CS142 – Lecture 9

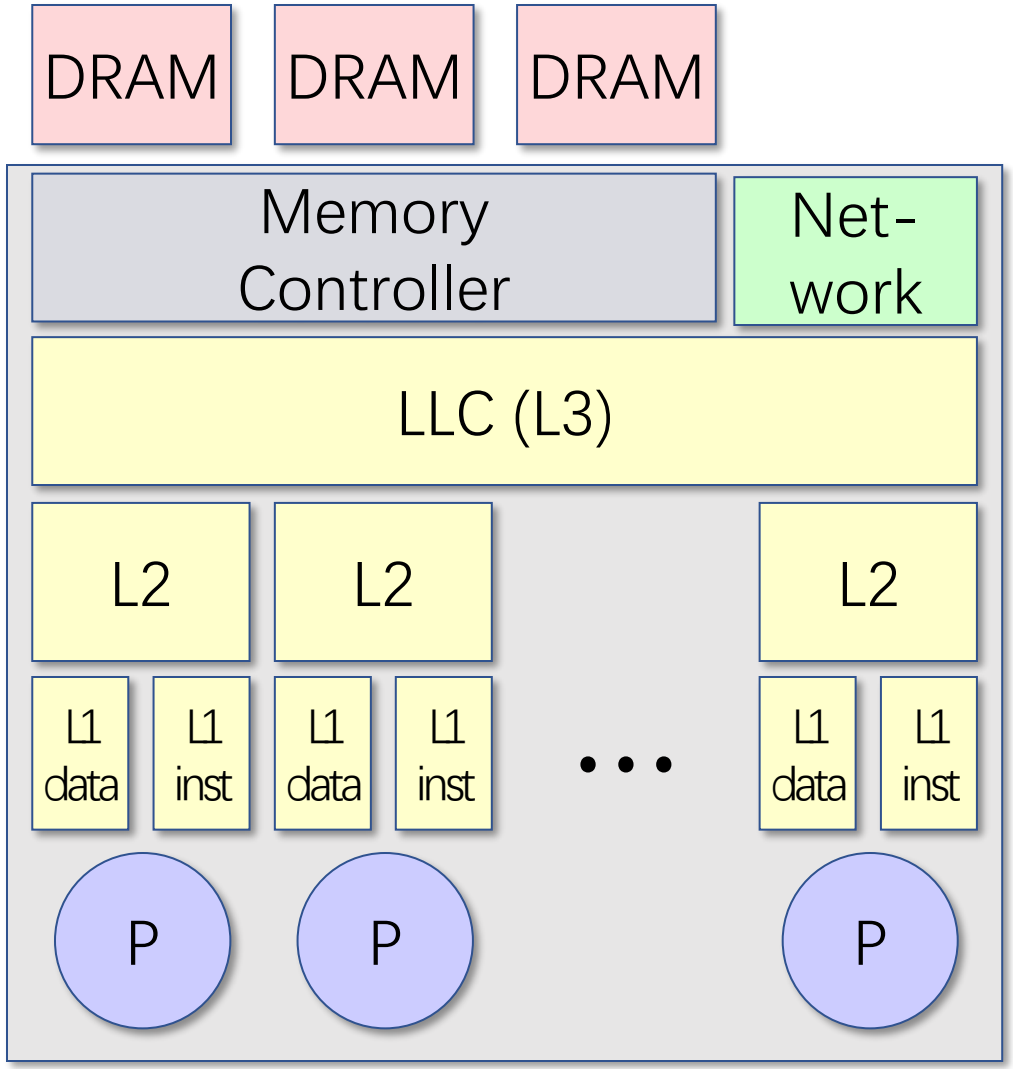
Yan Gu

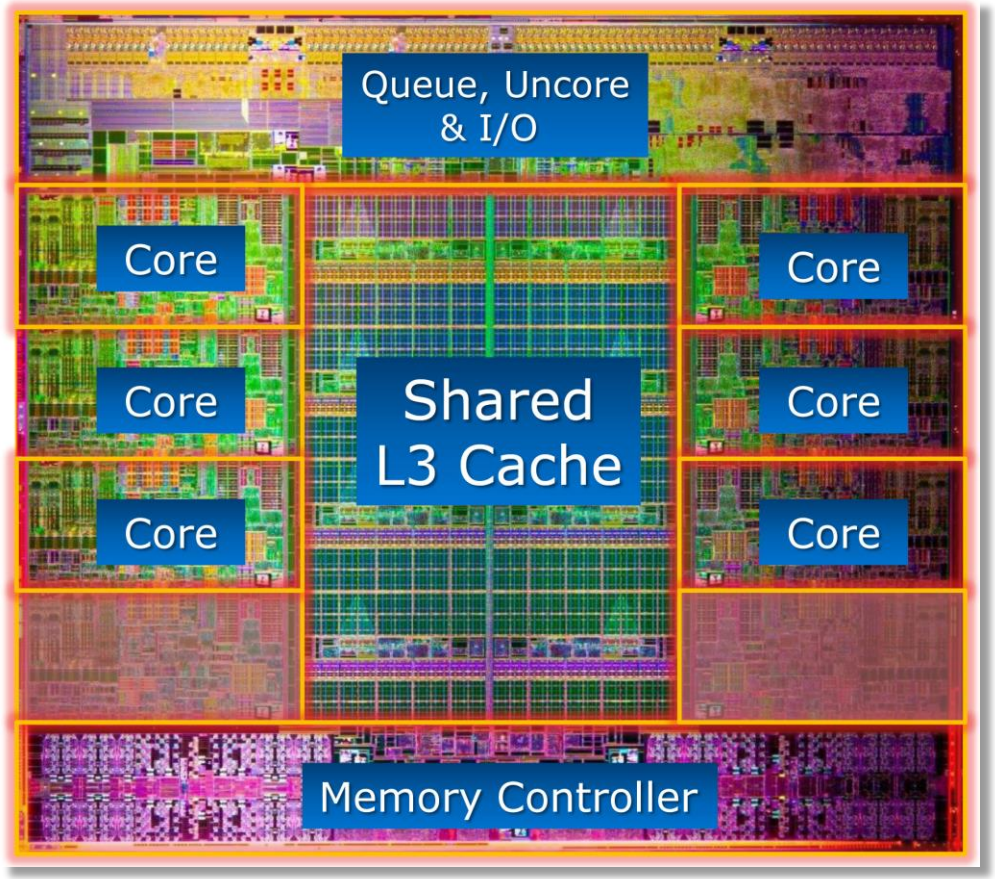
Algorithm Engineering
(aka. How to Write Fast Code)

I/O (Cache) Efficiency

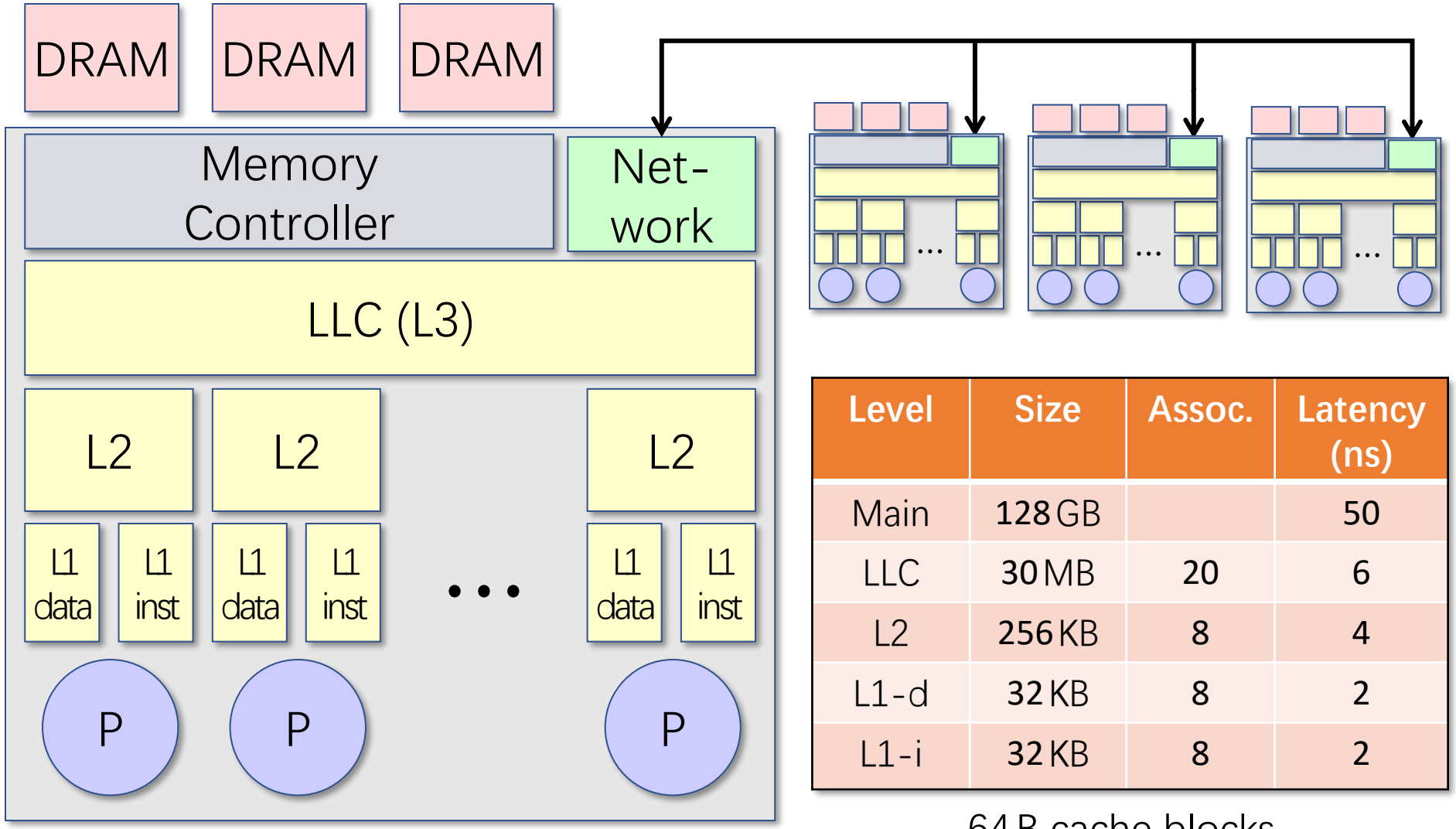
Many slides in this lecture are borrowed from Lecture 14 in 6.172 Performance Engineering of Software Systems at MIT. The credit is to Prof. Charles E. Leiserson, and the instructor appreciates the permission to use them in this course.

Multicore Cache Hierarchy





Multicore Cache Hierarchy

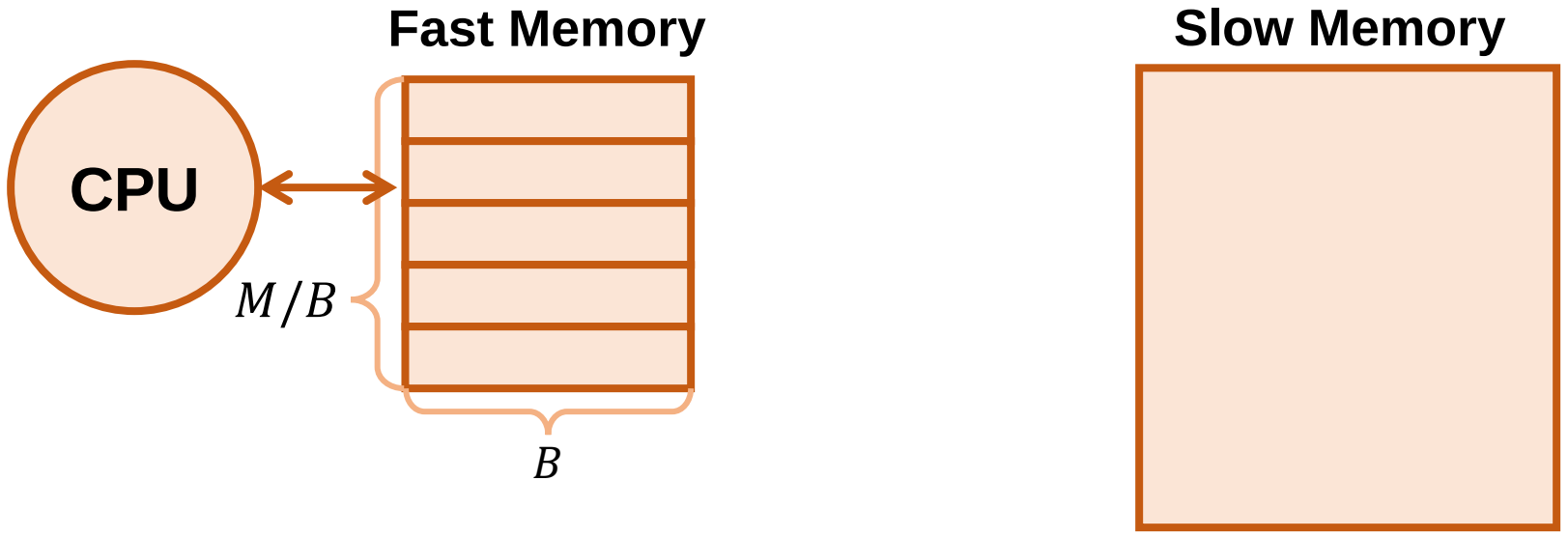


Level	Size	Assoc.	Latency (ns)
Main	128 GB		50
LLC	30 MB	20	6
L2	256 KB	8	4
L1-d	32 KB	8	2
L1-i	32 KB	8	2

64 B cache blocks

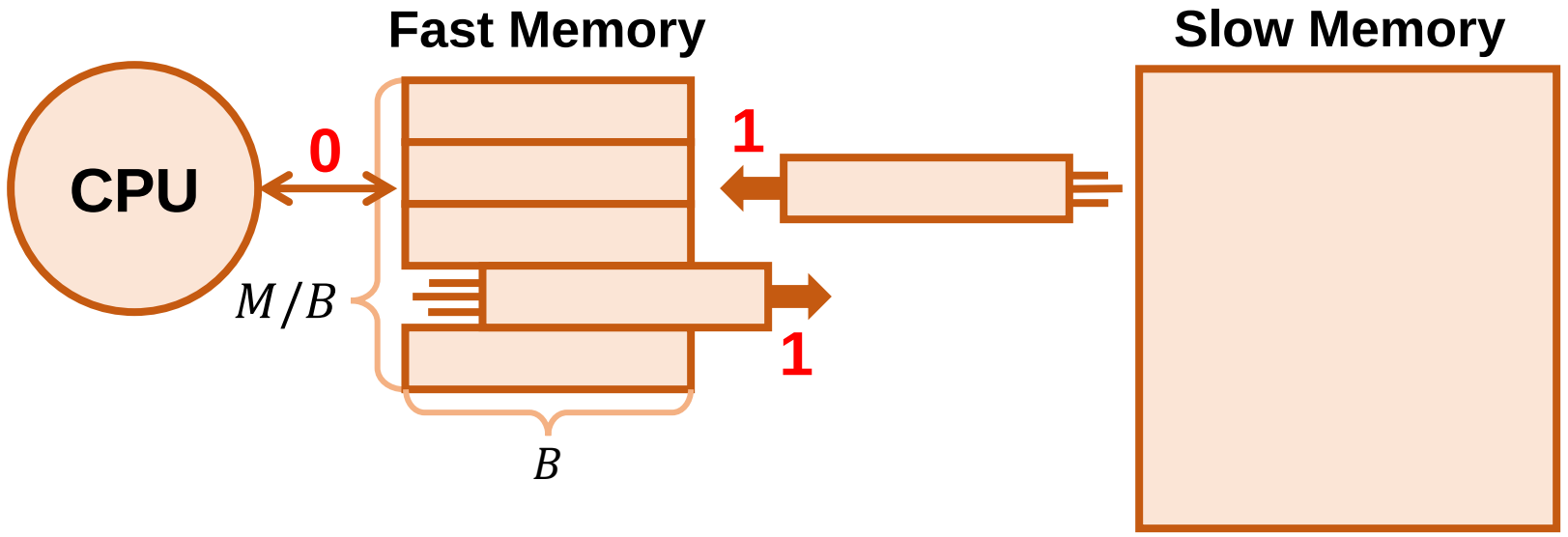
The I/O Model (External Memory-, Ideal Cache-)

