Lecture 13
Cache Performance

CS213 – Intro to Computer Systems

Branden Ghena – Winter 2023

Slides adapted from:
St-Amour, Hardavellas, Bustamente (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

Northwestern

Administrivia

- Attack Lab
    - Due this week on Wednesday

- Homework 4
    - Last homework!
    - Releases today
    - Due next week Wednesday

- SETI Lab
    - Last lab!
    - Releases on Wednesday
    - Two weeks to complete it

Today’s Goals

- Explore impacts of cache and code design
- Calculate cache performance based on array accesses
- Understand what it means to write “cache-friendly code”

Outline

- **Memory Mountain**
- Cache Metrics
- Cache Performance for Arrays
- Improving code
  - Rearranging Matrix Math
  - Matrix Math in Blocks

Writing Cache-Friendly Code

- Caches are key to program performance
  - CPU accessing main memory = CPU twiddling its thumbs = bad
  - Want to avoid as much as possible

- Minimize cache misses in the inner loops of core functions
  - That's usually where your program spends most of its time ("hot" code)
    - Programmers are notoriously bad at guessing these spots
    - Use a profiler to find them (e.g., `gprof`)

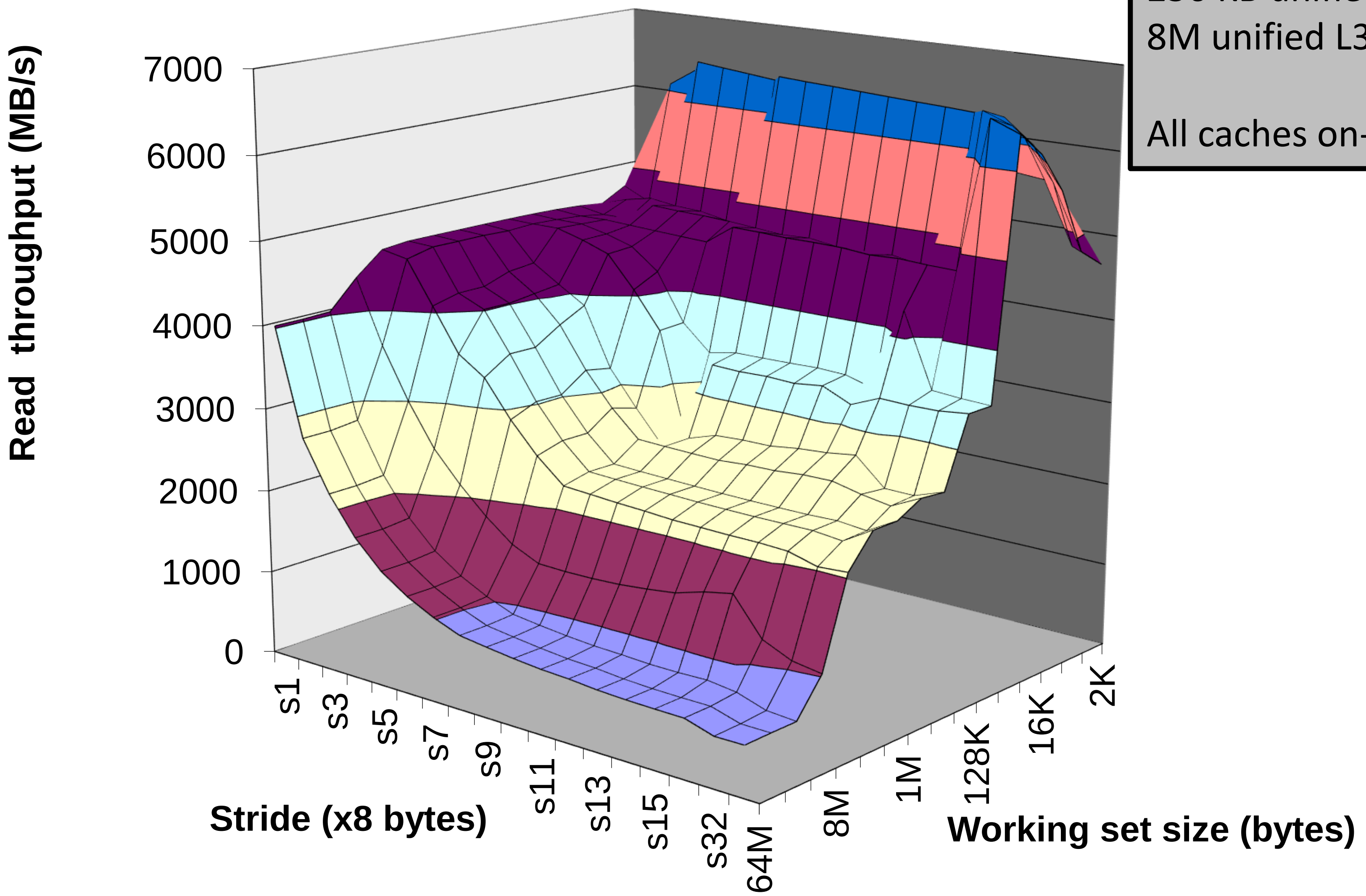
  - Repeated references to variables are good (***temporal locality***)
  - Stride-1 reference patterns are good (***spatial locality***)
    - I.e., accessing array elements in sequence, not jumping around

- Now that we know how cache memories work
  - We can quantify the effect of locality on performance

A Memory Mountain

Intel Core i7
32 KB L1 i-cache
32 KB L1 d-cache
256 KB unified L2 cache
8M unified L3 cache

All caches on-chip



The Memory Mountain

- *Read throughput* (read bandwidth)
  - Number of bytes read from the memory subsystem per second (MB/s)
  - The higher it is, the less likely your CPU is to be waiting on memory

- *Memory mountain*: Measures read throughput as a function of spatial and temporal locality.
  - We run variants of the same program with different levels of spatial and temporal locality, then measure read throughput
  - Compact way to characterize memory system performance
  - Different systems (with different caches) have different mountains!

- Observation: if you decrease locality, bandwidth drops
  - As we'd expect; locality is key to having the right data in the cache
  - And if data is not in the cache, need to get it from next level down

Mapping the Memory Mountain

Basically: a ton of memory reads in a loop
and nothing else (that takes much time)

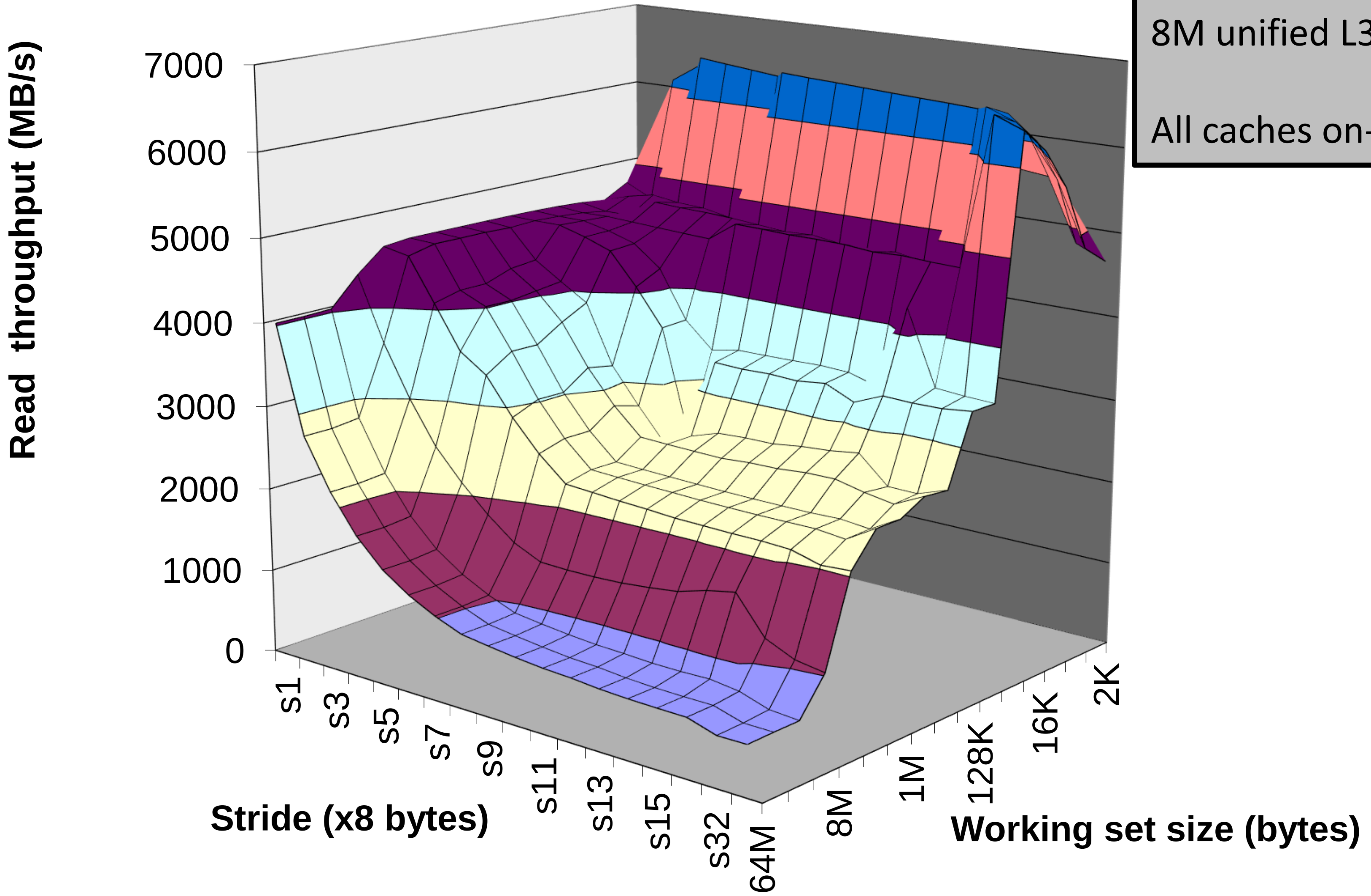
```
/* The test function */
void test(int elems, int stride) {
    int i, result = 0;
    volatile int sink;

    for (i = 0; i < elems; i += stride)
        result += data[i];
    sink = result; /* So compiler doesn't optimize away the loop */
}

/* Run test(elems, stride) and return read throughput (MB/s) */
double run(int size, int stride, double Mhz)
{
    double cycles;
    int elems = size / sizeof(int);

    test(elems, stride);
    cycles = fcyc2(test, elems, stride, 0);
    return (size / stride) / (cycles / Mhz);
}
```

Lower = more temporal locality (fewer elements = less likely to get kicked out by conflicts)

Lower = more spatial locality (we visit close-by addresses one after the other)

Harness code
- Warms up cache
  (don’t want to count cold misses)
- Measures read throughput

