Memory & Caches IV
CSE 351 Spring 2024

Instructor:

Elba Garza

Teaching Assistants:

Ellis Haker
Adithi Raghavan
Aman Mohammed
Brenden Page
Celestine Buendia
Chloe Fong
Claire Wang
Hamsa Shankar
Maggie Jiang
Malak Zaki
Naama Amiel
Nikolas McNamee
Shananda Dokka
Stephen Ying
Will Robertson

The cache when you ask for something that was just evicted:

Playlist: [CSE 351 24Sp Lecture Tunes!]

Announcements, Reminders

- Happy Midterm madness!
- Mid-Quarter Survey on Canvas due tonight!
- HW 16 also due tonight! HW 17/18 due Friday (10 May).
- Lab 3 due Wednesday by 11:59 PM
- Lab 4 releasing on Wednesday-ish.
  - HW 19 helps you prepare for Lab 4 🫡

Reading Review

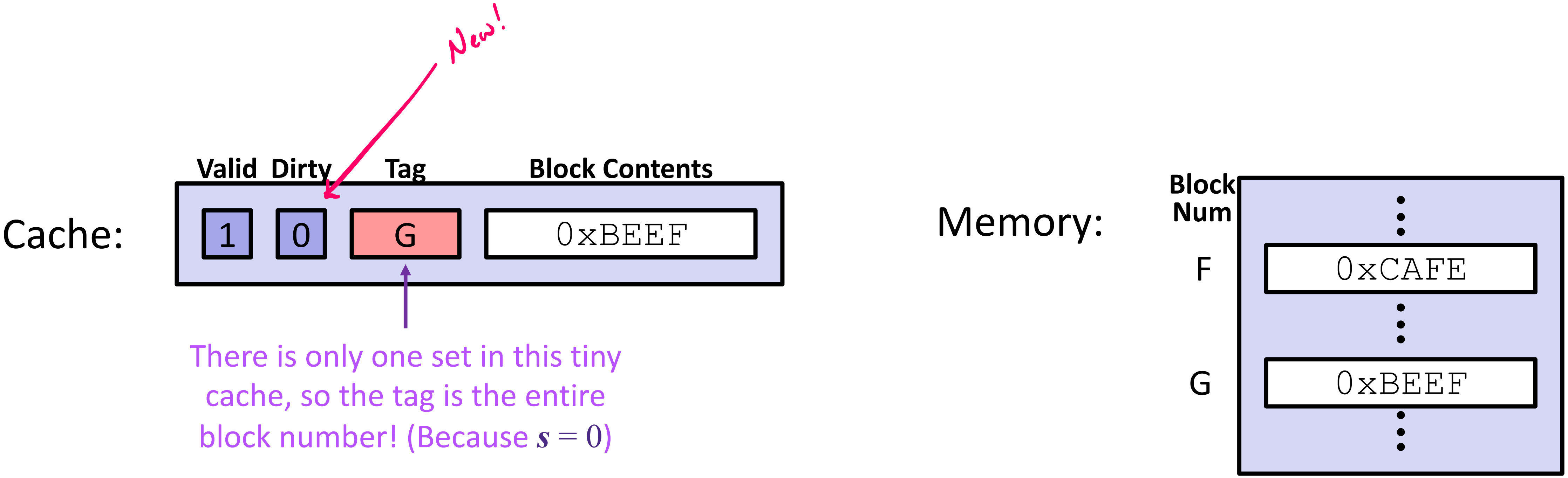
- Terminology:
  - Write-hit policies: write-back, write-through
  - Write-miss policies: write allocate, no-write allocate
  - Cache blocking

What about writes? (Review)

- Multiple copies of data may exist:
  - multiple levels of cache and main memory
- What to do on a write-hit (data already in cache)?
  - Write-through: write immediately to next level
  - Write-back: defer write to next level until line is evicted (replaced)
    - Must track which cache lines have been modified (using the "dirty bit")
- What to do on a write-miss (data not in cache)?
  - Write allocate: ("fetch on write") load into cache, then execute the write-hit policy
    - Good if more writes or reads to the location follow
  - No-write allocate: ("write around") just write immediately to next level
- Typical caches:
  - **Write-back + Write allocate, usually**
  - Write-through + No-write allocate, occasionally

Write-back, Write Allocate Example

Write-back: defer write to next level until line is evicted
Write-allocate: on a miss, bring the data into cache



There is only one set in this tiny cache, so the tag is the entire block number! (Because $s = 0$)

Note: We are making some unrealistic simplifications to keep this example simple and focus on the cache policies!