- **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 (External Memory-, Ideal Cache-)

- We assume the cache is fully associative, and the it takes unit cost to load and evict a pair of blocks
- The complexity of an algorithm on the I/O model (I/O complexity) assumes an optimal cache replacement policy



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];
}
```

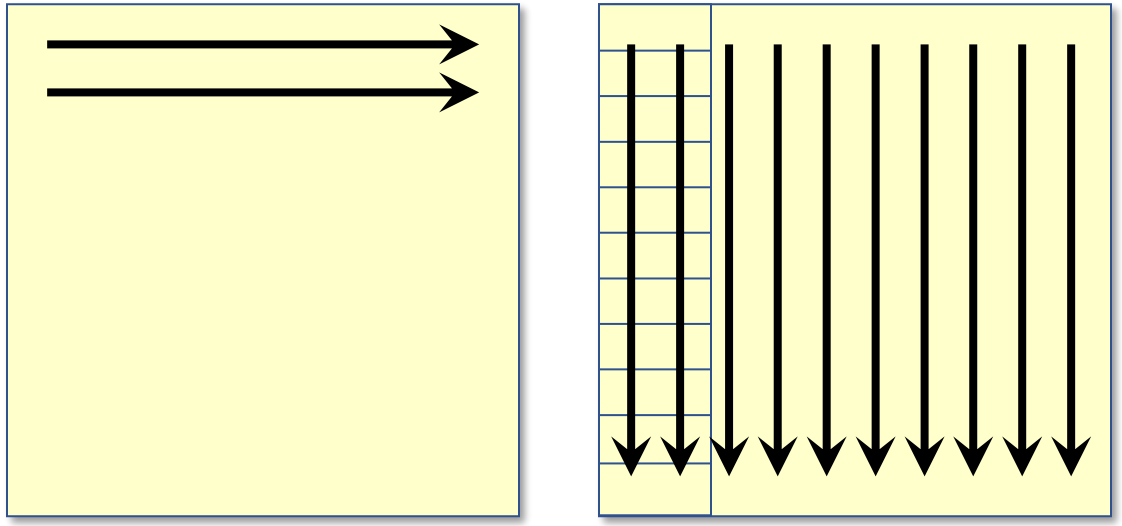
Analysis of work:

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

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.

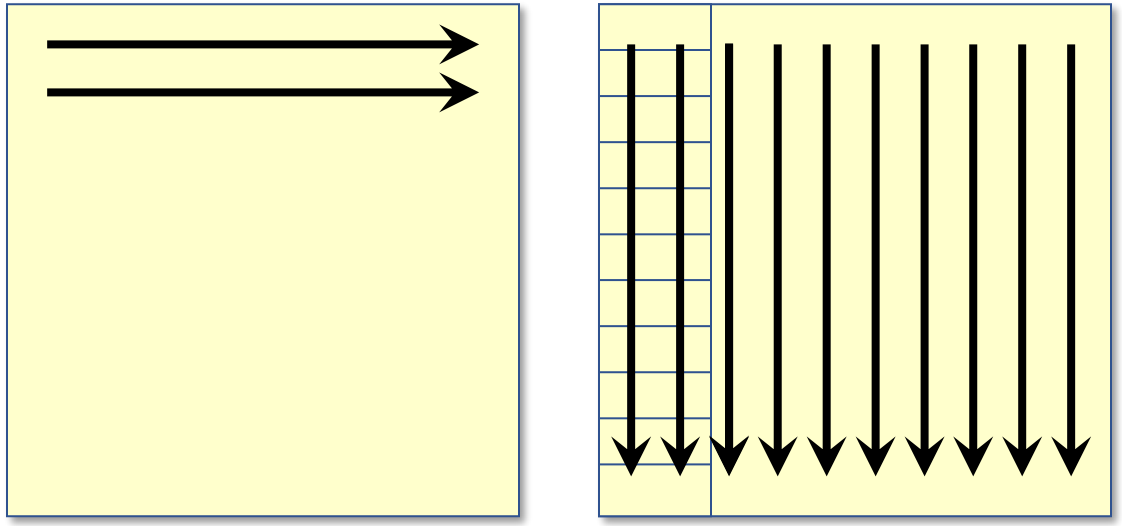
Assume LRU.

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

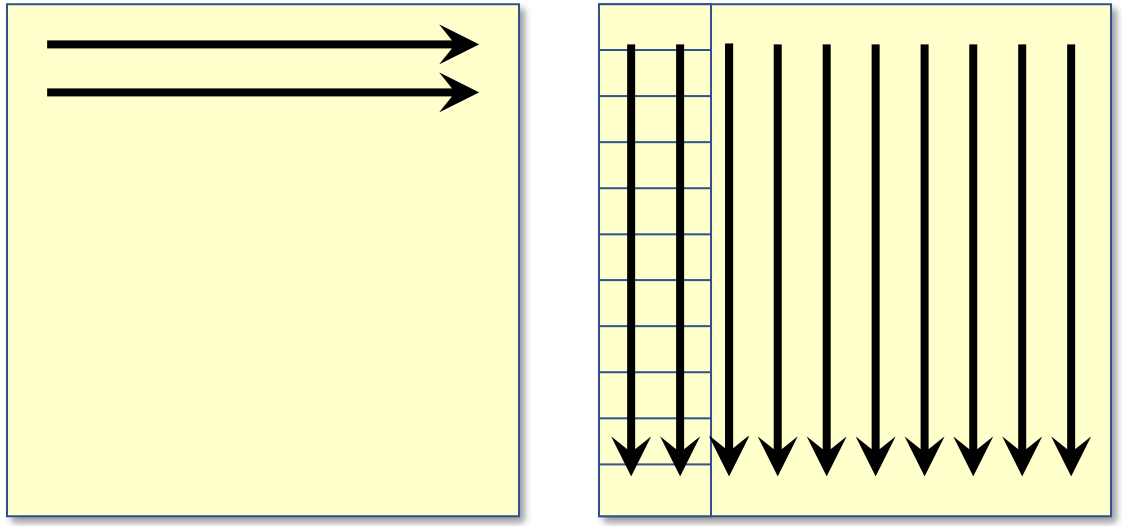
Assume LRU.

$Q(n) = n \cdot \Theta(n^2/\mathcal{B}) = \Theta(n^3/\mathcal{B})$, 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.

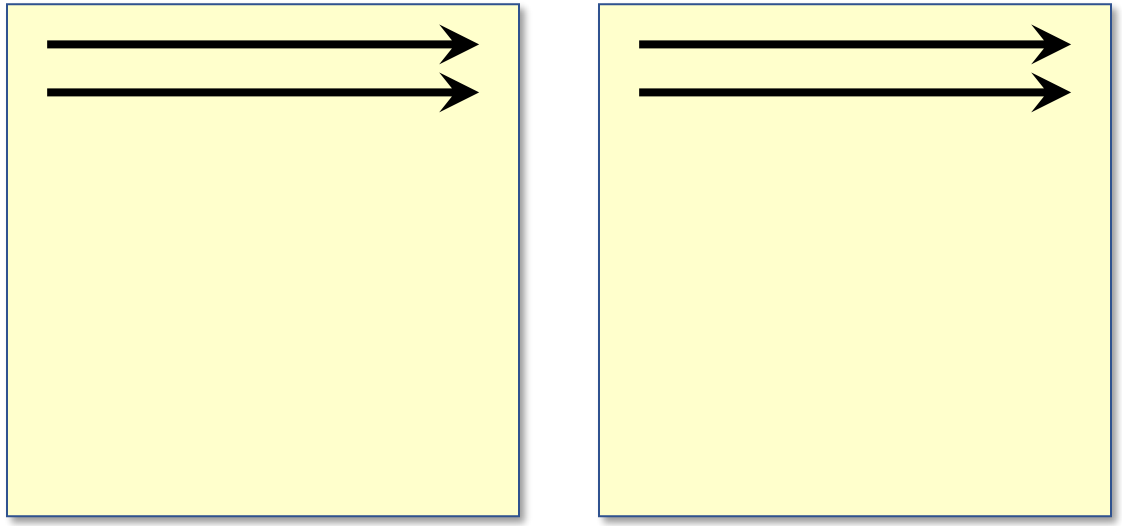
Assume LRU.

$Q(n) = \Theta(n^2/\mathcal{B})$, 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

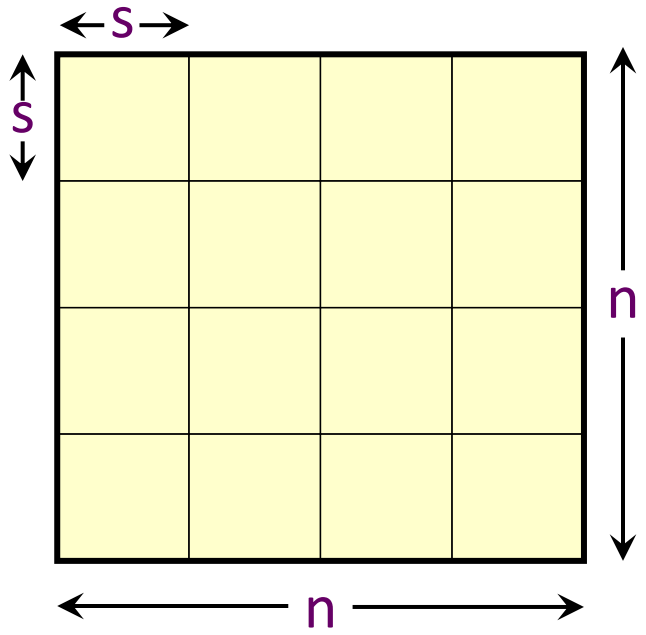
Analyze matrix B.
Assume LRU.

$Q(n) = n \cdot \Theta(n^2/\mathcal{B}) = \Theta(n^3/\mathcal{B})$, 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];
}
```