A Memory Mountain



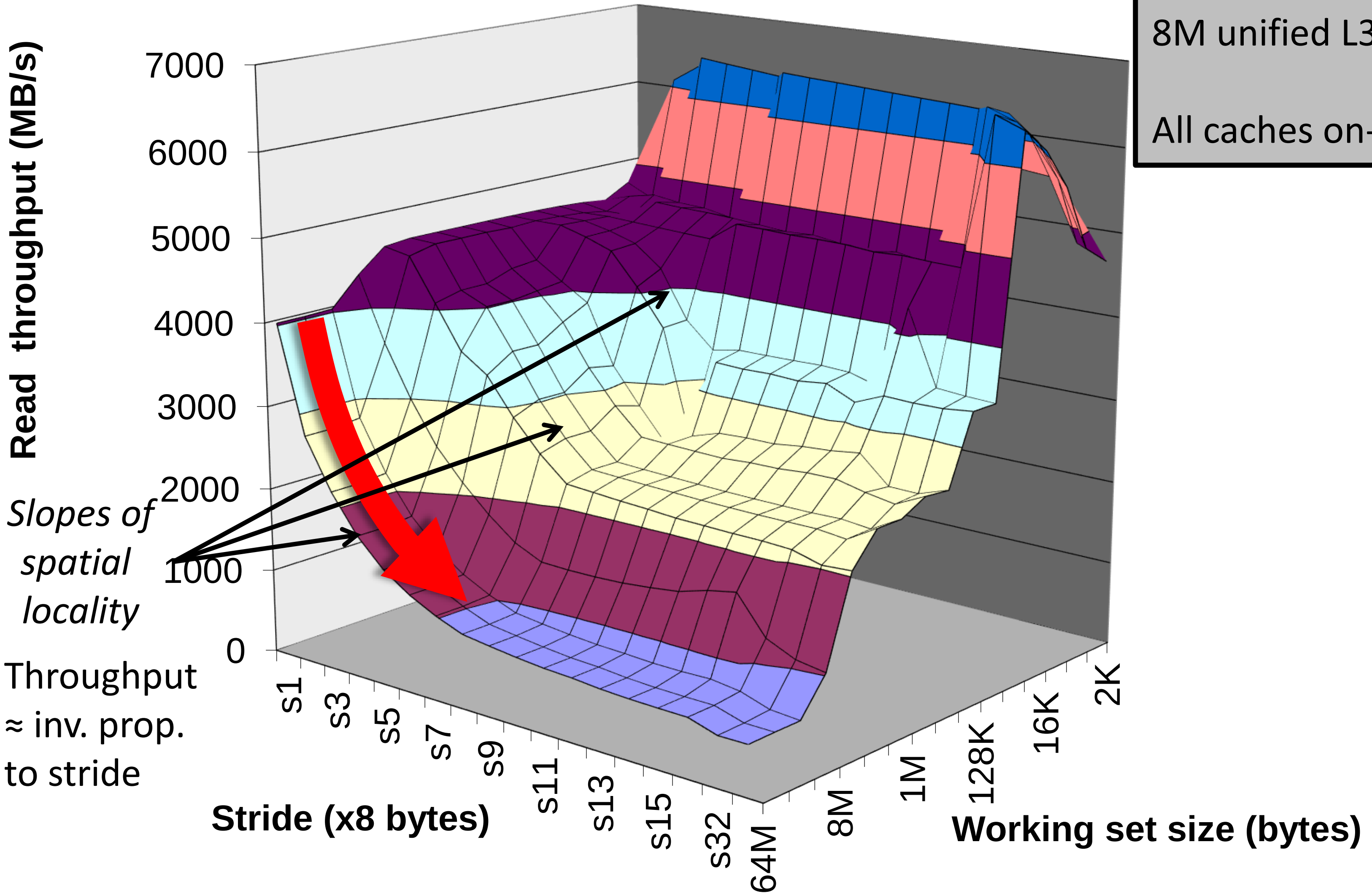
Intel Core i7
32 KB L1 i-cache
32 KB L1 d-cache
256 KB unified L2 cache
8M unified L3 cache

All caches on-chip

A Memory Mountain

Intel Core i7
32 KB L1 i-cache
32 KB L1 d-cache
256 KB unified L2 cache
8M unified L3 cache

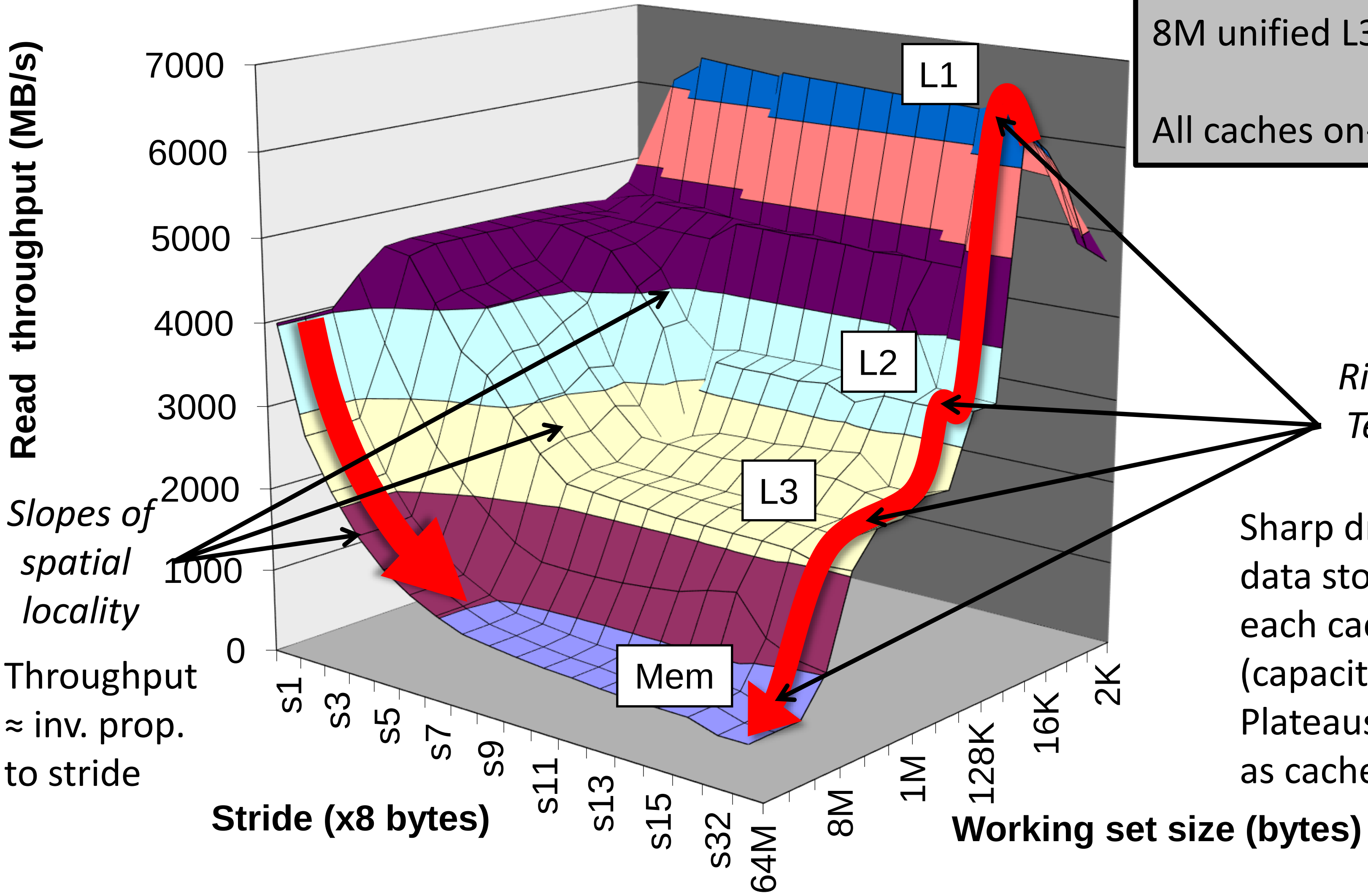
All caches on-chip



A Memory Mountain

Intel Core i7
32 KB L1 i-cache
32 KB L1 d-cache
256 KB unified L2 cache
8M unified L3 cache

All caches on-chip



Slopes of spatial locality

Throughput ≈ inv. prop. to stride

Ridges of Temporal locality

Sharp drops when data stops fitting in each cache level (capacity misses)

Plateaus get wider as caches get larger

Outline

- Memory Mountain
- **Cache Metrics**
- Cache Performance for Arrays
- Improving code
  - Rearranging Matrix Math
  - Matrix Math in Blocks

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 dataset size, etc.
    - However, many applications have >30% miss rate in L2 cache

- Hit Time
  - Time to deliver a line in the cache to the processor
    - Includes time to determine whether the line is in the cache
    - Assumption: always check first cache *before* going to the next level
  - Typical numbers:
    - 1-2 clock cycles for L1
    - 5-20 clock cycles for L2

- Miss Penalty
  - Additional time required because of a miss
  - Typically 50-200 cycles for main memory
    - Not really a "penalty", just how long it takes to read from memory

Let's think about those numbers

- Huge difference between a hit and a miss
  - Could be 100x, if comparing L1 and main memory
- Would you believe a 99% hit rate is twice as good as 97%?
  - Consider:
    cache hit time of 1 cycle
    miss penalty of 100 cycles
  - Average access time:

    97% hits: 100 instructions: 100*1 (L1 accesses) + 3*100 (misses)

    on average: 1 cycle/instr. + 0.03 * 100 cycles/instr. = **4 cycles/instr.**

    99% hits: on average: 1 cycle/instr. + 0.01 * 100 cycles/instr. = **2 cycles/instr.**
- This is why "miss rate" is used instead of "hit rate"
  - In our example, 1% miss rate vs. 3% miss rate
  - Makes the radical performance difference more obvious
- "Computation is what happens between cache misses."

Average Memory Access Time (AMAT)

- AMAT = Hit time + Miss rate × Miss penalty
  - Generalization of previous formula

- Can extend for multiple layers of caching
  - AMAT = Hit Time L1 + Miss Rate L1 × Miss Penalty L1
    - Miss Penalty L1 = Hit Time L2 + Miss Rate L2 × Miss Penalty L2
    - Miss Penalty L2 = Hit Time Main Memory

- Generally: multi-level caching helps minimize AMAT

Example Memory Access Time Problem

- Computer specs: One layer of cache plus main memory
  - Cache Hit Time: 5 nanoseconds
  - Cache Miss Rate: 2%
    - Memory Access Time: 100 nanoseconds

- Calculate Average Memory Access Time (Hit Time + Miss Rate * Miss Penalty)
  - 5 ns + 0.02 * 100 ns
  - = 5 ns + 2 ns
  - = 7 ns

Break + Practice

- Computer specs: Two layers of cache plus main memory
  - L1 Cache Hit Time: 4 nanoseconds
  - L1 Cache Miss Rate: 10%
    - L2 Cache Hit Time: 8 nanoseconds
    - L2 Cache Miss Rate: 2%
      - Memory Access Time: 100 nanoseconds

- Calculate Average Memory Access Time (Hit Time + Miss Rate * Miss Penalty)

Break + Practice

- Computer specs: Two layers of cache plus main memory
  - L1 Cache Hit Time: 4 nanoseconds
  - L1 Cache Miss Rate: 10%
    - L2 Cache Hit Time: 8 nanoseconds
    - L2 Cache Miss Rate: 2%
      - Memory Access Time: 100 nanoseconds

- Calculate Average Memory Access Time (Hit Time + Miss Rate * Miss Penalty)
  - 4 ns + 0.10 * (8 ns + 0.02 * 100 ns)
  - = 4 ns + 0.10 * (8 ns + 2 ns)
  - = 4 ns + 0.10 * 10 ns
  - = 5 ns

Outline

- Memory Mountain
- Cache Metrics
- **Cache Performance for Arrays**
- Improving code
  - Rearranging Matrix Math
  - Matrix Math in Blocks

Contiguous Memory vs Indirection

- The rest of this lecture will focus on loops over arrays
  - I.e., operating on contiguous blocks of memory

- Not all programs are like that
  - "Pointer-chasing" is common
    - E.g., traversing a linked list, following a pointer for every node
  - (Usually) terrible for locality
    - See earlier comment about some programs having >30% L2 misses
    - A good allocator (**`malloc`**) can help some, but no miracles

- Specialized data structures can improve locality while still having a linked structure, e.g., for trees
  - E.g., ropes, B-trees, HAMTs, etc.