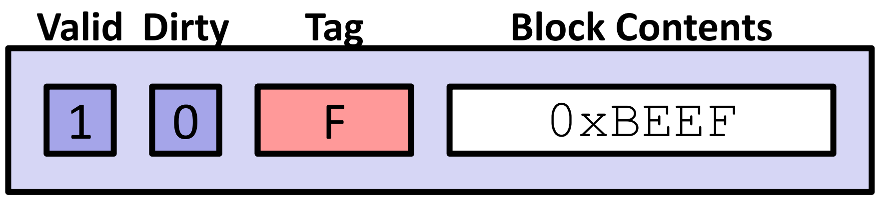
Write-back, Write Allocate Example

Write-back: defer write to next level until line is evicted
Write-allocate: on a miss, bring the data into cache

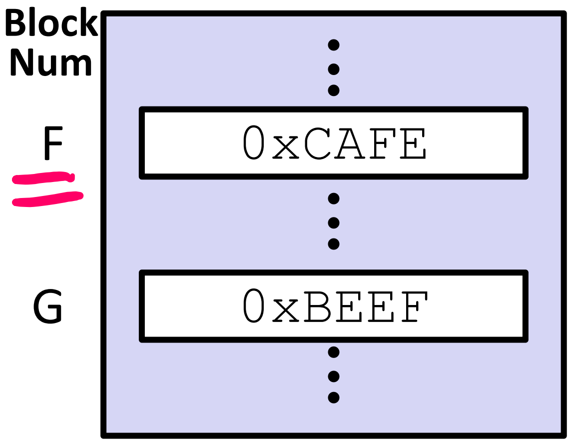
1) `mov $0xFACE, (F)` ← Not valid x86, assume we mean an address associated with this block num

Write Miss

Cache:



Memory:



Step 1: Bring **F** into cache

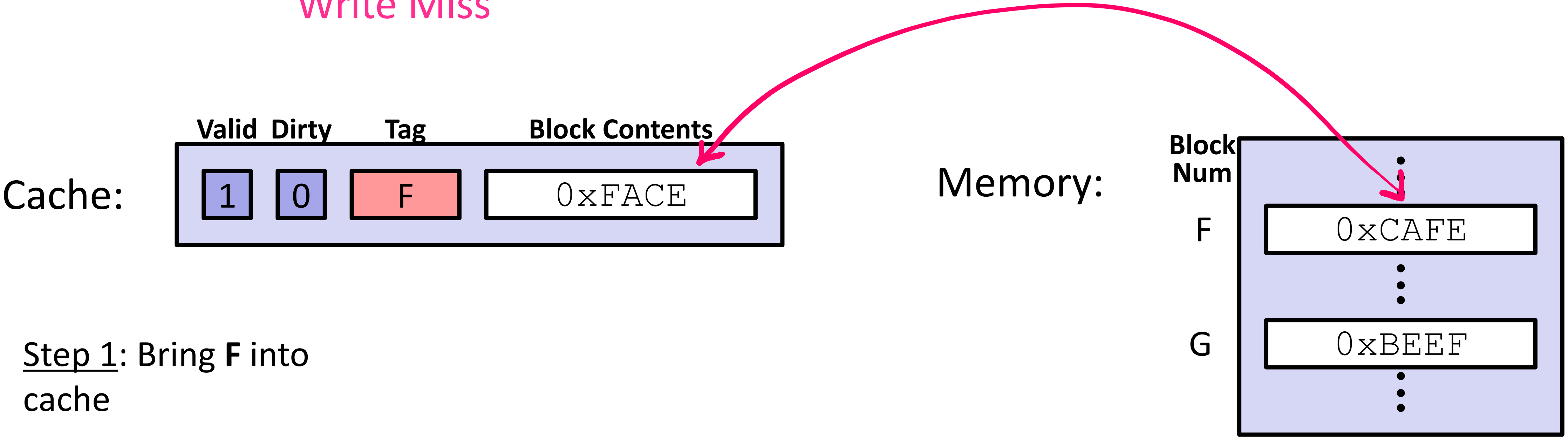
Write-back, Write Allocate Example

Write-back: defer write to next level until line is evicted
Write-allocate: on a miss, bring the data into cache

1) `mov $0xFACE, (F)`

Write Miss

Difference!



Step 1: Bring **F** into cache

Step 2: Write `0xFACE` to cache only and set the dirty bit. Why? Look at the values!

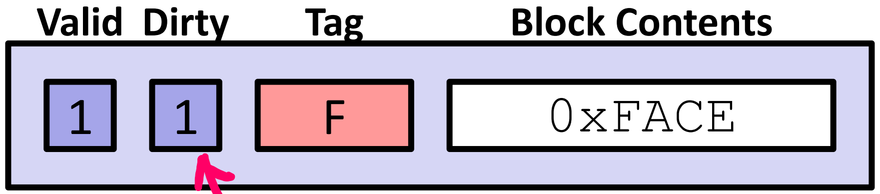
Write-back, Write Allocate Example

Write-back: defer write to next level until line is evicted
Write-allocate: on a miss, bring the data into cache

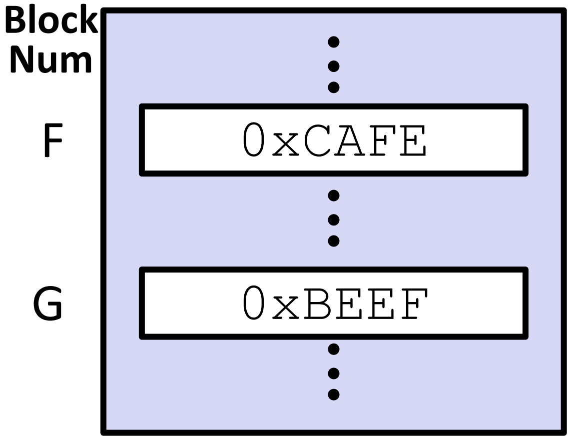
1) `mov $0xFACE, (F)`

Write Miss

Cache:



Memory:



This is set to 1 because the contents differ.

