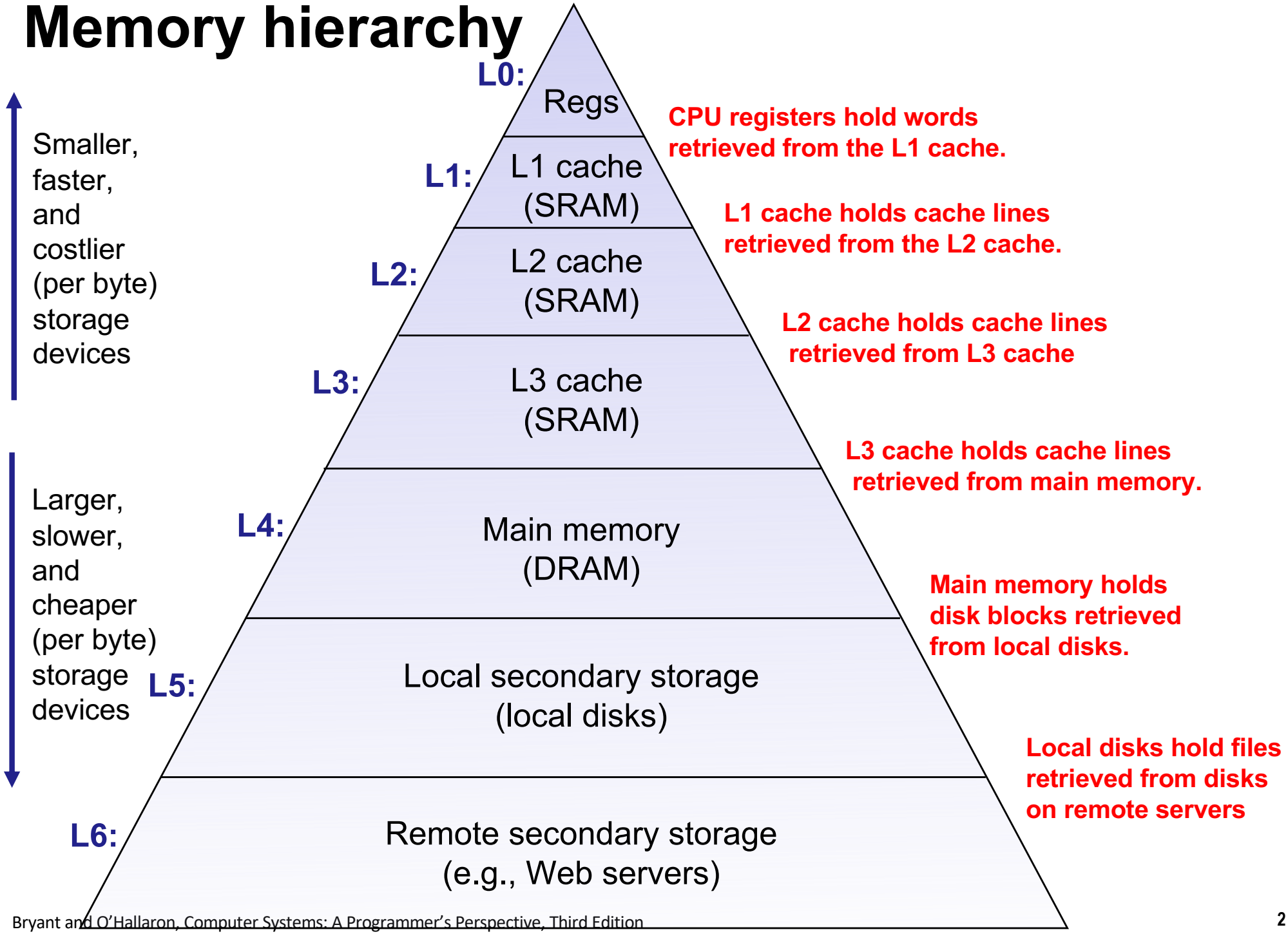
# Cache Memories and locality

IN3030
Spring March 3, 2022

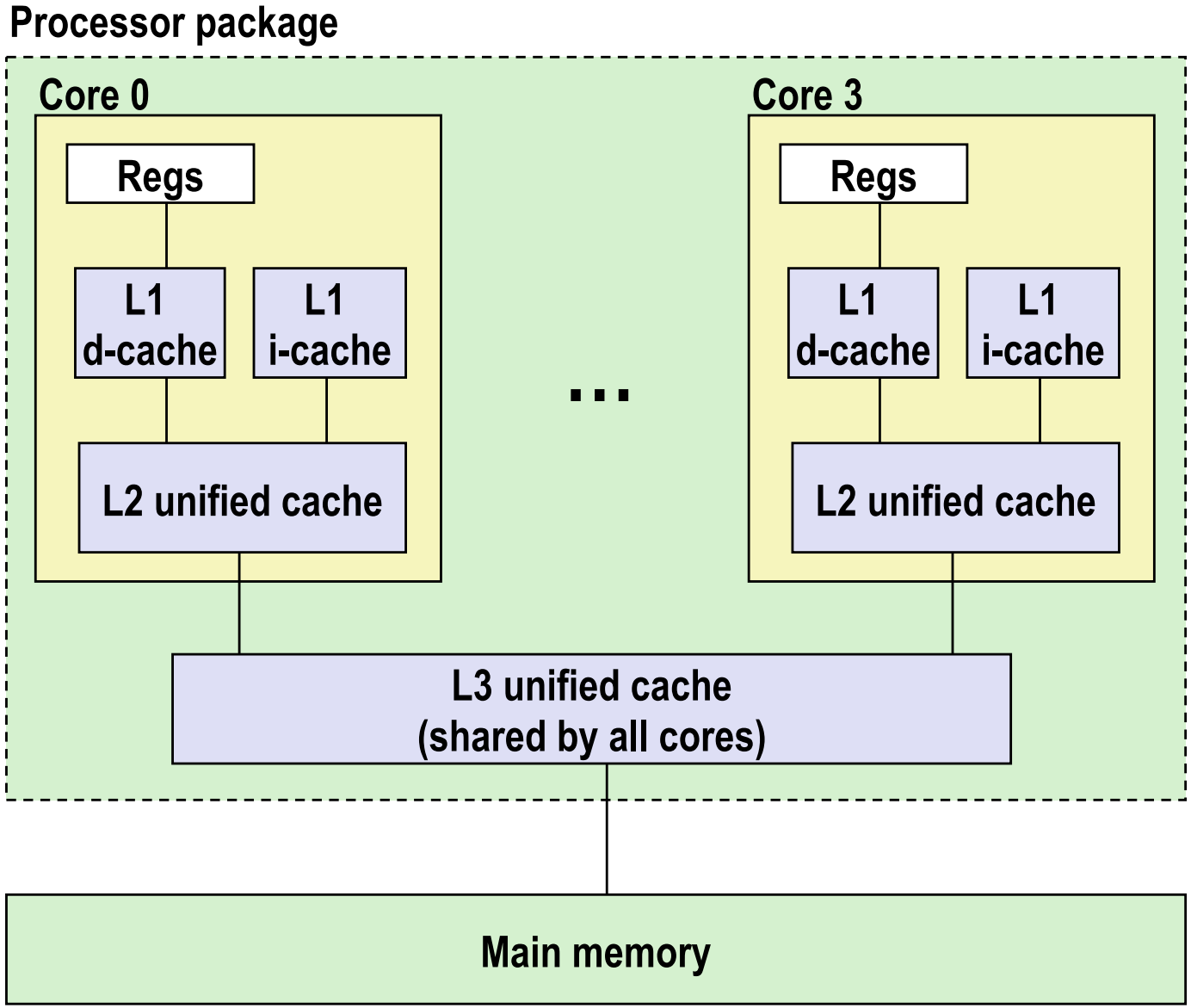
**Michael Kirkedal Thomsen**

*Based on slides by Randal E. Bryant and David R. O'Hallaron*

# Memory hierarchy



# Intel Core i7 Cache Hierarchy



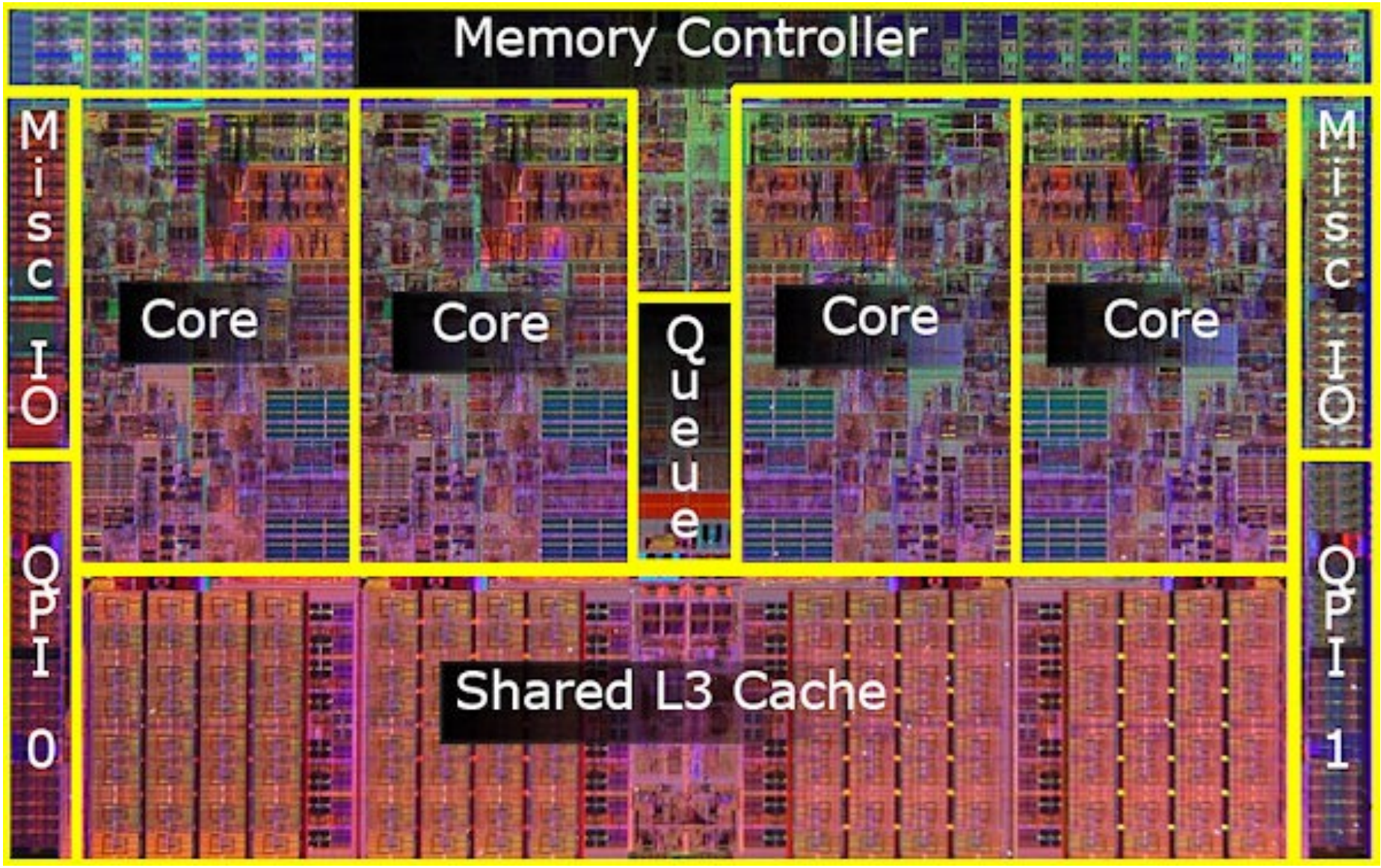

**L1 i-cache and d-cache:**
32 KB, 8-way,
Access: 4 cycles

**L2 unified cache:**
256 KB, 8-way,
Access: 10 cycles

**L3 unified cache:**
8 MB, 16-way,
Access: 40-75 cycles

**Block size**: 64 bytes for all caches.

# Intel Core i7 - 4-core CPU



# Cache Performance Metrics

- **Miss Rate**
  - Fraction of memory references not found in cache (misses / accesses) = 1 – hit rate
  - Typical numbers (in percentages):
    - 3-10% for L1
    - can be quite small (e.g., < 1%) for L2, depending on size, etc.
- **Hit Time**
  - Time to deliver a line in the cache to the processor
    - includes time to determine whether the line is in the cache
  - Typical numbers:
    - 4 clock cycle for L1
    - 10 clock cycles for L2
- **Miss Penalty**
  - Additional time required because of a miss
    - typically 50-200 cycles for main memory (Trend: increasing!)

# Let's think about those numbers

- **Huge difference between a hit and a miss**
  - Could be 100x, if just L1 and main memory

- **Would you believe 99% hits is twice as good as 97%?**
  - Consider:
    cache hit time of 1 cycle
    miss penalty of 100 cycles

  - Average access time:

    97% hits: 1 cycle + 0.03 * 100 cycles = **4 cycles**

    99% hits: 1 cycle + 0.01 * 100 cycles = **2 cycles**

- **This is why "miss rate" is used instead of "hit rate"**

# Writing Cache Friendly Code

- **Make the common case go fast**