Layout of C Arrays in Memory (review)

- C arrays allocated in row-major order
  - Each row in contiguous memory locations
  - Here, let's assume we have a matrix of **`long`** or **`double`** (8 bytes)
  - That matrix is so large that we can't even fit a whole row in the cache

- Stepping through columns in one row:
  - ```
    for (i = 0; i < N; i++)
       sum += a[0][i];
    ```
  - accesses successive elements
  - if cache block size (B) > 8 bytes (element size), exploit spatial locality
    - cold/compulsory miss rate = 1 miss / Elements in Block = 1/(Block size / 8) = 8 / Block size

- Stepping through rows in one column:
  - ```
    for (j = 0; j < M; j++)
       sum += a[j][0];
    ```
  - accesses distant elements
  - no spatial locality!
    - cold/compulsory miss rate = 1 (i.e. 100%)

Example cache performance problem

- Cache parameters
  - Direct-mapped data cache
  - 256-byte total size
  - 16-byte blocks

  - Blocks per set: 1
  - Sets: 256/16 = 16

- Assume data starts at address 0 and cache starts empty

```
int mat[6][16];
```

- First, think about how array maps to the cache
  - Element size: 4 bytes
  - Array size: 384 bytes (too big)

  - 4 elements per cache block
  - Array row takes up 4 cache blocks

  - First 4 rows * 16 cols fit in cache without overlap
    - Next 2 rows overlap with first 2 rows

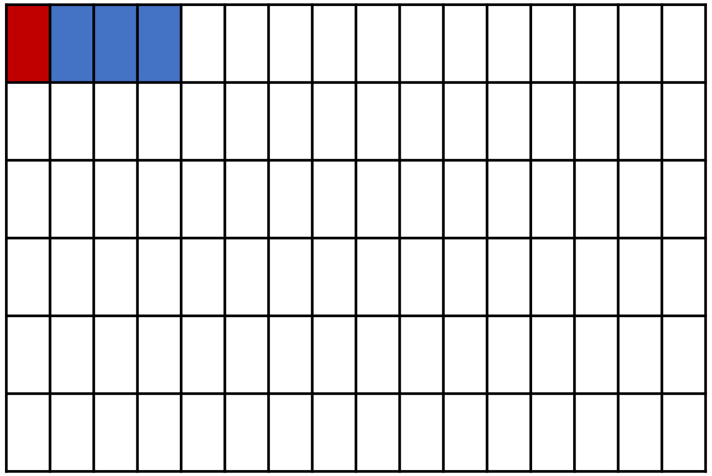
Example: accessing elements in a row

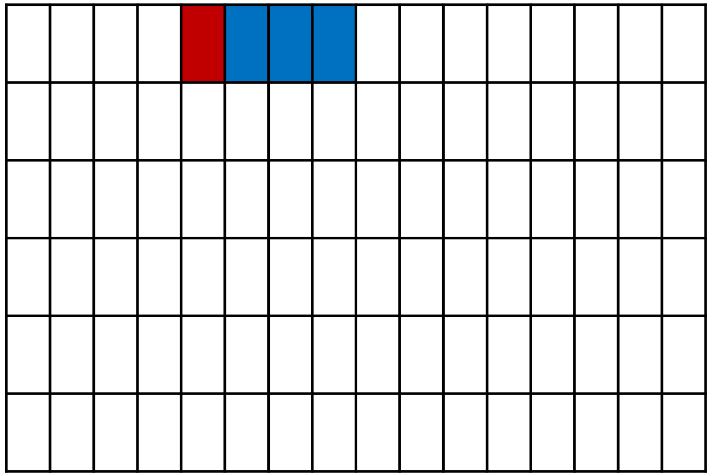
```
int mat[6][16];
```

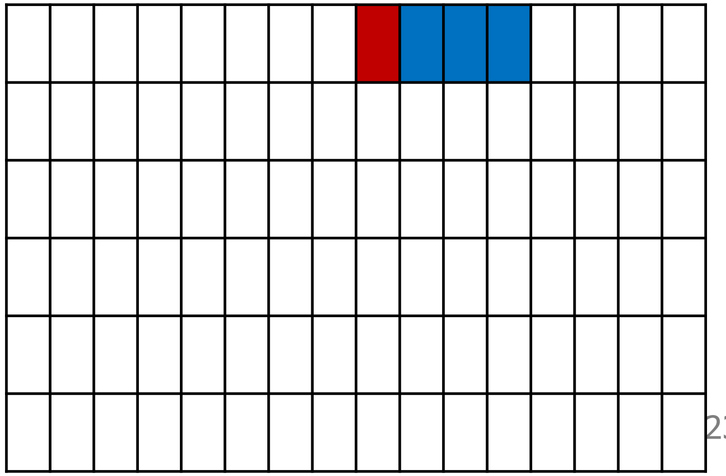
- First, think about how array maps to the cache
  - Element size: 4 bytes
  - Array size: 384 bytes (too big)
  - 4 elements per cache block
  - Array row takes up 4 cache blocks
  - First 4 row * 16 cols fit in cache without overlap
    - Next 2 rows overlap with first 2 rows

```
for (int i = 0; i < 6; i = i+1) {
  for (int j = 0; j < 16; j = j+4) {
    mat[i][j]   = 0;
    mat[i][j+1] = 1;
    mat[i][j+2] = 2;
    mat[i][j+3] = 3;
  }
}
```

- Calculate miss rate







Example: accessing elements in a row

```
int mat[6][16];
```

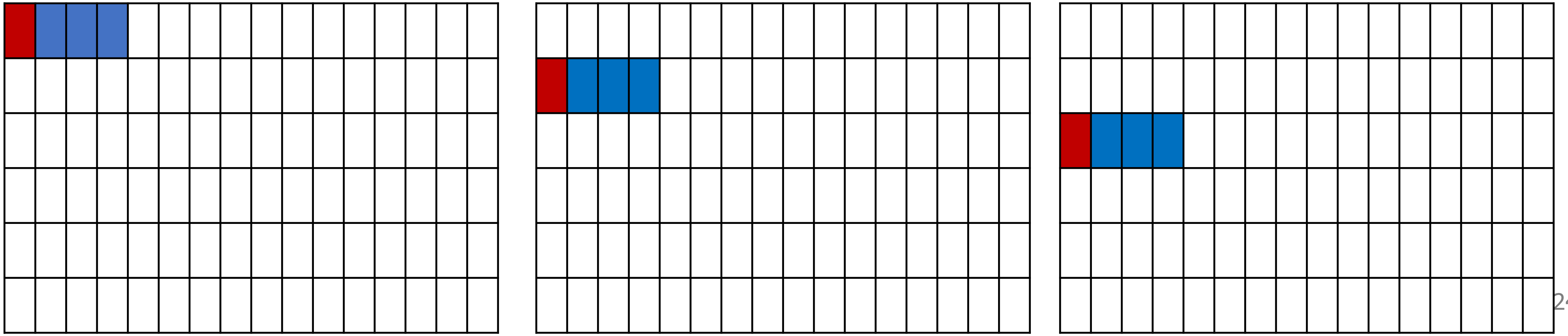
- First, think about how array maps to the cache
  - Element size: 4 bytes
  - Array size: 384 bytes (too big)

  - 4 elements per cache block
  - Array row takes up 4 cache blocks

  - First 4 row * 16 cols fit in cache without overlap
    - Next 2 rows overlap with first 2 rows

```
for (int i = 0; i < 6; i = i+1) {
  for (int j = 0; j < 16; j = j+4) {
    mat[i][j]   = 0;
    mat[i][j+1] = 1;
    mat[i][j+2] = 2;
    mat[i][j+3] = 3;
  }
}
```

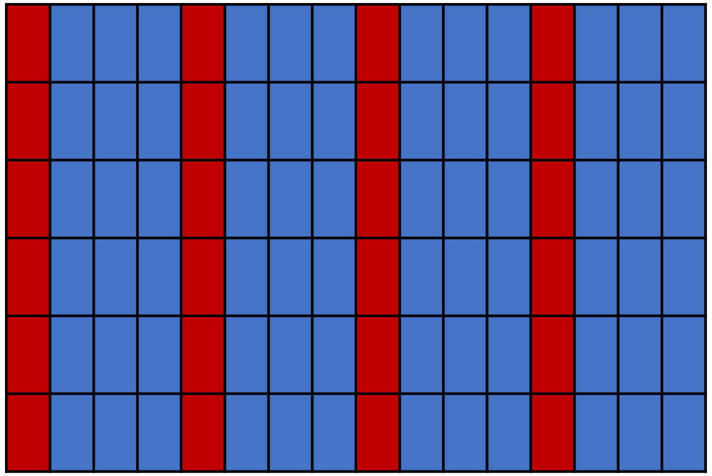
- Calculate miss rate



Example: accessing elements in a row

```
int mat[6][16];
```

- First, think about how array maps to the cache
  - Element size: 4 bytes
  - Array size: 384 bytes (too big)
  - 4 elements per cache block
  - Array row takes up 4 cache blocks
  - First 4 row * 16 cols fit in cache without overlap
    - Next 2 rows overlap with first 2 rows



```
for (int i = 0; i < 6; i = i+1) {
  for (int j = 0; j < 16; j = j+4) {
    mat[i][j]   = 0;
    mat[i][j+1] = 1;
    mat[i][j+2] = 2;
    mat[i][j+3] = 3;
  }
}
```

- Calculate miss rate
  - All four accesses within loop fit in a cache block!
    - 1 miss, 3 hits
  - The next set of columns repeat pattern
  - The next row repeats pattern
    - Nothing already in cache from before
    - Never reference old cells again
  - **Miss rate: 25%**

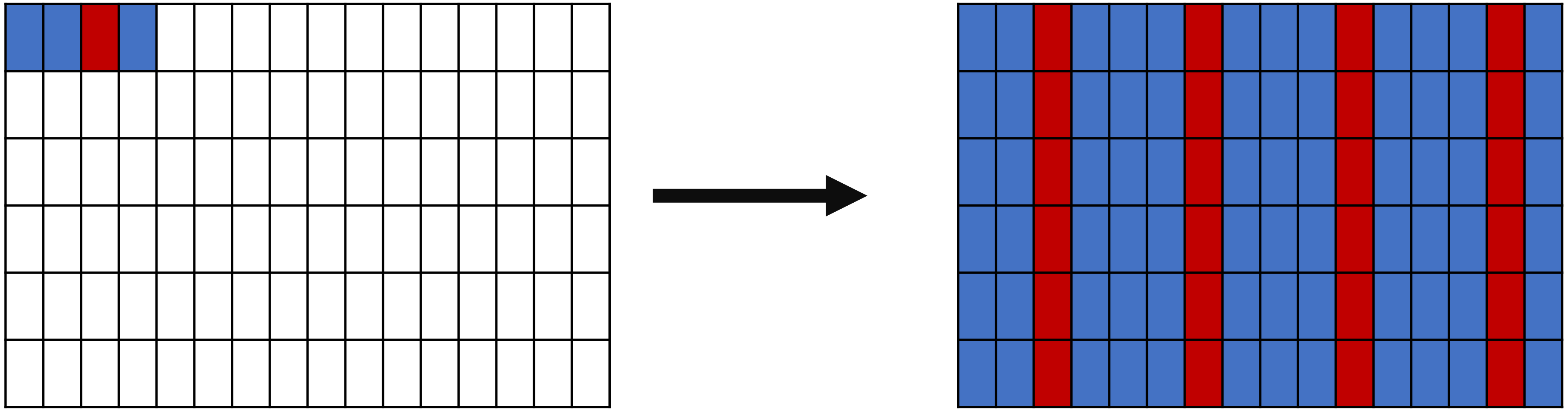
Example: reordering element access

```
int mat[6][16];
```

- First, think about how array maps to the cache
  - Element size: 4 bytes
  - Array size: 384 bytes (too big)
  - 4 elements per cache block
  - Array row takes up 4 cache blocks
  - First 4 row * 16 cols fit in cache without overlap
    - Next 2 rows overlap with first 2 rows

```
for (int i = 0; i < 6; i = i+1) {
  for (int j = 0; j < 16; j = j+4) {
    mat[i][j+2] = 2;
    mat[i][j]   = 0;
    mat[i][j+3] = 3;
    mat[i][j+1] = 1;
  }
}
```

- Does this change anything?
  - No! First access brings in entire block
  - Later accesses within block are hits



