15-213 Recitation 6: Caches & Blocking

Your TAs

Monday, February 26th, 2024

Agenda

- Logistics
- Code Reviews
- Cache Lab
- Blocking
- Intro to Git

Logistics

- Cache Lab is due **Thursday, Feb. 29th** at **11:59pm**
- Written Midterm due **Wednesday, Feb. 28th**!
- Drop deadline **today (Monday, Feb. 26th)**
- Make sure you have Github working so you can commit your code!

Cache Lab: Cache Simulator Hints

- Goal: Count hits, misses, evictions and # of dirty bytes
- Procedure
  - Least Recently Used (LRU) replacement policy
  - Structs are good for storing cache line parts (valid bit, tag, LRU counter, etc.)
  - A cache is like a 2D array of cache lines

```
struct cache_line cache[S][E];
```

- Your simulator needs to handle different values of S, E, and b (block size) given at run time
  - Dynamically allocate memory!
- Dirty bytes: any payload byte whose corresponding cache block's dirty bit is set (i.e. the payload of that block has been modified, but not yet written back to main memory)

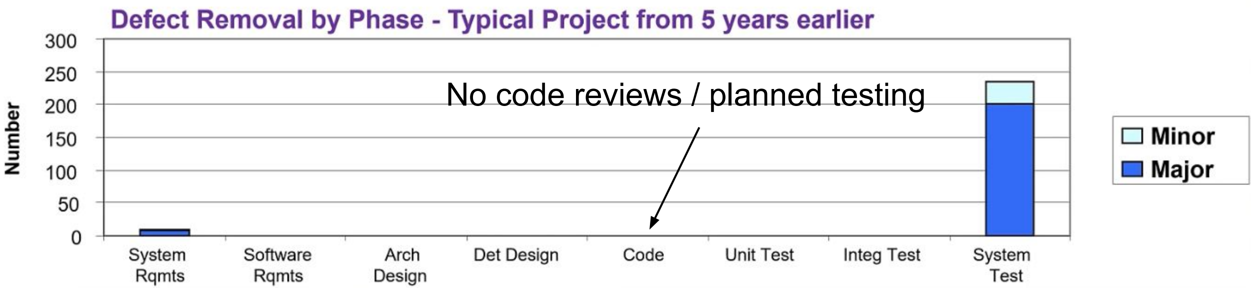
Code Reviews

Code Reviews

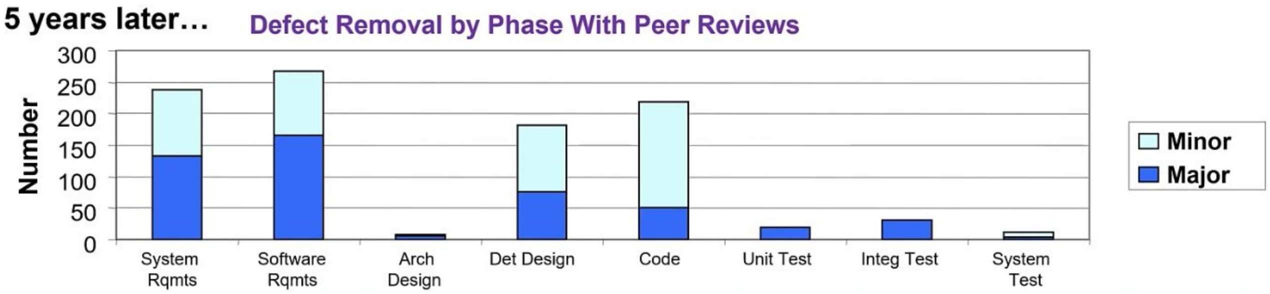
- Why code reviews?
  - Used in industry - Nearly all companies utilize code reviews
  - Systematic code reviews are highly effective at finding bugs efficiently and effectively.

Code Reviews

- Industry example from an embedded system machine critical pipeline flow device requiring high software quality



The same team implemented testing and code reviews. This is a similar project done 5 years later.



Example from CMU 18-642 https://course.ece.cmu.edu/~ece642/ , Sourced from Roger G. 2005

Code Review Signup

- All students in the course will receive an email with a link to signup for a code review timeslot.
- All students will receive a final style score from 0-4 points
- 213 code reviews will be short (<= 15 minutes) and cover code style and code quality.