  - Focus on the inner loops of the core functions

- **What influence does that have on parallelism?**

# Today

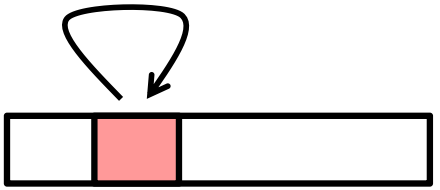
- Cache organization and operation
- **Performance impact of caches**
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

# Writing Cache Friendly Code

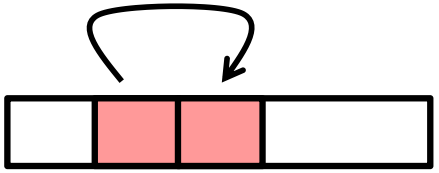
- **Make the common case go fast**
  - Focus on the inner loops of the core functions

# Locality

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



- **Temporal locality:**
  - Recently referenced items are likely to be referenced again in the near future



- **Spatial locality:**
  - Items with nearby addresses tend to be referenced close together in time

# Writing Cache Friendly Code

- **Make the common case go fast**
  - Focus on the inner loops of the core functions

- **Minimize the misses in the inner loops**
  - Repeated references to variables are good (temporal locality)
  - Stride-1 reference patterns are good (spatial locality)

**Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories**

# The Memory Mountain

- **Read throughput (read bandwidth)**
  - Number of bytes read from memory per second (MB/s)

- **Memory mountain: Measured read throughput as a function of spatial and temporal locality.**
  - Compact way to characterize memory system performance.

# Memory Mountain Test Function

```
long data[MAXELEMS];  /* Global array to traverse */

/* test - Iterate over first "elems" elements of
 *        array "data" with stride of "stride", using
 *        using 4x4 loop unrolling.
 */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}
```

*mountain/mountain.c*

Call `test()` with many combinations of `elems` and `stride`.