Example: accessing elements by column

```
int mat[6][16];
```

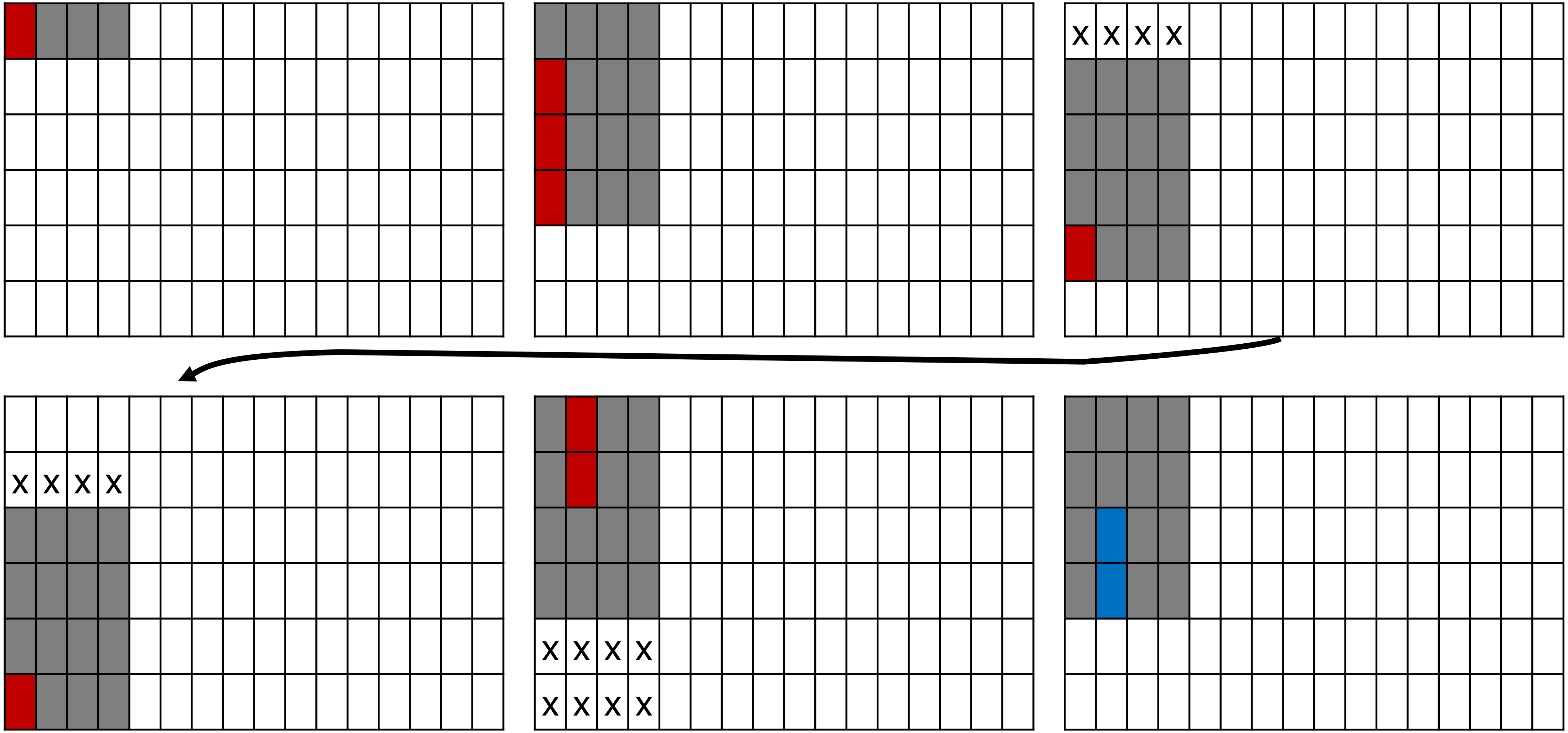
- First, think about how array maps to the cache
  - Element size: 4 bytes
  - Array size: 384 bytes (too big)
  - 4 elements per cache block
  - Array row takes up 4 cache blocks
  - First 4 row * 16 cols fit in cache without overlap
    - Next 2 rows overlap with first 2 rows

```
for (int j = 0; j < 16; j = j+1) {
  for (int i = 0; i < 6; i = i+1) {
    mat[i][j] = 7;
  }
}
```

- Calculate miss rate

Example: accessing elements by column (graphically)

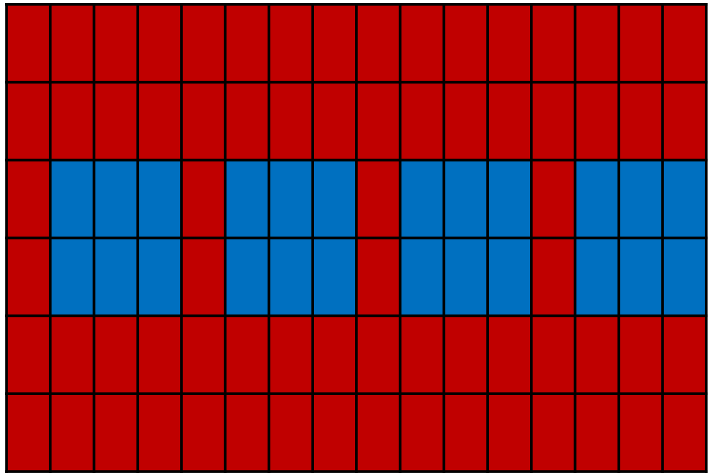
Grey blocks are loaded into the cache, but not accessed at this time



Example: accessing elements by column

```
int mat[6][16];
```

- First, think about how array maps to the cache
  - Element size: 4 bytes
  - Array size: 384 bytes (too big)
  - 4 elements per cache block
  - Array row takes up 4 cache blocks
  - First 4 row * 16 cols fit in cache without overlap
    - Next 2 rows overlap with first 2 rows



```
for (int j = 0; j < 16; j = j+1) {
  for (int i = 0; i < 6; i = i+1) {
    mat[i][j] = 7;
  }
}
```

- Calculate miss rate
  - 6 misses for 1st load of each row
  - 4 misses for 2nd column in the row (2 hits)
  - 4 misses for 3rd column in the row (2 hits)
  - 4 misses for 4th column in the row (2 hits)
  - Repeat
  - Miss rate = (6+4+4+4)/24 = 75%

Break + Question

```
int mat[4][16];
```

- Same cache from before:
  - Direct-mapped data cache
  - 256-byte total size
  - 16-byte blocks

- Change matrix to be 4 rows of 16 columns (not 6 rows)

```
for (int j = 0; j < 16; j = j+1) {
  for (int i = 0; i < 4; i = i+1) { // 4!
    mat[i][j] = 7;
  }
}
```

- Calculate miss rate

Break + Question

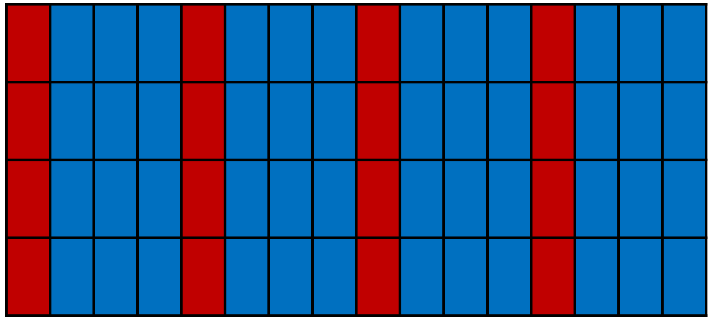
```
int mat[4][16];
```

- Same cache from before:
  - Direct-mapped data cache
  - 256-byte total size
  - 16-byte blocks

- Change matrix to be 4 rows of 16 columns (not 6 rows)

```
for (int j = 0; j < 16; j = j+1) {
  for (int i = 0; i < 4; i = i+1) { // 4!
    mat[i][j] = 7;
  }
}
```

- Calculate miss rate
  - Entire array fits in cache!
    - No conflicts
  - 1 miss per four accesses
  - **Miss rate = 25%**



Outline

- Memory Mountain
- Cache Metrics
- Cache Performance for Arrays
- **Improving code**
    - **Rearranging Matrix Math**
    - Matrix Math in Blocks

Our Benchmark: Matrix Multiplication

- Review from your linear algebra class

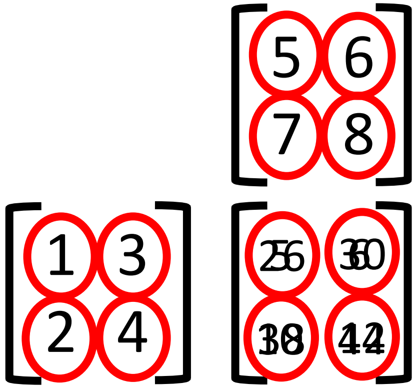
$$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 26 & 30 \\ 38 & 44 \end{bmatrix}$$

$1 \times 5 + 3 \times 7 = 26$

$1 \times 6 + 3 \times 8 = 30$

$2 \times 5 + 4 \times 7 = 38$

$2 \times 6 + 4 \times 8 = 44$

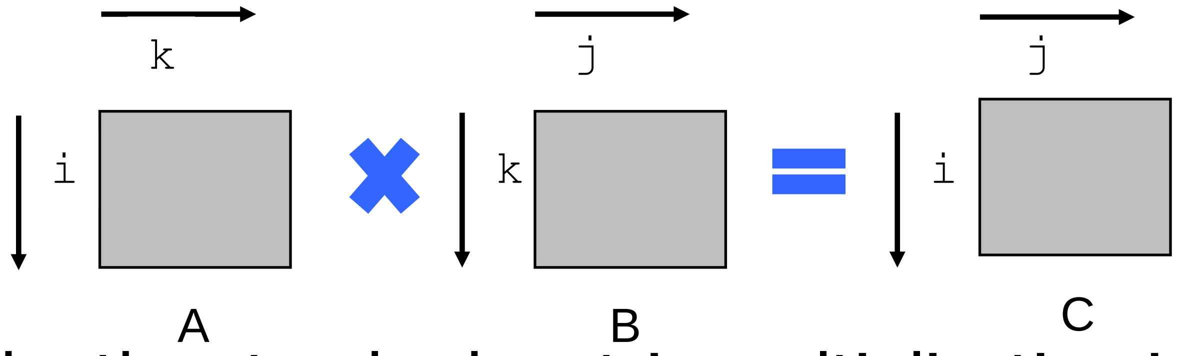


When is matrix multiplication important?

- ML and AI algorithms!!

Miss Rate Analysis for Matrix Multiply

- Assume:
  - Line size = 32B (big enough for four 64-bit longs)
  - Matrix dimension (N) is very large
    - Approximate 1/N as 0.0
  - Cache is not big enough to hold even one row
- Analysis Method:
  - Look at access pattern of inner loop