Step 1: Bring **F** into cache

Step 2: Write `0xFACE` to cache only and set the dirty bit. Why? Look at the values!

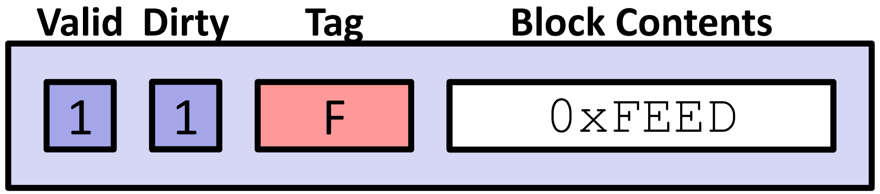
Write-back, Write Allocate Example

Write-back: defer write to next level until line is evicted
Write-allocate: on a miss, bring the data into cache

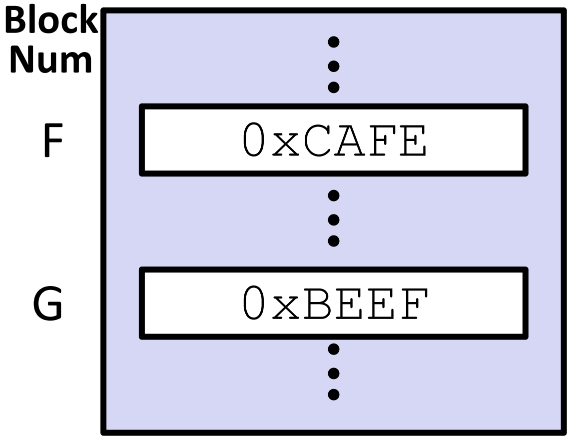
1) `mov $0xFACE, (F)` 2) `mov $0xFEED, (F)`

Write Miss Write Hit

Cache:



Memory:



Step: Write `0xFEED` to cache only (and set the dirty bit)

Redondant? Yes, but just do it. It's protocol!

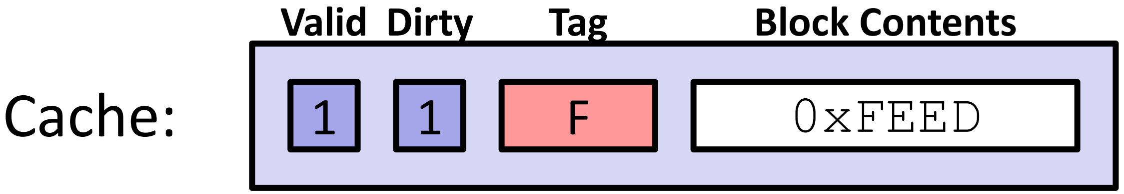
Write-back: defer write to next level until line is evicted
Write-allocate: on a miss, bring the data into cache

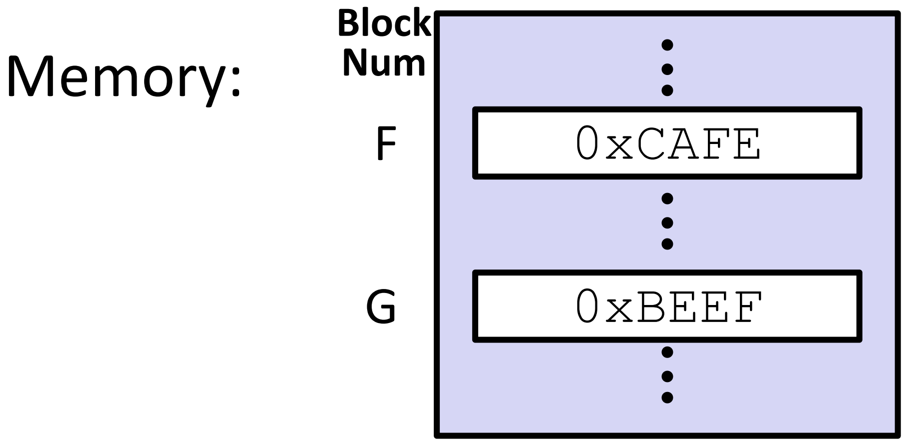
Write-back, Write Allocate Example

1) `mov $0xFACE, (F)` Write Miss

2) `mov $0xFEED, (F)` Write Hit

3) `mov (G), %ax` Read Miss





<u>Step 1</u>: Write **F** back to memory since it is dirty

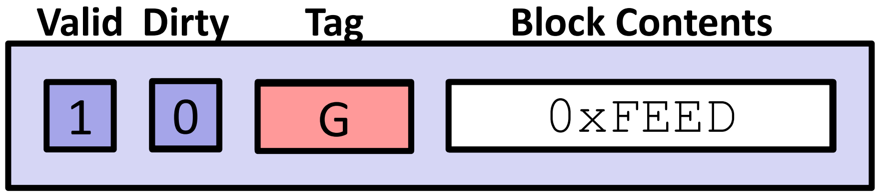
Write-back: defer write to next level until line is evicted
Write-allocate: on a miss, bring the data into cache

