L10 Cache Optimizations

Zonghua Gu, 2018

Acknowledgement: some slides taken from UC Berkeley CS61C

Optimizations for the Memory Hierarchy

- Write code that has locality!
  - Spatial: access data contiguously
  - Temporal: make sure access to the same data is not too far apart in time

- How can you achieve locality?
  - Adjust memory accesses in *code* (software) to improve miss rate (MR)
    - Requires knowledge of *both* how caches work as well as your system's parameters
  - Proper choice of algorithm
  - Loop transformations

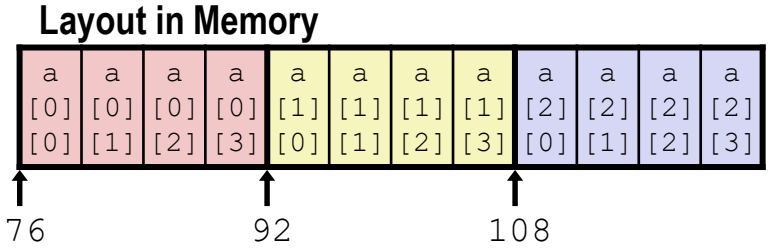
Example: Matrix Sum

- Consider the following two loops, which calculate the sum of the entries in a 3x4 matrix A of 32-bit integers:

Loop A	*Loop B*

`sum = 0;`
`for (i = 0; i < 3; i++)`
`  for (j = 0; j < 4; j++)`
`  sum += A[i][j];`
	`sum = 0;`
	`for (j = 0; j < 4; j++)`
	`  for (i = 0; i < 3; i++)`
	`  sum += A[i][j];`

- Matrix A is stored contiguously in memory in row-major order. Row major order means that elements in the same row of the matrix are adjacent in memory (A[i][j] resides in memory location [4*(64*i + j)]).

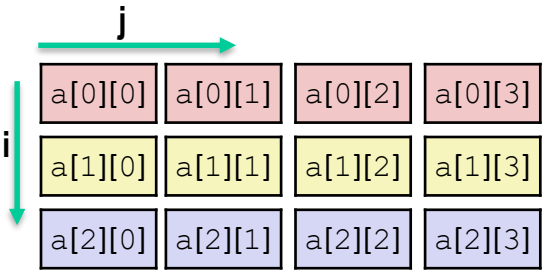


Note: 76 is just one possible starting address of A

Matrix Sum Loop A

Loop A

```
sum = 0;
for (i = 0; i < 3; i++)
  for (j = 0; j < 4; j++)
    sum += A[i][j];
```



Access Pattern:
stride = 1

1) a[0][0]
2) a[0][1]
3) a[0][2]
4) a[0][3]
5) a[1][0]
6) a[1][1]
7) a[1][2]
8) a[1][3]
9) a[2][0]
10) a[2][1]
11) a[2][2]
12) a[2][3]

Layout in Memory

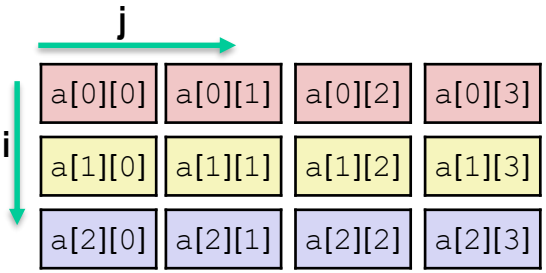
a [0] [0]	a [0] [1]	a [0] [2]	a [0] [3]	a [1] [0]	a [1] [1]	a [1] [2]	a [1] [3]	a [2] [0]	a [2] [1]	a [2] [2]	a [2] [3]

- Index j is incremented in the inner loop
- Good spatial locality for a[][]
- No temporal locality for a[][]
  - Each element is used only once

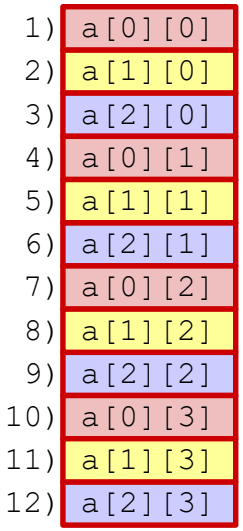
Matrix Sum Loop B

Loop B

```
sum = 0;
for (j = 0; j < 4; j++)
  for (i = 0; i < 3; i++)
    sum += A[i][j];
```



