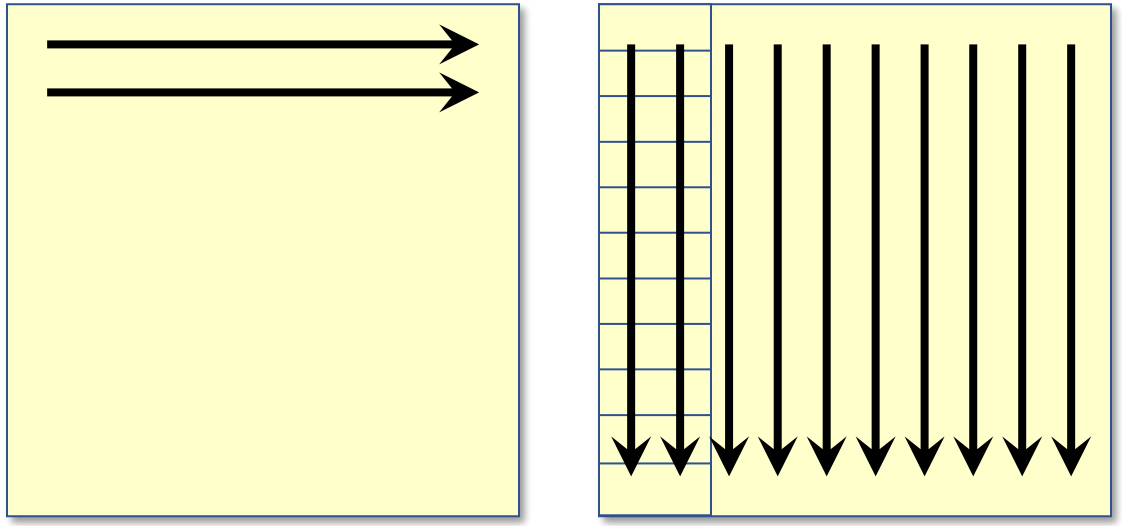
Analysis of work:

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

Analysis of Cache Misses

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

Assume row major and tall cache



A B

Case 1

$n > c\mathcal{M}/\mathcal{B}$.

Analyze matrix B.

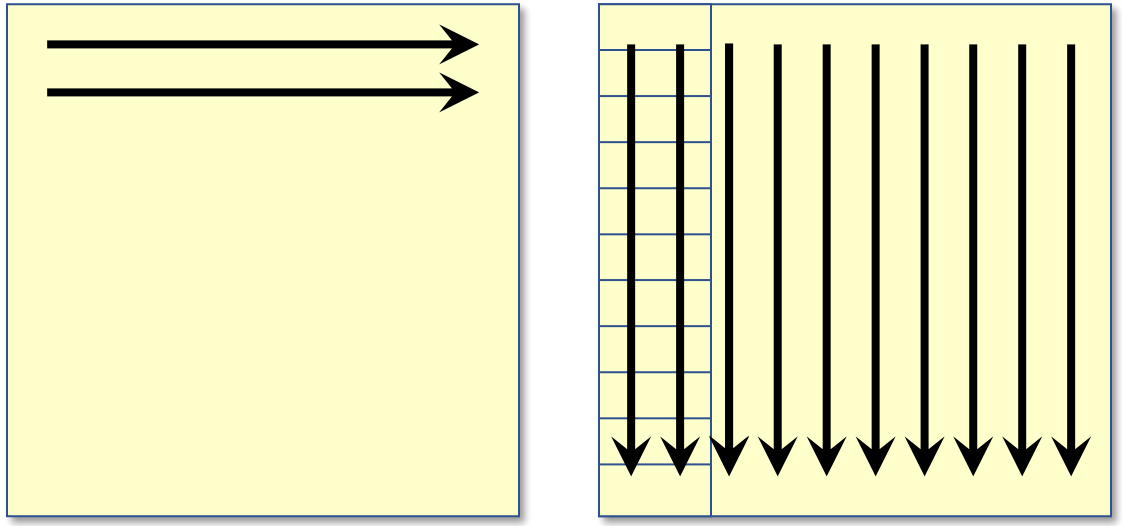
Assume LRU.

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

Analysis of Cache Misses

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

Assume row major and tall cache



A B

Case 2

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

Analyze matrix B.

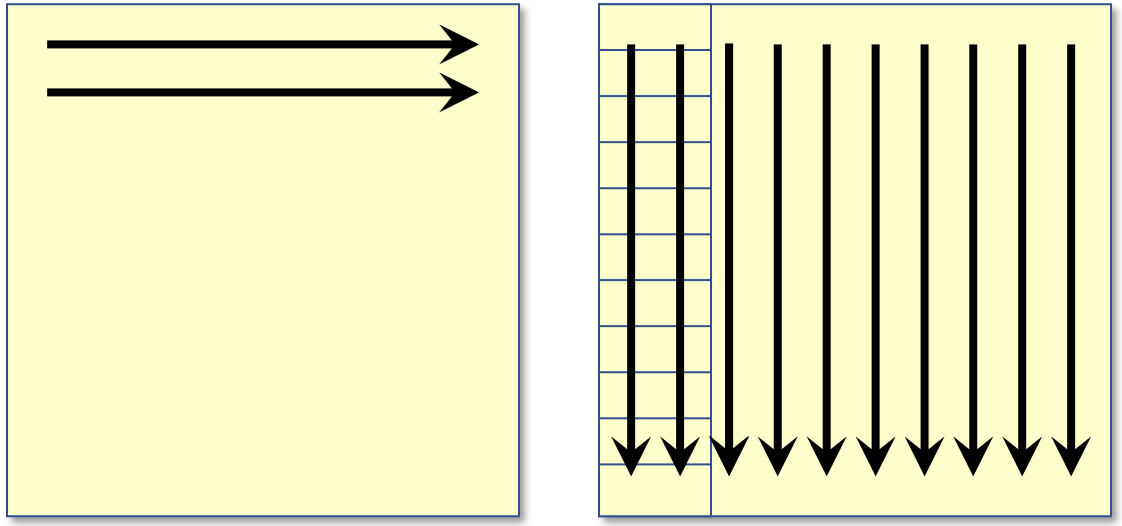
Assume LRU.

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

Analysis of Cache Misses

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

Assume row major and tall cache



A B

Case 3

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

Analyze matrix B.