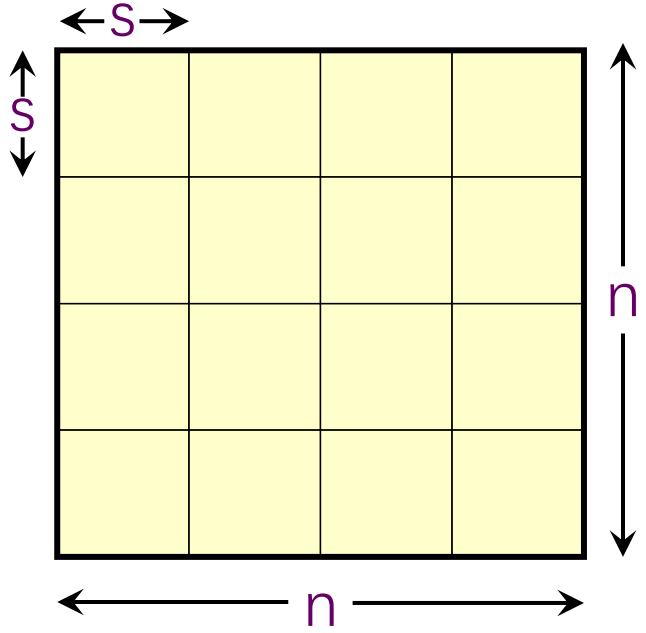


Analysis of work

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

Tiled Matrix Multiplication

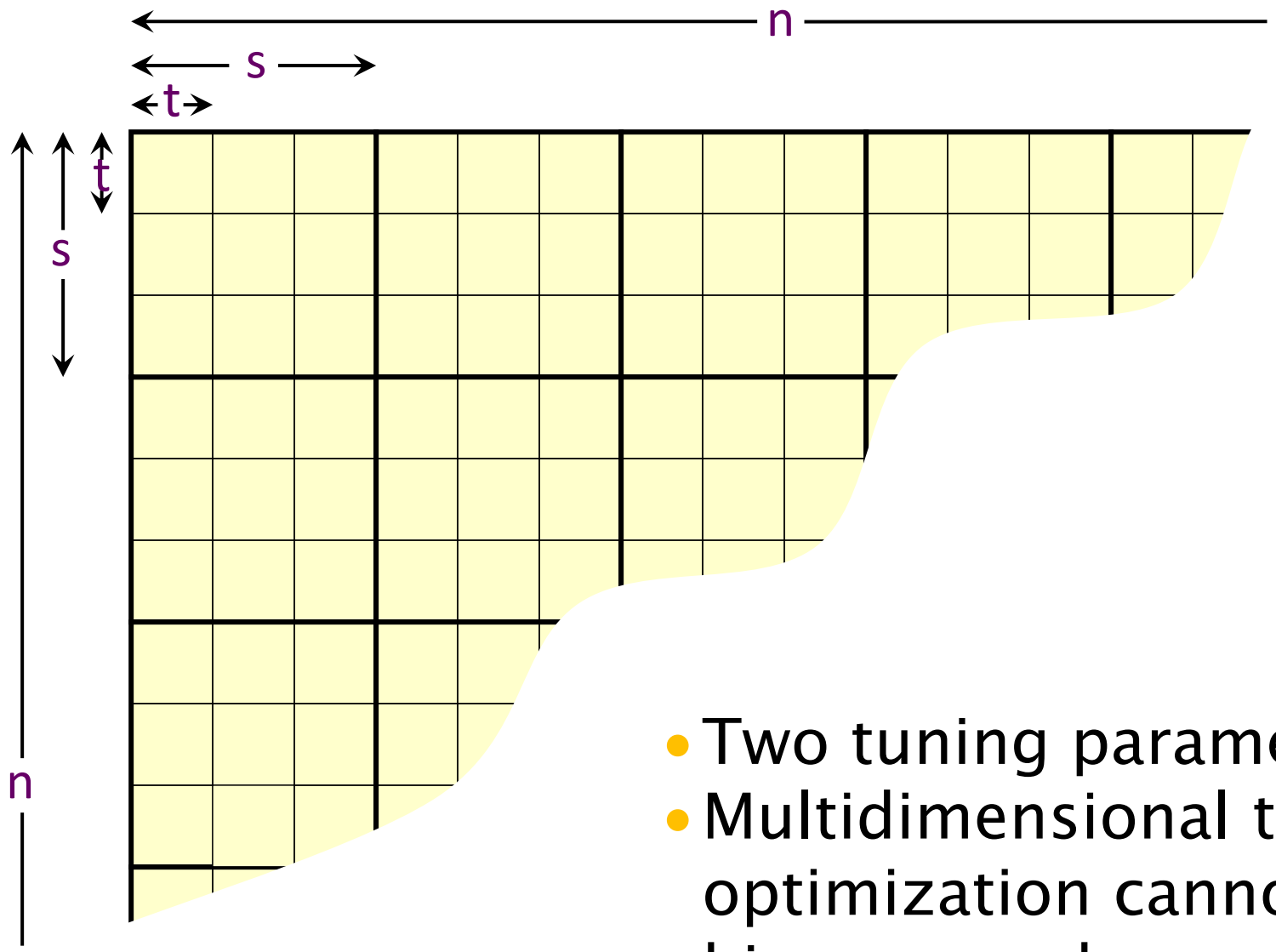
```
void Tiled_Mult(double *C, double *A, double *B, int n) {
  for (int i1=0; i1<n; i1+=s)
    for (int j1=0; j1<n; j1+=s)
      for (int k1=0; k1<n; k1+=s)
        for (int i=i1; i<i1+s && i<n; i++)
          for (int j=j1; j<j1+s && j<n; j++)
            for (int k=k1; k<k1+s && k<n; k++)
       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



Analysis of cache misses

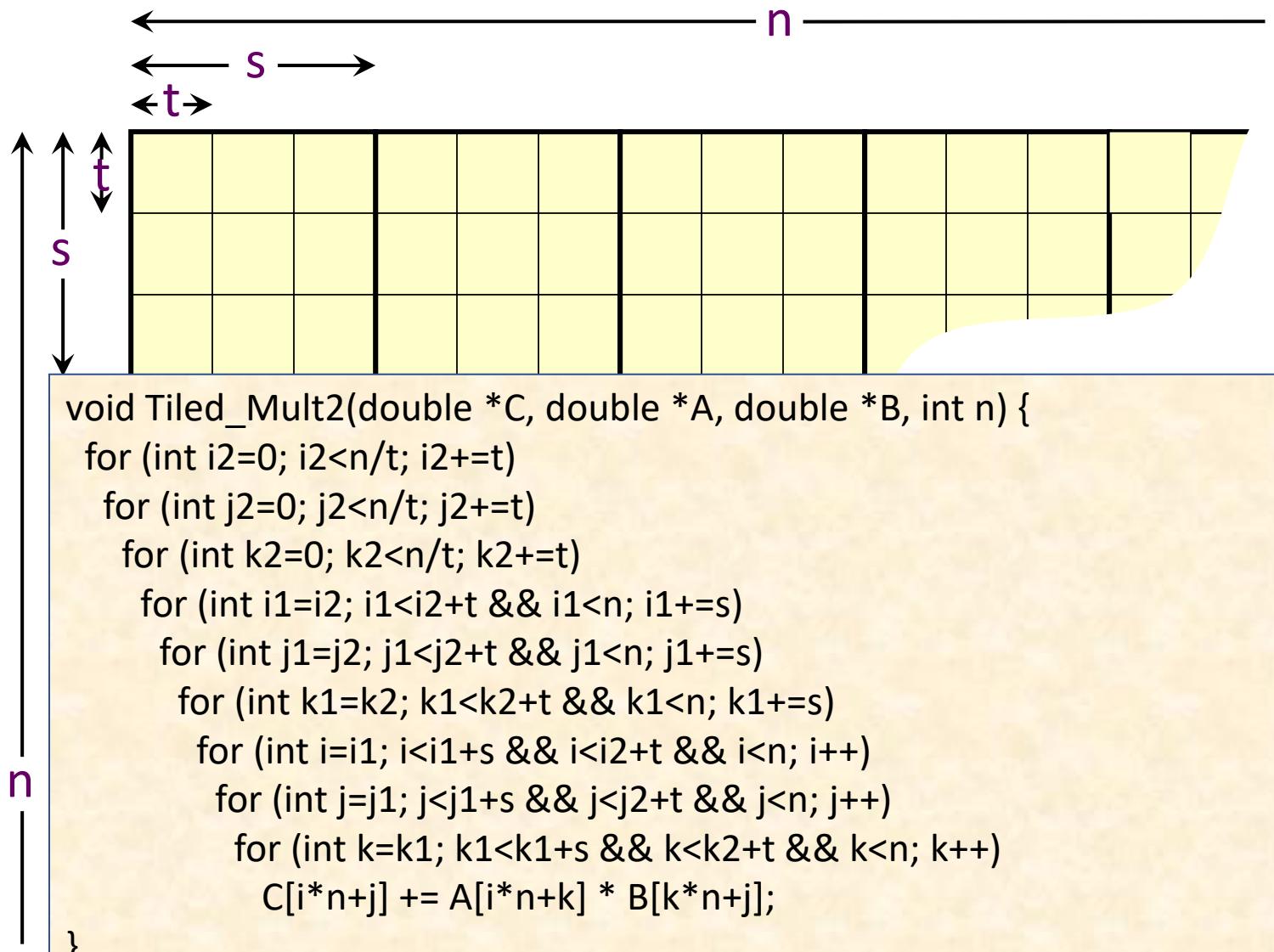
- Tune s so that the submatrices just fit into cache $\Rightarrow s = \Theta(\sqrt{M})$
- Submatrix Caching Lemma implies $\Theta(s^2/B)$ misses per submatrix
- $Q(n) = \Theta((n/s)^3(s^2/B))$
  $= \Theta(n^3/(B\sqrt{M}))$ ***Remember this!***
- Optimal [HK81]

Two-Level Cache



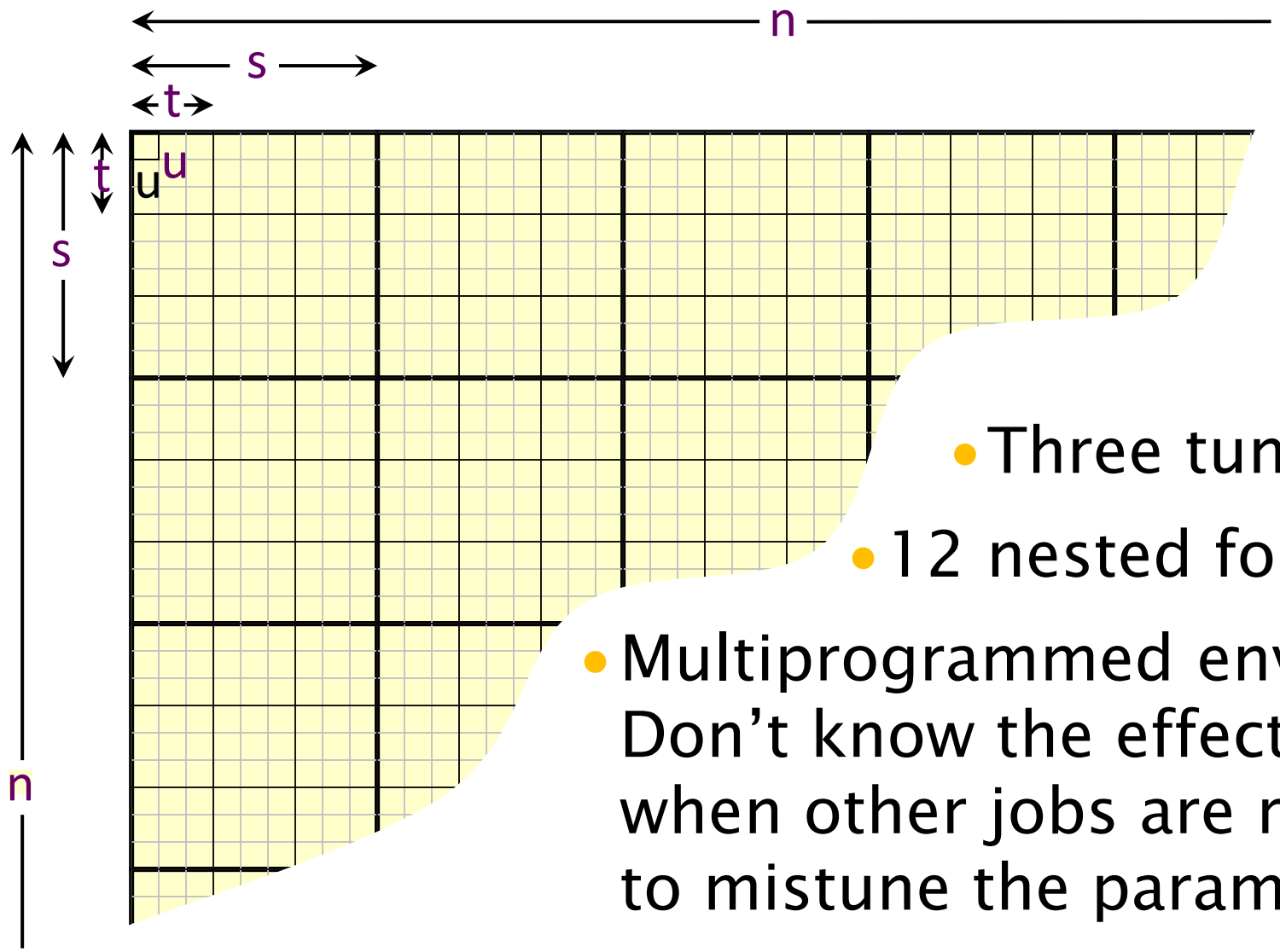
- Two tuning parameters s and t
- Multidimensional tuning optimization cannot be done with binary search

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

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

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

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

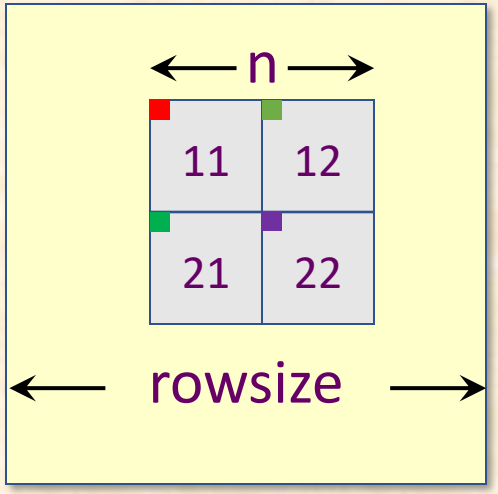
Recursive Code

```
// Assume that n is an exact power of 2.
void Rec_Mult(double *C, double *A, double *B,
              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

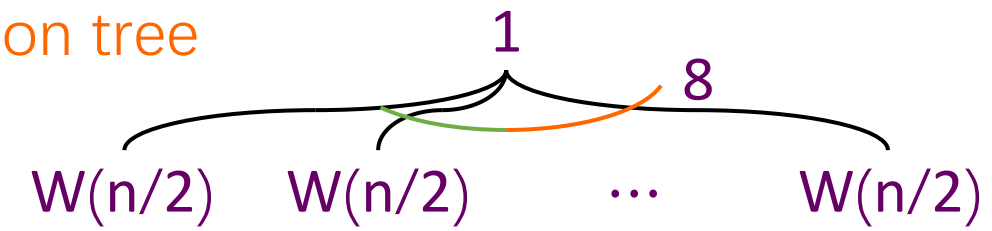
$$W(n) = 8W(n/2) + \Theta(1)$$

recursion tree $W(n)$

Analysis of Work

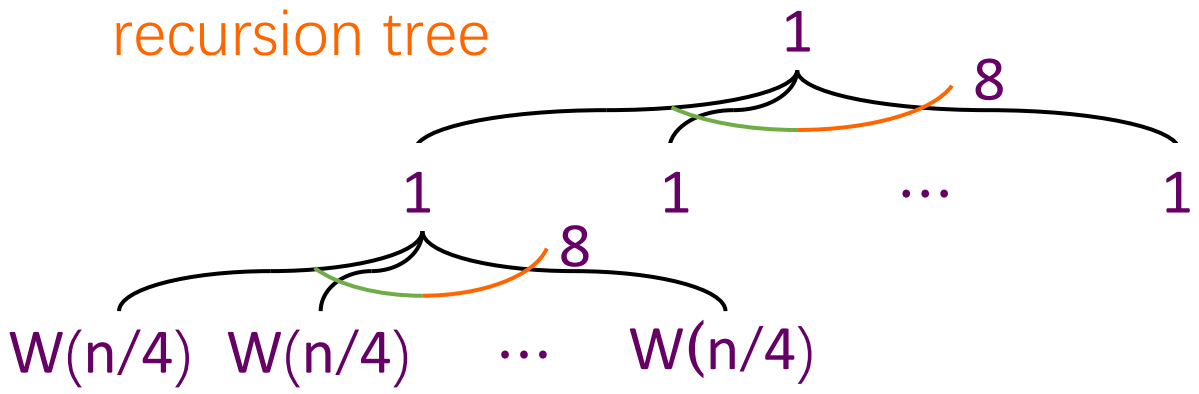
$$W(n) = 8W(n/2) + \Theta(1)$$

recursion tree



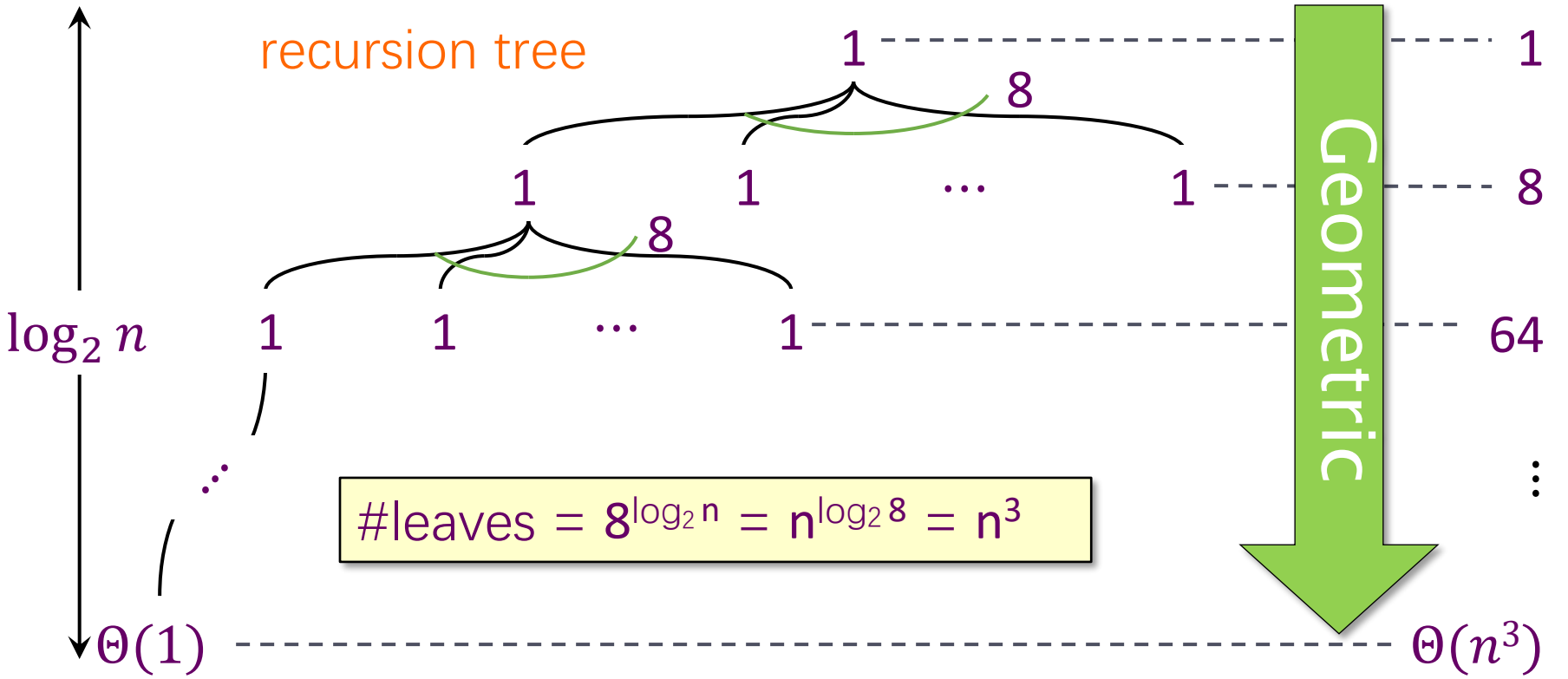
Analysis of Work

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



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

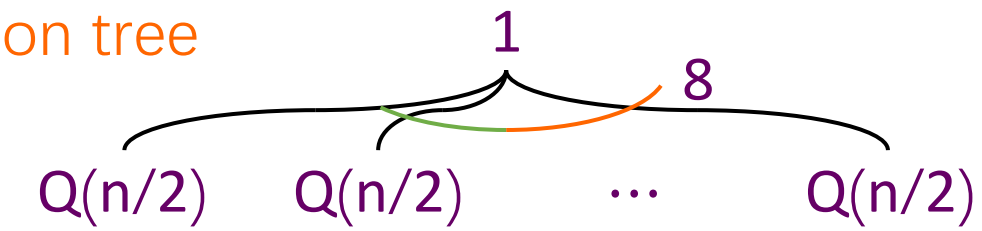
$$Q(n) = \begin{cases} \Theta(n^2/B) \text{ if } n^2 < cM \text{ for suff. small const } c \leq 1, \\ 8Q(n/2) + \Theta(1) \text{ otherwise} \end{cases}$$

recursion tree Q(n)