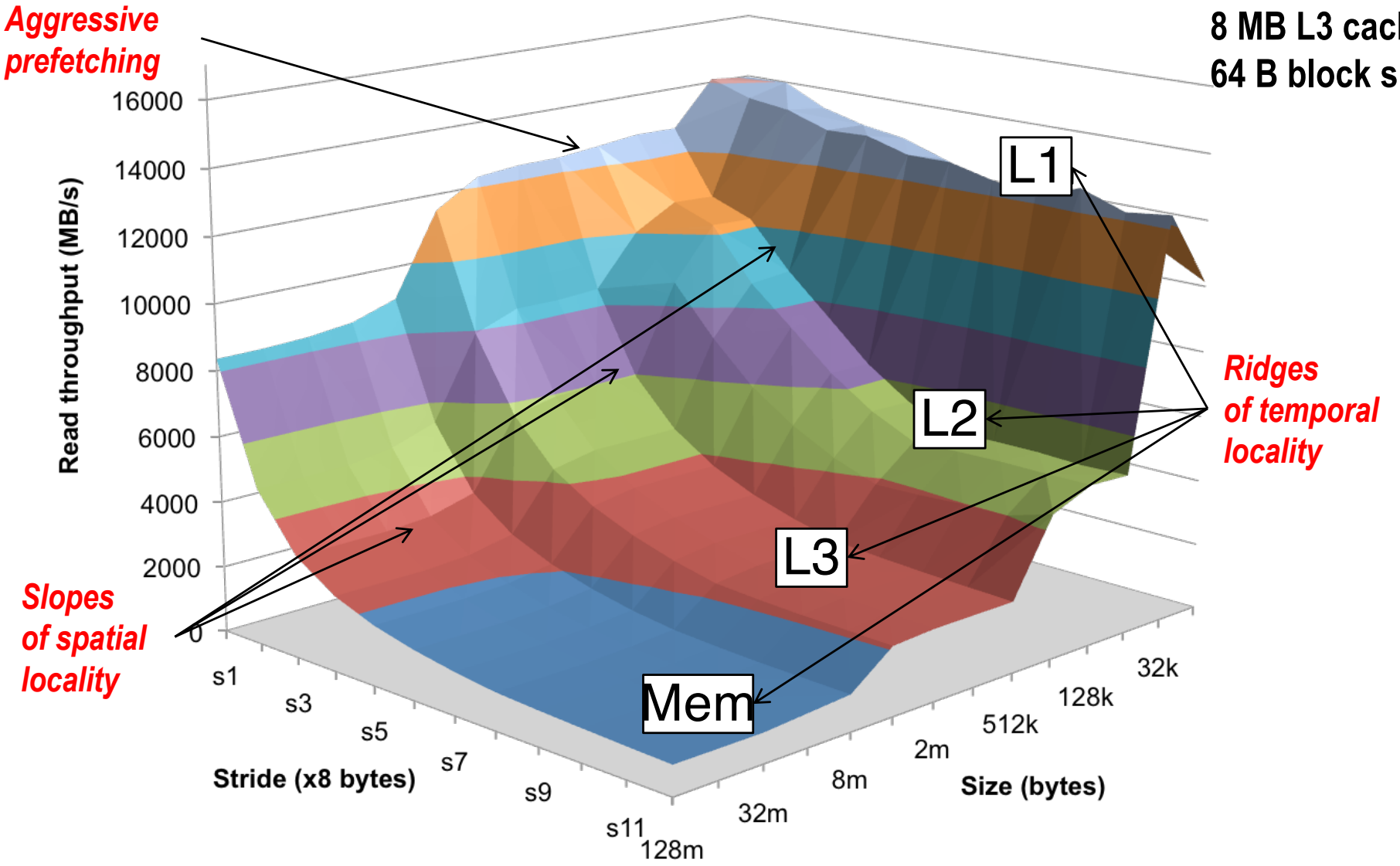
For each `elems` and `stride`:

1. Call `test()` once to warm up the caches.

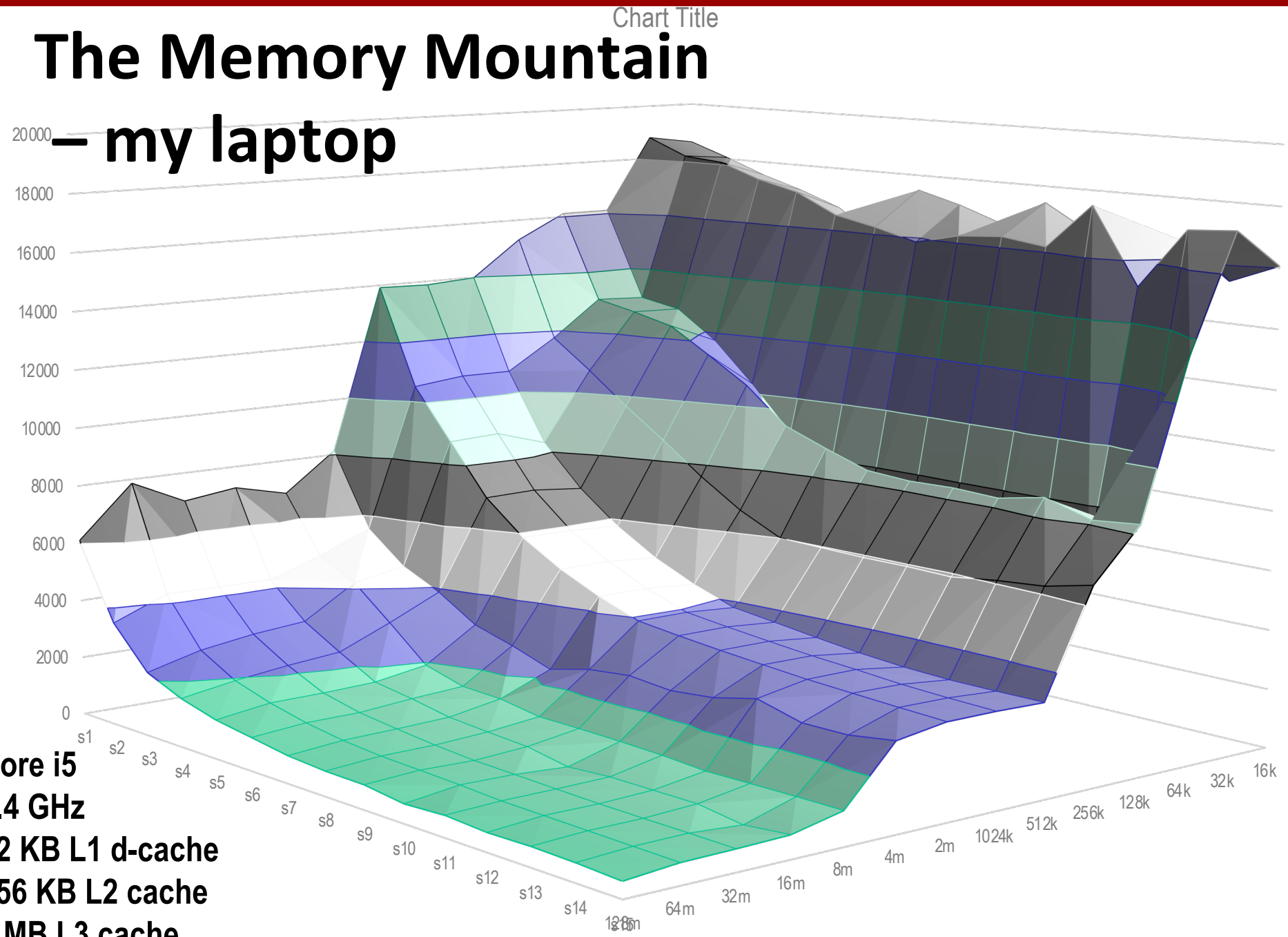
2. Call `test()` again and measure the read throughput (MB/s)