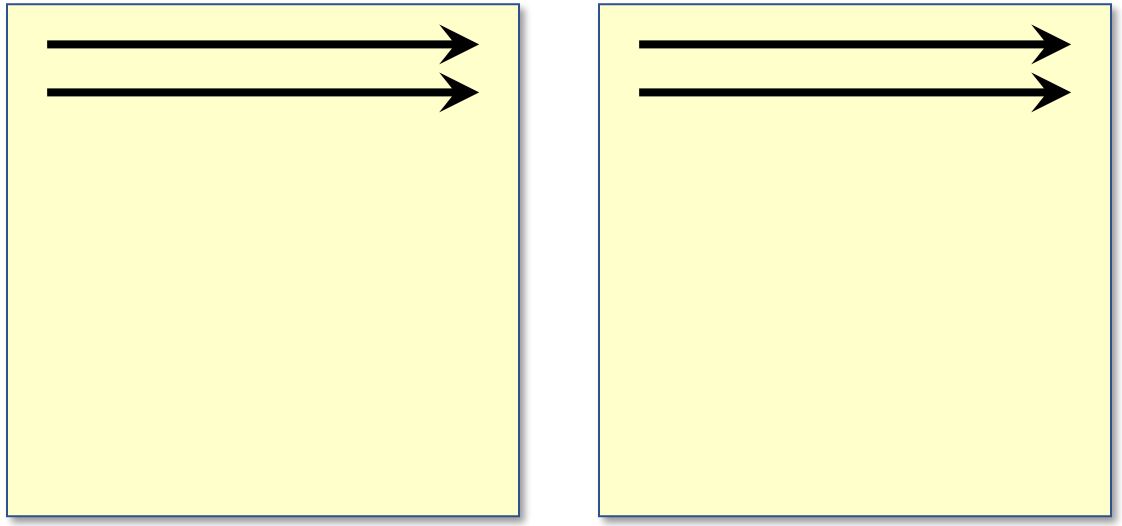
Assume LRU.

$Q(n) = \Theta(n^2/\mathcal{B})$, since everything fits in cache!

Swapping Inner Loop Order

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

Assume row major and tall cache



C B

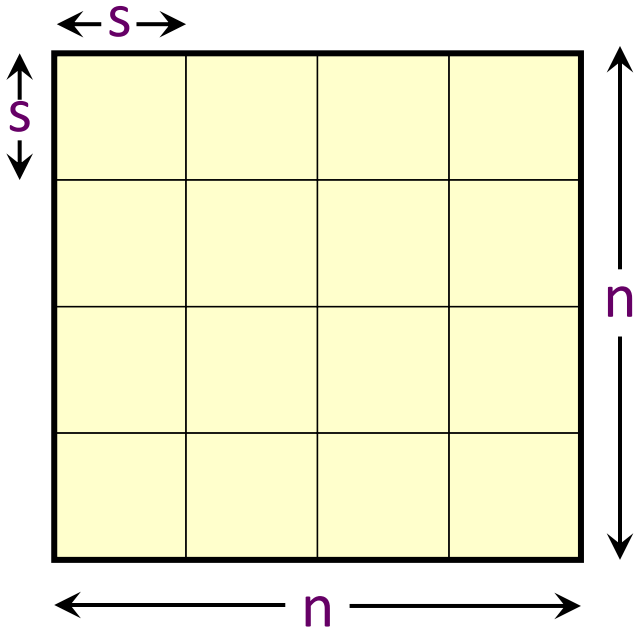
Analyze matrix B.
Assume LRU.