Analysis of Cache Misses

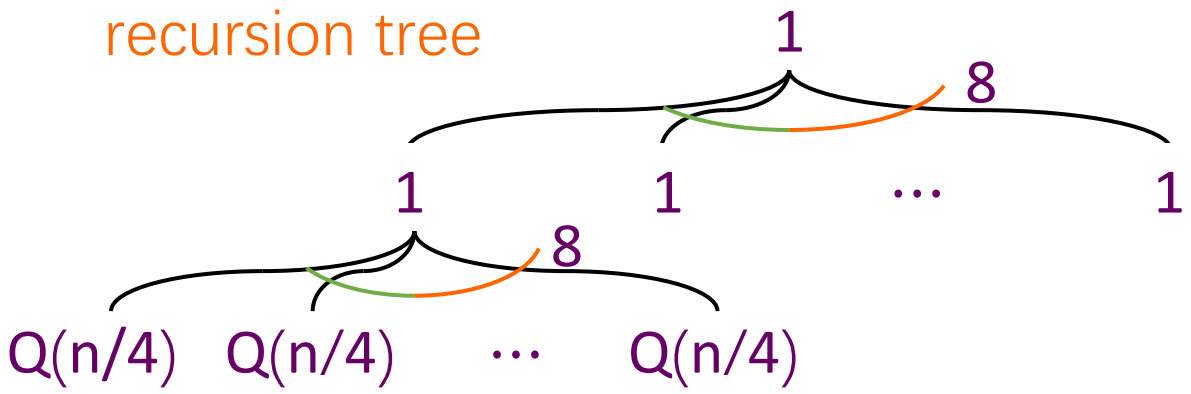
$$Q(n) = \begin{cases} \Theta(n^2/B) \text{ if } n^2 < cM \text{ for suff. small const } c \leq 1, \\ 8Q(n/2) + \Theta(1) \text{ otherwise} \end{cases}$$

recursion tree



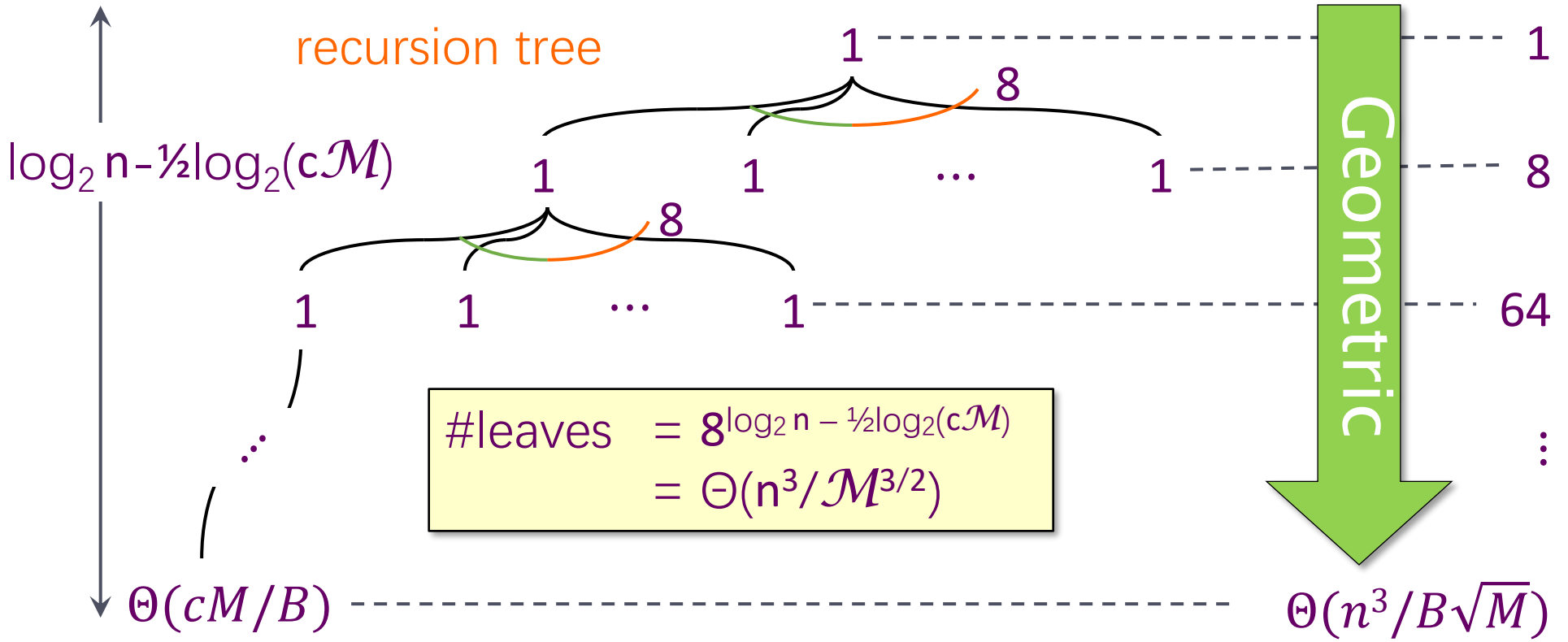
Analysis of Cache Misses

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



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 multi-level caches automatically
- Good for parallelism

Matrix multiplication

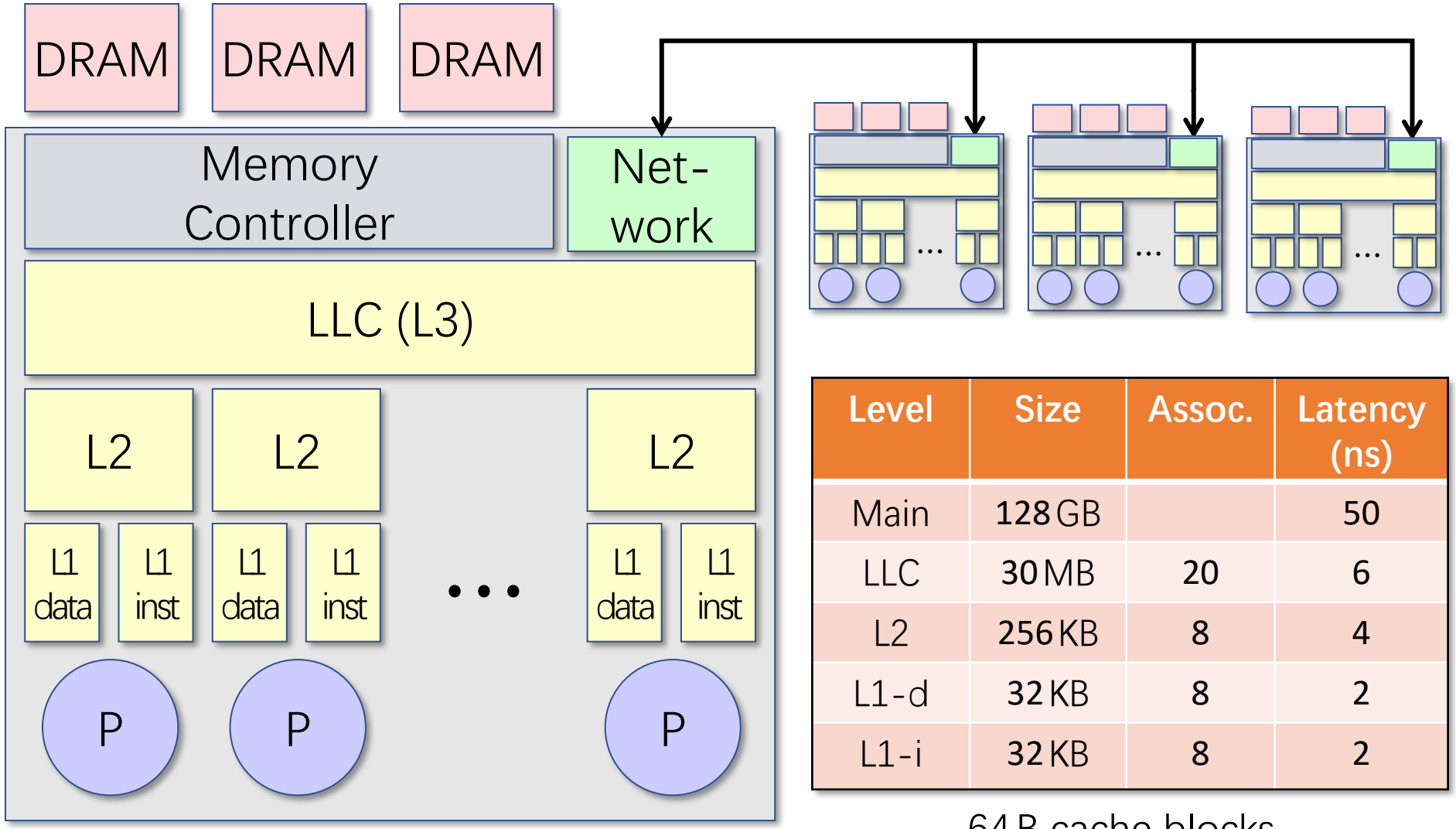
The best cache-oblivious codes to date work on arbitrary rectangular matrices and perform binary splitting (instead of 8-way) on the largest of i, j, and k

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

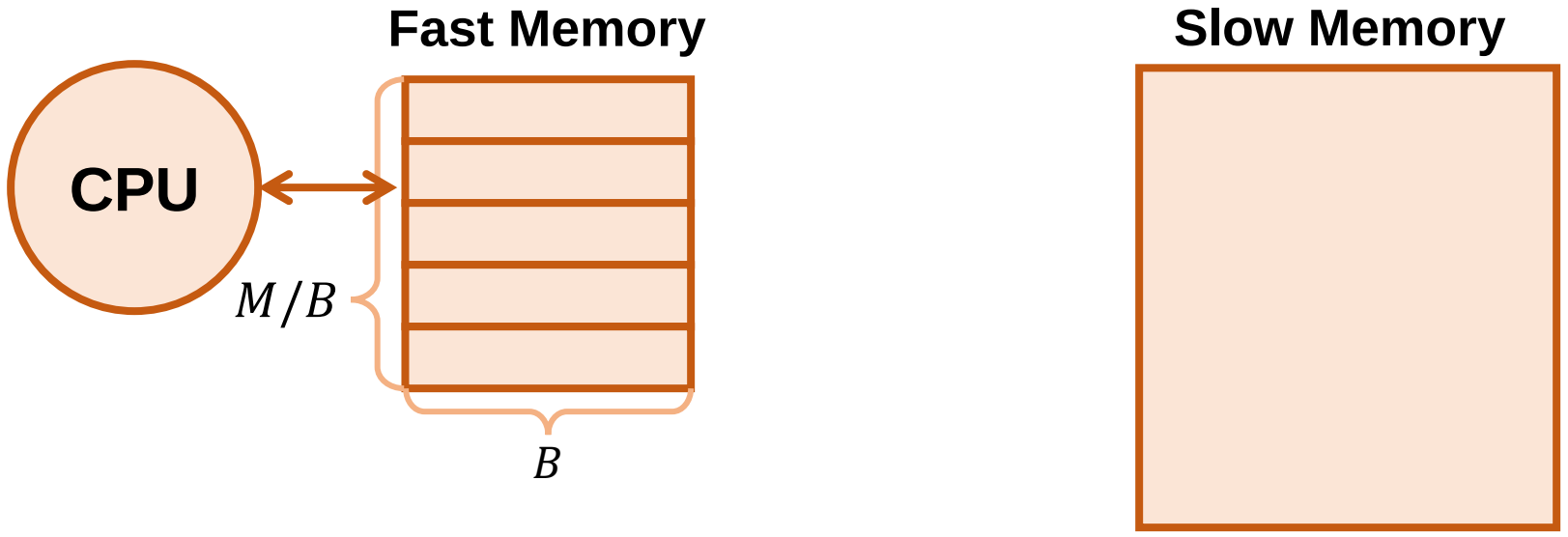


Level	Size	Assoc.	Latency (ns)
Main	128 GB		50
LLC	30 MB	20	6
L2	256 KB	8	4
L1-d	32 KB	8	2
L1-i	32 KB	8	2

64 B cache blocks

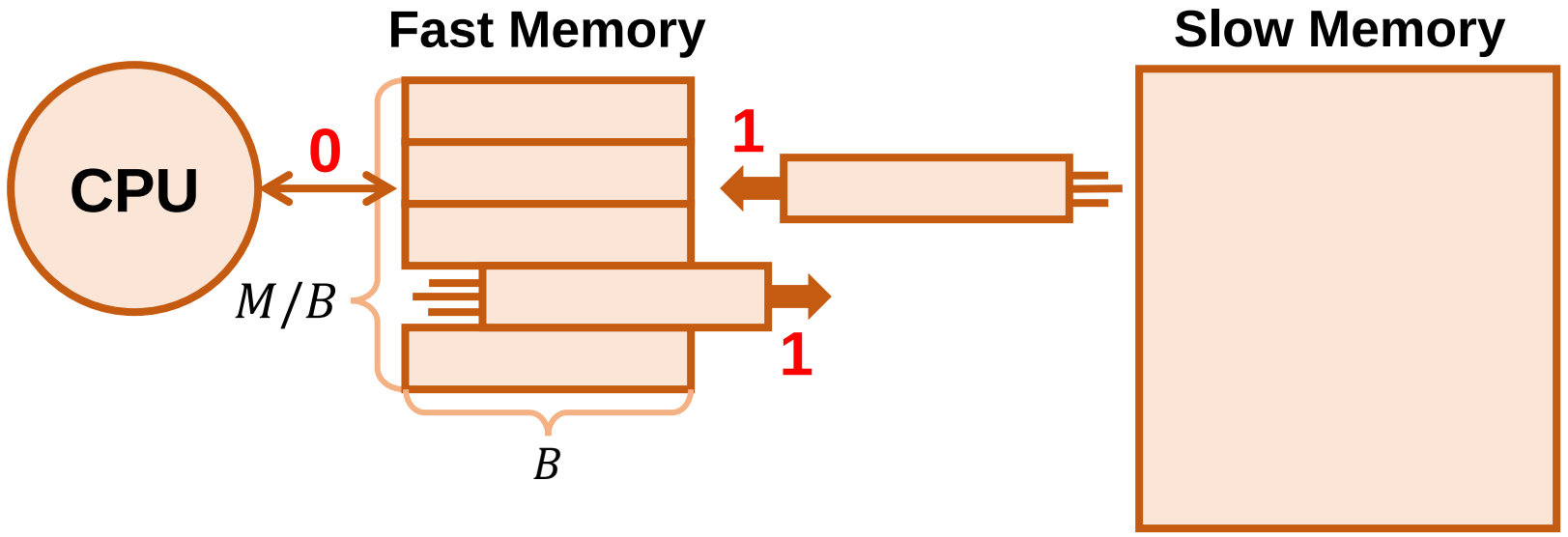
The I/O Model (External Memory-, Ideal Cache-)

- **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 (External Memory-, Ideal Cache-)

- We assume the cache is fully associative, and the it takes unit cost to load and evict a pair of blocks
- The complexity of an algorithm on the I/O model (I/O complexity) assumes an optimal cache replacement policy



I/O-efficient algorithms

- **Matrix multiplication:** $O\left(\frac{n^3}{B\sqrt{M}}\right)$ **(sequential)**
- **Next lecture: engineering matrix multiplication and check the performance**
- **Next discussion: I/O-efficient mergesort**

Putting everything together

Square-Matrix Multiplication

$$\begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$

C A B

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Assume for simplicity that $n = 2^k$.