Access Pattern:
stride = 4

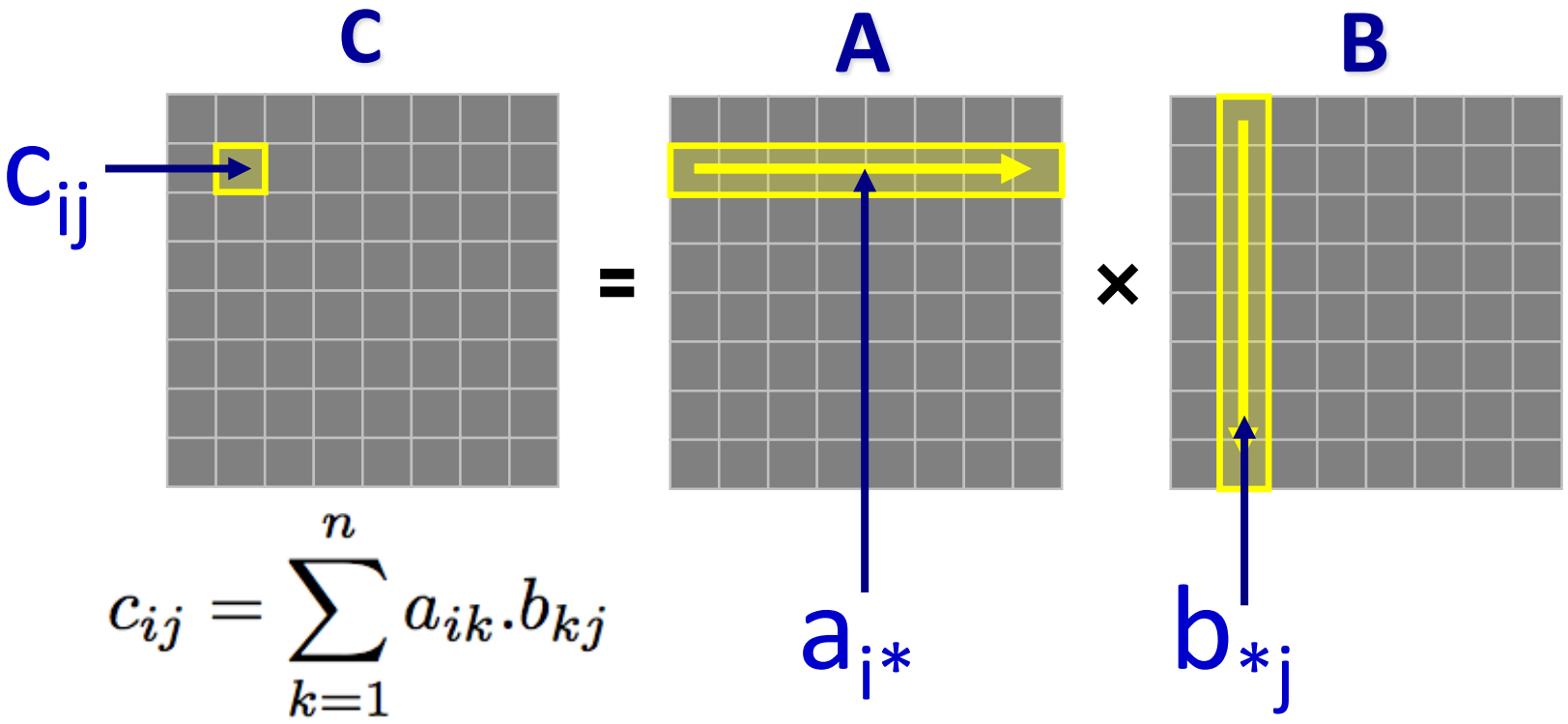


Layout in Memory

a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[2][0]	a[2][1]	a[2][2]	a[2][3]

- Index i is incremented in the inner loop
- Poor spatial locality for a[][]
- No temporal locality for a[][]
  - Each element is used only once