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

Tiling

Tiled Matrix Multiplication

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

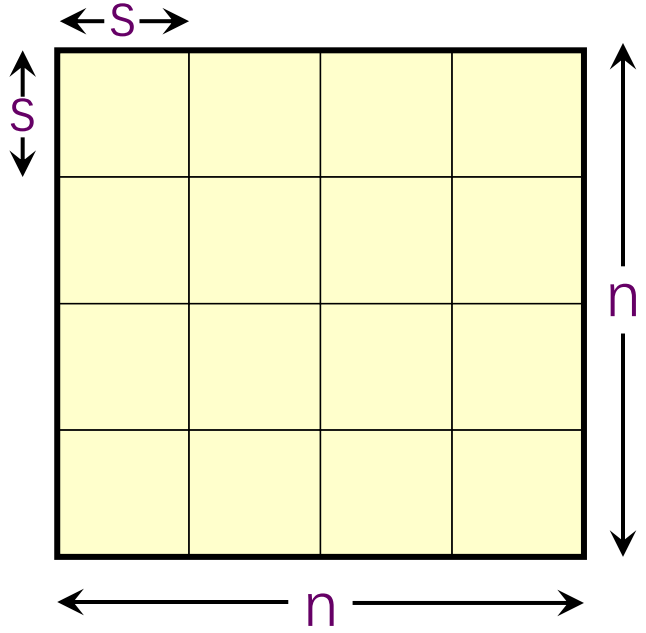


Analysis of work

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

Tiled Matrix Multiplication

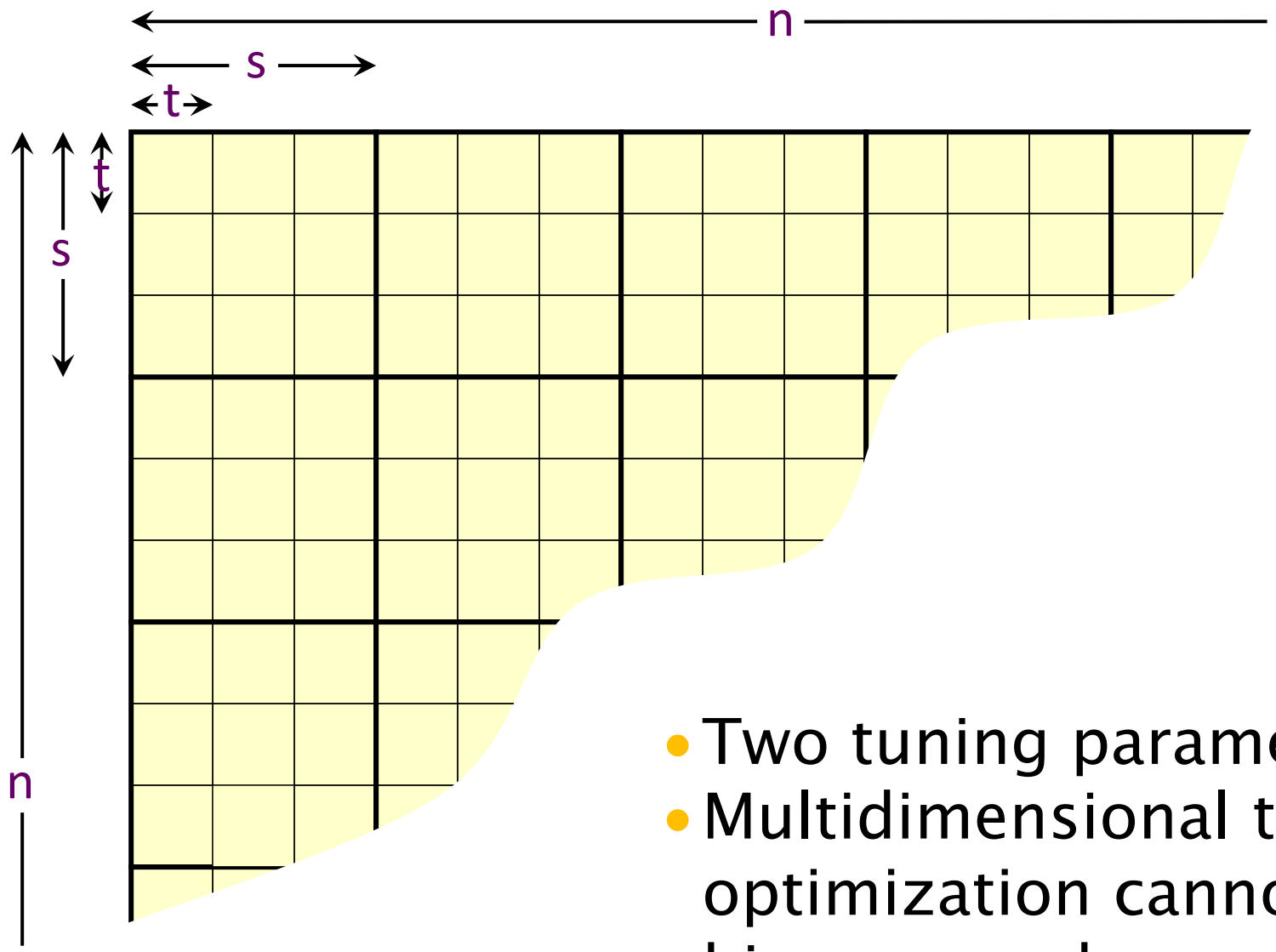
```
void Tiled_Mult(double *C, double *A, double *B, int n) {
  for (int i1=0; i1<n/s; i1+=s)
    for (int j1=0; j1<n/s; j1+=s)
      for (int k1=0; k1<n/s; k1+=s)
        for (int i=i1; i<i1+s && i<n; i++)
          for (int j=j1; j<j1+s && j<n; j++)