# The Memory Mountain

Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size



# The Memory Mountain – my laptop



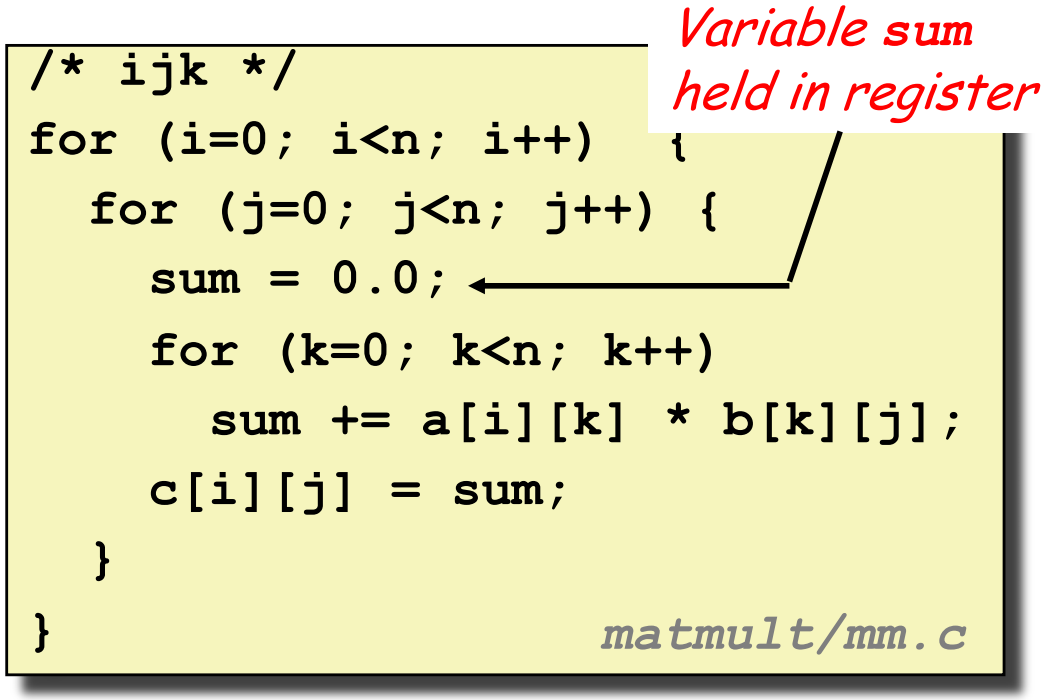

**Core i5**
**2.4 GHz**
**32 KB L1 d-cache**
**256 KB L2 cache**
**6 MB L3 cache**
**64 B block size**

# Today

- Cache organization and operation
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

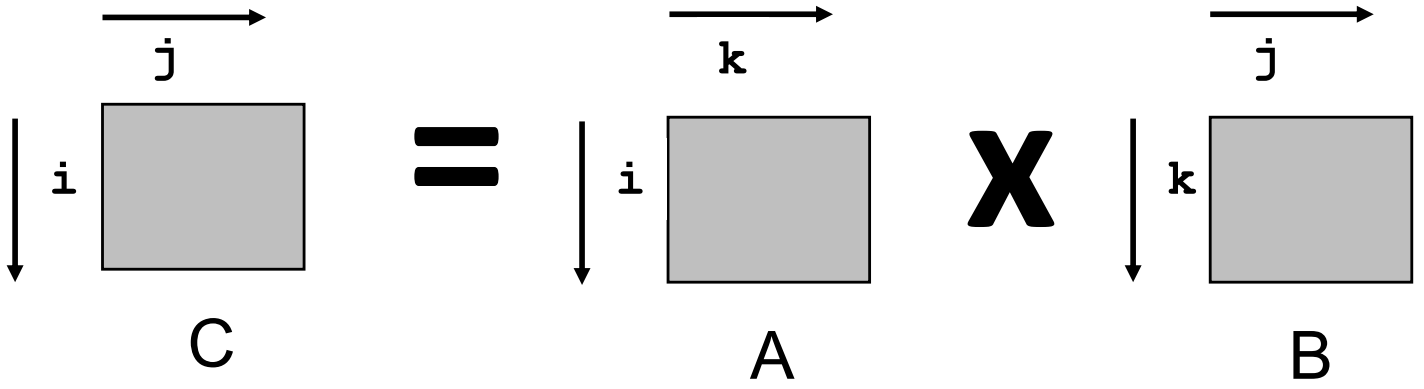
# Matrix Multiplication Example

- **Description:**
  - Multiply N x N matrices
  - Matrix elements are doubles (8 bytes)
  - $O(N^3)$ total operations
  - N reads per source element
  - N values summed per destination
    - but may be able to hold in register



# Miss Rate Analysis for Matrix Multiply

- **Assume:**
  - Block size = 32B (big enough for four doubles)
  - Matrix dimension (N) is very large
    - Approximate 1/N as 0.0
  - Cache is not even big enough to hold multiple rows
- **Analysis Method:**
  - Look at access pattern of inner loop



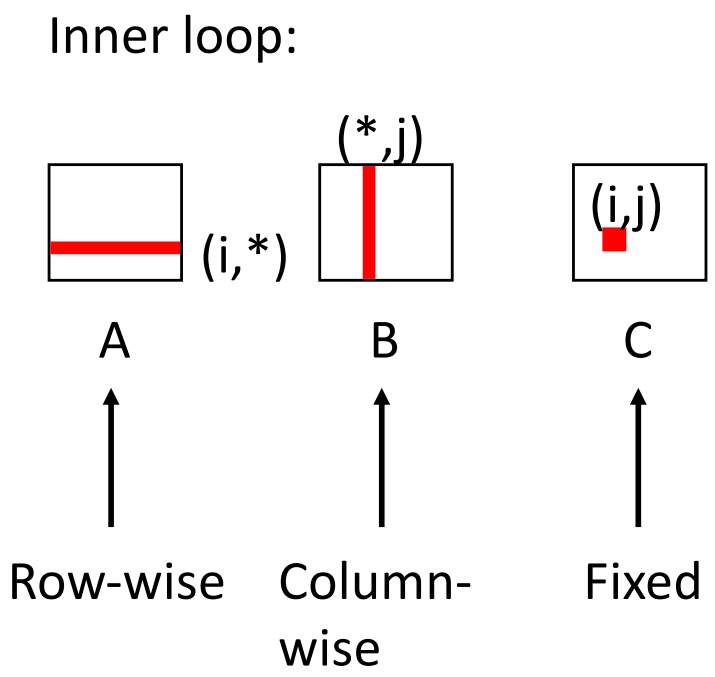
# Layout of C Arrays in Memory (review)

- **C arrays allocated in row-major order**
  - each row in contiguous memory locations
- **Stepping through columns in one row:**
  - `for (i = 0; i < N; i++)`
    `sum += a[0][i];`
  - accesses successive elements
  - if block size (B) > sizeof($a_{ij}$) bytes, exploit spatial locality
    - miss rate = sizeof($a_{ij}$) / B
- **Stepping through rows in one column:**
  - `for (i = 0; i < n; i++)`
    `sum += a[i][0];`
  - accesses distant elements
  - no spatial locality!
    - miss rate = 1 (i.e. 100%)

# Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

*matmult/mm.c*



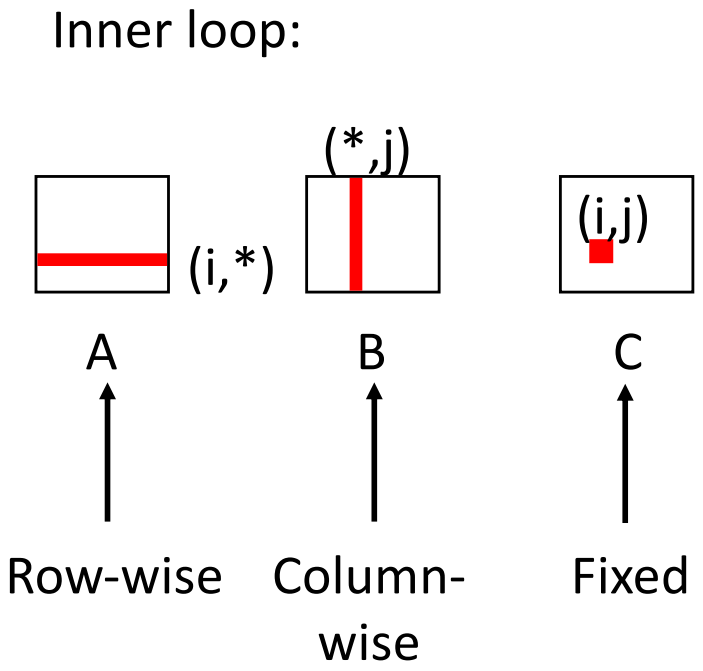

Misses per inner loop iteration:

| A | B | C |
|---|---|---|
| 0.25 | 1.0 | 0.0 |

# Matrix Multiplication (jik)

```
/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum
  }
}
```

*matmult/mm.c*




Misses per inner loop iteration:

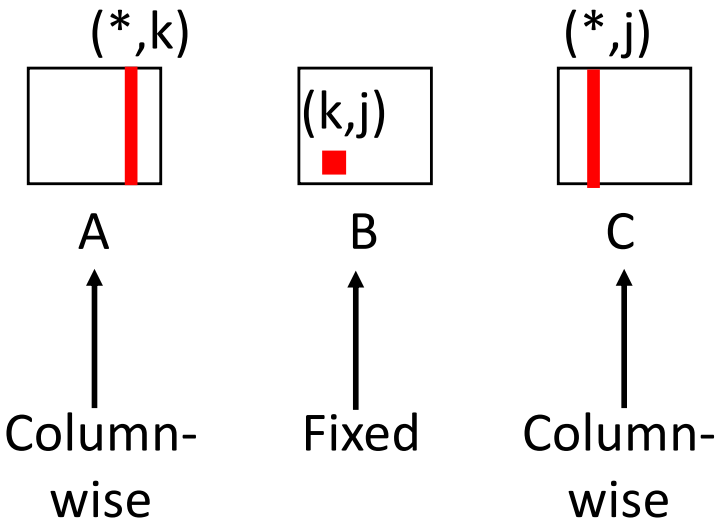
| A | B | C |
|---|---|---|
| 0.25 | 1.0 | 0.0 |

# Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

*matmult/mm.c*

Inner loop:




Misses per inner loop iteration:

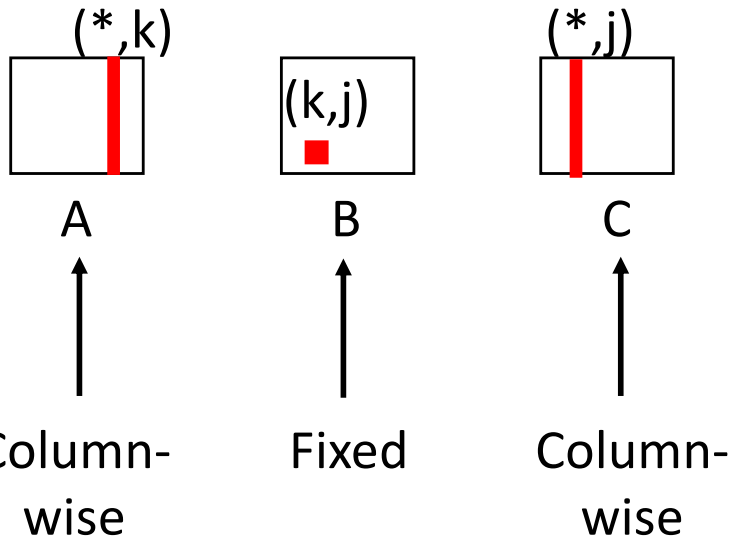
| A | B | C |
|---|---|---|
| 1.0 | 0.0 | 1.0 |

# Matrix Multiplication (kji)

```
/* kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

*matmult/mm.c*

Inner loop:




Misses per inner loop iteration:

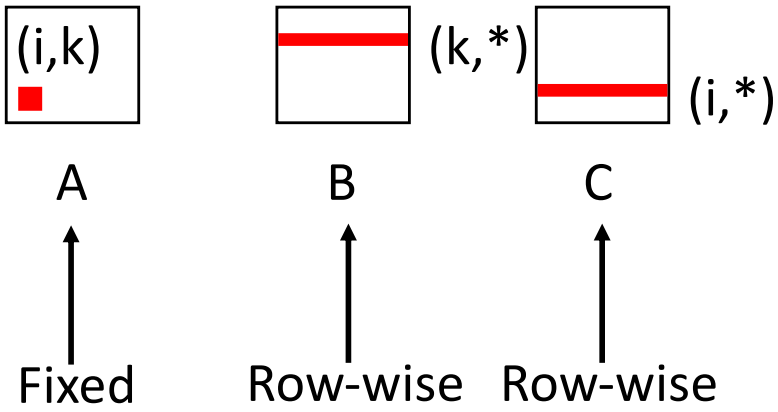
| A | B | C |
|---|---|---|
| 1.0 | 0.0 | 1.0 |

# Matrix Multiplication (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

*matmult/mm.c*

Inner loop:



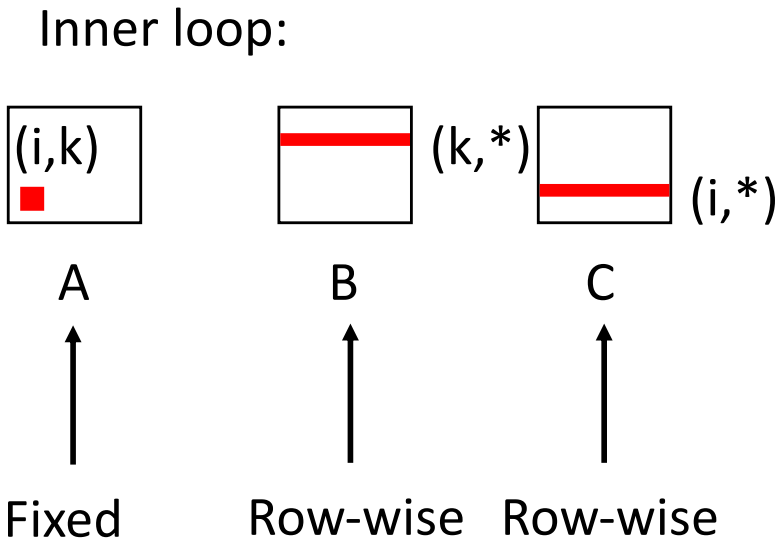

Misses per inner loop iteration:

| A | B | C |
|---|---|---|
| 0.0 | 0.25 | 0.25 |

# Matrix Multiplication (ikj)