Example: Matrix Multiplication

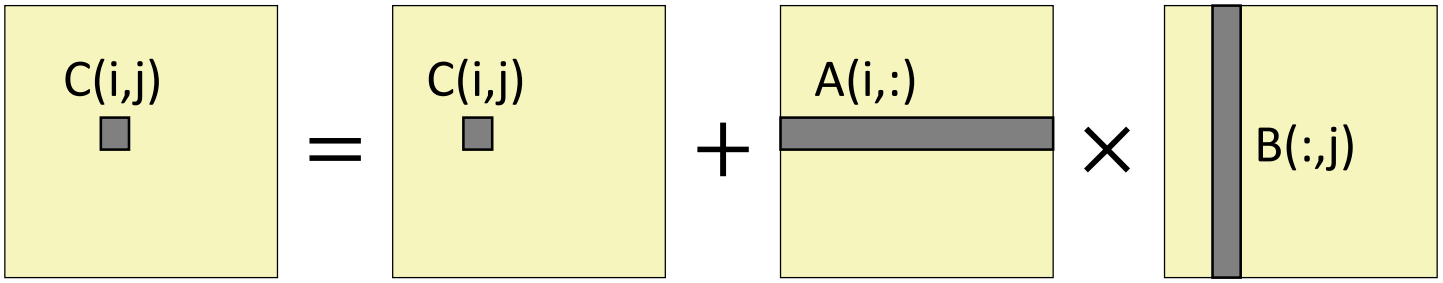


$$c_{ij} = \sum_{k=1}^{n} a_{ik}.b_{kj}$$

The mathematical cache miss analysis in this example will not be covered in the exam. You only need to grasp the basic concept of cache blocking.

Naïve Matrix Multiply

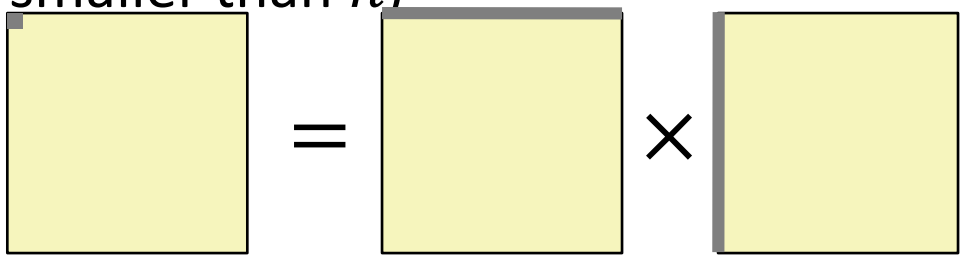
```
# move along rows of A
for (i = 0; i < n; i++)
  # move along columns of B
  for (j = 0; j < n; j++)
    # EACH k loop reads row of A, col of B
    # Also read & write c(i,j) n times
    for (k = 0; k < n; k++)
      C[i][j] += A[i][k] * B[k][j];
      //Equivalent form
      //C[i*n+j] += A[i*n+k] * B[k*n+j];
```



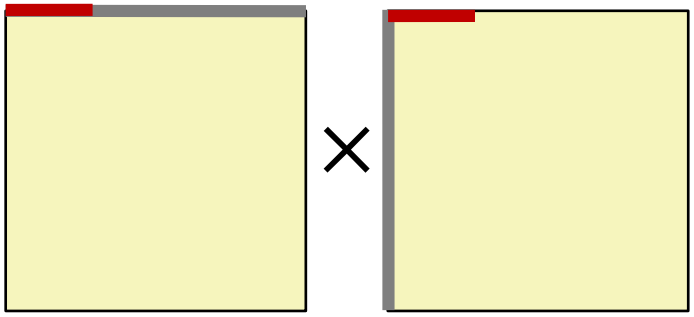
Cache Miss Analysis (Naïve)

Ignoring matrix C

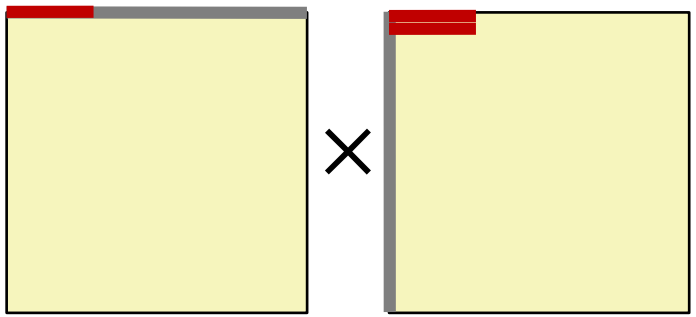
- Scenario Parameters:
  - Square matrix ($n \times n$), elements are `doubles`
  - FA (Fully Associative) LRU cache, block size B = 64 B = 8 doubles
  - Cache size $C \ll n$ (much smaller than n)
- Each iteration:
  - $\frac{n}{8} + n = \frac{9n}{8}$ misses