            for (int k=k1; k<k1+s && k<n; k++)
 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



Analysis of cache misses

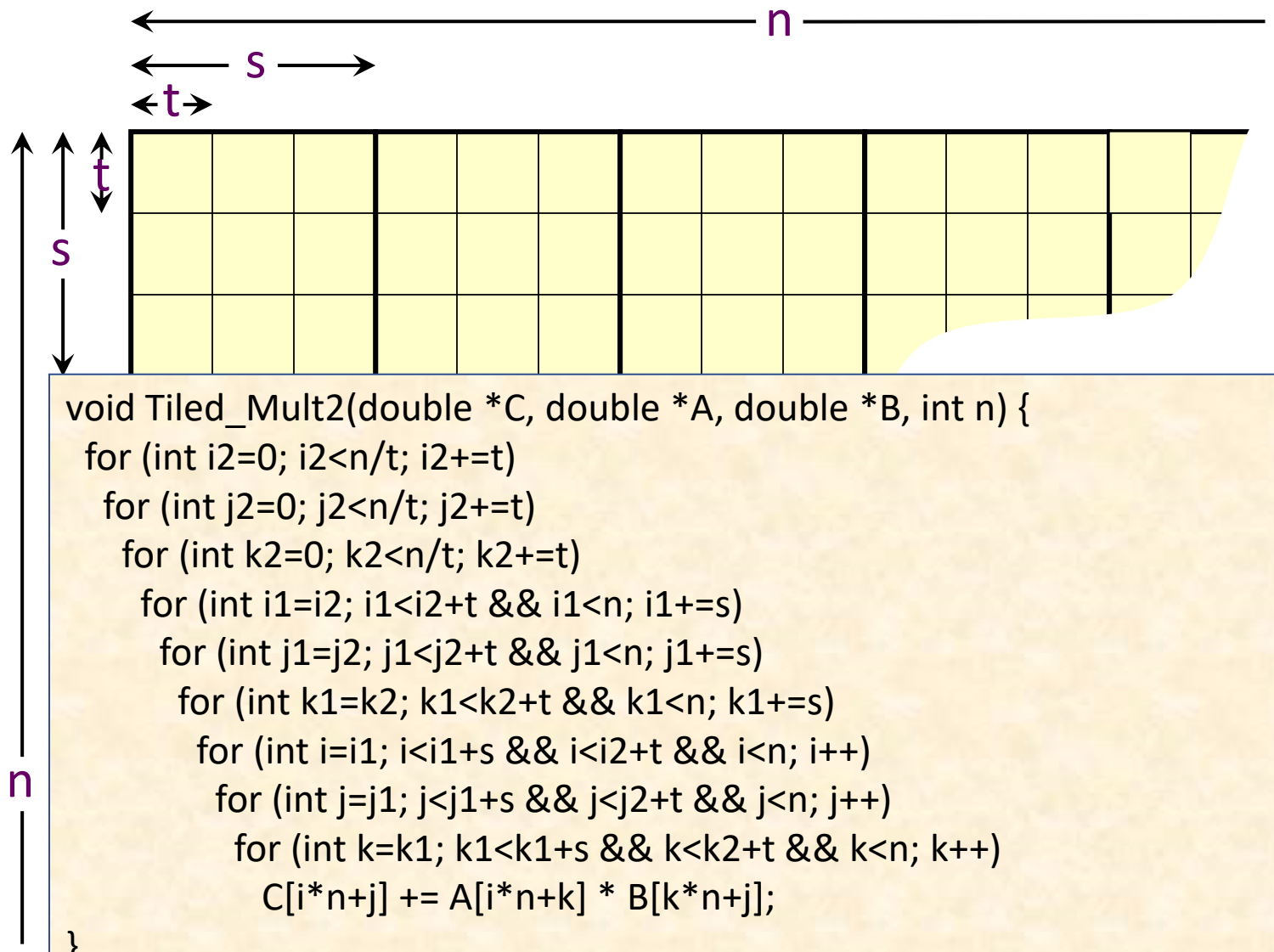
- Tune s so that the submatrices just fit into cache $\Rightarrow s = \Theta(\sqrt{M})$
- Submatrix Caching Lemma implies $\Theta(s^2/B)$ misses per submatrix
- $Q(n) = \Theta((n/s)^3(s^2/B))$
  $= \Theta(n^3/(B\sqrt{M}))$ ***Remember this!***
- Optimal [HK81]

Two-Level Cache



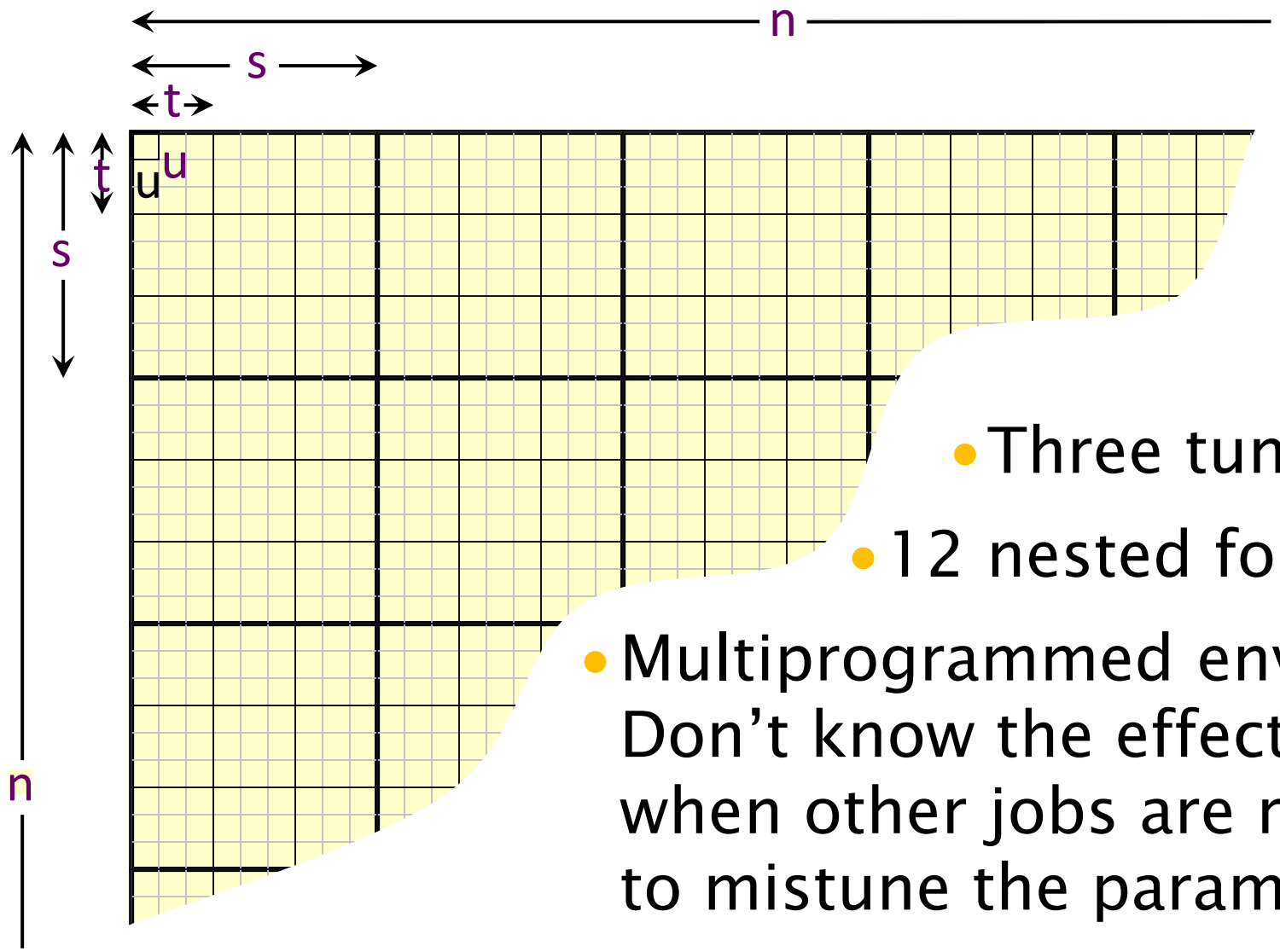
- Two tuning parameters s and t
- Multidimensional tuning optimization cannot be done with binary search

Two-Level Cache