AWS c4.8xlarge Machine Specs

Feature	Specification
Microarchitecture	Haswell (Intel Xeon E5-2666 v3)
Clock frequency	2.9 G
Processor chips	2
Processing cores	9 per processor chip
Hyperthreading	2 way
Floating-point unit	8 do fuse
Cache-line size	64 B
L1-icache	32 KB private 8-way set associative
L1-dcache	32 KB
L2-cache	256
L3-cache (LLC)	25 M
DRAM	60 GB

c4.8xlarge	$1.591 per Hour	$0.27 per Hour
c5.18xlarge	$3.06 per Hour	$0.27 per Hour
c5.24xlarge	$4.08 per Hour	$0.27 per Hour

Model	vCPU	Memory (GiB)
c5.18xlarge	72	144
c5.24xlarge	96	192

Peak = $(2.9 \times 10^9) \times 2 \times 9 \times 16 = 836$ GFLOPS

Version 1: Nested Loops in Python

```
import sys, random
from time import *

n = 4096

A = [[random.random()
      for row in xrange(n)]
     for col in xrange(n)]
B = [[random.random()
      for row in xrange(n)]
     for col in xrange(n)]
C = [[0 for row in xrange(n)]
     for col in xrange(n)]

start = time()
for i in xrange(n):
    for j in xrange(n):
        for k in xrange(n):
            C[i][j] += A[i][k] * B[k][j]
end = time()

print '%0.6f' % (end - start)
```

A. 6 milliseconds

B. 6 seconds

C. 6 minutes

D. 6 hours

E. 6 weeks

Version 1: Nested Loops in Python

```
import sys, random
from time import *

n = 4096

A = [[random.random()
      for row in xrange(n)]
     for col in xrange(n)]
B = [[random.random()
      for row in xrange(n)]
     for col in xrange(n)]
C = [[0 for row in xrange(n)]
     for col in xrange(n)]

start = time()
for i in xrange(n):
    for j in xrange(n):
        for k in xrange(n):
            C[i][j] += A[i][k] * B[k][j]
end = time()

print '%0.6f' % (end - start)
```

Running time = 21042 seconds ≈ 6 hours

Is this fast?

Should we expect more?

Version 1: Nested Loops in Python

```
import sys, random
from time import *

n = 4096

A = [[random.random()
      for row in xrange(n)]
     for col in xrange(n)]
B = [[random.random()
      for row in xrange(n)]
     for col in xrange(n)]
C = [[0 for row in xrange(n)]
     for col in xrange(n)]

start = time()
for i in xrange(n):
    for j in xrange(n):
        for k in xrange(n):
            C[i][j] += A[i][k]
end = time()

print '%0.6f' % (end - start)
```

Running time = 21042 seconds ≈ 6 hours

Is this fast?

Should we expect more?

Back-of-the-envelope calculation

$2n^3 = 2(2^{12})^3 = 2^{37}$ floating-point operations

Running time = 21042 seconds

∴ Python gets $2^{37}/21042 \approx 6.25$ MFLOPS

Peak ≈ 836 GFLOPS

Python gets ≈ 0.00075% of peak

Version 2: Java

```java
import java.util.Random;

public class mm_java {
  static int n = 4096;
  static double[][] A = new double[n][n];
  static double[][] B = new double[n][n];
  static double[][] C = new double[n][n];

  public static void main(String[] args) {
    Random r = new Random();

    for (int i=0; i<n; i++) {
      for (int j=0; j<n; j++) {
        A[i][j] = r.nextDouble();
        B[i][j] = r.nextDouble();
        C[i][j] = 0;
      }
    }

    long start = System.nanoTime();

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

    long stop = System.nanoTime();

    double tdiff = (stop - start) * 1e-9;
    System.out.println(tdiff);
  }
}
```

Running time = 2,738 seconds ≈ 46 minutes

⋯ about 8.8× faster than Python.

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

Version 3: C

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>

#define n 4096
double A[n][n];
double B[n][n];
double C[n][n];

float tdiff(struct timeval *start,
            struct timeval *end) {
  return (end->tv_sec-start->tv_sec) +
    1e-6*(end->tv_usec-start->tv_usec);
}

int main(int argc, const char *argv[]) {
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      A[i][j] = (double)rand() / (double)RAND_MAX;
      B[i][j] = (double)rand() / (double)RAND_MAX;
      C[i][j] = 0;
    }
  }

  struct timeval start, end;
  gettimeofday(&start, NULL);

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

  gettimeofday(&end, NULL);
  printf("%0.6f\n", tdiff(&start, &end));
  return 0;
}
```

Using the Clang/LLVM 5.0 compiler

Running time = 1,156 seconds ≈ 19 minutes

About 2× faster than Java and
about 18× faster than Python

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

Where We Stand So Far

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.007	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.119	0.014

Why is Python so slow and C so fast?

- Python is interpreted
- C is compiled directly to machine code
- Java is compiled to byte-code, which is then interpreted and just-in-time (JIT) compiled to machine code

Loop Order

We can change the order of the loops in this program without affecting its correctness

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

Loop Order

We can change the order of the loops in this program without affecting its correctness

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

Does the order of loops matter for performance?

Performance of Different Orders

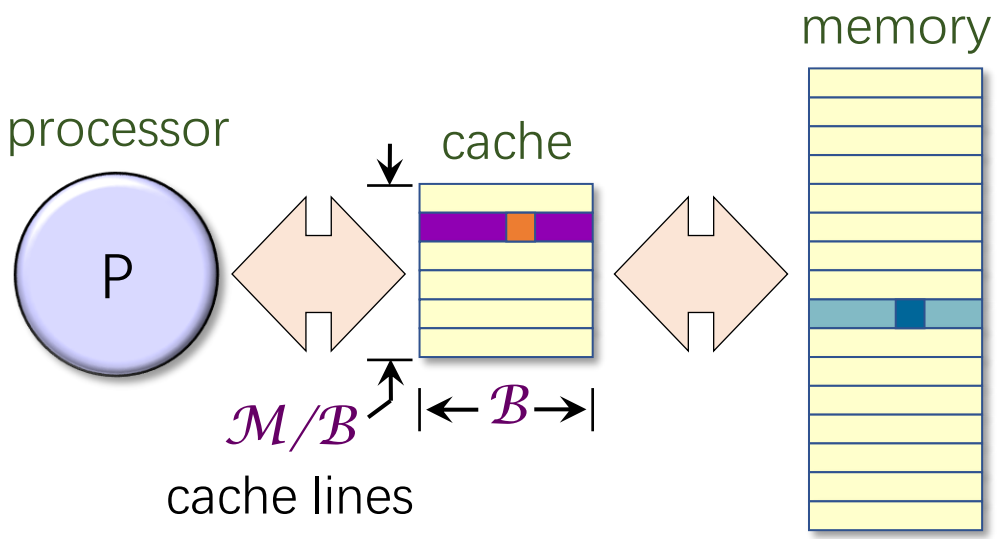
Loop order (outer to inner)	Running time (s)
i, j, k	1155.77
i, k, j	177.68
j, i, k	1080.61
j, k, i	3056.63
k, i, j	179.21
k, j, i	3032.82

- Loop order affects running time by a factor of **18**!
- What's going on?!

Hardware Caches

Each processor reads and writes main memory in contiguous blocks, called *cache lines*

- **Previously accessed cache lines are stored in a smaller memory, called a *cache*, that sits near the processor**
- ***Cache hits* — accesses to data in cache — are fast**
- ***Cache misses* — accesses to data not in cache — are slow**



Memory Layout of Matrices

In this matrix-multiplication code, matrices are laid out in memory in *row-major order*

Matrix

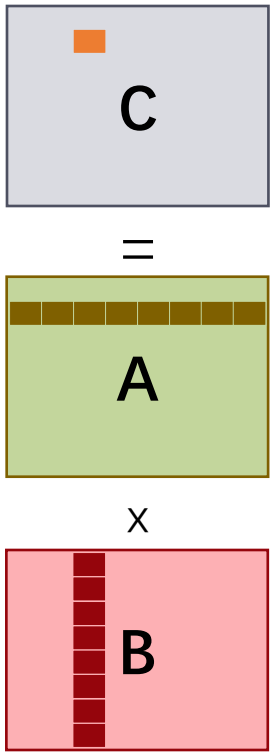
Row 1
Row 2
Row 3
Row 4
Row 5
Row 6
Row 7
Row 8

What does this layout imply about the performance of different loop orders?

Access Pattern for Order i, j, k

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

Running time:
1155.77s

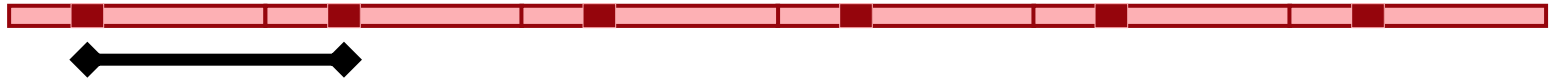


In-memory layout

Excellent spatial locality

Good spatial locality

Poor spatial locality

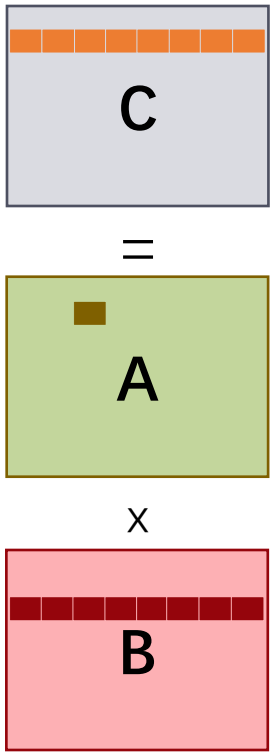


4096 elements apart

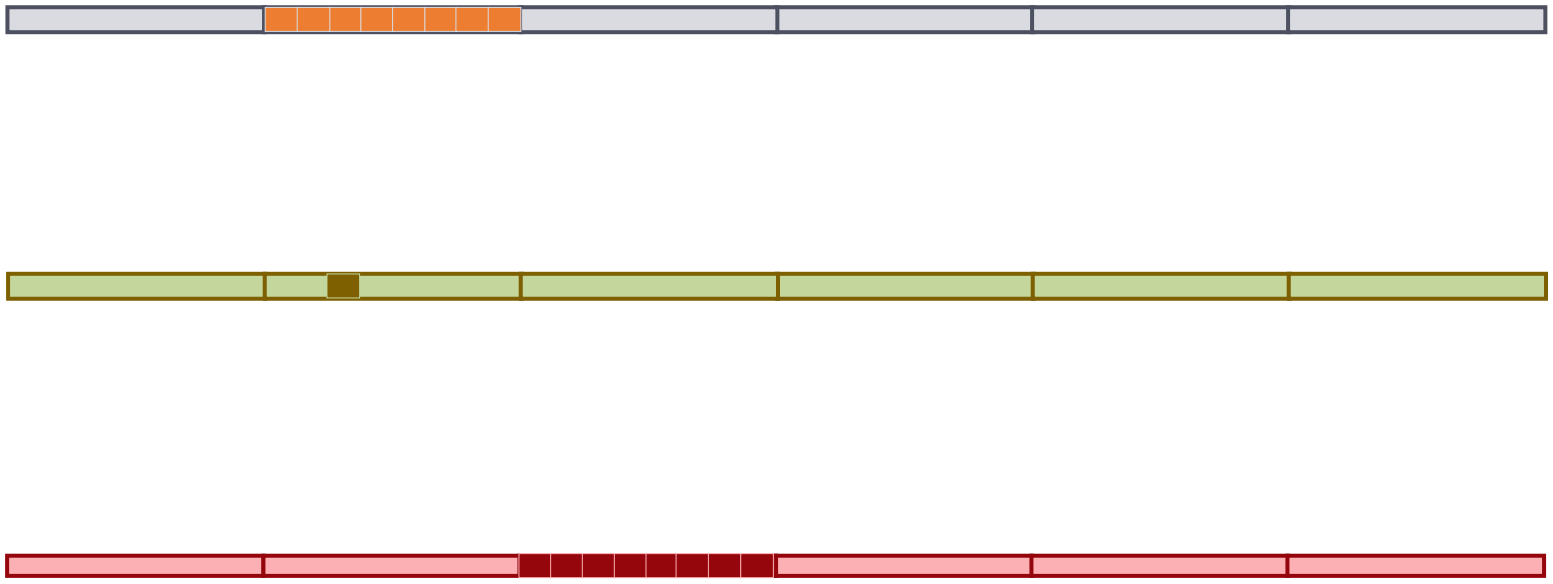
Access Pattern for Order i, k, j

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

Running time:
177.68s



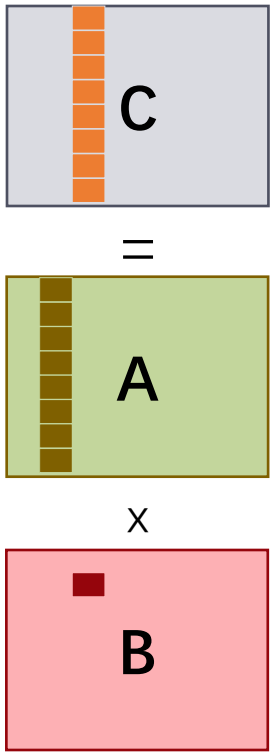
In-memory layout



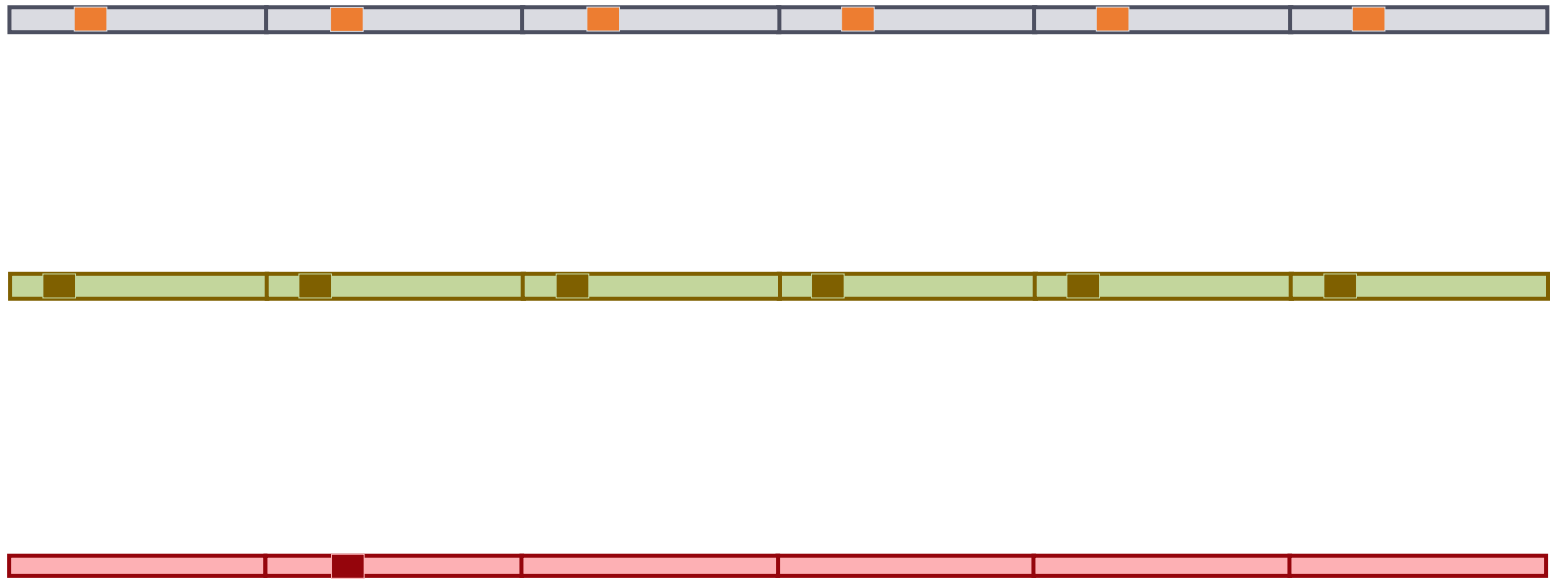
Access Pattern for Order j, k, i

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

Running time:
3056.63s



In-memory layout



Performance of Different Orders

We can measure the effect of different access patterns using the Cachegrind cache simulator:

```
$ valgrind --tool=cachegrind ./mm
```

Loop order (outer to inner)	Running time (s)	Last-level-cache miss rate
i, j, k	1155.77	7.7%
i, k, j	177.68	1.0%
j, i, k	1080.61	8.6%
j, k, i	3056.63	15.4%
k, i, j	179.21	1.0%
k, j, i	3032.82	15.4%

Version 4: Interchange Loops

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093

What other simple changes we can try?

Compiler Optimization

Clang provides a collection of optimization switches. You can specify a switch to the compiler to ask it to optimize

Opt. level	Meaning	Time (s)
`-O0`	Do not optimize	177.54
`-O1`	Optimize	66.24
`-O2`	Optimize even more	54.63
`-O3`	Optimize yet more	55.58

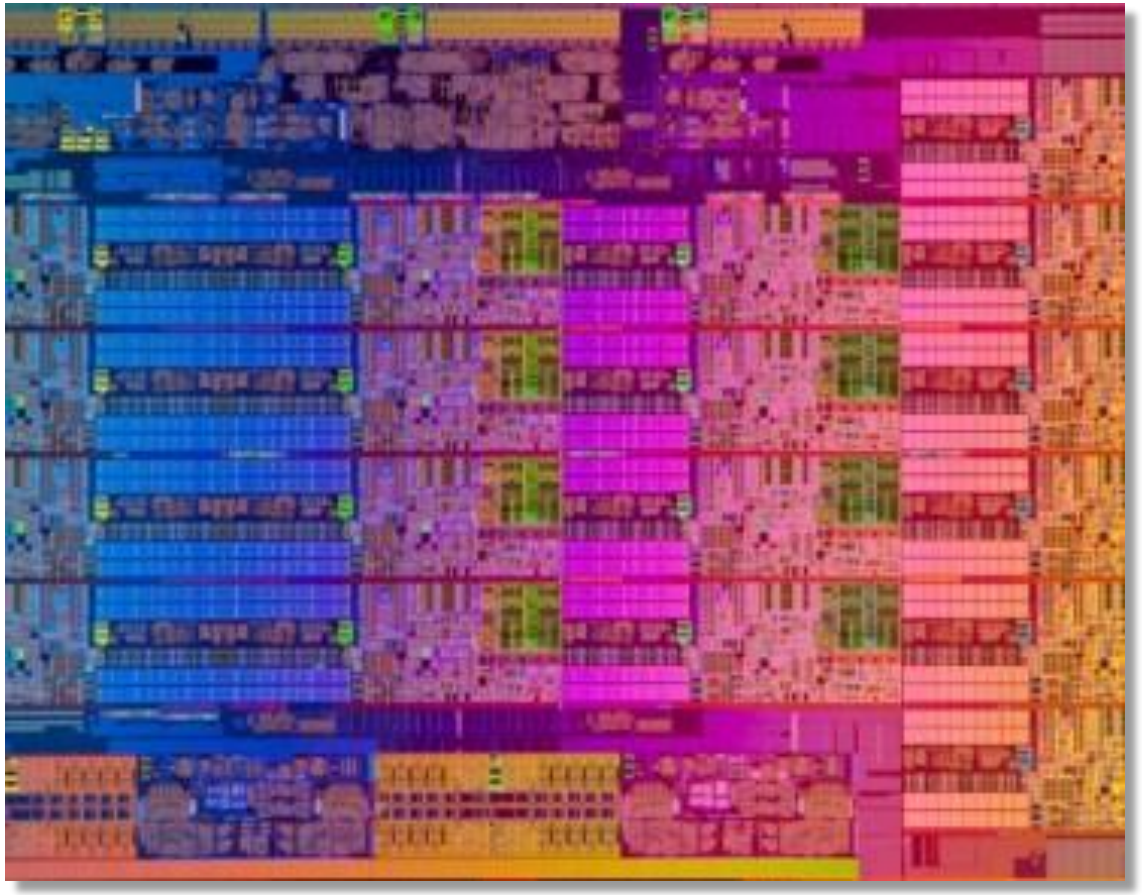
Version 5: Optimization Flags

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301

With simple code and compiler technology, we can achieve 0.3% of the peak performance of the machine

What's causing the low performance?

Multicore Parallelism



Intel Haswell E5:
9 cores per chip

The AWS test machine has 2 of these chips

We're running on just 1 of the 18 parallel-processing cores on this system. Let's use them all!

Parallel Loops

The cilk_for loop allows all iterations of the loop to execute in parallel

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

These loops can be (easily) parallelized.

Which parallel version works best?

Experimenting with Parallel Loops

Parallel `i` loop

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

Running time: 3.18s

Parallel `j` loop

```
for (int i = 0; i < n; ++i)
  for (int k = 0
    cilk_for (in
      C[i][j] +=
```

Running time: 531.71s

Rule of Thumb

Parallelize outer loops rather than inner loops

Parallel `i` and `j`

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

Running time: 10.64s

Version 6: Parallel Loops

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408

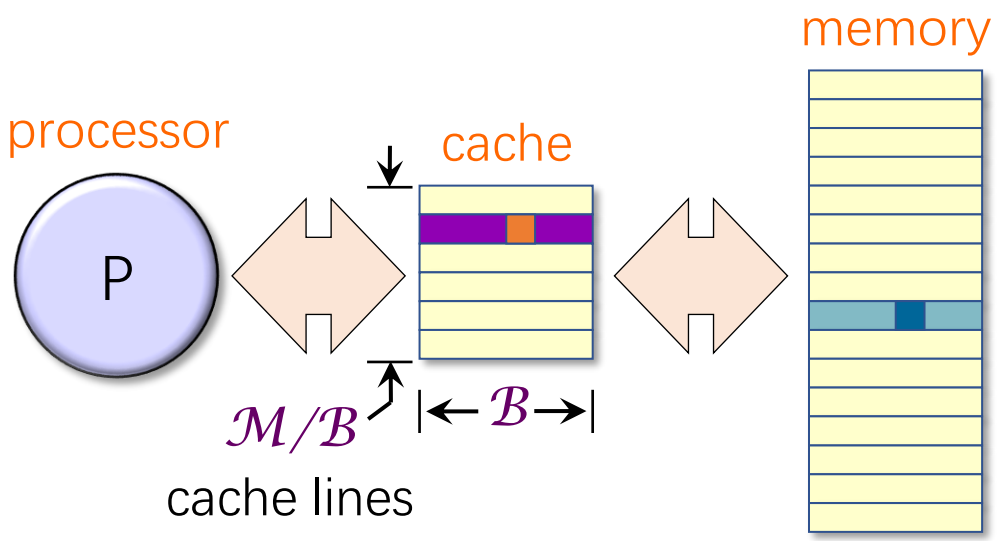
Using parallel loops gets us almost 18× speedup on 18 cores!
(Disclaimer: Not all code is so easy to parallelize effectively.)

Why are we still getting just 5% of peak?

Hardware Caches, Revisited

IDEA: Restructure the computation to reuse data in the cache as much as possible

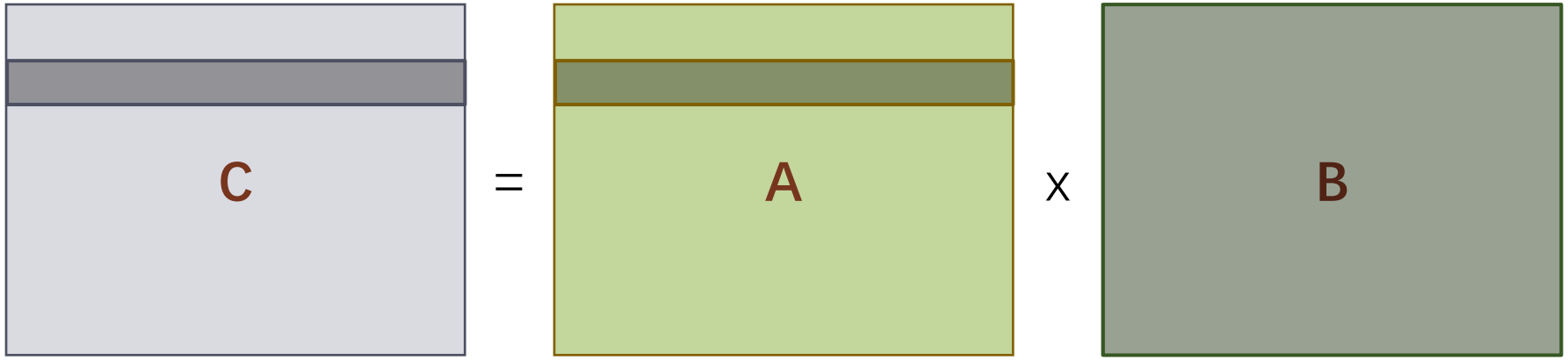
- **Cache misses are slow, and cache hits are fast**
- **Try to make the most of the cache by reusing the data that's already there**



Data Reuse: Loops

How many memory accesses must the looping code perform to fully compute 1 row of C?

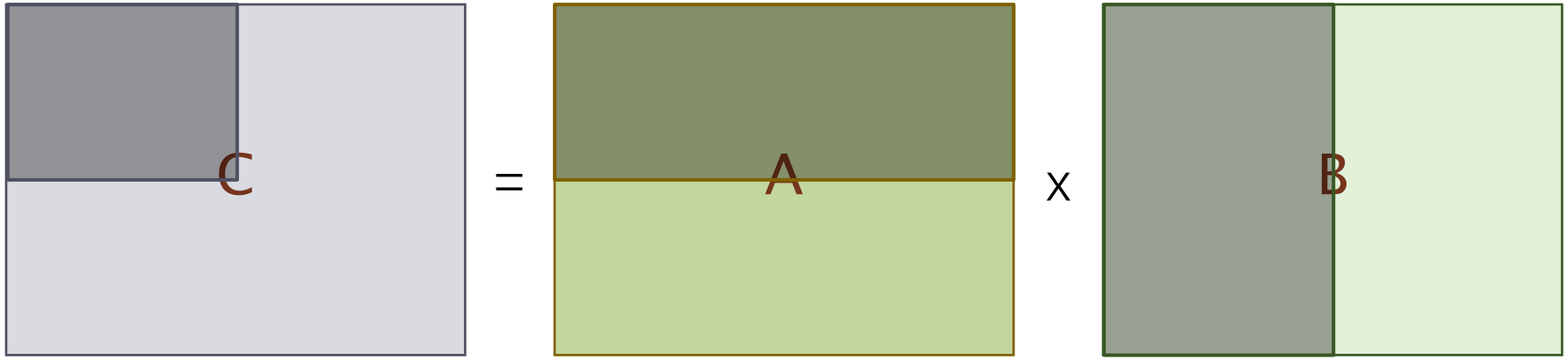
- **4096 * 1 = 4096 writes to C,**
- **4096 * 1 = 4096 reads from A, and**
- **4096 * 4096 = 16,777,216 reads from B, which is**
- **16,785,408 memory accesses total**



Data Reuse: Blocks

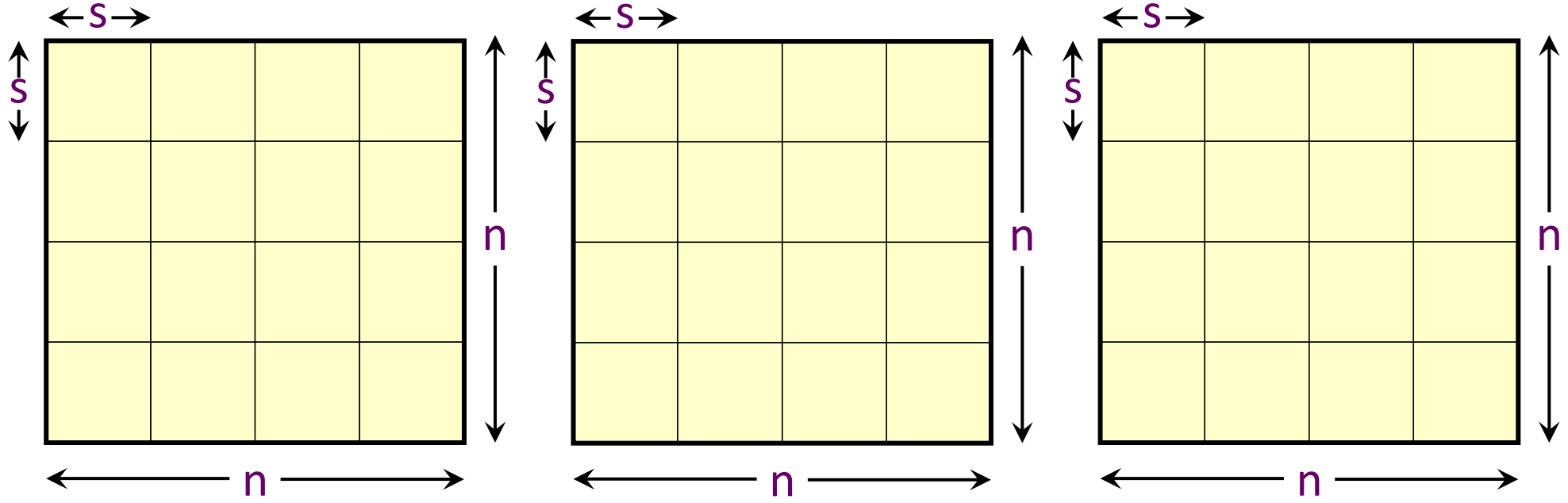
How about to compute a 64 × 64 block of C?

- **64 · 64 = 4096 writes to C,**
- **64 · 4096 = 262,144 reads from A, and**
- **4096 · 64 = 262,144 reads from B, or**
- **528,384 memory accesses total**



Tiled Matrix Multiplication