Write-back, Write Allocate Example

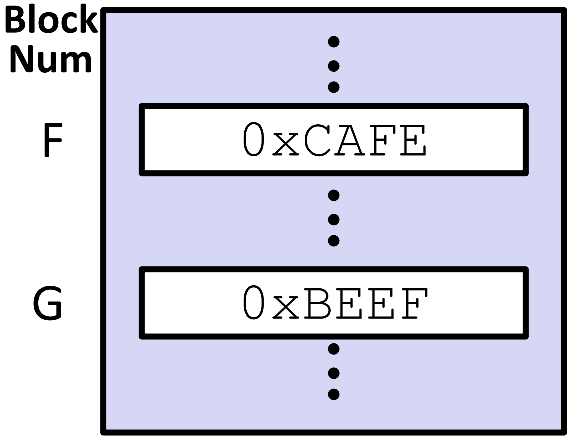
1) `mov $0xFACE, (F)` 2) `mov $0xFEED, (F)` 3) `mov (G), %ax`

Write Miss — Write Hit — Read Miss

Cache:



Memory:



Step 1: Write **F** back to memory since it is dirty

Step 2: Bring **G** into the cache so that we can copy it into `%ax`

Cache Simulator

- Want to play around with cache parameters and policies? Check out our cache simulator!
  - https://courses.cs.washington.edu/courses/cse351/cachesim/
- Way to use:
  - Take advantage of "explain mode" and navigable history to test your own hypotheses and answer your own questions
  - Self-guided Cache Sim Demo posted along with Section 7
  - Will be used in HW19 – Lab 4 Preparation

Polling Question

- Which of the following cache statements is FALSE?

A. A write-through cache will always match data with the memory hierarchy level below it

B. We can reduce compulsory misses by ~~decreasing~~ our block size

to make it true: increasing, not decreasing

C. A write-back cache will save time for code with good temporal locality on writes

D. We can reduce conflict misses by increasing associativity

E. We're lost…

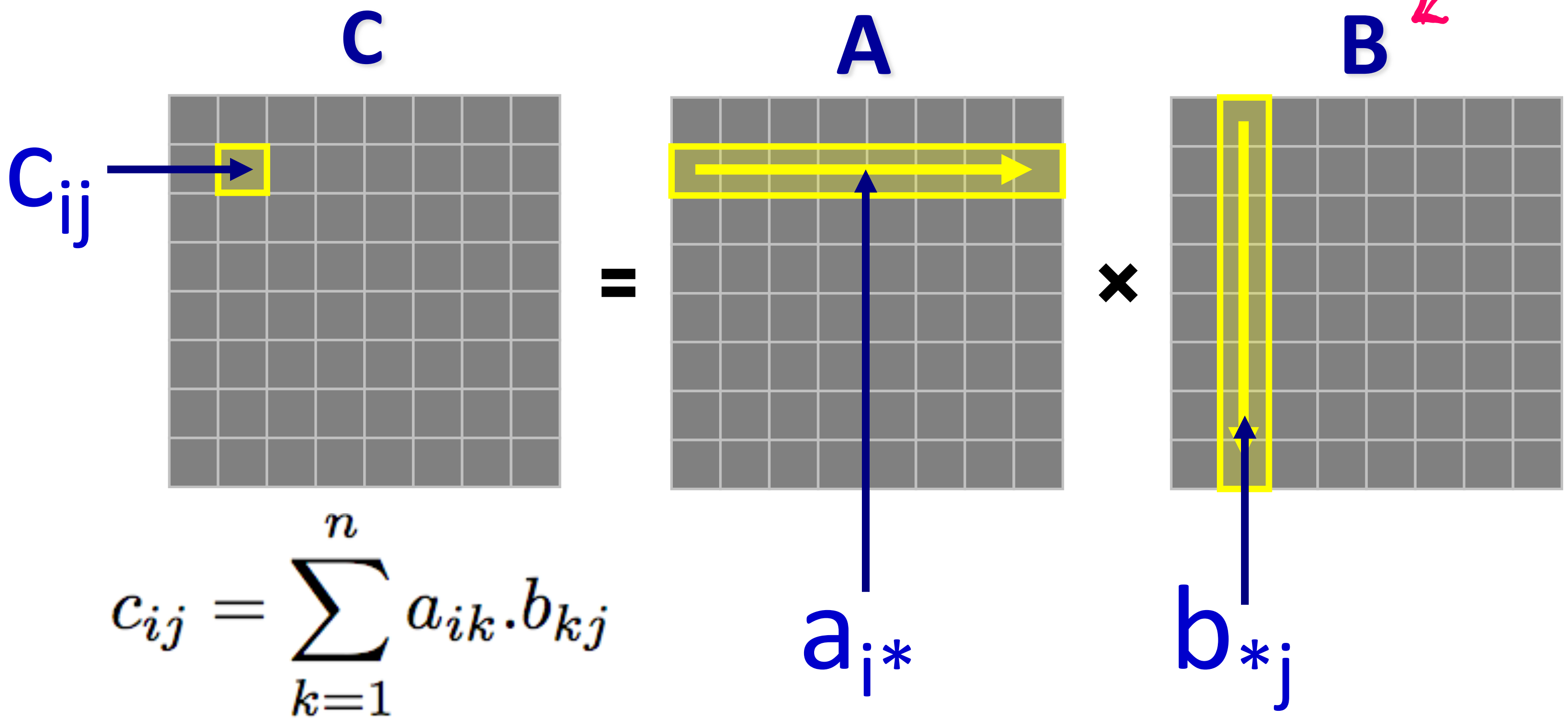
Optimizations for the Memory Hierarchy

- 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 Multiplication (Why?)