```
void Tiled_Mult2(double *C, double *A, double *B, int n) {
  for (int i2=0; i2<n/t; i2+=t)
    for (int j2=0; j2<n/t; j2+=t)
      for (int k2=0; k2<n/t; k2+=t)
        for (int i1=i2; i1<i2+t && i1<n; i1+=s)
          for (int j1=j2; j1<j2+t && j1<n; j1+=s)
            for (int k1=k2; k1<k2+t && k1<n; k1+=s)
              for (int i=i1; i<i1+s && i<i2+t && i<n; i++)
                for (int j=j1; j<j1+s && j<j2+t && j<n; j++)
                  for (int 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

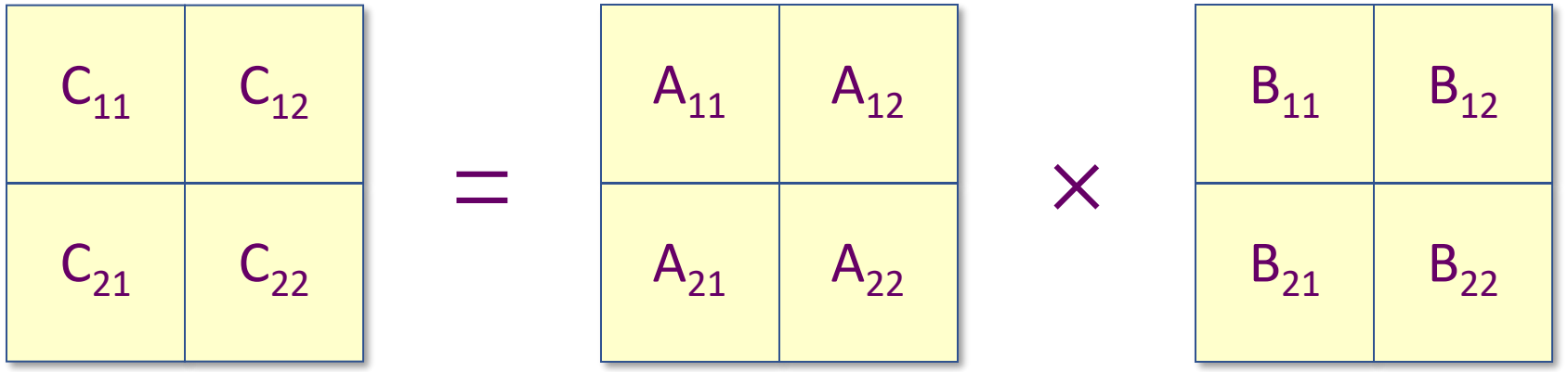


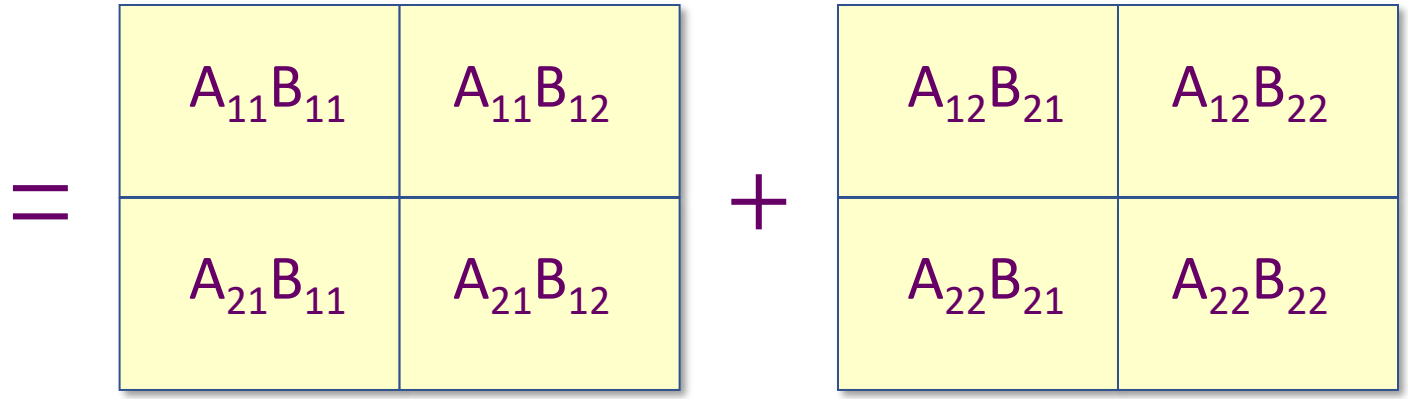
- Three tuning parameters
- 12 nested for loops
- Multiprogrammed environment: Don't know the effective cache size when other jobs are running ⇒ easy to mistune the parameters!