```
cilk_for (int ih = 0; ih < n; ih += s)
  cilk_for (int jh = 0; jh < n; jh += s)
    for (int kh = 0; kh < n; kh += s)
      for (int il = 0; il < s; ++il)
        for (int kl = 0; kl < s; ++kl)
          for (int jl = 0; jl < s; ++jl)
            C[ih+il][jh+jl] += A[ih+il][kh+kl] * B[kh+kl][jh+jl];
```

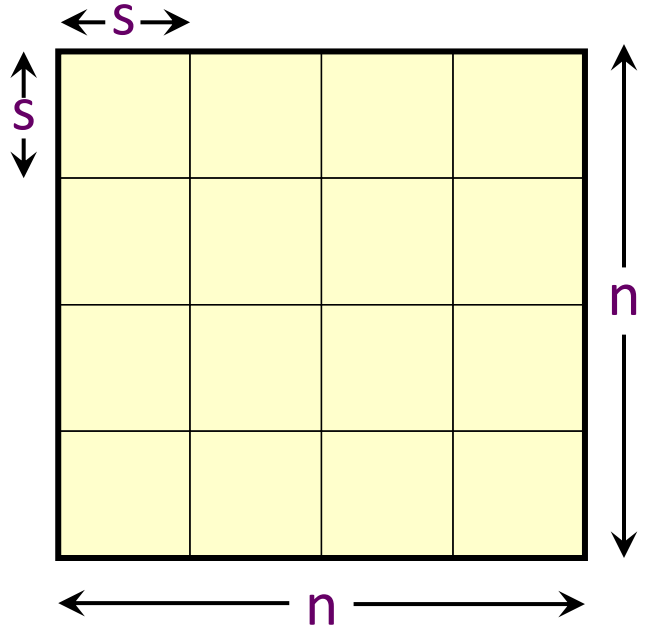


Tiled Matrix Multiplication

```
cilk_for (int ih = 0; ih < n; ih += s)
  cilk_for (int jh = 0; jh < n; jh += s)
    for (int kh = 0; kh < n; kh += s)
      for (int il = 0; il < s; ++il)
        for (int kl = 0; kl < s; ++kl)
          for (int jl = 0; jl < s; ++jl)
            C[ih+il][jh+jl] += A[ih+il][kh+kl] * B[kh+kl][jh+jl];
```

Tuning parameter

How do we find the right value of s? Experiment!



Tile size	Running time (s)
4	6.74
8	2.76
16	2.49
32	1.74
64	2.33
128	2.13

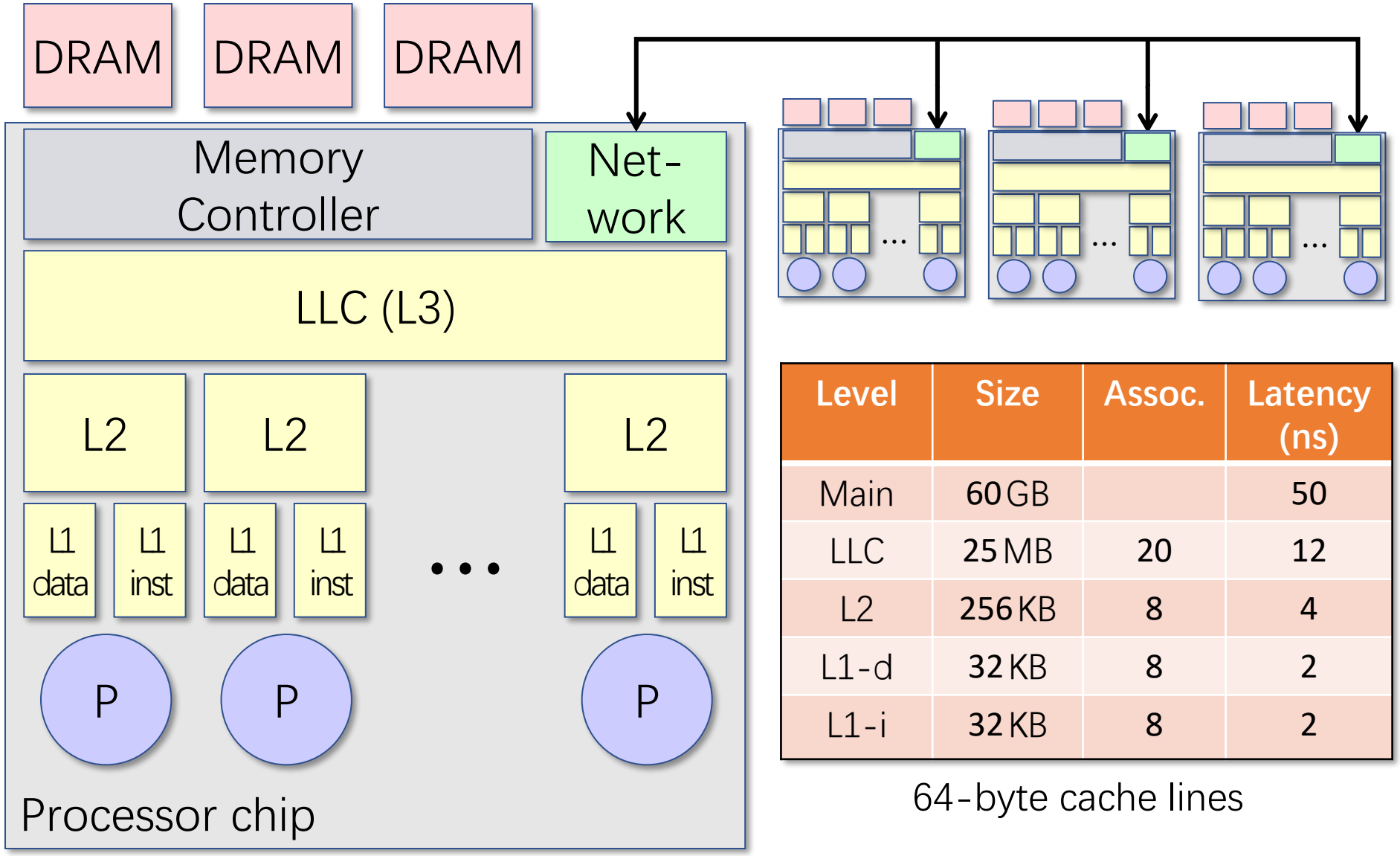
Version 7: Tiling

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408
7	+ tiling	1.74	1.70	11,772	76.782	9.184

Implementation	Cache references (millions)	L1-d cache misses (millions)	Last-level cache misses (millions)
Parallel loops	104,090	17,220	8,600
+ tiling	64,690	11,777	416

The tiled implementation performs about 62% fewer cache references and incurs 68% fewer cache misses.

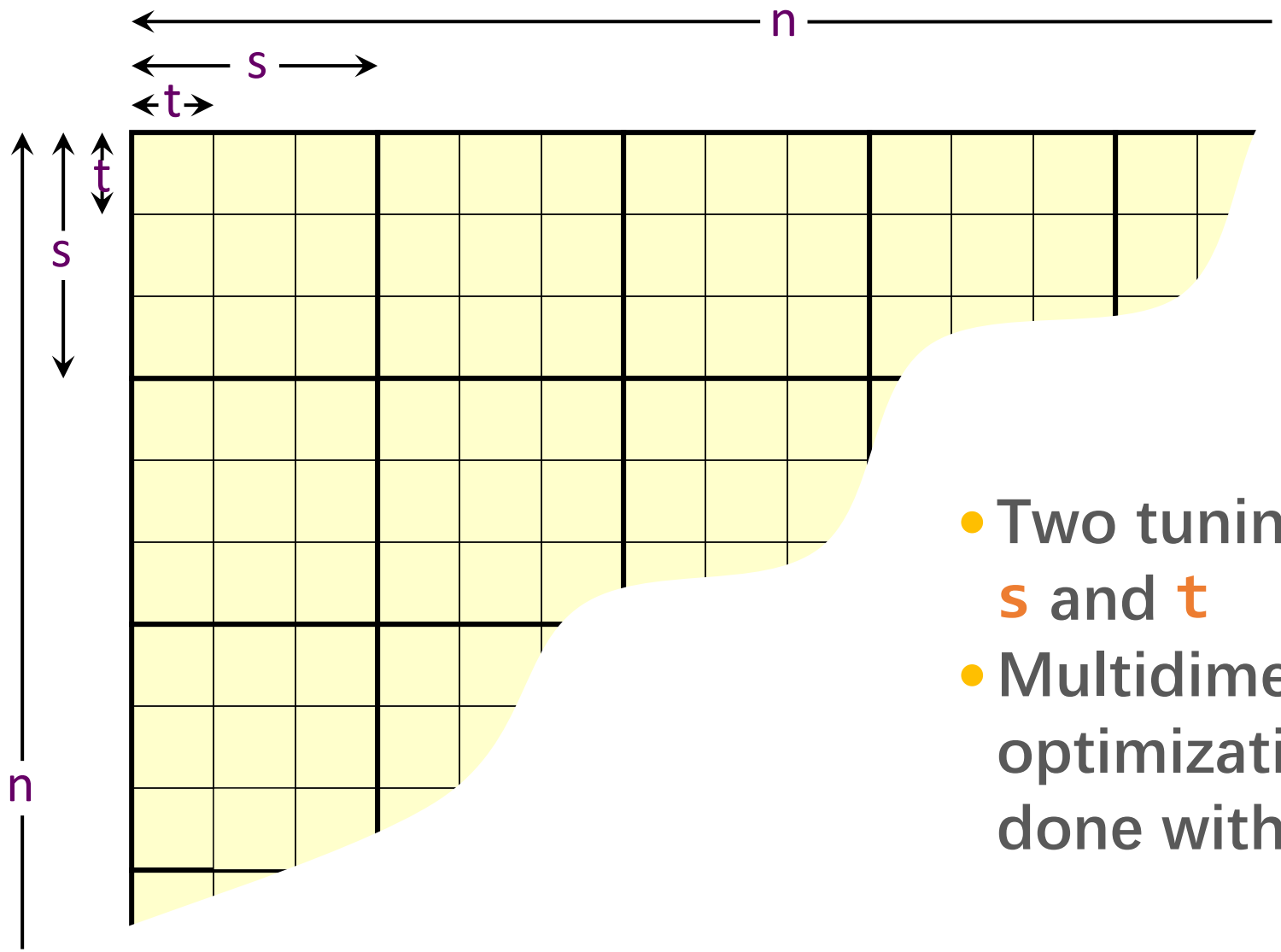
Multicore Cache Hierarchy



Level	Size	Assoc.	Latency (ns)
Main	60 GB		50
LLC	25 MB	20	12
L2	256 KB	8	4
L1-d	32 KB	8	2
L1-i	32 KB	8	2

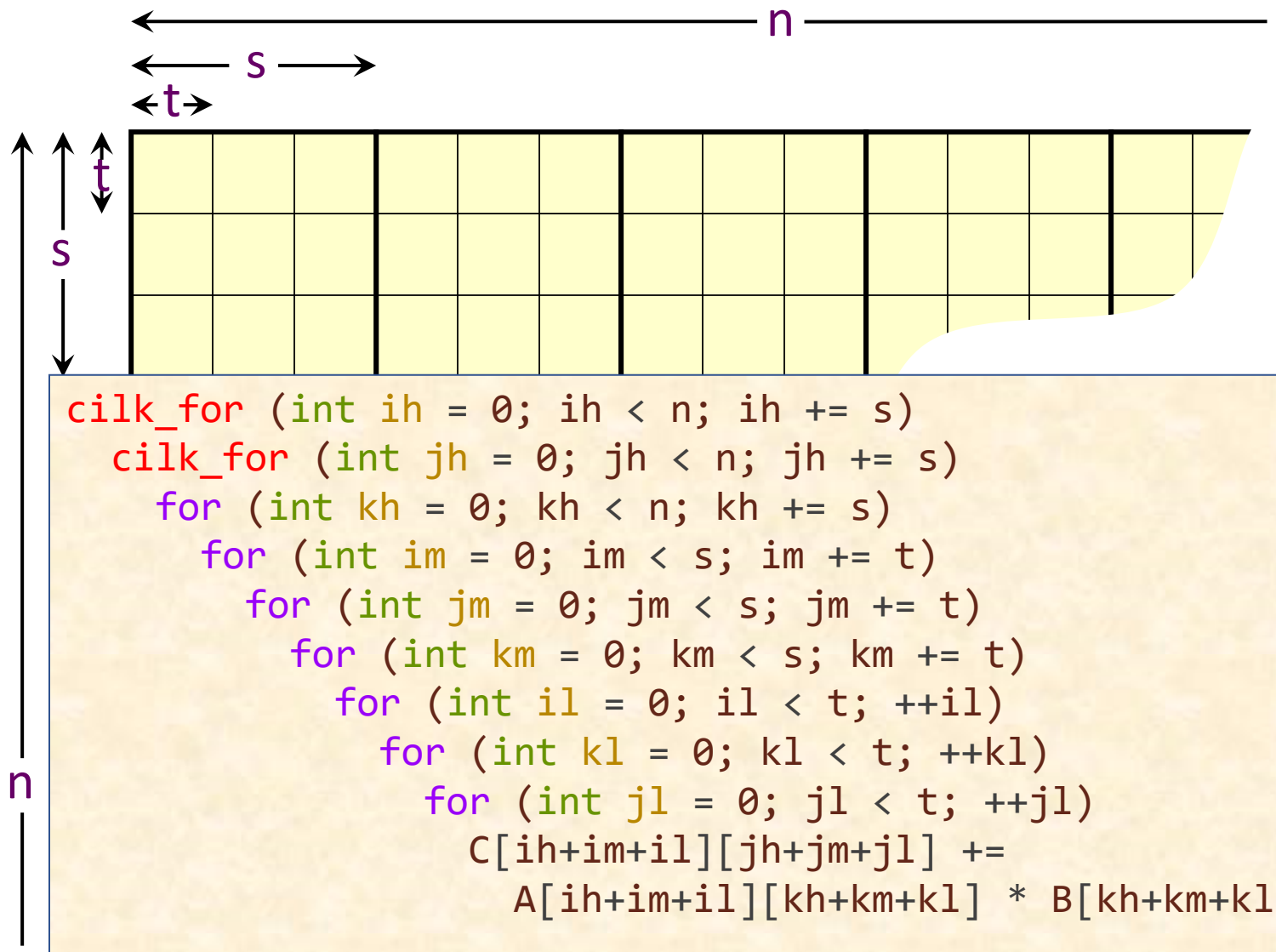
64-byte cache lines

Tiling for a Two-Level Cache



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

Tiling for a Two-Level Cache




```
cilk_for (int ih = 0; ih < n; ih += s)
  cilk_for (int jh = 0; jh < n; jh += s)
    for (int kh = 0; kh < n; kh += s)
      for (int im = 0; im < s; im += t)
        for (int jm = 0; jm < s; jm += t)
          for (int km = 0; km < s; km += t)
            for (int il = 0; il < t; ++il)
              for (int kl = 0; kl < t; ++kl)
                for (int jl = 0; jl < t; ++jl)
                  C[ih+im+il][jh+jm+jl] +=
                    A[ih+im+il][kh+km+kl] * B[kh+km+kl][jh+jm+jl];
```

Recursive Matrix Multiplication

IDEA: Tile for every power of 2 simultaneously

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$= \begin{pmatrix} A_{00}B_{00} & A_{00}B_{01} \\ A_{10}B_{00} & A_{10}B_{01} \end{pmatrix} + \begin{pmatrix} A_{01}B_{10} & A_{01}B_{11} \\ A_{11}B_{10} & A_{11}B_{11} \end{pmatrix}$$

8 multiplications of $n/2 \times n/2$ matrices
1 addition of $n \times n$ matrices

Recursive Parallel Matrix Multiply

The child function call is spawned, meaning it may execute in parallel with the parent caller

Control may not pass this point until all spawned children have returned.