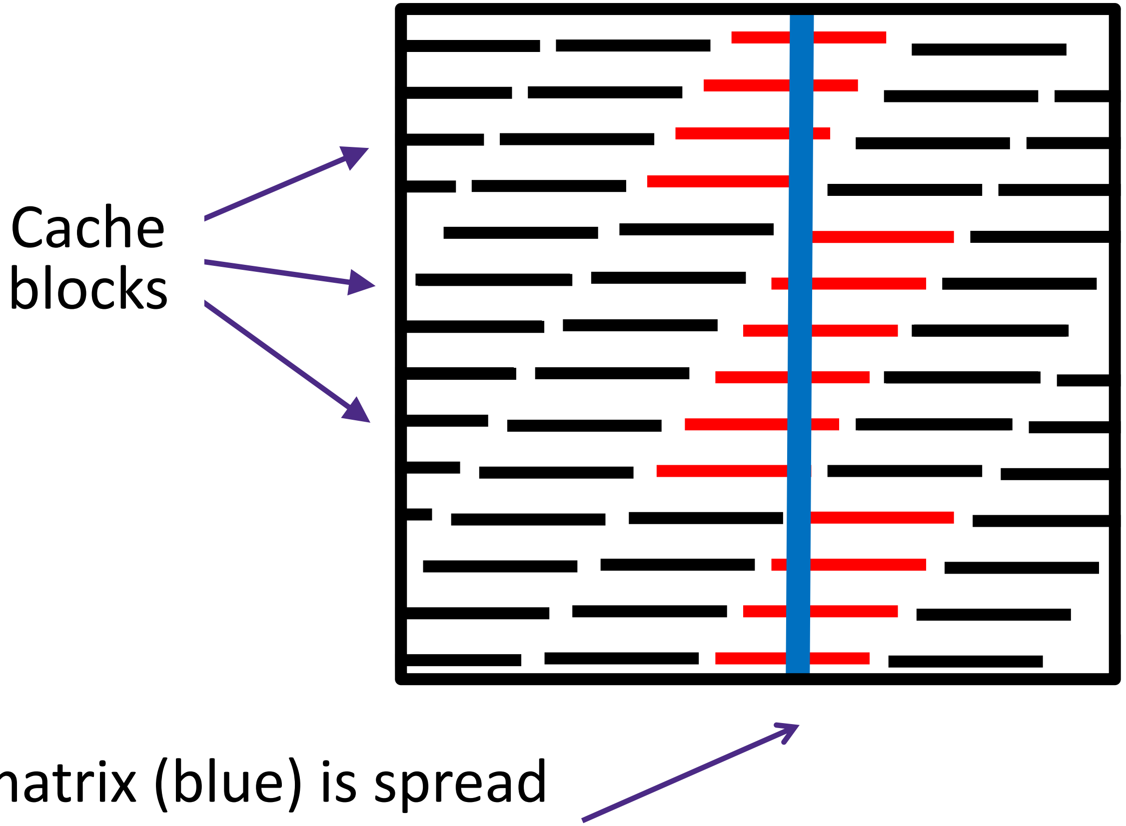
Sigh, our column-major problem child.



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

Matrices in Memory

- How do cache blocks fit into this scheme?
  - Row major matrix in memory:



column of matrix (blue) is spread among cache blocks shown in red

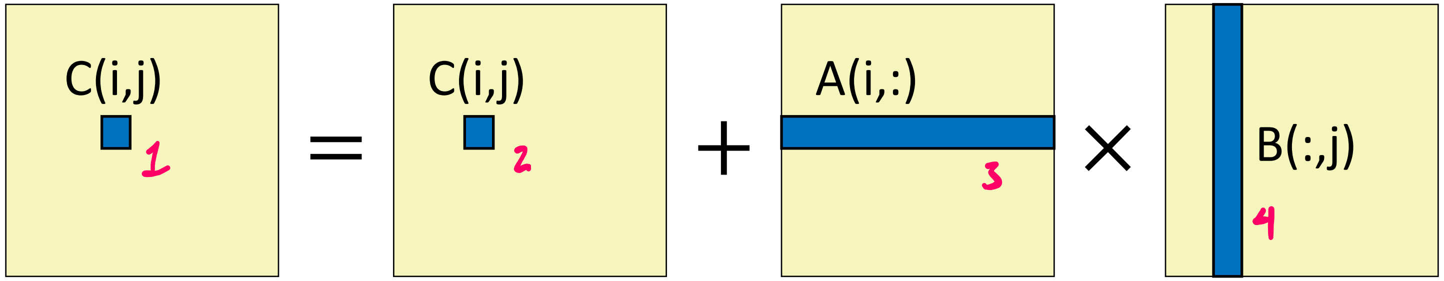
Aaand, this is why..