2	Zoom Link				
3					
4	Time Slots	Location	TA	Andrew ID	Status
5	EX: 10/10 1:00 PM - 1:15 PM	Zoom	Sachit	jwli2	DONE
6	EX: 10/10 1:15 PM - 1:30 PM	Zoom	Sachit	jwli3	DONE
7	EX: 10/10 1:30 PM - 1:45 PM	Zoom	Sachit	jwli4	DONE
8	EX: 10/10 1:45 PM - 2:00 PM	Zoom	Sachit	jwli5	
9	EX: 10/10 2:00 PM - 2:15 PM	Zoom	Sachit		
10	EX: 10/10 2:15 PM - 2:30 PM	Zoom	Sachit		
11					
12	EX: 10/11 1:00 PM - 1:15 PM	Recitation Room	Shravya		
13	EX: 10/11 1:15 PM - 1:30 PM	Recitation Room	Shravya		
14	EX: 10/11 1:30 PM - 1:45 PM	Recitation Room	Shravya		
15	EX: 10/11 1:45 PM - 2:00 PM	Recitation Room	Shravya		
16					
17	EX: 10/10 1:00 PM - 1:15 PM	Zoom	Sachit		
18	EX: 10/10 1:15 PM - 1:30 PM	Zoom	Shravya		
19	EX: 10/10 1:30 PM - 1:45 PM	Zoom	Shravya		
20	EX: 10/10 1:45 PM - 2:00 PM	Zoom	Shravya		
21	EX: 10/10 2:00 PM - 2:15 PM	Zoom	Shravya		
22	EX: 10/10 2:15 PM - 2:30 PM	Zoom	Shravya		
23					
24	Conflicts (Andrew ID):				
25					

Code Style

- Properly document your code
  - Function + File header comments, overall operation of large blocks, any tricky bits
- Write robust code – check error and failure conditions
- Write modular code
  - Use interfaces for data structures, e.g. create/insert/remove/free functions for a linked list
  - No magic numbers – use `#define` or `static const`
- Formatting
  - 80 characters per line (use Autolab's highlight feature to double-check)
  - Consistent braces and whitespace
- No memory or file descriptor leaks

Blocking

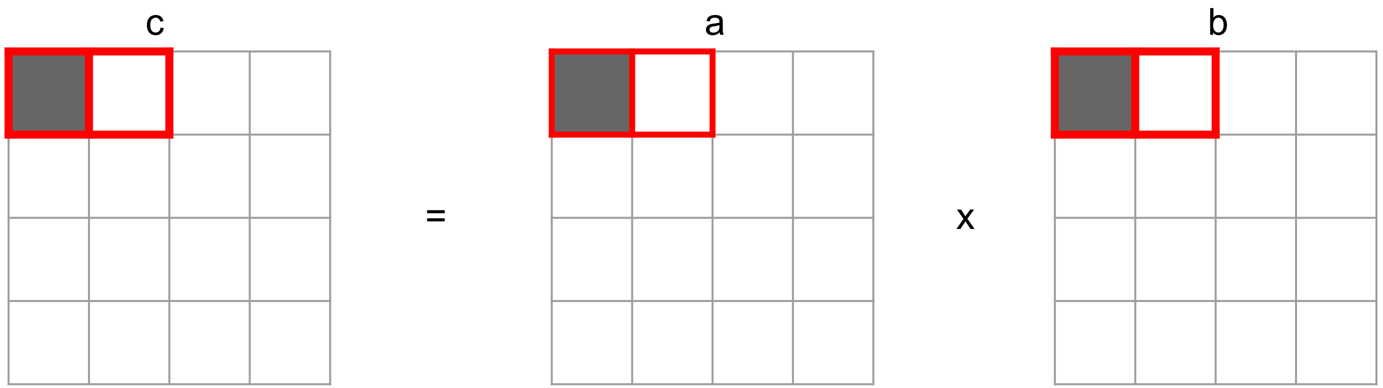
Example: Matrix Multiplication

```
/* multiply 4x4 matrices */
void mm(int a[4][4], int b[4][4], int c[4][4]) {
    int i, j, k;
    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            for (k = 0; k < 4; k++)
                c[i][j] += a[i][k] * b[k][j];
```