```
void mm_dac(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 <= 1) {
    *C += *A * *B;
  } else {
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
               mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_sync;
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
               mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    cilk_sync;
  }
}
```

Recursive Parallel Matrix Multiply

The base case is too small. We must coarsen the recursion to overcome function-call overheads.

Running time: 93.93s

··· about 50× **slower** than the last version!

```
void mm_dac(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 <= 1) {
    *C += *A * *B;
  } else {
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
               mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_sync;
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
               mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    cilk_sync;
  }
}
```

Coarsening The Recursion

Just one tuning parameter, for the size of the base case.

```
void mm_dac(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,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
               mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_sync;
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
               mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    cilk_sync;
  }
}
```

Coarsening The Recursion

```
void mm_dac(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,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);

    dac(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
```

```
void mm_base(double *restrict C, int n_C,
             double *restrict A, int n_A,
             double *restrict B, int n_B,
             int n)
{ // C = A * B
  for (int i = 0; i < n; ++i)
    for (int k = 0; k < n; ++k)
      for (int j = 0; j < n; ++j)
        C[i*n_C+j] += A[i*n_A+k] * B[k*n_B+j];
}
```

Coarsening The Recursion

Base-case size	Running time (s)
4	3.00
8	1.34
16	1.34
32	1.30
64	1.95
128	2.08

```
void mm_dac(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,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
               mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_sync;
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
               mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    cilk_sync;
  }
}
```

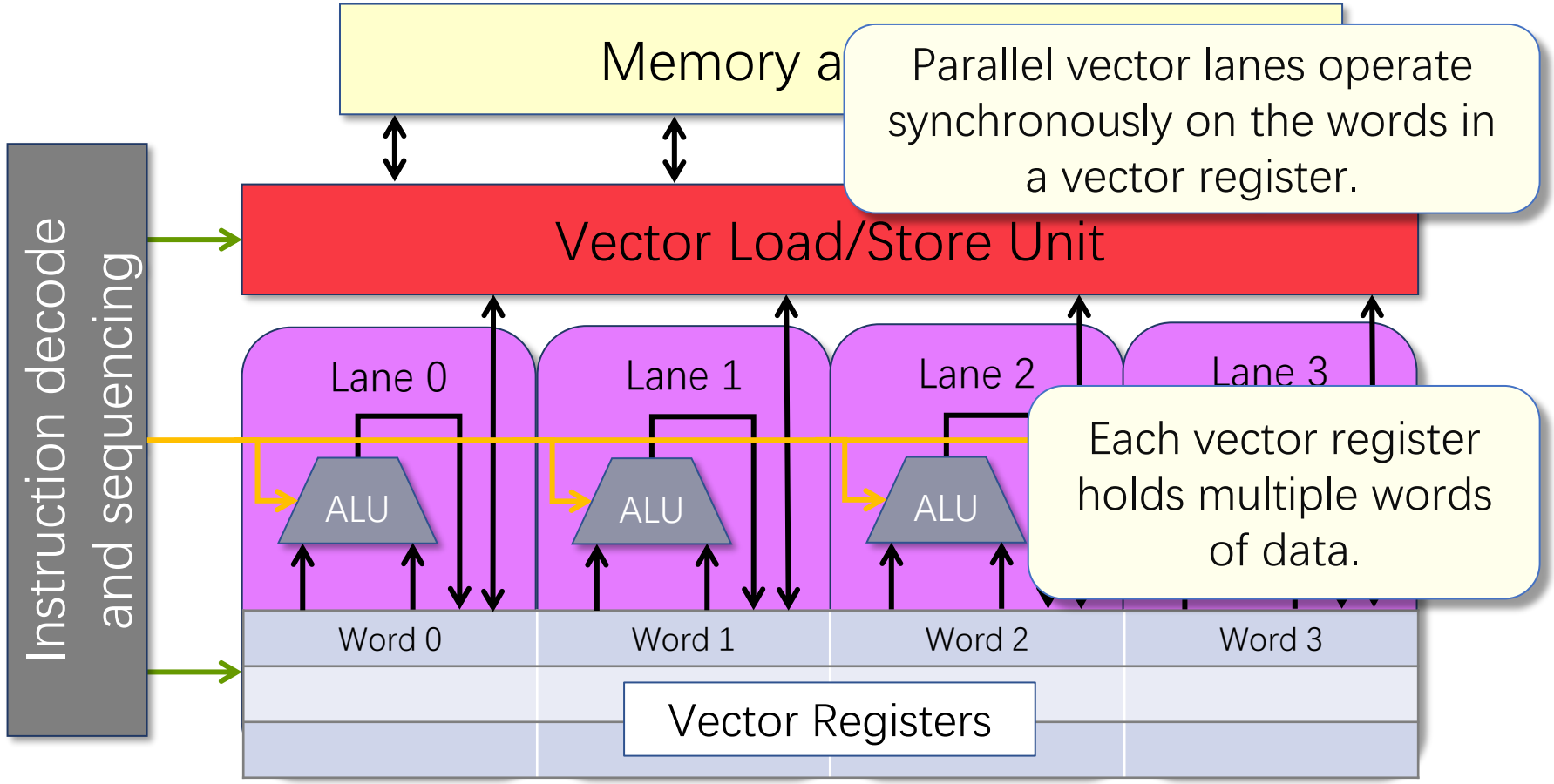
8. Divide-and-Conquer

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408
7	+ tiling	1.79	1.70	11,772	76.782	9.184
8	Parallel divide-and-conquer	1.30	1.38	16,197	105.722	12.646

Implementation	Cache references (millions)	L1-d cache misses (millions)	Last-level cache misses (millions)
Parallel loops	104,090	17,220	8,600
+ tiling	64,690	11,777	416
Parallel divide-and-conquer	58,230	9,407	64

Vector Hardware

Modern microprocessors incorporate vector hardware to process data in single-instruction stream, multiple-data stream (SIMD) fashion



Compiler Vectorization

Clang/LLVM uses vector instructions automatically when compiling at optimization level `-O2` or higher

Can be checked in a *vectorization report* as follows:

```
$ clang -O3 -std=c99 mm.c -o mm -Rpass=vector
mm.c:42:7: remark: vectorized loop (vectorization width: 2,
interleaved count: 2) [-Rpass=loop-vectorize]
      for (int j = 0; j < n; ++j) {
      ^
```

Many machines don't support the newest set of vector instructions, however, so the compiler uses vector instructions conservatively by default

Vectorization Flags

Programmers can direct the compiler to use modern vector instructions using compiler flags such as the following:

- -mavx: Use Intel AVX vector instructions
- -mavx2: Use Intel AVX2 vector instructions
- -mfma: Use fused multiply-add vector instructions
- -march=<string>: Use whatever instructions are available on the specified architecture
- -march=native: Use whatever instructions are available on the architecture of the machine doing compilation

Due to restrictions on floating-point arithmetic, additional flags, such as -ffast-math, might be needed for these vectorization flags to have an effect

Version 9: Compiler Vectorization

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408
7	+ tiling	1.79	1.70	11,772	76.782	9.184
8	Parallel divide-and-conquer	1.30	1.38	16,197	105.722	12.646
9	+ compiler vectorization	0.70	1.87	30,272	196.341	23.486

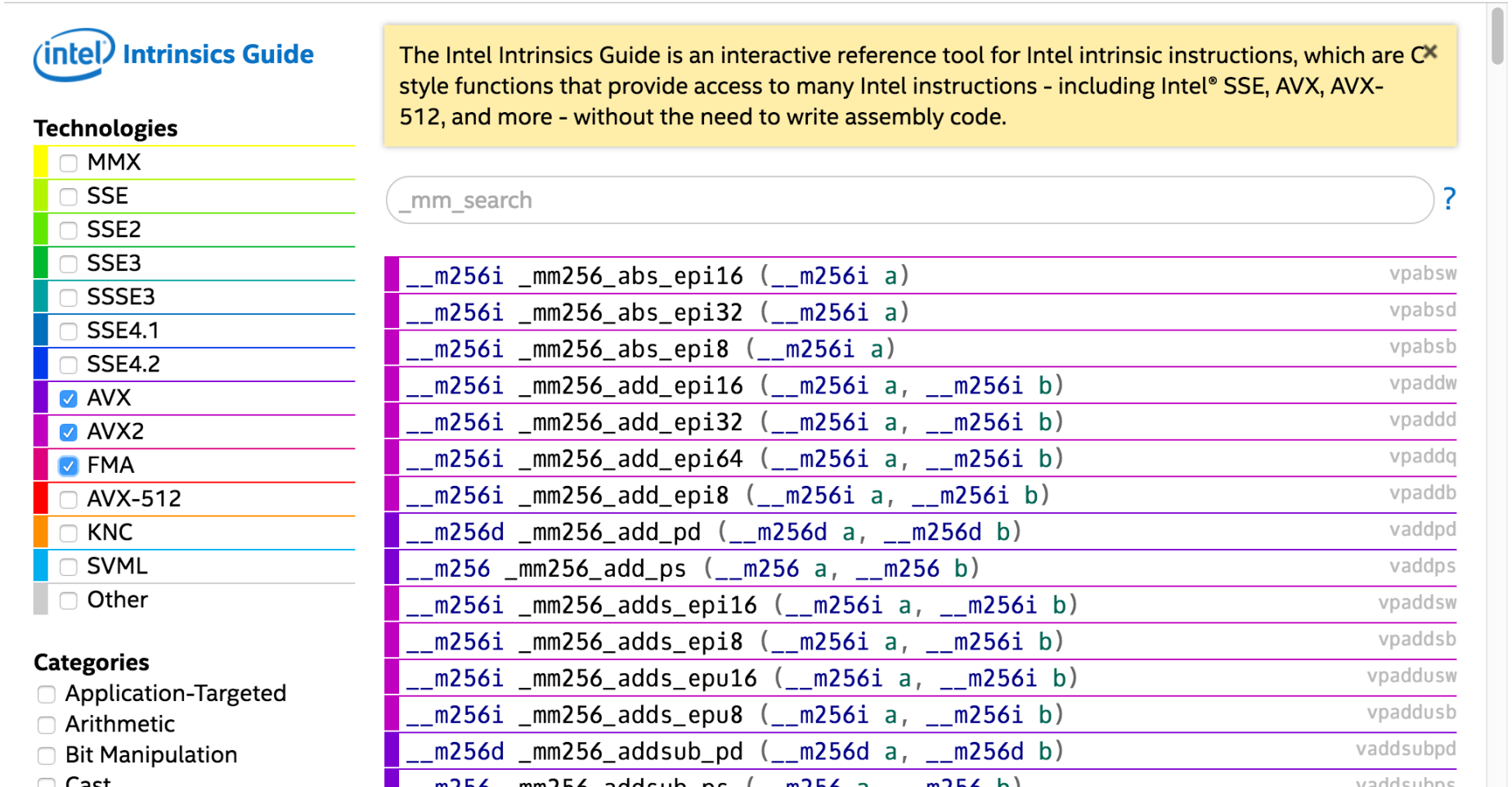
Using the flags `-march=native -ffast-math` nearly doubles the program's performance!

Can we be smarter than the compiler?

AVX Intrinsic Instructions

- **Intel provides C-style functions, called *intrinsic instructions*, that provide direct access to hardware vector operations:**

https://software.intel.com/sites/landingpage/IntrinsicsGuide/



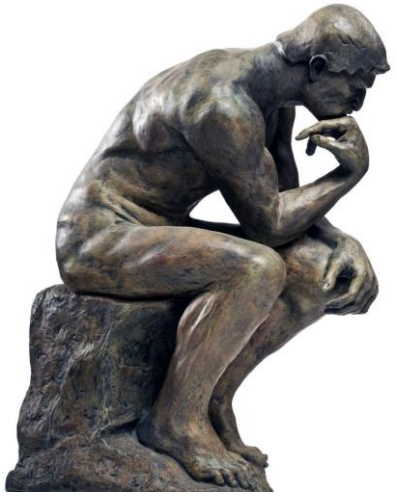
Plus More Optimizations

We can apply several more insights and performance-engineering tricks to make this code run faster, including:

- **Preprocessing**
- **Matrix transposition**
- **Data alignment**
- **Memory-management optimizations**
- **A clever algorithm for the base case that uses AVX intrinsic instructions explicitly**

Plus Performance Engineering

Think,



code,

run, run, run…

…to test and measure many different implementations

Version 10: AVX Intrinsics

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408
7	+ tiling	1.79	1.70	11,772	76.782	9.184
8	Parallel divide-and-conquer	1.30	1.38	16,197	105.722	12.646
9	+ compiler vectorization	0.70	1.87	30,272	196.341	23.486
10	+ AVX intrinsics	0.39	1.76	53,292	352.408	41.677

Version 11: Final Reckoning

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408
7	+ tiling	1.79	1.70	11,772	76.782	9.184
8	Parallel divide-and-conquer	1.30	1.38	16,197	105.722	12.646
9	+ compiler vectorization	0.70	1.87	30,272	196.341	23.486
10	+ AVX intrinsics	0.39	1.76	53,292	352.408	41.677
11	Intel MKL	0.41	0.97	51,497	335.217	40.098

Version 10 is competitive with Intel's professionally engineered Math Kernel Library!

Engineering the Performance of your Algorithms

Gas economy MPG

53,292×

- You won't generally see the magnitude of performance improvement we obtained for matrix multiplication
- But in this course, you will learn how to print the currency of performance all by yourself

Design and Analysis of Algorithms

- Work W, depth D, I/O cost Q (sequential / random)
- Parallelism for work: $\frac{W}{P}$
- Time for I/O: $\max\left(\frac{Q}{P}, \frac{Q}{B_{max}}\right)$
- Number of steals: $O(PD)$
- Most combinatorial algorithms are I/O bottlenecked

Overall Structure in this Course

Performance Engineering

Parallelism
I/O efficiency
New Bentley rules
Brief overview of architecture

Algorithm Engineering

Sorting / Semisorting
Matrix multiplication
Graph algorithms
Geometry Algorithms

EE/CS217 GPU Architecture and Parallel Programming
CS211 High Performance Computing
CS213 Multiprocessor Architecture and Programming (Stanford CS149)
CS247 Principles of Distributed Computing