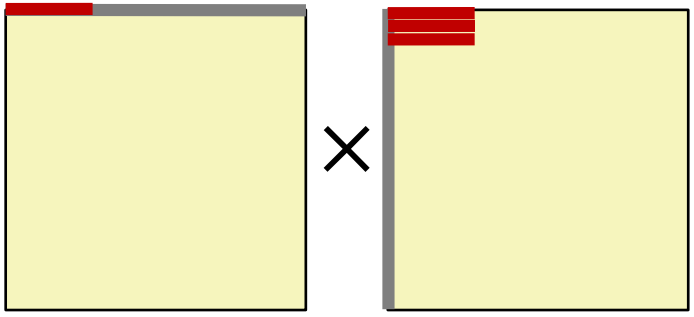
- Each cache block holds 8 elements
- Reading one row of Matrix A
  - good spatial locality with stride 1
  - 1 cache miss every 8 elements; total cache misses due to reading one row of A is n/8
- Reading one column of Matrix B
  - poor spatial locality with stride n
  - 1 cache miss every element; total cache misses due to reading one column of B is n



- 1st cache miss brings 8 elements A[0][0], A[0][1] … A[0][7] into cache
- 2nd cache miss brings 8 elements B[0][0], B[0][1]… B[0][7] into cache
- Compute A[0][0]*B[0][0]; Only B[0][0] is used

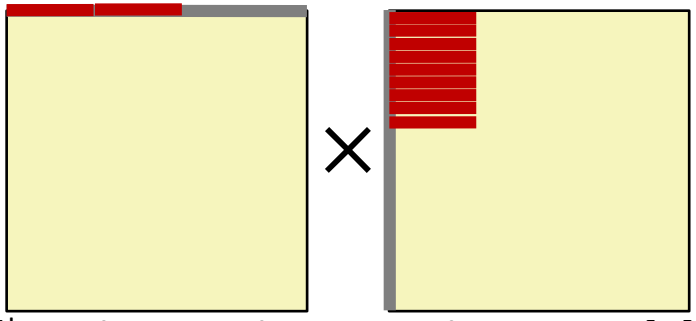


- 3rd cache miss brings 8 elements B[1][0], B[1][1] … B[1][7] into cache
- Compute A[0][1]*B[1][0]
- Only B[1][0] is used

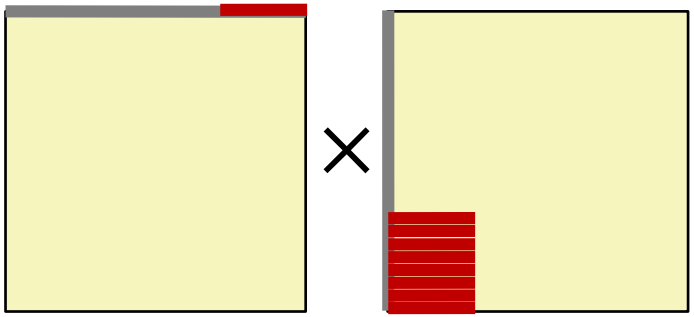


- 4th cache miss brings 8 elements B[2][0], B[2][1] … B[2][7] into cache
- Compute A[0][2]*B[2][0]
- Only B[2][0] is used

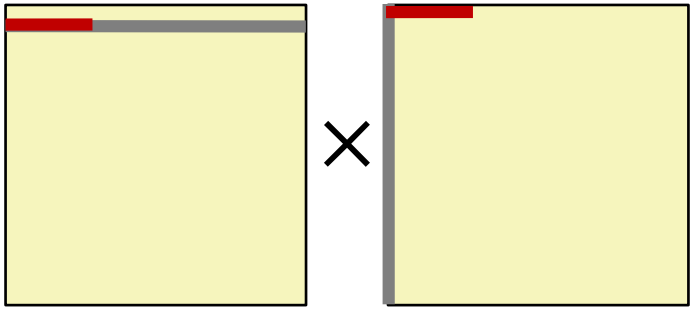
…



- 9th cache miss brings 8 elements A[0][8], A[0][9] … A[0][15] into cache
- 10th cache miss brings 8 elements B[8][0], B[8][1] … B[8][7] into cache
- Compute A[0][8]*B[8][0]



- At the end of multiplying 0^{th} row of A[0][0…n-1] with 0^{th} column of B[0…n-1][0], this is what the cache contains (assuming cache is quite small compared to matrix size)