Let’s step through this to see what’s actually happening

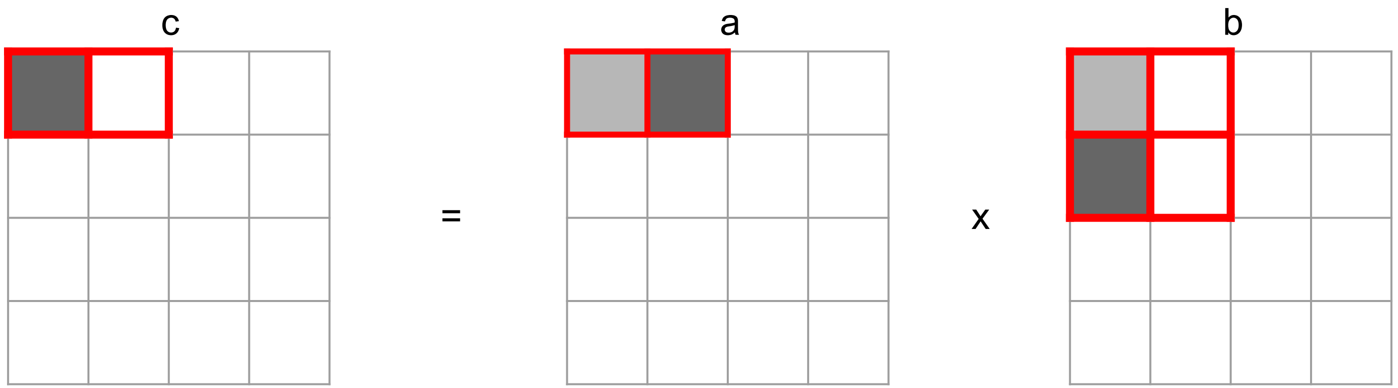
Example: Matrix Multiplication

- Assume a tiny cache with 4 lines of 8 bytes (2 ints)
  - S = 1, E = 4, B = 8
- Let's see what happens if we don't use blocking



iter	i	j	k	operation	miss?
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	(m, m)

Key:
Grey = accessed
Dark grey = currently accessing
Red border = in cache



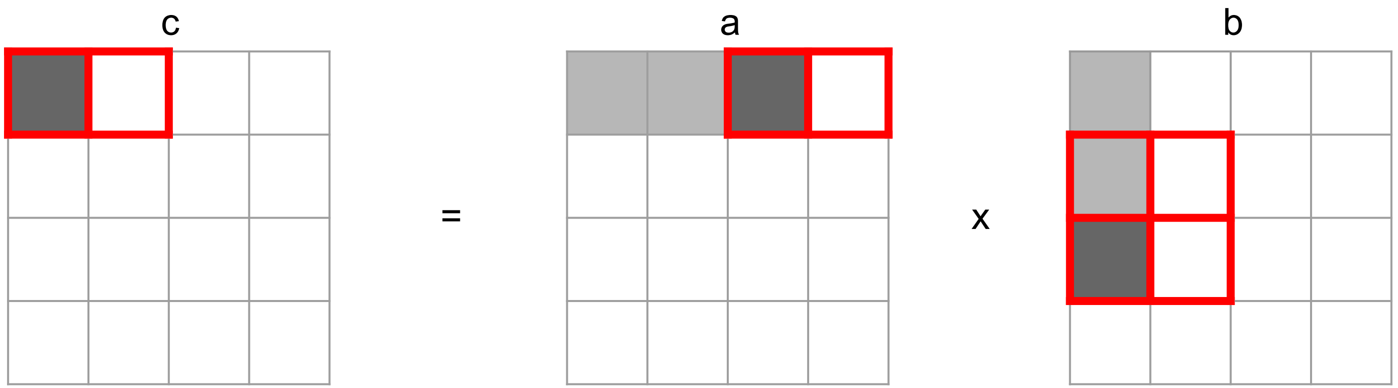
iter	i	j	k	operation	miss?
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	(m, m)
1	0	0	1	c[0][0] += a[0][1] * b[1][0]	(h, m)