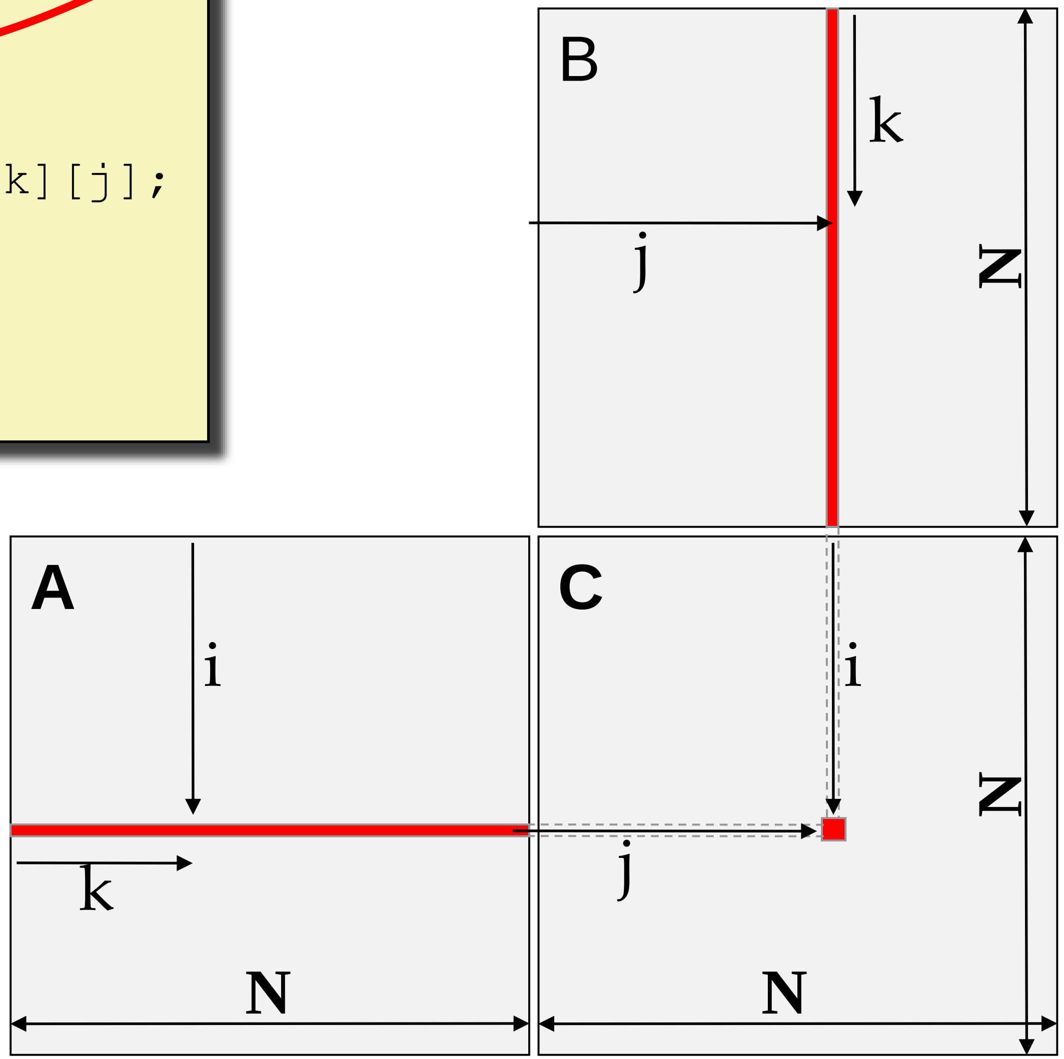
- Now we'll see why the standard matrix multiplication is bad!
  - From a performance standpoint, that is

Matrix Multiplication Example

```
/* 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;
  }
}
```

Variable `sum` held in register

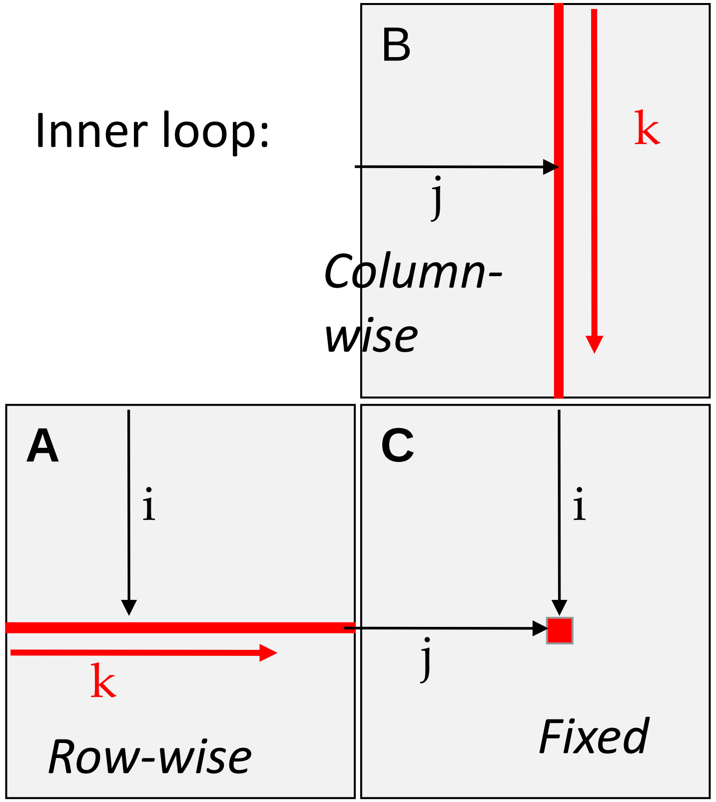
- Multiply N x N matrices
- $O(N^3)$ total operations
- Each source element read N times
- N values summed per destination



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;
  }
}
```

Inner loop:



Misses per inner loop iteration:

A	B	C
0.25	1	0

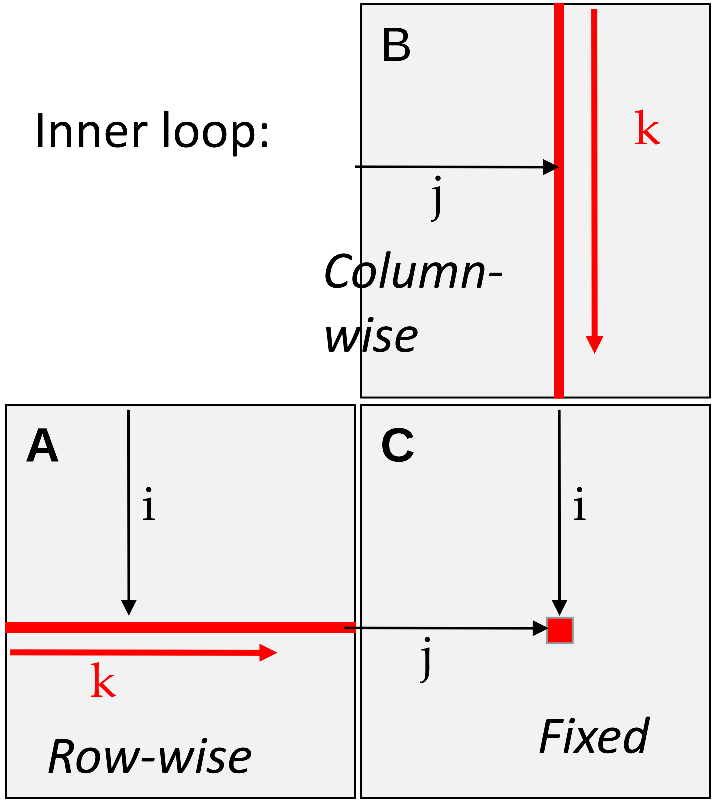
Total misses/iteration: 1.25

Remember: Line size = 32B
(big enough for four 64-bit longs)

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;
  }
}
```

Inner loop:



Misses per inner loop iteration:

A	B	C
0.25	1	0

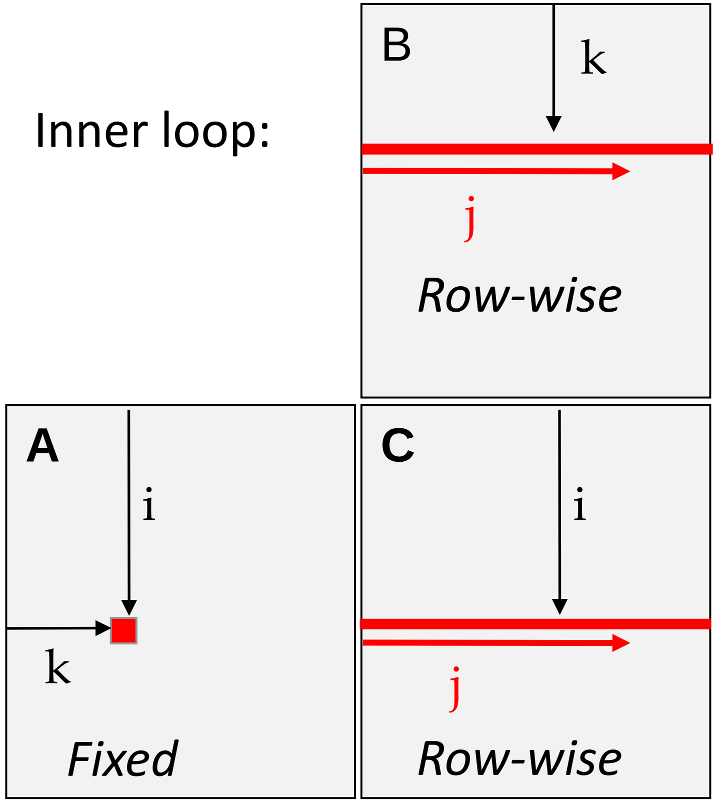
Total misses/iteration: 1.25

Remember: Line size = 32B
(big enough for four 64-bit longs)

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];
  }
}
```

Inner loop:



Misses per inner loop iteration:

A	B	C
0	0.25	0.25

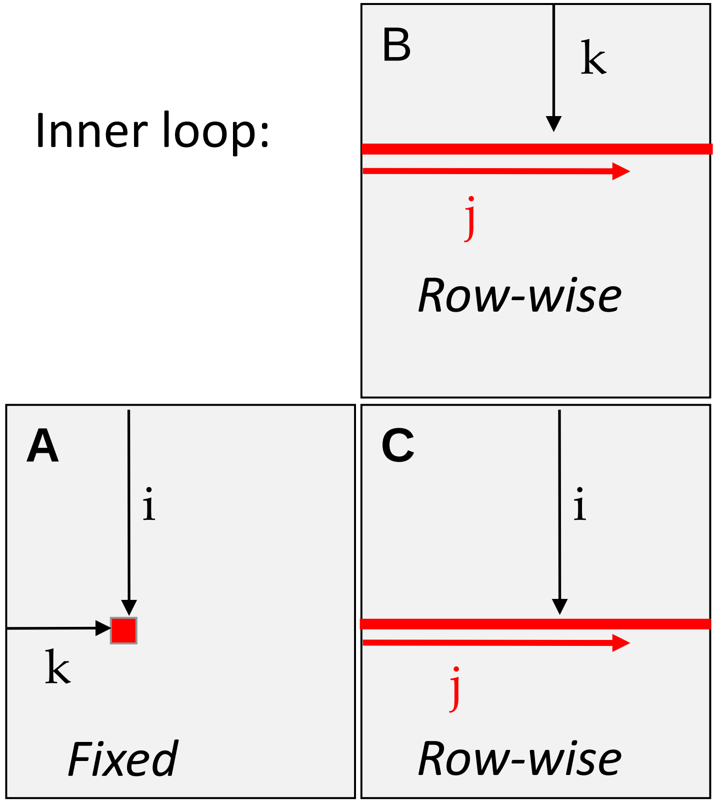
Total misses/iteration: 0.5

Remember: Line size = 32B
(big enough for four 64-bit longs)

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];
  }
}
```

Inner loop:



Misses per inner loop iteration:

A	B	C
0	0.25	0.25

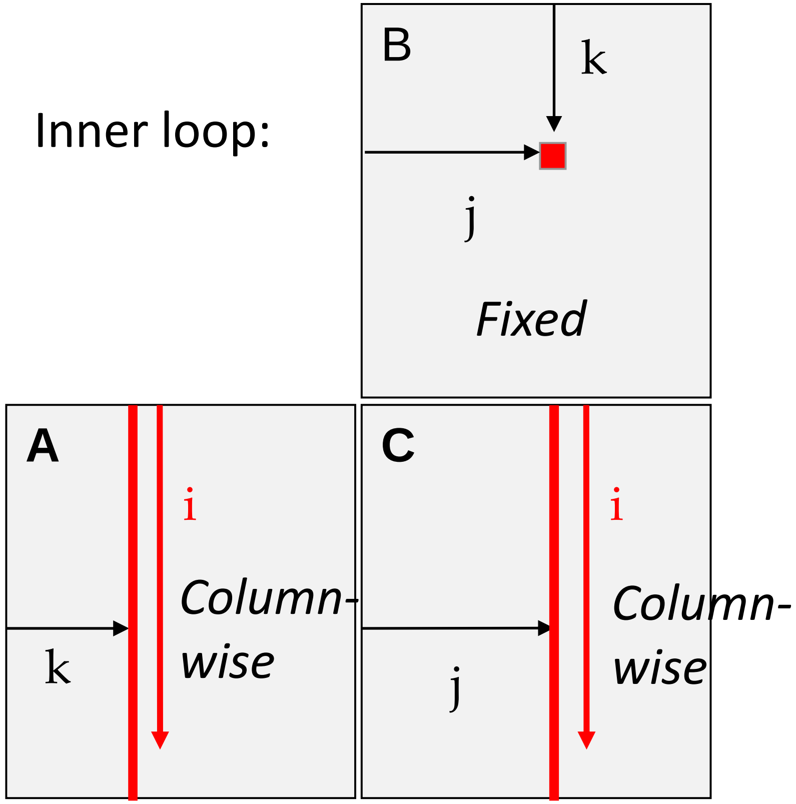
Total misses/iteration: 0.5

Remember: Line size = 32B
(big enough for four 64-bit longs)