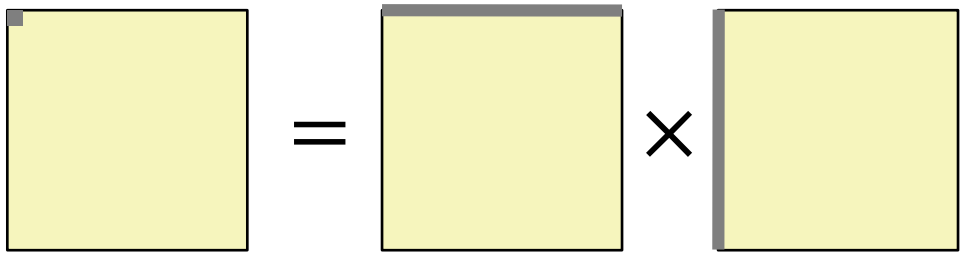
- By the time we get to 1^{st} row of A[1][0…n-1], the top left elements of B have been evicted from the cache, so every element of B has to be fetched again from memory, even though they have been used before. So the process repeats.

Cache Miss Analysis (Naïve)

Ignoring matrix `C`

- Scenario Parameters:
  - Square matrix ($n \times n$), elements are `doubles`
  - Cache block size B = 64 B = 8 doubles
  - Cache size $C \ll n$ (much smaller than n)

- Each iteration:
  - $\frac{n}{8} + n = \frac{9n}{8}$ misses



- Total misses: $\frac{9n}{8} \times n^2 = \frac{9}{8}n^3$

n iterations of the doubly-nested loop

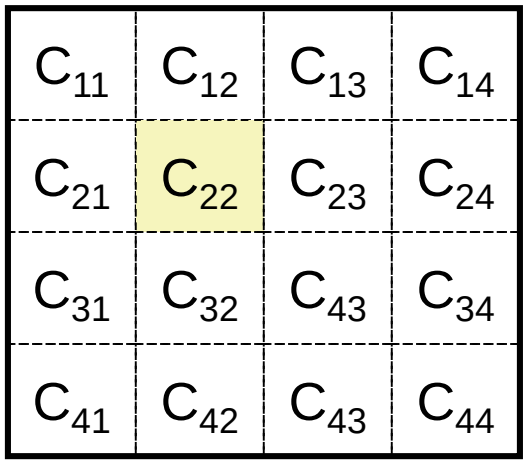
Cache Blocking (1)

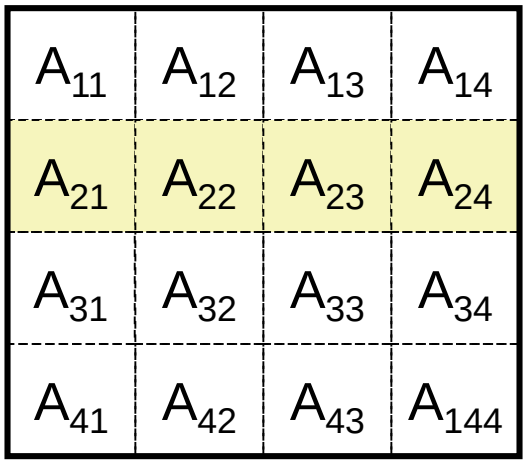
- Can get the same result of a matrix multiplication by splitting the matrices into smaller submatrices (matrix "blocks")
- For example, multiply two 4×4 matrices:

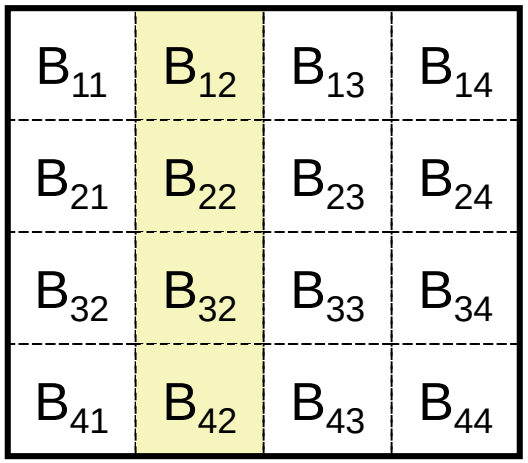
$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \text{ with } B \text{ defined similarly.}$$

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

Cache Blocking (2)







Matrices of size $n \times n$, split into 4 blocks of size r (n=$4r$)