```
/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

matmult/mm.c




Misses per inner loop iteration:

| A | B | C |
|---|---|---|
| 0.0 | 0.25 | 0.25 |

# Summary of Matrix Multiplication

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
   sum = 0.0;
   for (k=0; k<n; k++)
     sum += a[i][k] * b[k][j];
   c[i][j] = sum;
 }
}
```

**ijk (& jik):**

- 2 loads, 0 stores
- misses/iter = **1.25**

```
for (k=0; k<n; k++) {
 for (i=0; i<n; i++) {
  r = a[i][k];
  for (j=0; j<n; j++)
   c[i][j] += r * b[k][j];
 }
}
```

**kij (& ikj):**

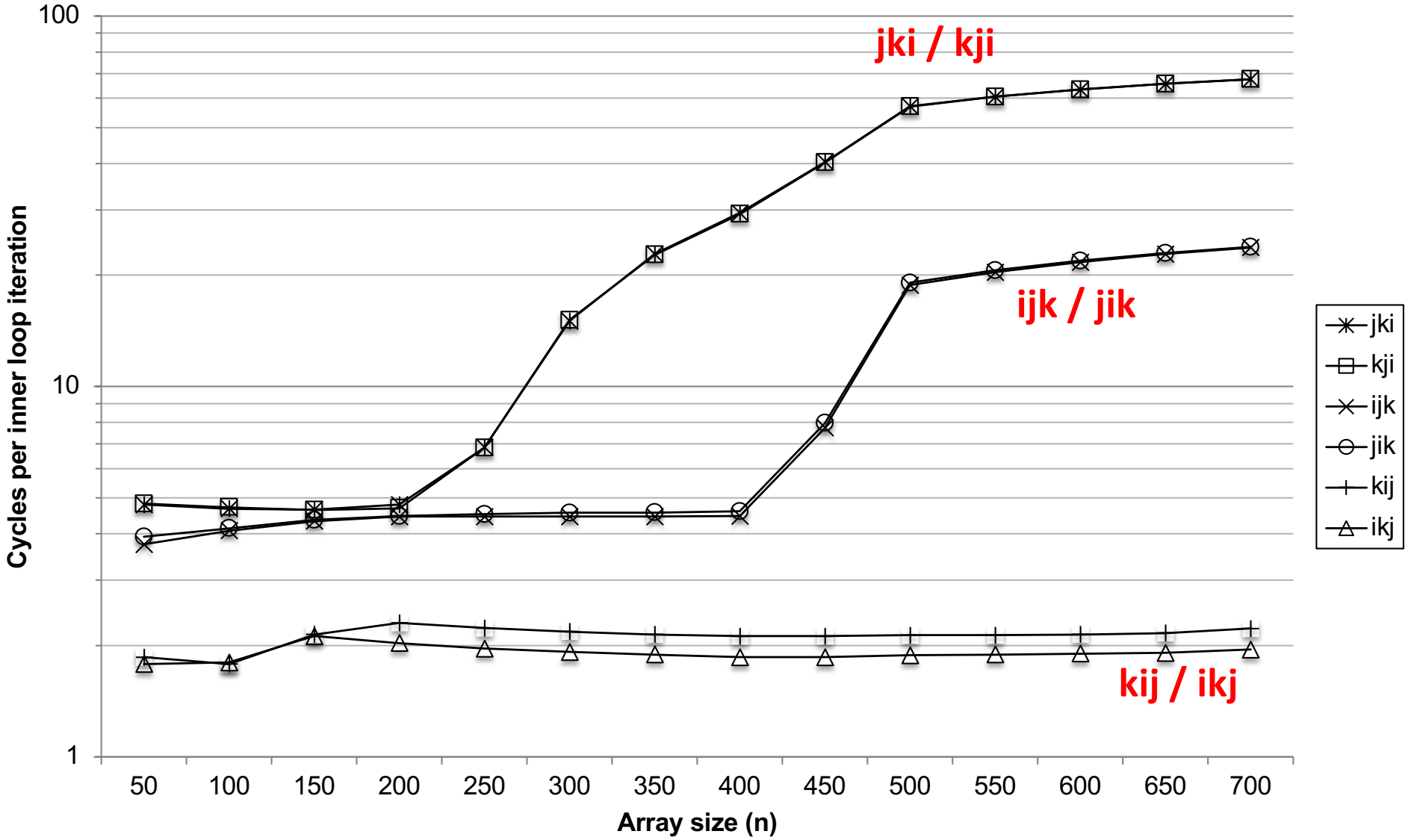
- 2 loads, 1 store
- misses/iter = **0.5**