Key:

Grey = accessed

Dark grey = currently accessing

Red border = in cache



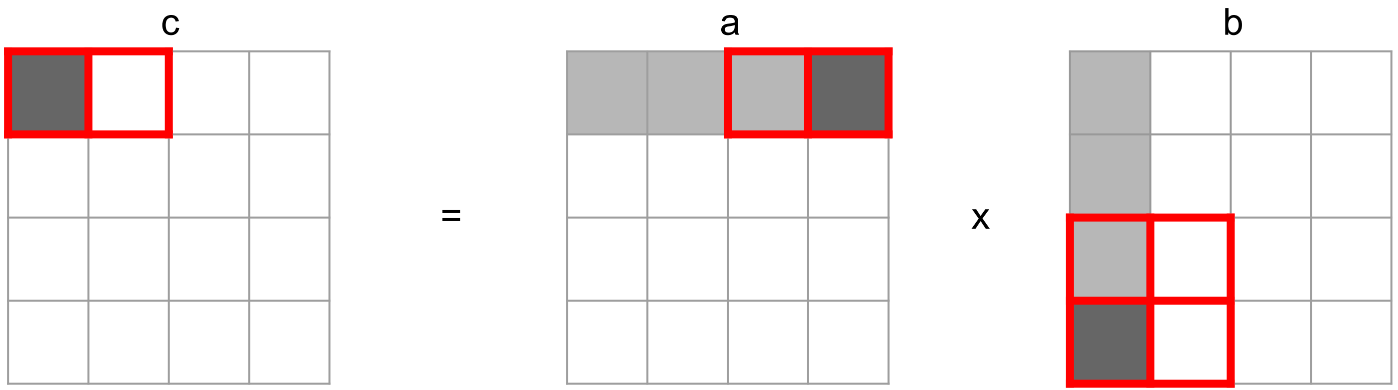
iter	i	j	k	operation	miss?
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	(m, m)
1	0	0	1	c[0][0] += a[0][1] * b[1][0]	(h, m)
2	0	0	2	c[0][0] += a[0][2] * b[2][0]	(m, m)

Key:

Grey = accessed

Dark grey = currently accessing

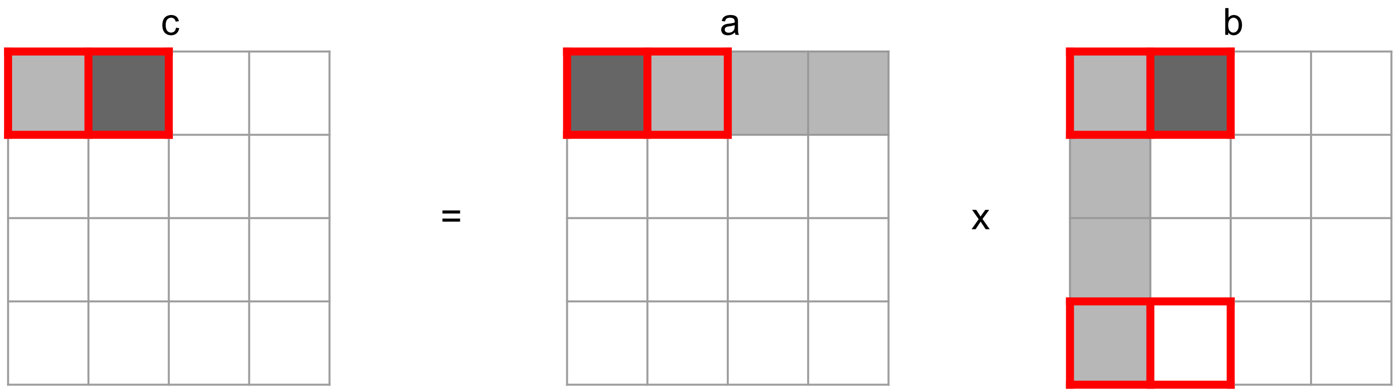
Red border = in cache



iter	i	j	k	operation	miss?
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	(m, m)
1	0	0	1	c[0][0] += a[0][1] * b[1][0]	(h, m)
2	0	0	2	c[0][0] += a[0][2] * b[2][0]	(m, m)
3	0	0	3	c[0][0] += a[0][3] * b[3][0]	(h, m)