Divide-and-conquer

Recursive Matrix Multiplication

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





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

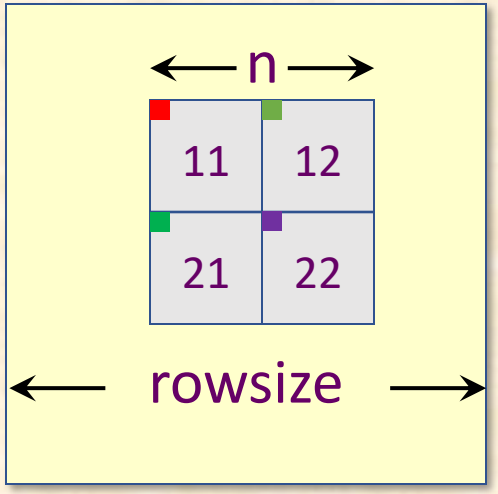
Recursive Code

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

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

Analysis of Work

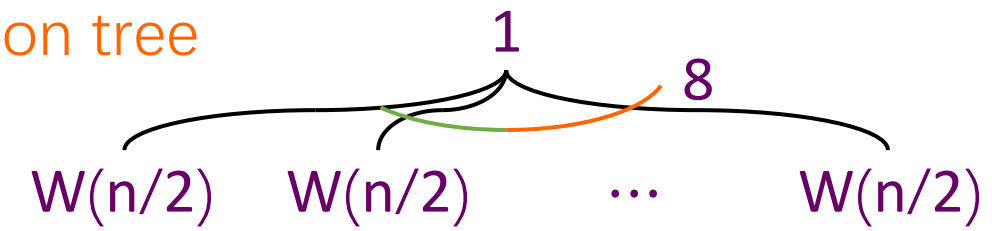
$$W(n) = 8W(n/2) + \Theta(1)$$

recursion tree $W(n)$

Analysis of Work

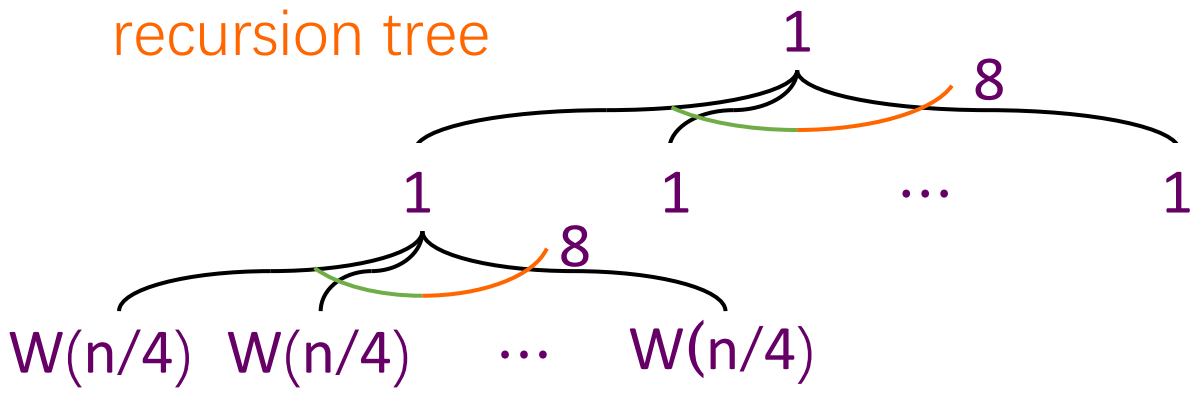
$$W(n) = 8W(n/2) + \Theta(1)$$

recursion tree



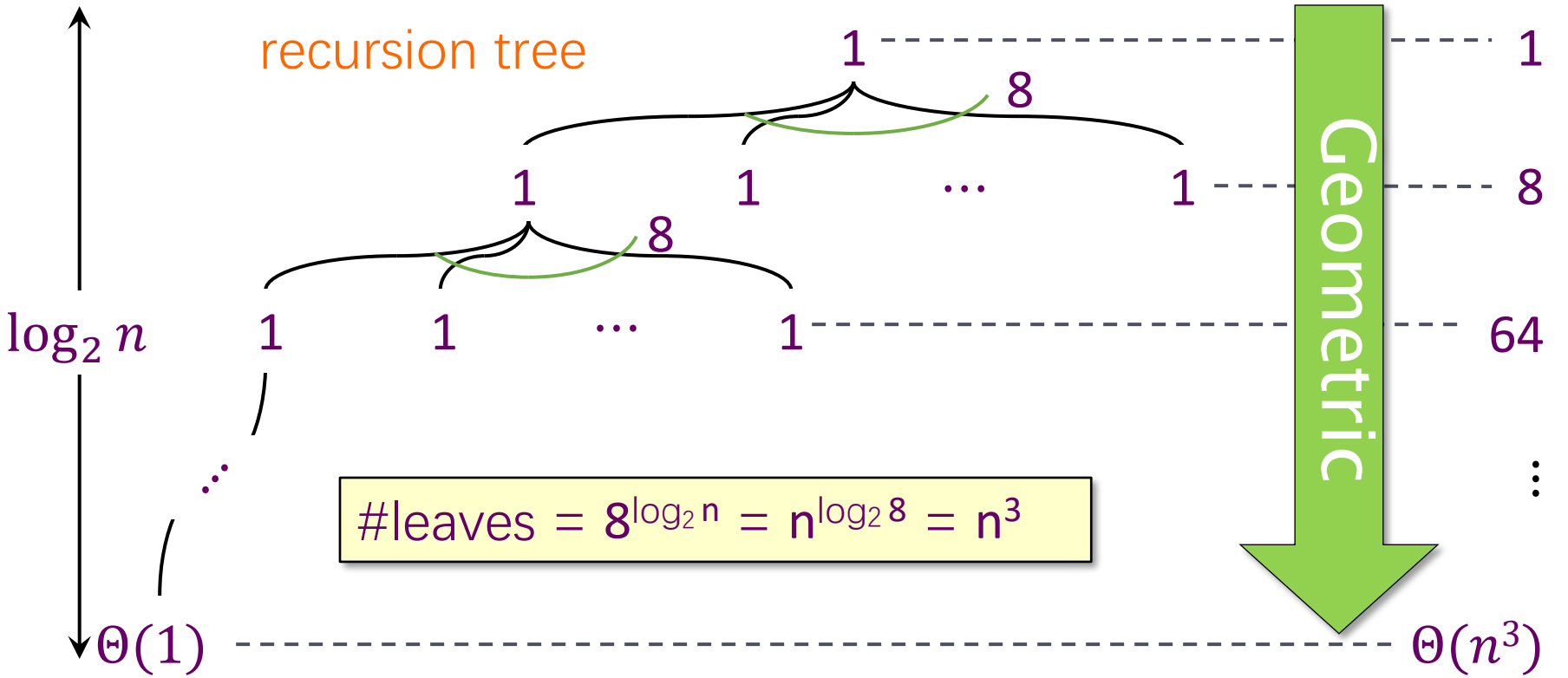
Analysis of Work

$$W(n) = 8W(n/2) + \Theta(1)$$



Analysis of Work

$$W(n) = 8W(n/2) + \Theta(1)$$



Note: Same work as looping versions.

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

Analysis of Cache Misses

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

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

Analysis of Cache Misses

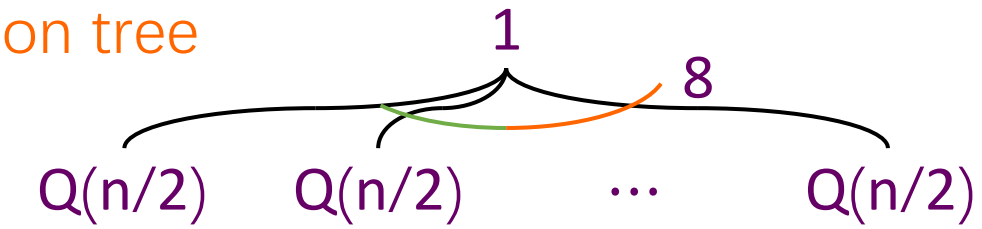
$$Q(n) = \begin{cases} \Theta(n^2/B) \text{ if } n^2 < cM \text{ for suff. small const } c \leq 1, \\ 8Q(n/2) + \Theta(1) \text{ otherwise} \end{cases}$$

recursion tree Q(n)

Analysis of Cache Misses

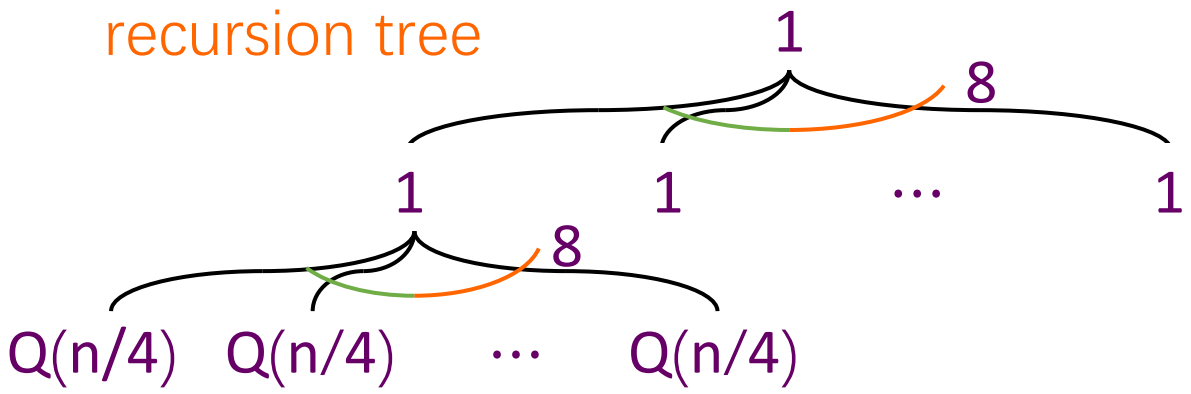
$$Q(n) = \begin{cases} \Theta(n^2/B) \text{ if } n^2 < cM \text{ for suff. small const } c \leq 1, \\ 8Q(n/2) + \Theta(1) \text{ otherwise} \end{cases}$$