```
for (j=0; j<n; j++) {
 for (k=0; k<n; k++) {
   r = b[k][j];
   for (i=0; i<n; i++)
    c[i][j] += a[i][k] * r;
 }
}
```

**jki (& kji):**

- 2 loads, 1 store
- misses/iter = **2.0**

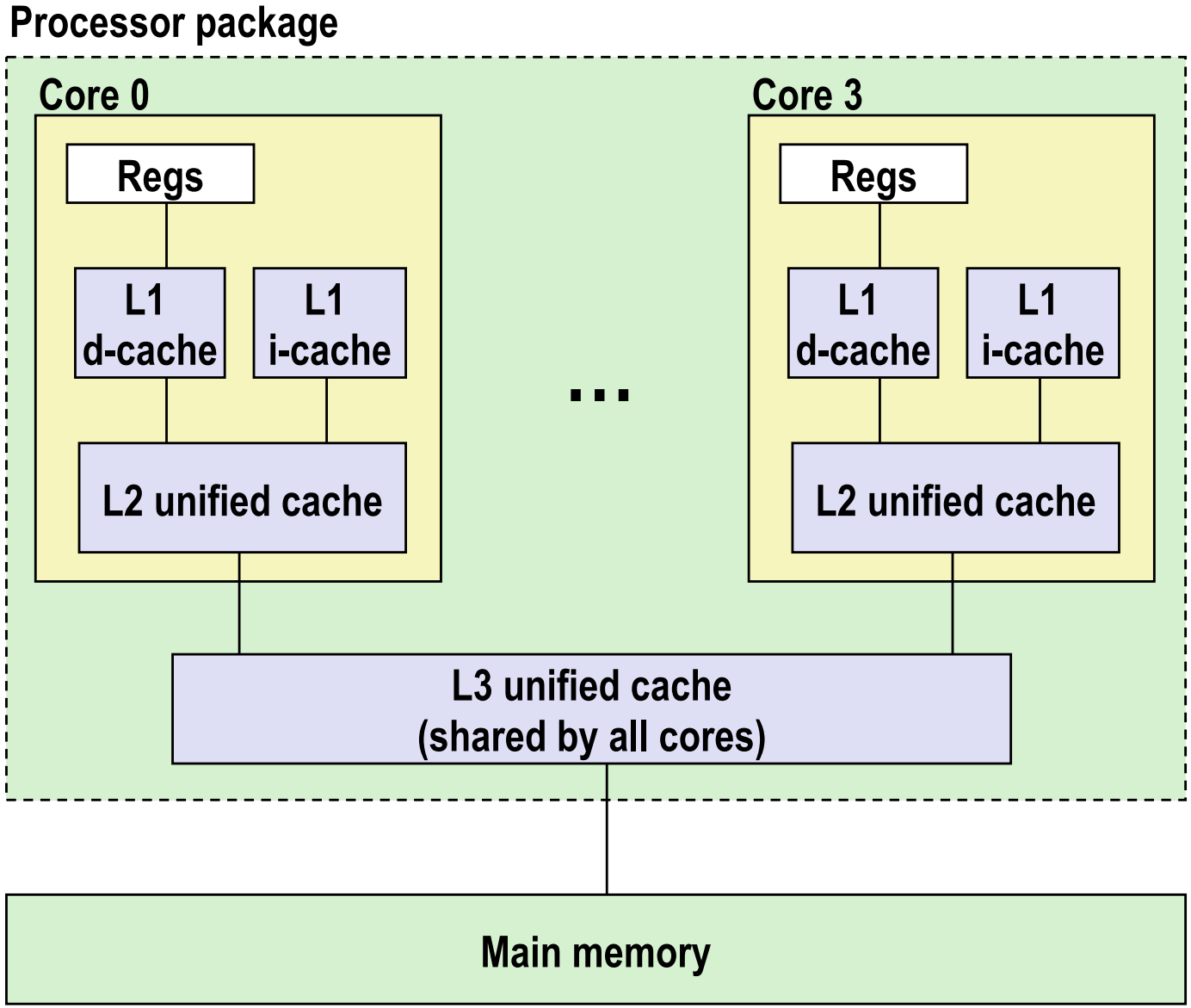
# Core i7 Matrix Multiply Performance



# What influence does that have on parallelism?

- ?

# Intel Core i7 Cache Hierarchy




**L1 i-cache and d-cache:**
32 KB, 8-way,
Access: 4 cycles

**L2 unified cache:**
256 KB, 8-way,
Access: 10 cycles

**L3 unified cache:**
8 MB, 16-way,
Access: 40-75 cycles

**Block size**: 64 bytes for all caches.

# What influence does that have on parallelism?

- Memory is an even larger limitation
- The shared L3$ is
- Cache coherence!!  (:-o)

# Strategies for Parallelization

- ?

$$\begin{pmatrix} A1 & B1 \\ C1 & D1 \end{pmatrix} \times \begin{pmatrix} A2 & B2 \\ C2 & D2 \end{pmatrix} =$$

$$\begin{pmatrix} A1A2 + B1C2 & A1B2+B1D2 \\ C1A2 + D1C2 & C1B2+D1D2 \end{pmatrix}$$

# Strategies for Parallelization

- **Tilling:**

$$\begin{pmatrix} A1 & B1 \\ C1 & D1 \end{pmatrix} \times \begin{pmatrix} A2 & B2 \\ C2 & D2 \end{pmatrix} =$$

$$\begin{pmatrix} A1A2 + B1C2 & A1B2+B1D2 \\ C1A2 + D1C2 & C1B2+D1D2 \end{pmatrix}$$

  - Synchronization?

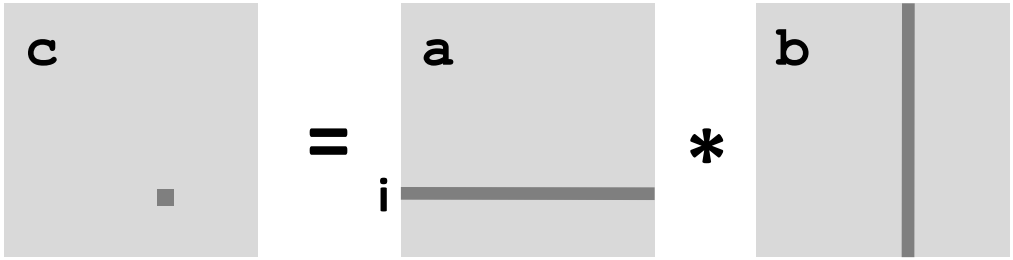
# Today

- Cache organization and operation
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using tilling to improve temporal locality

# Example: Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

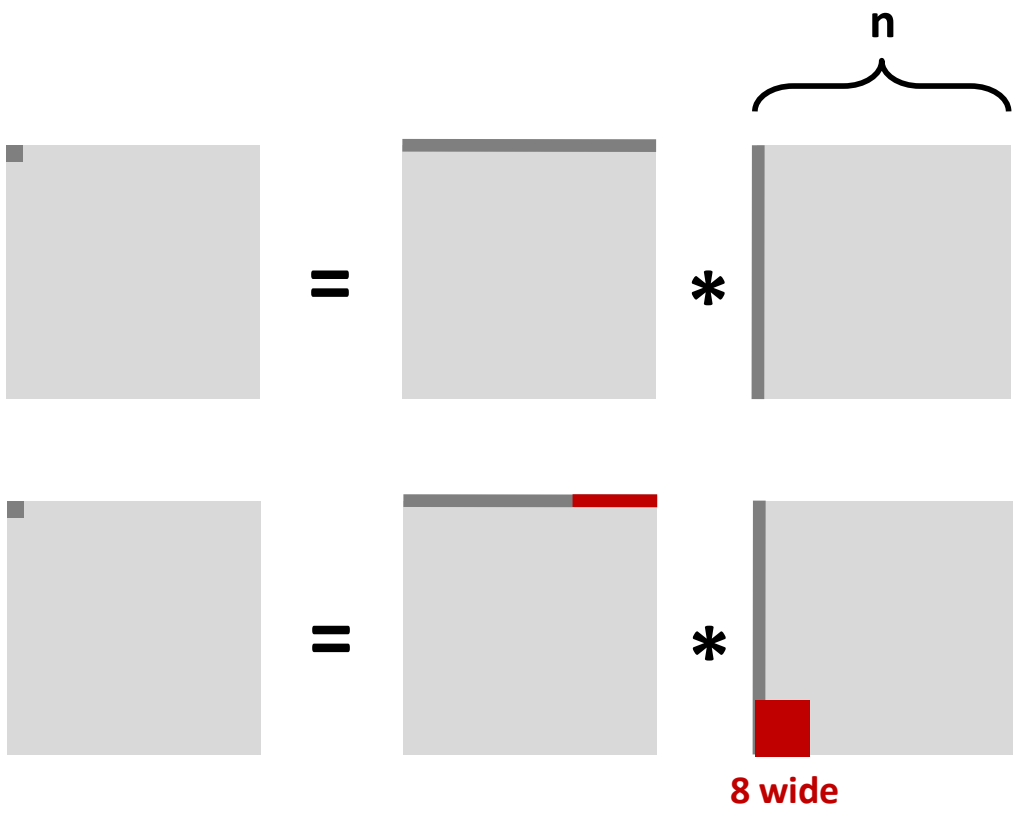
/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n + j] += a[i*n + k] * b[k*n + j];
}
```



# Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

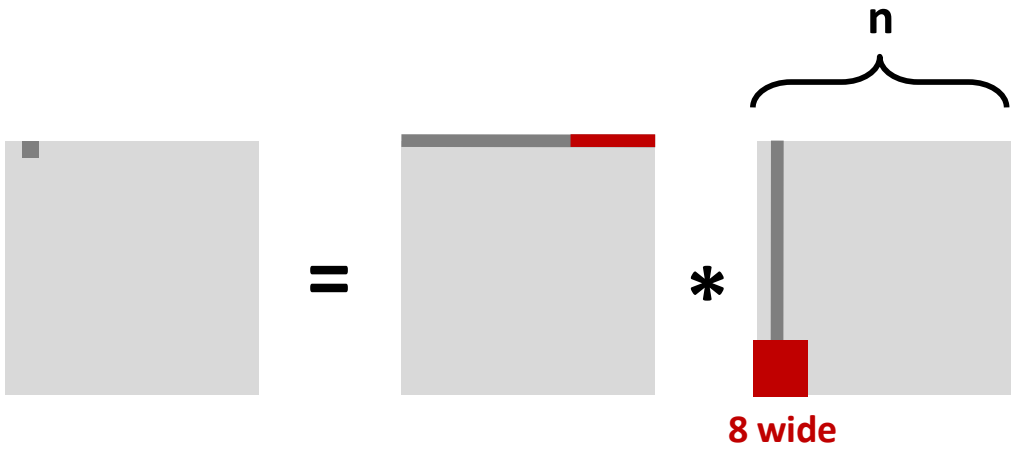
- **First iteration:**
  - n/8 + n = 9n/8 misses
  - Afterwards in cache: (schematic)



# Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

- **Second iteration:**
  - Again:
    n/8 + n = 9n/8 misses



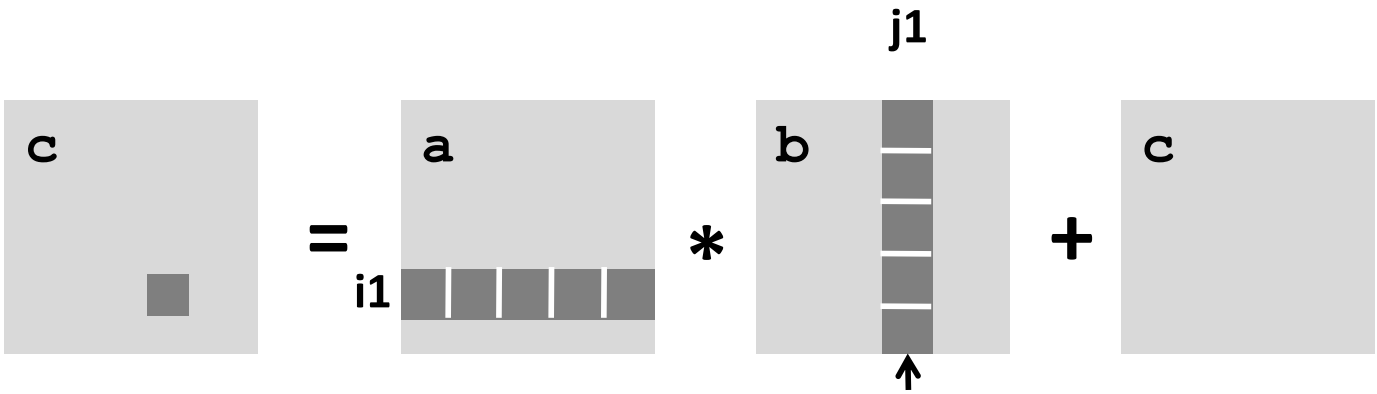

- **Total misses:**
  - $9n/8 * n^2 = (9/8) * n^3$

# Blocked Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i++)
                    for (j1 = j; j1 < j+B; j++)
                        for (k1 = k; k1 < k+B; k++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

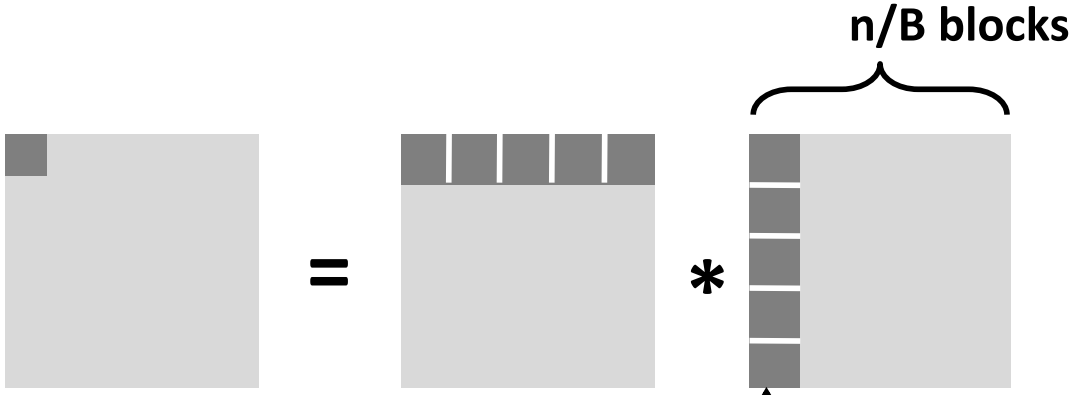
*matmult/bmm.c*



# Cache Miss Analysis

- **Assume:**
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks ■ fit into cache: $3B^2 < C$

- **First (block) iteration:**
  - $B^2/8$ misses for each block
  - $2n/B * B^2/8 = nB/4$ (omitting matrix c)



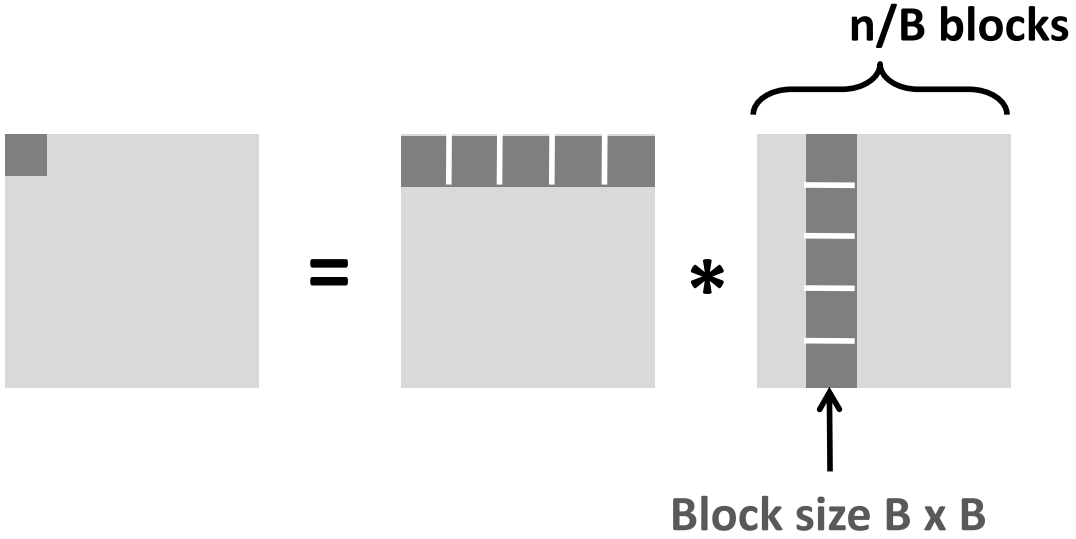

  - Afterwards in cache (schematic)



# Cache Miss Analysis

- **Assume:**
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks ■ fit into cache: $3B^2 < C$

- **Second (block) iteration:**
  - Same as first iteration
  - $2n/B * B^2/8 = nB/4$




- **Total misses:**
  - $nB/4 * (n/B)^2 = n^3/(4B)$

# Blocking Summary

- **No blocking: (9/8) * $n^3$**
- **Blocking: 1/(4B) * $n^3$**

- **Suggest largest possible block size B, but limit $3B^2 < C$!**
  - My machine has 32 KB L1-dcache
  - Implies B < 104

- **Reason for dramatic difference:**
  - Matrix multiplication has inherent temporal locality:
    - Input data: $3n^2$, computation $2n^3$
    - Every array elements used O(n) times!
  - But program has to be written properly

# Cache Summary

- **Cache memories can have significant performance impact**

- **You can write your programs to exploit this!**
  - Focus on the inner loops, where bulk of computations and memory accesses occur.
  - Try to maximize spatial locality by reading data objects with sequentially with stride 1.
  - Try to maximize temporal locality by using a data object as often as possible once it's read from memory.