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;
  }
}
```

Inner loop:



Misses per inner loop iteration:

A	B	C
1	0	1

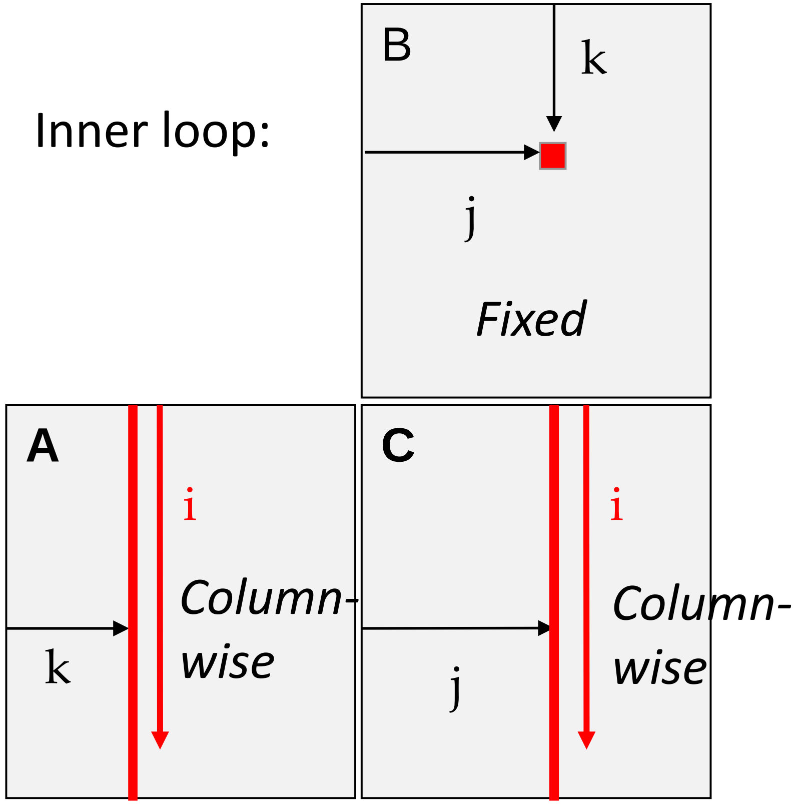
Total misses/iteration: 2

Remember: Line size = 32B
(big enough for four 64-bit longs)

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;
  }
}
```

Inner loop:



Misses per inner loop iteration:

A	B	C
1	0	1

Total misses/iteration: 2

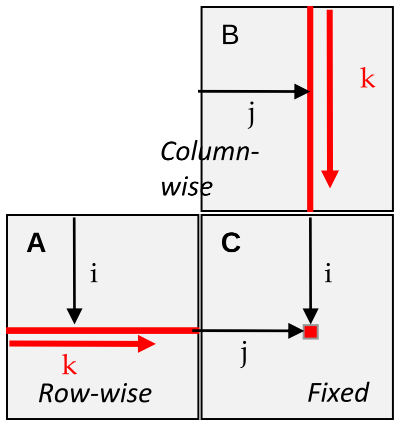
Remember: Line size = 32B
(big enough for four 64-bit longs)

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):

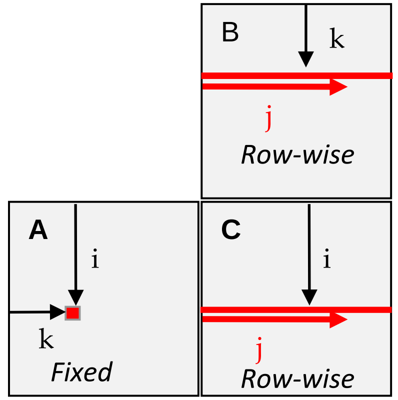
- 2 loads, 0 stores
- misses/iter = 1.25




```
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):

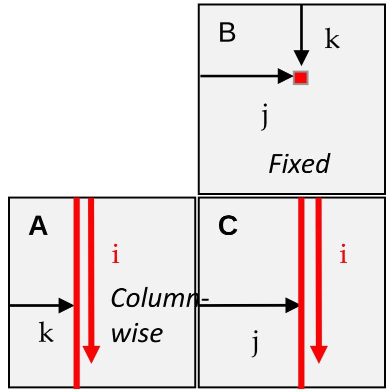
- 2 loads, 1 store
- misses/iter = 0.5




```
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;
 }
}
```

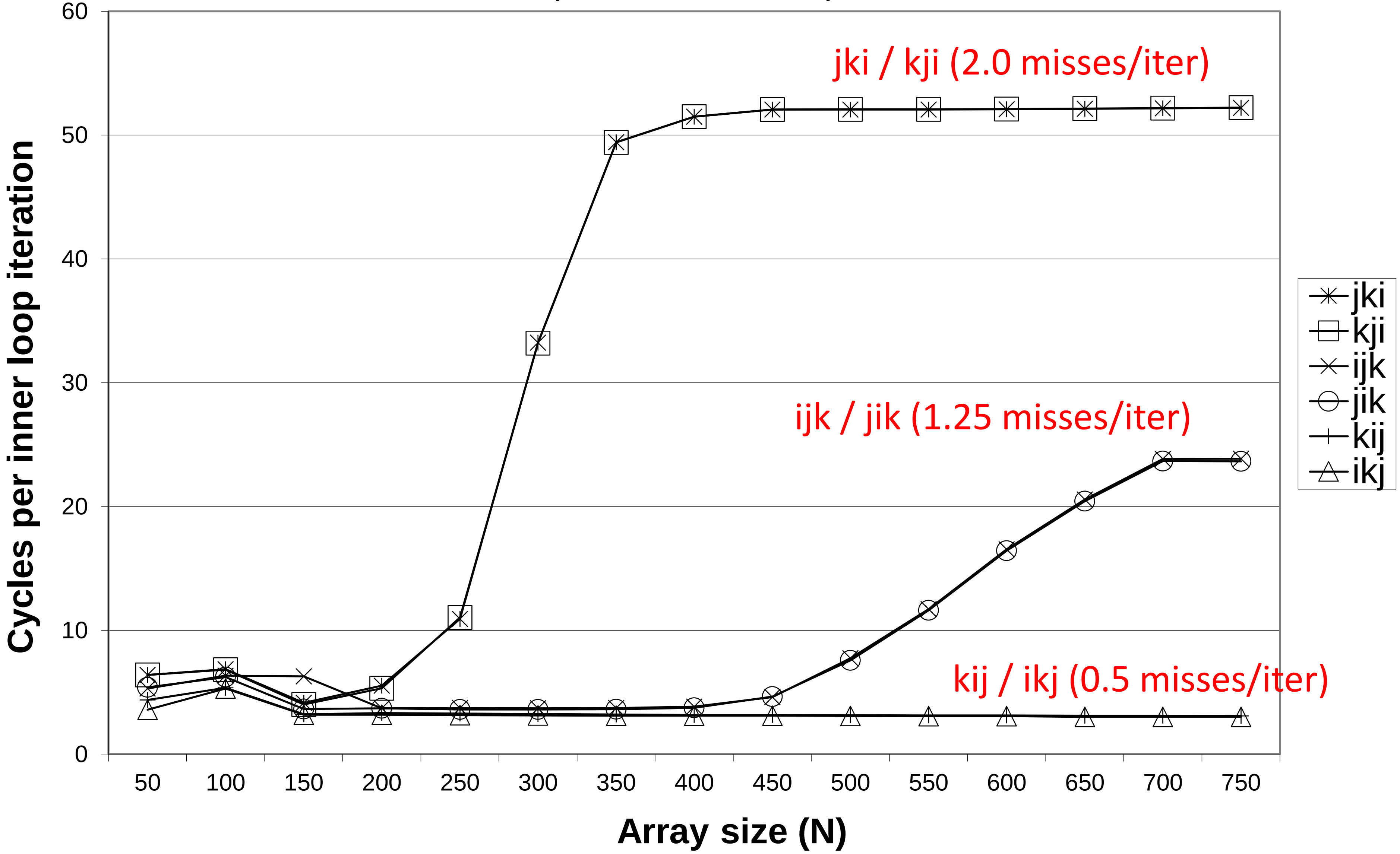
jki (& kji):

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



Core i7 Matrix Multiply Performance

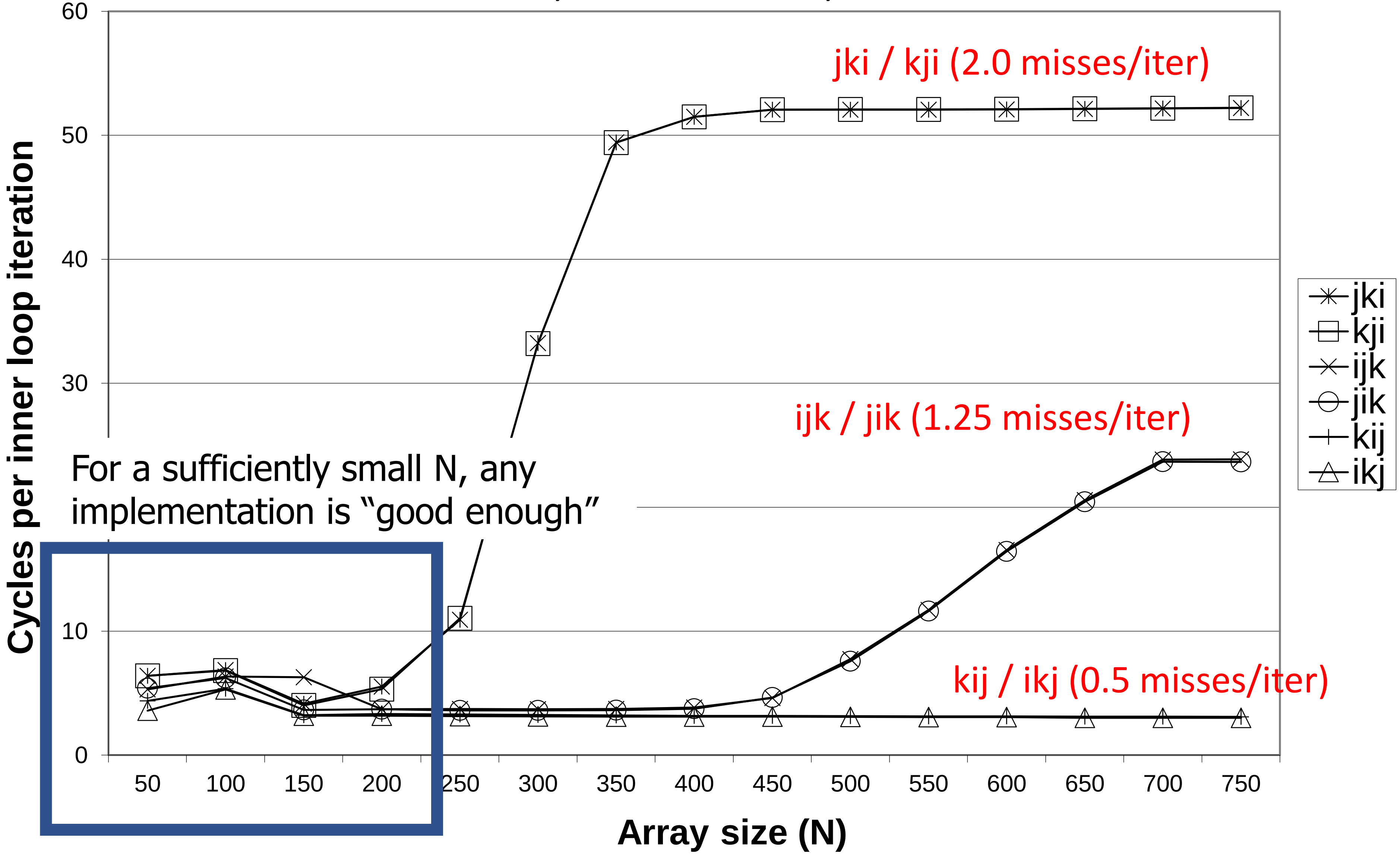
Essentially the same algorithm, just different data access patterns!
The most natural way to write code may not be the best one!



Core i7 Matrix Multiply Performance

Essentially the same algorithm, just different data access patterns!

The most natural way to write code may not be the best one!



Break + Open Question

- What about those writes? Do they have additional costs?

Break + Open Question

- What about those writes? Do they have additional costs?
  - Assumption: write-back cache such that they don't cost more than reads until evicted

  - As long as evictions of modified (dirty) data happen once per array cell, we're equivalent to the one write outside of the for loop
    - This is not the case here since entire row doesn't fit in cache

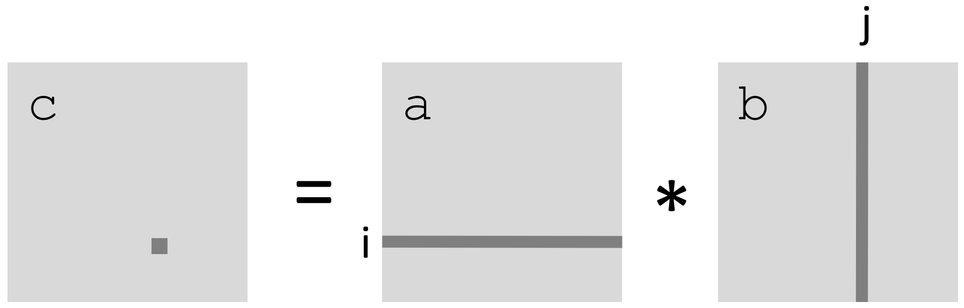
  - If evictions of modified (dirty) data happen multiple times per array cell, question becomes complicated
    - How much does that hurt compared to extra cache misses?
    - Writes can happen in the background (while processor is running)
    - Likely need to measure real-world performance to understand

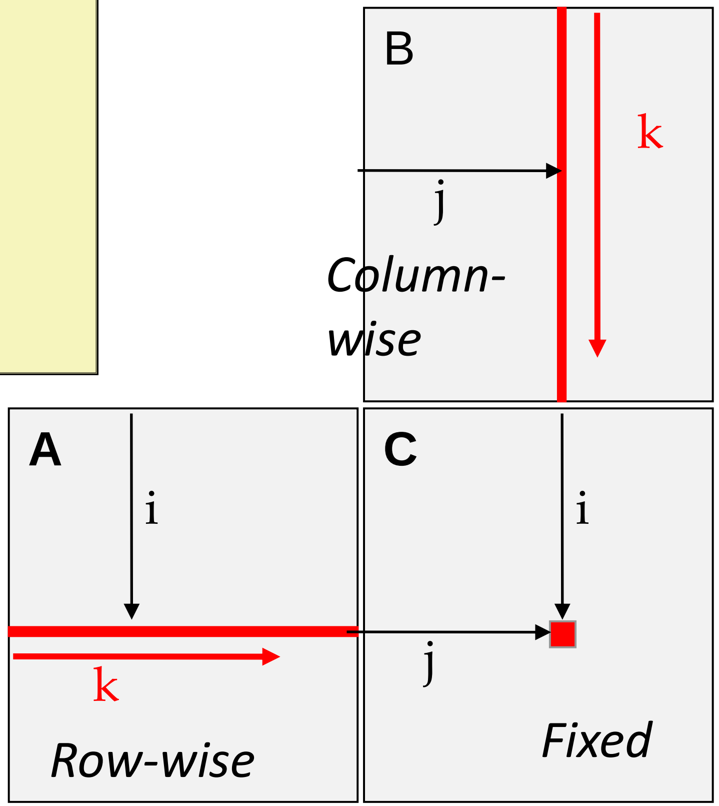
Outline

- Memory Mountain
- Cache Metrics
- Cache Performance for Arrays
- **Improving code**
  - Rearranging Matrix Math
  - **Matrix Math in Blocks**

Example: Matrix Multiplication