recursion tree



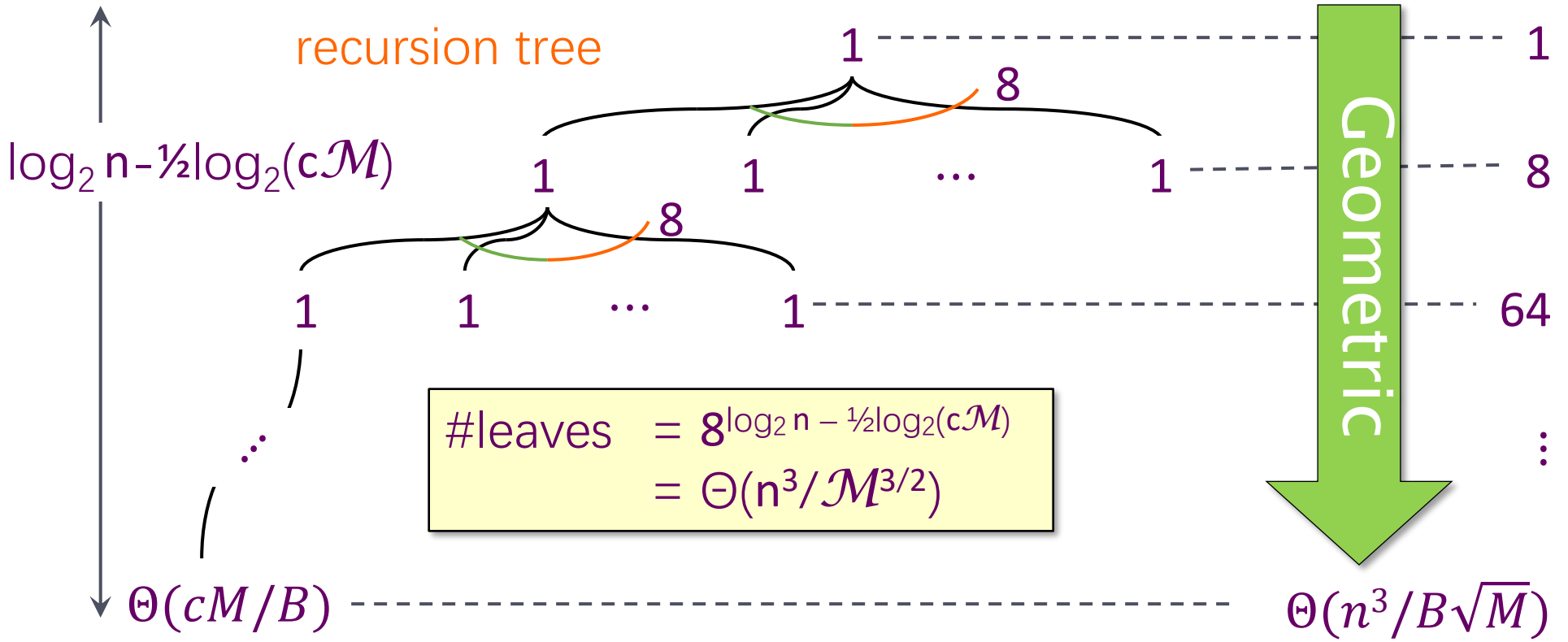
Analysis of Cache Misses

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



Analysis of Cache Misses

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



Same cache misses as with tiling!

$$Q(n) = \Theta(n^3/B\sqrt{M})$$

Efficient Cache-Oblivious Algorithms

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

Matrix multiplication

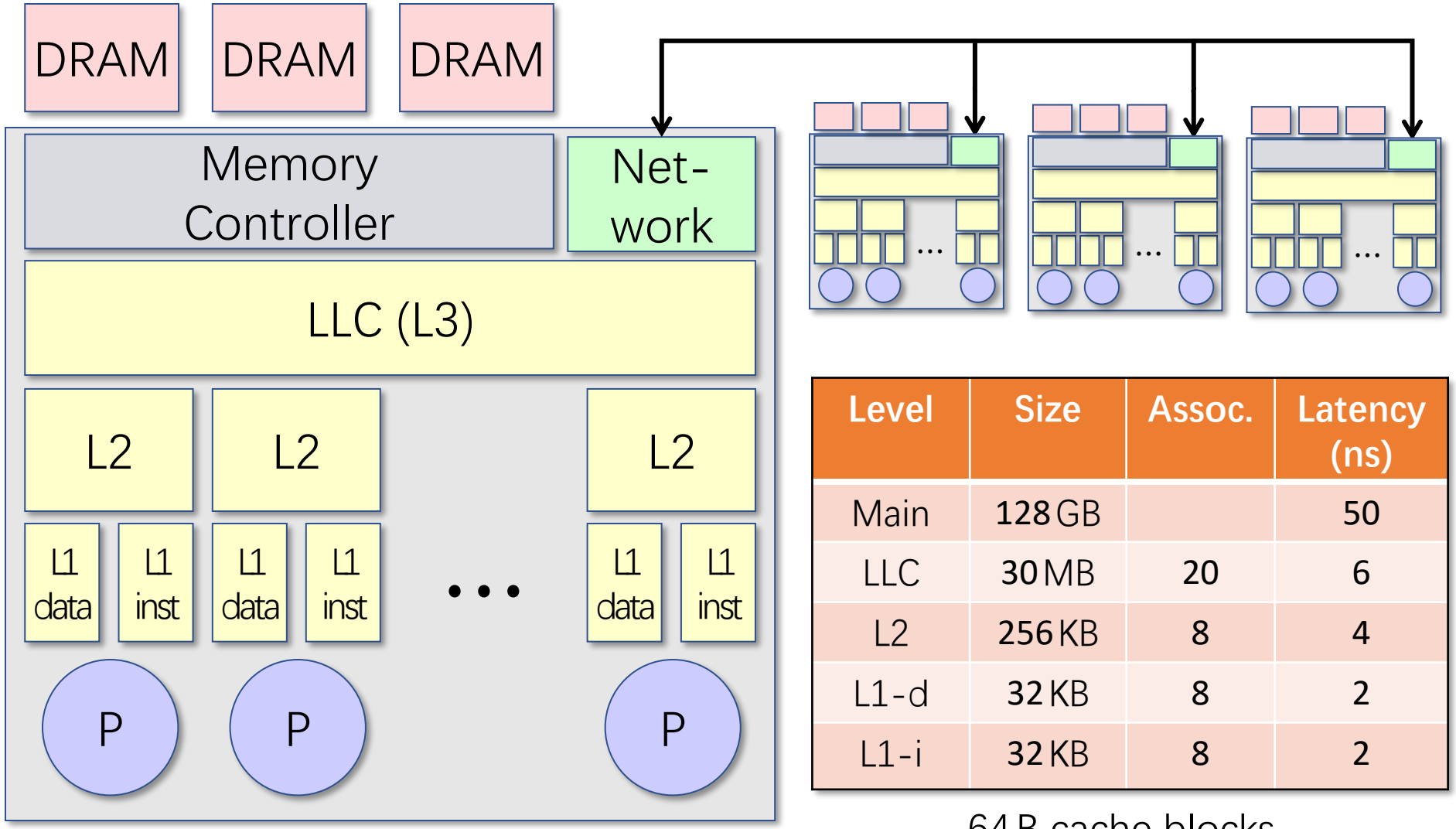
The best cache-oblivious codes to date work on arbitrary rectangular matrices and perform binary splitting (instead of 8-way) on the largest of i, j, and k