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*n+j] += a[i*n+k] * b[k*n+j];
```

Something to think about: How many memory accesses in this line?

4



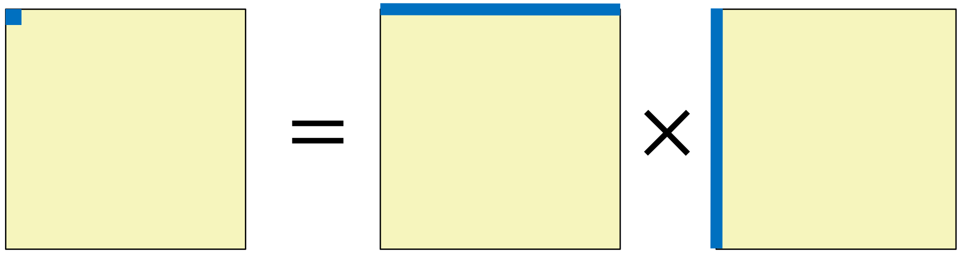
Cache Miss Analysis (Naïve)

Ignoring matrix `c`

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

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

matrix: A B

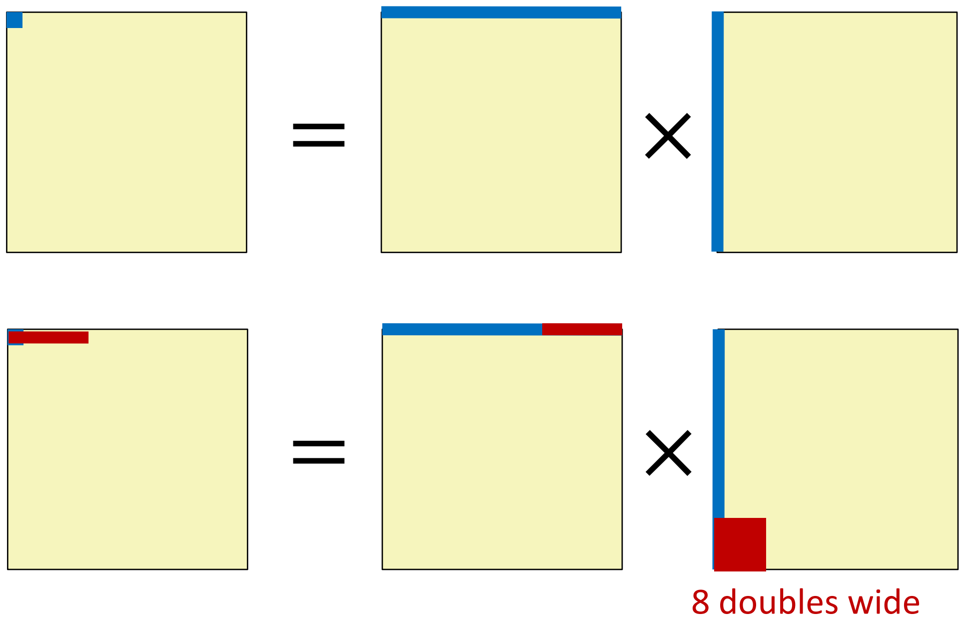


Cache Miss Analysis (Naïve)

Ignoring matrix `c`

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

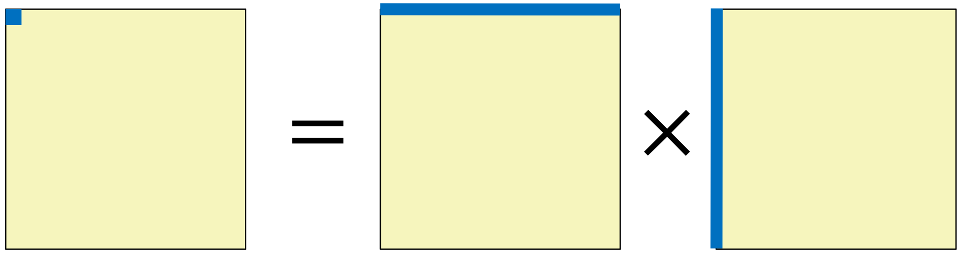
- Each iteration:
  - $\frac{n}{8} + n = \frac{9n}{8}$ misses
  - Afterwards in cache: (schematic)



Cache Miss Analysis (Naïve)

- Scenario Parameters:
  - Square matrix ($n \times n$), elements are `doubles`
  - Cache block size K = 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: $\frac{9n}{8} \times n^2 = \frac{9}{8} n^3$

once per element in the $n \times n$ product matrix

Linear Algebra to the Rescue (1)

Ignoring matrix `c`

- 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:

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

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

Linear Algebra to the Rescue (2)

Ignoring matrix C

C_{11}	C_{12}	C_{13}	C_{14}
C_{21}	C_{22}	C_{23}	C_{24}
C_{31}	C_{32}	C_{43}	C_{34}
C_{41}	C_{42}	C_{43}	C_{44}

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_{144}

B_{11}	B_{12}	B_{13}	B_{14}
B_{21}	B_{22}	B_{23}	B_{24}
B_{32}	B_{32}	B_{33}	B_{34}
B_{41}	B_{42}	B_{43}	B_{44}

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 "*cache blocking*"

Blocked Matrix Multiply

- Blocked version of the naïve algorithm (wtf???):