$C_{22} = A_{21}B_{12} + A_{22}B_{22} + A_{23}B_{32} + A_{24}B_{42} = \sum_k A_{2k} * B_{k2}$

- Multiplication operates on small "block" matrices
  - Choose size so that they fit in the cache!
  - This technique called "*blocking*"

Blocked Matrix Multiply

- Blocked version of the naïve algorithm:

```
# move by rxr BLOCKS now
for (i = 0; i < n; i += r)
  for (j = 0; j < n; j += r)
    for (k = 0; k < n; k += r)
      # block matrix multiplication
      for (ib = i; ib < i+r; ib++)
        for (jb = j; jb < j+r; jb++)
          for (kb = k; kb < k+r; kb++)
            c[ib][jb] += a[ib][kb]*b[kb][jb];
            //Equivalently
            //c[ib*n+jb] += a[ib*n+kb]*b[kb*n+jb];
```

- r = block matrix size (assume r divides n evenly)

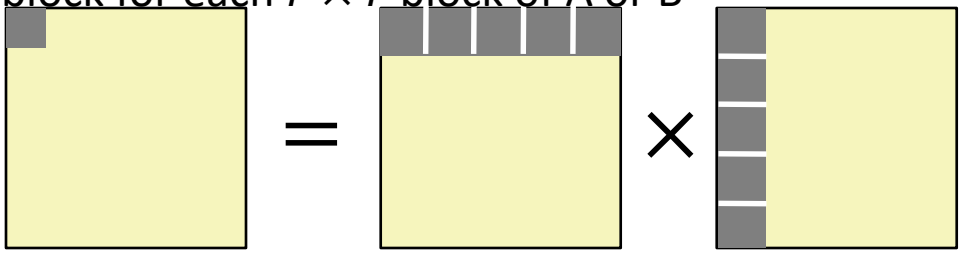
Cache Miss Analysis (Blocked)

Ignoring matrix C

- Scenario Parameters:
  - Cache block size B = 64 B = 8 doubles
  - Cache size $C \ll n$ (much smaller than n)
  - At least two blocks ■ $(r \times r)$ fit into cache: $2r^2 < C$; r^2 is multiple of 8

r^2 elements per block, 8 misses per cache block for each $r \times r$ block of A or B

n/r blocks



- Each block iteration:
  - $r^2/4$ misses per block
  - $n/r \times r^2/4 = nr/4$

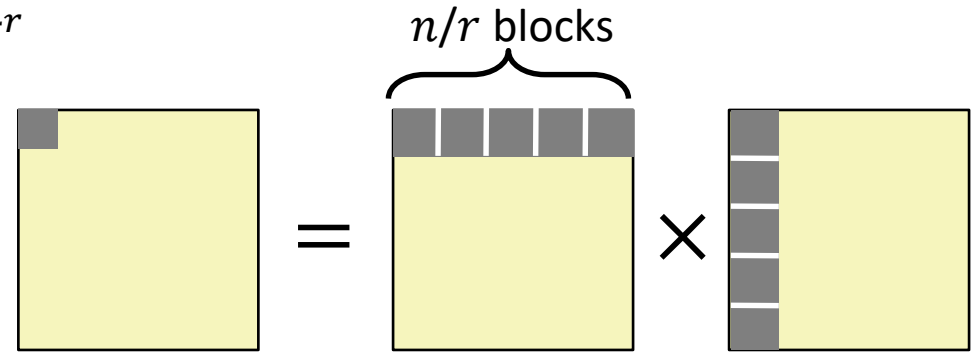
n/r blocks in row and column

- Total misses:
  - $\frac{nr}{4} \times \left(\frac{n}{r}\right)^2 = \frac{n^3}{4r} \ll \frac{9}{8}n^3$ (naive approach)

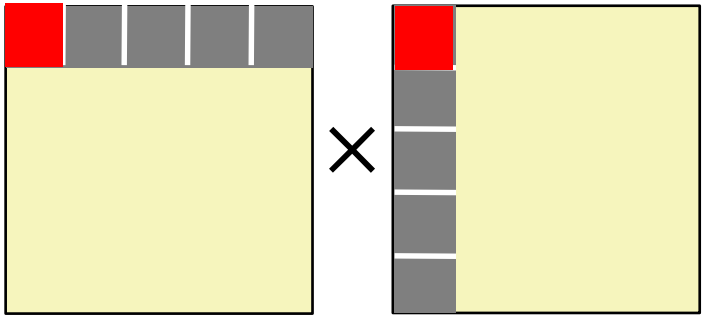
Q: why r=1 is not a special case of the naïve approach?
A: r=1 does not satisfy condition "r^2 is multiple of 8"