Recursive Parallel Matrix Multiply

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

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

Summary

Multicore Cache Hierarchy

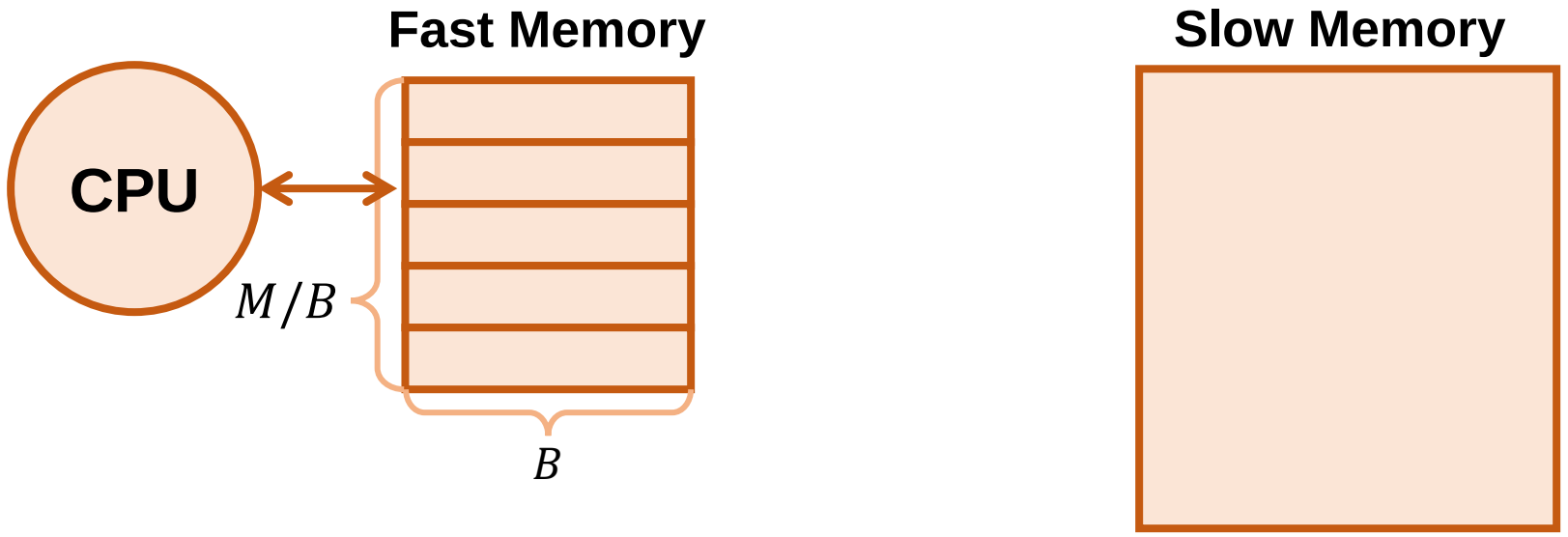


Level	Size	Assoc.	Latency (ns)
Main	128 GB		50
LLC	30 MB	20	6
L2	256 KB	8	4
L1-d	32 KB	8	2
L1-i	32 KB	8	2

64 B cache blocks

The I/O model

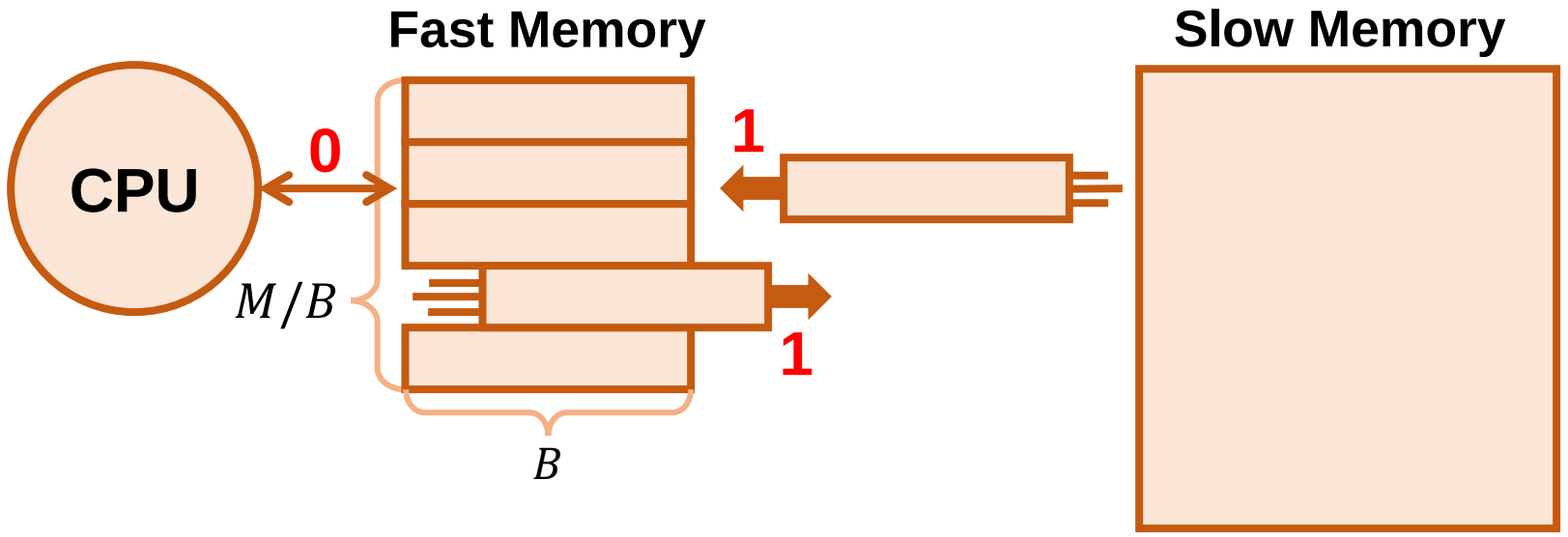
- **Two-level memory hierarchy:**
  - A small memory (fast memory, cache) of fixed size M
  - A large memory (slow memory) of unbounded size
- **Both are partitioned into blocks of size B**
- **Instructions can only apply to data in primary memory, and are free**



The I/O model

- **The I/O model has two special *memory transfer* instructions:**
  - **Read transfer**: load a block from slow memory
  - **Write transfer**: write a block to slow memory
- **The complexity of an algorithm on the I/O model (I/O complexity) is measured by:**

#(read transfers) + #(write transfers)



I/O-efficient algorithms

- **Matrix multiplication:** $O\left(\frac{n^3}{B\sqrt{M}}\right)$ **(sequential)**
- **Next two lectures: sorting and semisorting**
- **What is missing here: I/O efficiency for the parallel setting**