```
# 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*n+jb] += a[ib*n+kb]*b[kb*n+jb];
```

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

Cache Miss Analysis (Blocked)

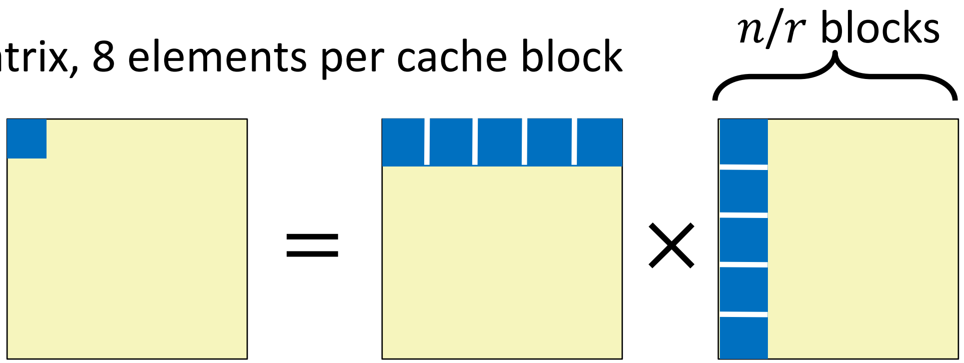
Ignoring matrix c

- Scenario Parameters:
  - Cache block size K = 64 B = 8 doubles
  - Cache size $C \ll n$ (much smaller than n)
  - Three blocks ■ ($r \times r$) fit into cache: $3r^2 < C$

r^2 elements per sub-matrix, 8 elements per cache block

- Each block iteration:
  - $r^2/8$ misses per block
  - $\frac{2n}{r} \times \frac{r^2}{8} = \frac{nr}{4}$

n/r blocks in row and in column



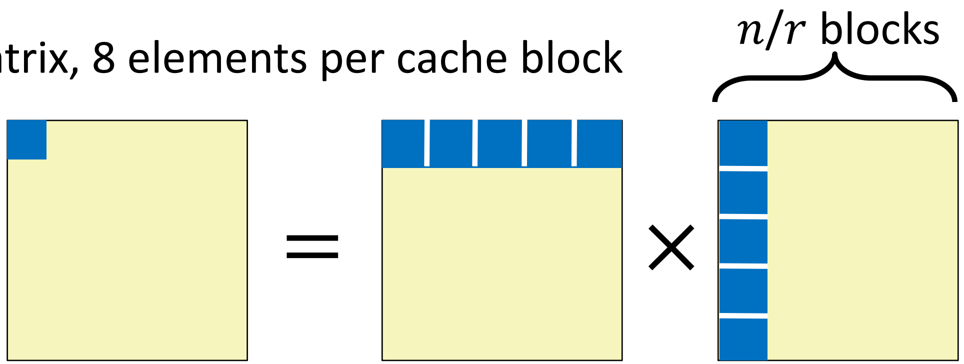
Cache Miss Analysis (Blocked)

Ignoring matrix `c`

- Scenario Parameters:
  - Cache block size K = 64 B = 8 doubles
  - Cache size $C \ll n$ (much smaller than n)
  - Three blocks ■ ($r \times r$) fit into cache: $3r^2 < C$

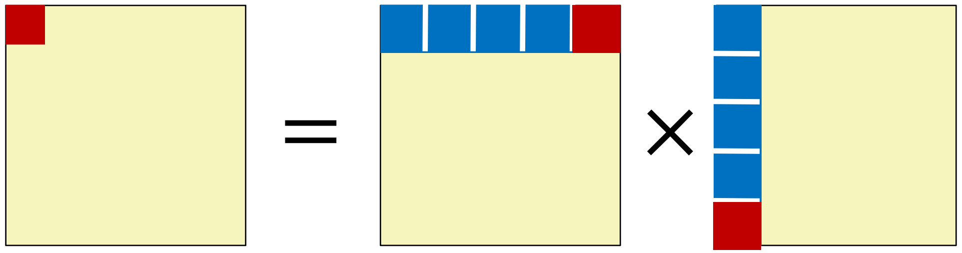
r^2 elements per sub-matrix, 8 elements per cache block

- Each block iteration:
  - $r^2/8$ misses per block