```
double *c = (double *) malloc(sizeof(double)*n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      double sum = 0.0;
      for (int k = 0; k < n; k++) {
        sum += a[i*n + k] * b[k*n + j];
      }
      c[i*n+j] = sum;
} } }
```

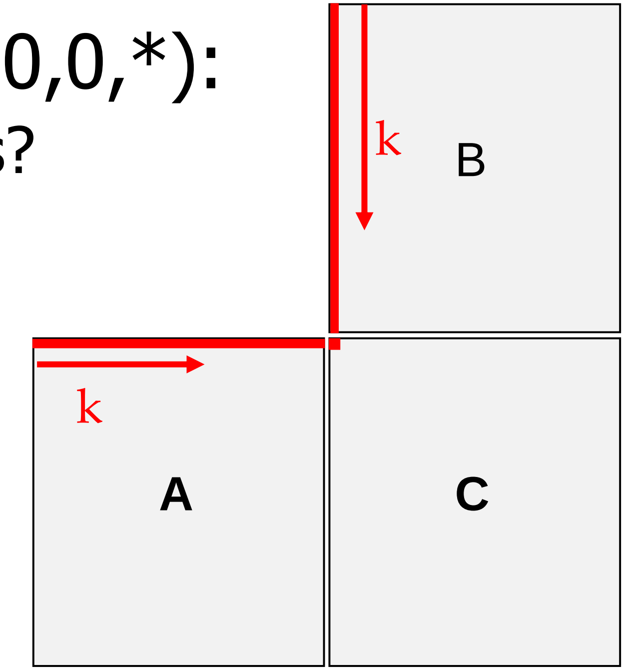




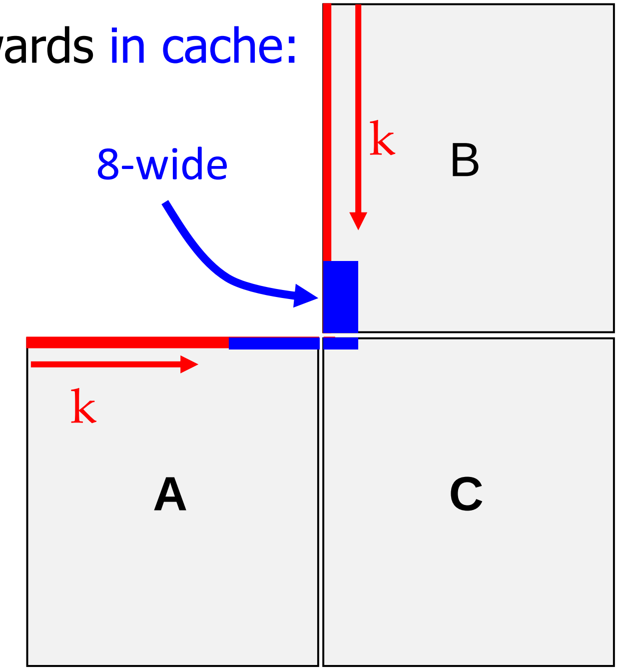
Cache Miss Analysis (approximate)

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

- 1st iteration (i,j,k=0,0,*):
  - How many misses?
  - n/8 + n + 1 = 9n/8+1 misses



Afterwards in cache:

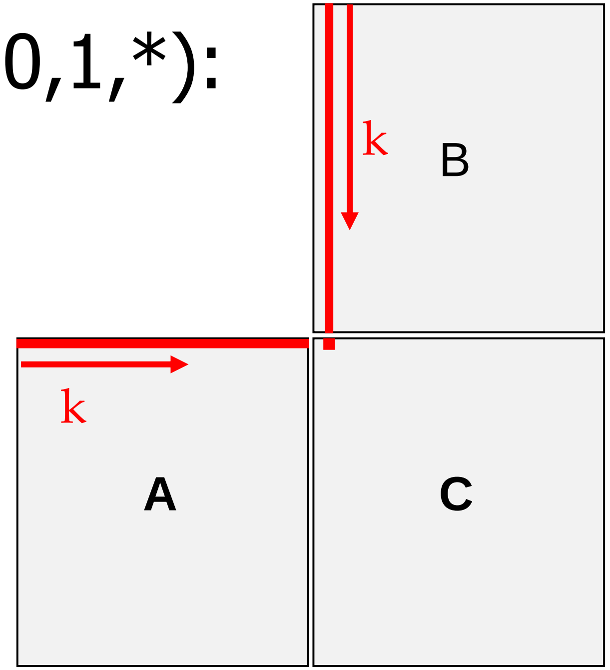


Cache Miss Analysis (approximate)

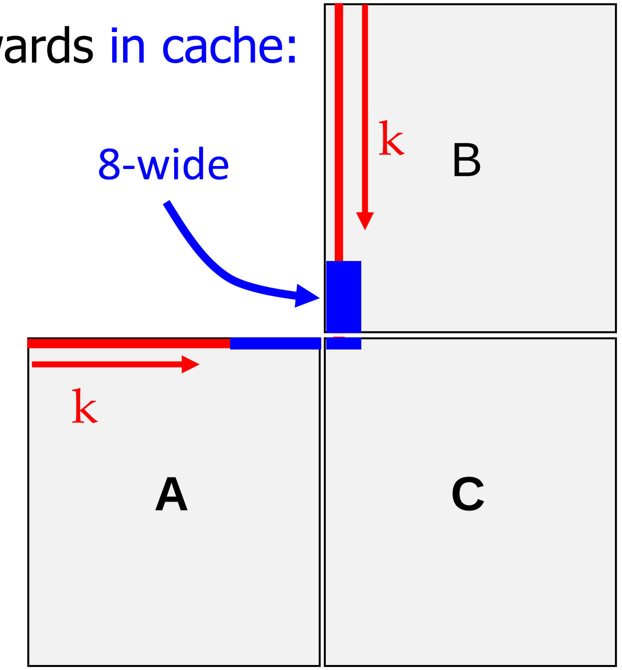
- Assume:
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C <<< n (much smaller than n)

- **Total misses:**
  - Every iteration: $9n/8 + 1$
  - # iterations: n^2
  - $(9n/8+1)*n^2 = (9/8)*n^3 + n^2$

- 2^{nd} iteration (i,j,k=0,1,*):
  - Again:
    $n/8 + n + 1 =$
    $9n/8+1$ misses



Afterwards in cache:

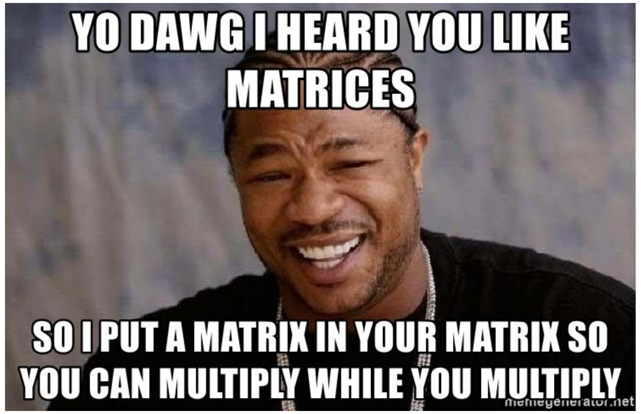


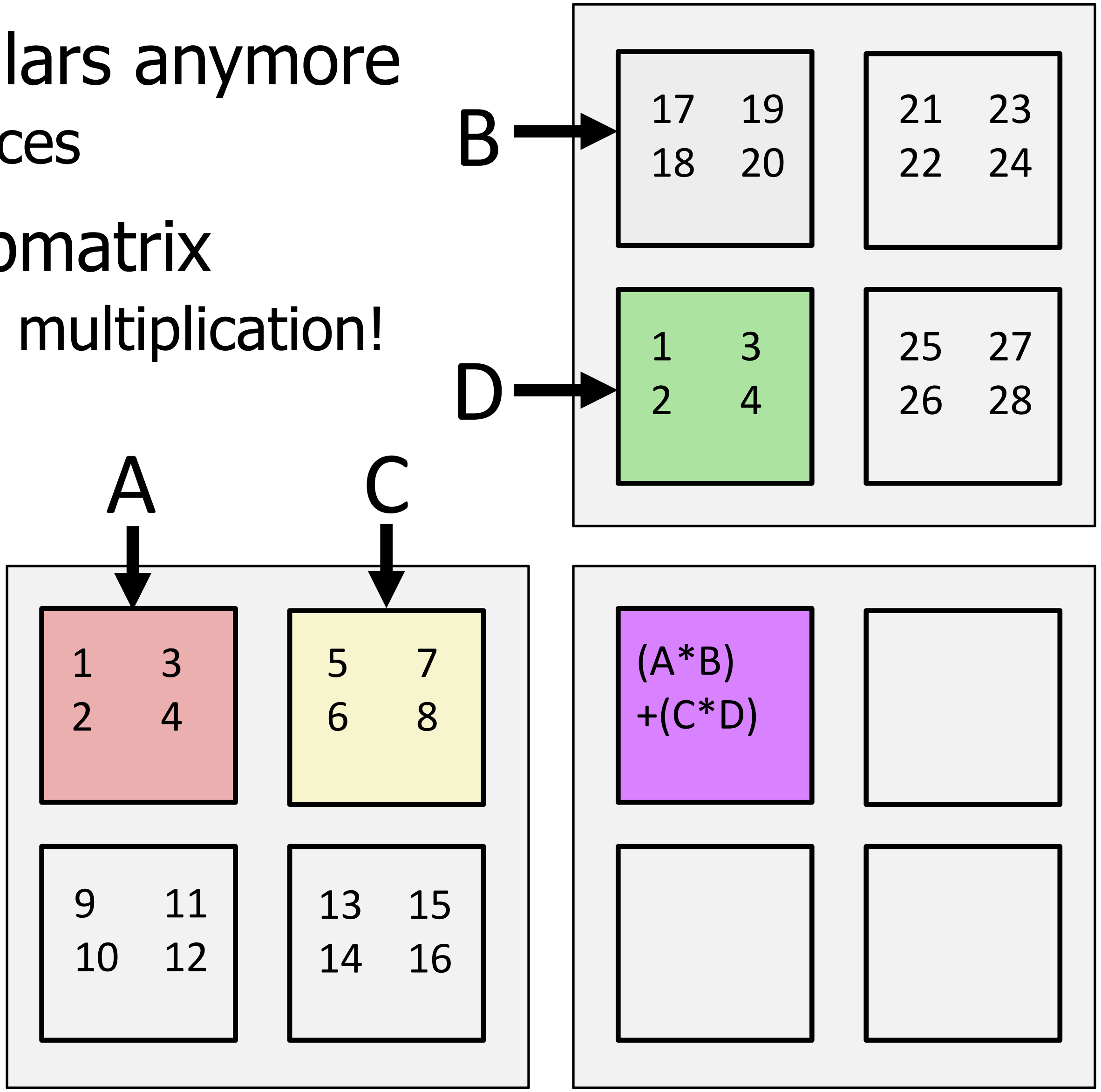
Enter Blocking Algorithms

- Special class of algorithms designed specifically to have excellent temporal and spatial locality
- Key idea: don't operate on individual elements; instead operate on *blocks* !
    - Treat the overall matrices as containing submatrices as elements
        - See next slide
- General principle: use a piece of data as much as we can
    - Then it's ok to kick it out of the cache
    - As opposed to using, kicking out, using again later, and so on
- Same result, but much nicer locality!
    - And thus can leverage the cache better (more hits, fewer misses)
    - Still same computational complexity
- May get a bit mind bending
    - I want you to understand the general principle
    - But you don't need to fully understand the details of the algorithm

Matrices as Matrices of Submatrices

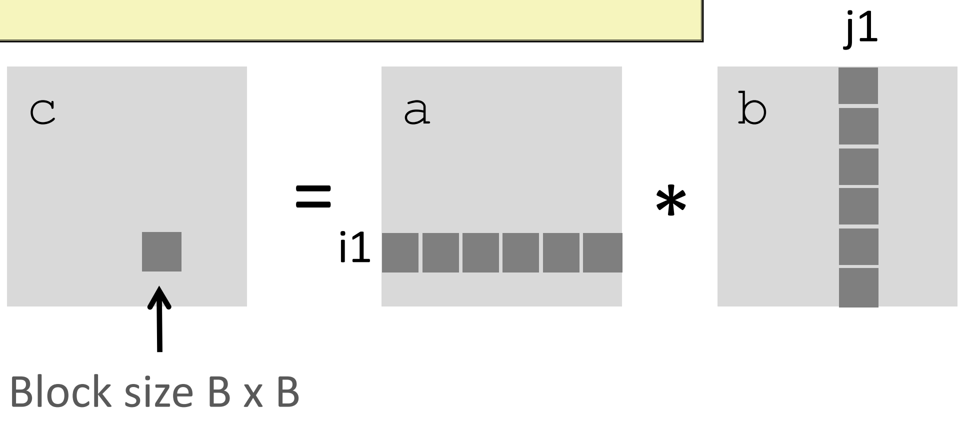
- Elements of are not scalars anymore
  - But rather smaller matrices
- To compute a result submatrix
  - Just do a smaller matrix multiplication!





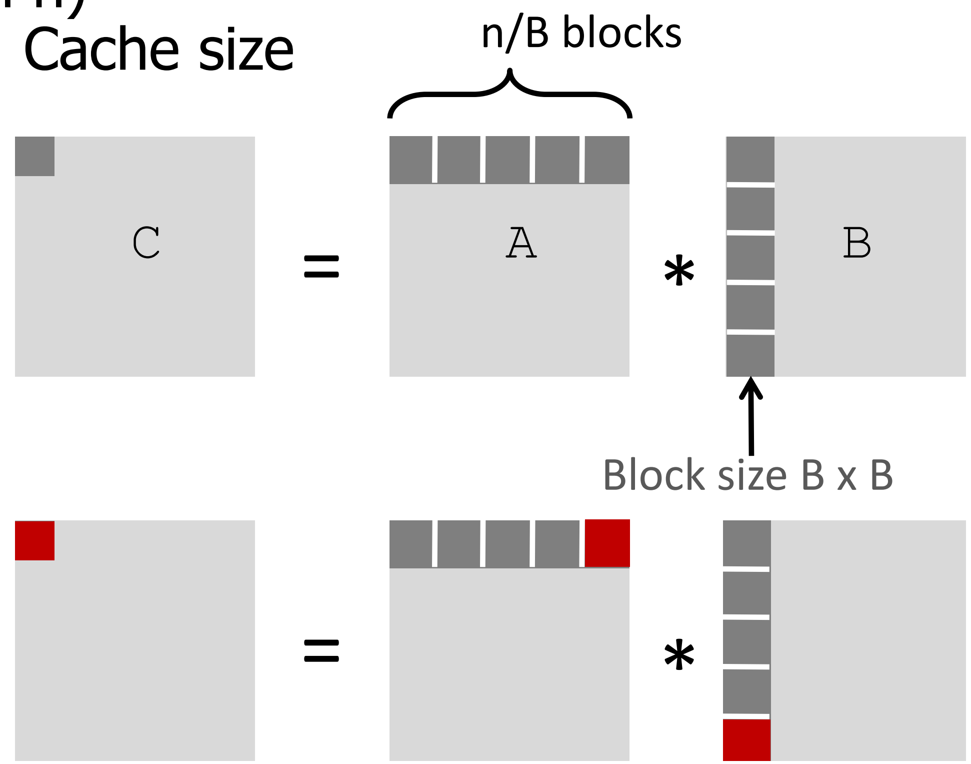
Blocked Matrix Multiplication