Detailed Cache Miss Analysis (Blocked)

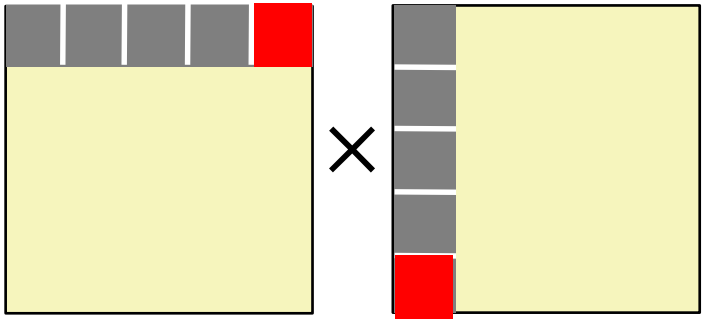
- Consider one small $r \times r$ block of C. We first multiply one $r \times r$ block of A with one $r \times r$ block of B, cache misses:
  - $\frac{r^2}{8} + \frac{r^2}{8} = \frac{r^2}{4}$
- Repeat this for all n/r $r \times r$ blocks of A (the top most "thick row", and all n/r $r \times r$ blocks of B (the leftmost "thick column"), # cache misses for computing the top left $r \times r$ block of C is:
  - $n/r \times r^2/4 = nr/4$
- You need to do this for all $(n/r)^2$ $r \times r$ blocks of C in order to compute the big $n \times n$ matrix C, so total # cache misses is:
  - $\frac{nr}{4} \times \left(\frac{n}{r}\right)^2 = \frac{n^3}{4r}$



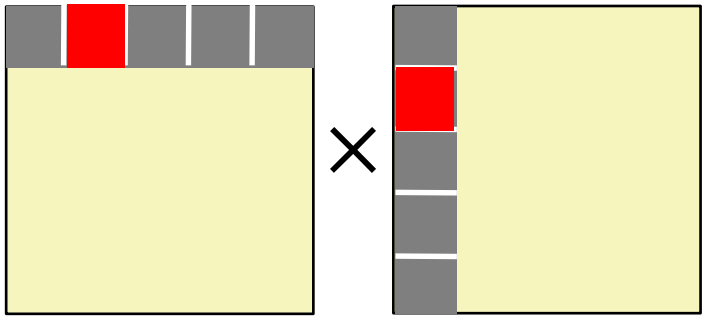
Cache Miss Analysis (Blocked)



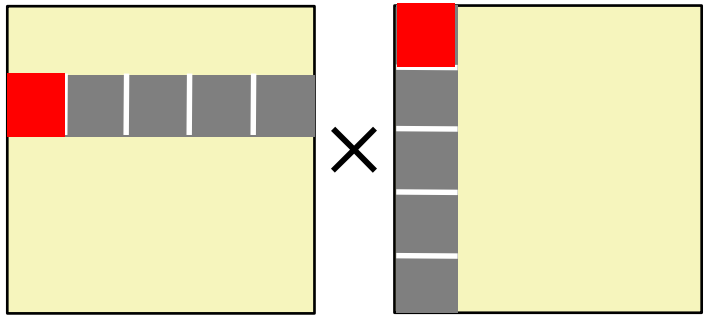
- After finishing 1st block matrix multiply: $r^2/8$ misses for A, and $r^2/8$ misses for B



- After finishing nth block matrix: $(r^2/8) * n/r$ misses for A, $(r^2/8) * n/r$ misses for B, with total of $nr/4$ misses.



- After finishing 2nd block matrix multiply: $r^2/8$ misses for B



- Process repeats for $\left(\frac{n}{r}\right)^2$ times
- Total misses: $\frac{nr}{4} \times \left(\frac{n}{r}\right)^2 = \frac{n^3}{4r}$

Cache-Friendly Code

- Programmer can optimize for cache performance
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - Blocking is a general technique
- All systems favor “cache-friendly code”
  - Getting absolute optimum performance is very platform specific
    - Cache size, cache block size, associativity, etc.
  - Can get most of the advantage with generic code
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)
    - Focus on inner loop code

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