  - $2n/r \times r^2/8 = nr/4$



n/r blocks in row and in column

  - Afterwards in cache (schematic)



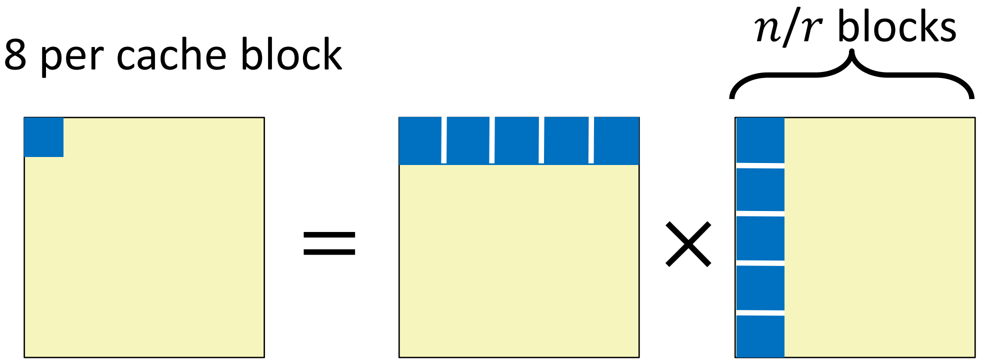
Cache Miss Analysis (Blocked)

Ignoring matrix c

- Scenario Parameters:
  - Cache block size K = 64 B = 8 doubles
  - Cache size $C \ll n$ (much smaller than n)
  - Three blocks ■ $(r \times r)$ fit into cache: $3r^2 < C$

r^2 elements per block, 8 per cache block

- Each block iteration:
  - $r^2/8$ misses per block
  - $2n/r \times r^2/8 = nr/4$



n/r blocks in row and column

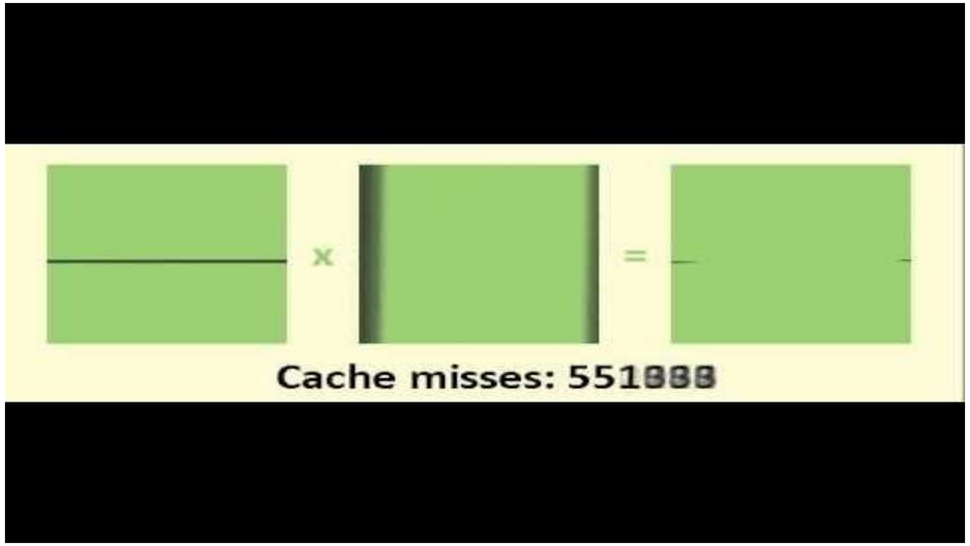
- Total misses:
  - $\frac{nr}{4} \times \left(\frac{n}{r}\right)^2 = \frac{n^3}{4r}$

number of blocks in product matrix

Compare this to $\frac{9}{8}n^3$!!!

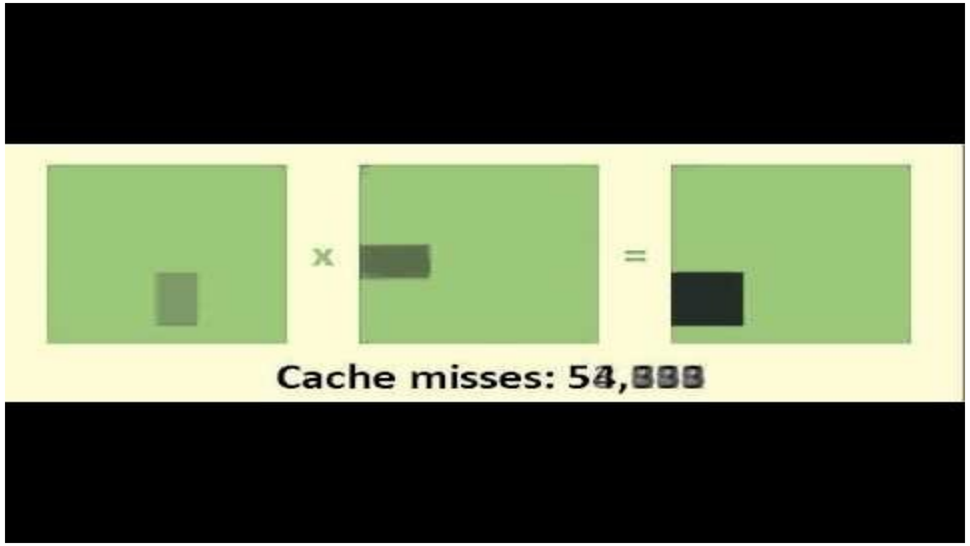
Matrix Multiply Visualization

Naïve:



≈ 1,020,000
cache misses

Blocked:



≈ 90,000
cache misses

Here n = 100, C = 32 KiB, r = 30

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 coding rules
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)
    - Focus on inner loop code