```
double * c = (double *) malloc(sizeof(double)*n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
  for (int i = 0; i < n; i+=B) {
    for (int j = 0; j < n; j+=B) {
      for (int k = 0; k < n; k+=B) {
        /* B x B mini matrix multiplications */
        for (int i1 = i; i1 < i+B; i1++) {
          for (int j1 = j; j1 < j+B; j1++) {
            double sum = 0.0;
            for (int k1 = k; k1 < k+B; k1++) {
              sum += a[i1*n + k1] * b[k1*n + j1];
            }
            c[i1*n + j1] = sum;
} } } } } }
```



Cache Miss Analysis (approximate)

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

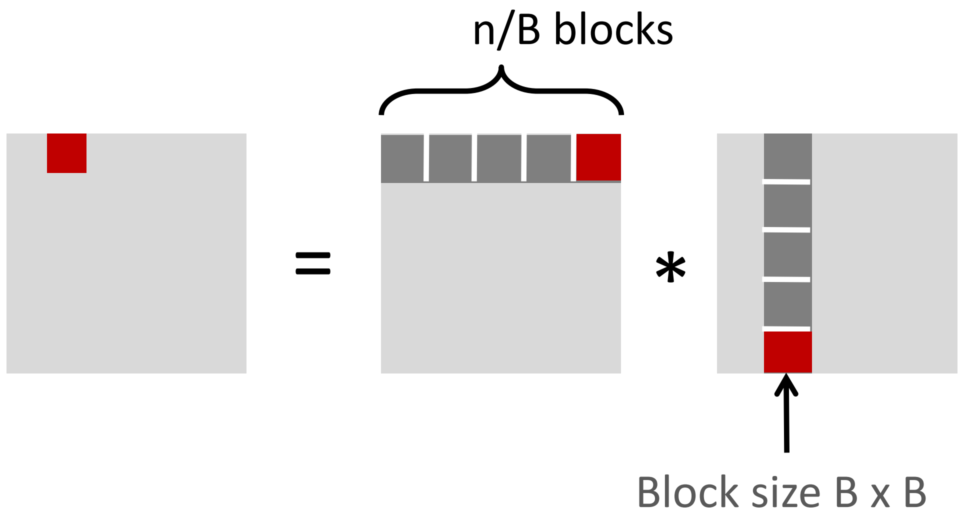


- First (block) iteration:
  - $B^2/8$ misses for any given block
  - $2B^2/8$ misses for each BxB-block multiplication (only counting A, B misses)
  - # BxB multiplications: n/B
  - $B^2/8$ misses for C[ ] block total
  - $2B^2/8 * n/B + B^2/8 = nB/4 + B^2/8$
  - Afterwards in cache
    - No waste! We used all that we brought in!

Cache Miss Analysis (approximate)

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

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



- Total misses:
  - #block iterations: $(n/B)^2$
  - $(nB/4 + B^2/8) * (n/B)^2 = n^3/(4B) + n^2/8$

Performance Impact

- Misses without blocking: $(9/8) * n^3 + n^2$
- Misses with blocking: $1/(4B) * n^3 + 1/8 * n^2$

- Largest possible block size B, but limit $3B^2 < C \rightarrow B = \lfloor \sqrt{C/3} \rfloor$
  - e.g., Cache size = 32K = 32,768 Bytes, then pick B = 104 (note: 104=13*8)
  - No blocking: $1.125*n^3 + n^2$

    468x 8x

  - Blocking: $0.0024*n^3 + 0.125*n^2$

- Reason for dramatic difference:
  - Matrix multiplication has inherent temporal locality
  - But program has to be written properly to take advantage of it

Takeaways

- Writing code to take advantage of the cache is challenging
  - It's totally possible, but high effort

- Generally: maximize spatial and temporal locality
  - Use elements close to each other (moving horizontally in 2D array)
  - Use the same element as many times as possible in a row (output)

- Well-designed math libraries will do this for you!
  - MATLAB, Mathematica, R, SciPy, etc.
  - Jack Dongarra won a Turing award for this!

Outline

- Memory Mountain
- Cache Metrics
- Cache Performance for Arrays
- Improving code
  - Rearranging Matrix Math
  - Matrix Math in Blocks