<u>Key:</u>

Grey = accessed
Dark grey = currently accessing
Red border = in cache



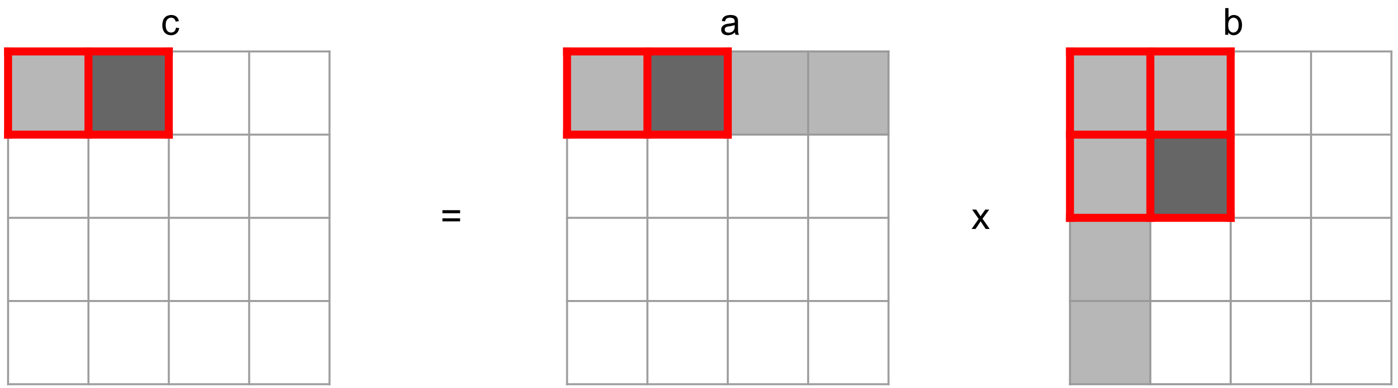
iter	i	j	k	operation	miss?
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	(m, m)
1	0	0	1	c[0][0] += a[0][1] * b[1][0]	(h, m)
2	0	0	2	c[0][0] += a[0][2] * b[2][0]	(m, m)
3	0	0	3	c[0][0] += a[0][3] * b[3][0]	(h, m)
4	0	1	0	c[0][1] += a[0][0] * b[0][1]	(m, m)

Key:

Grey = accessed

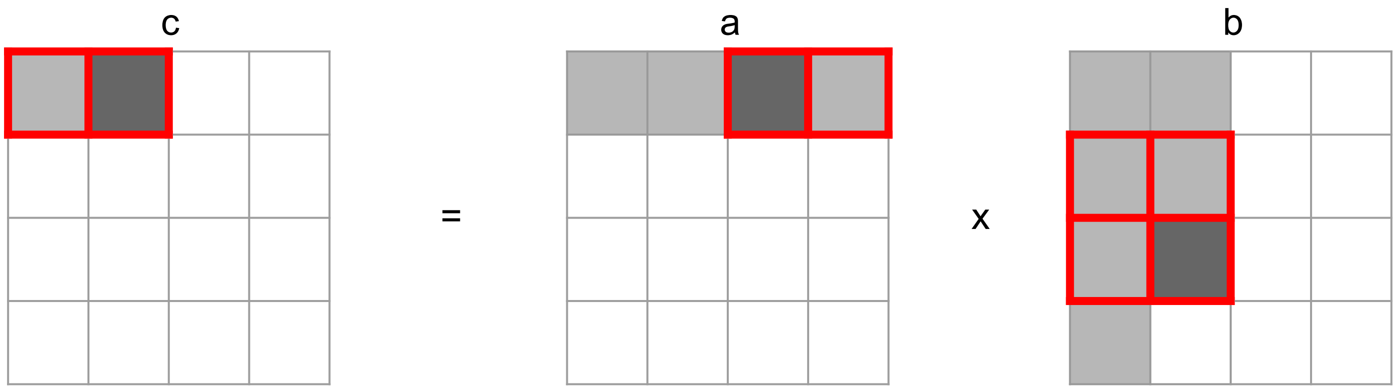
Dark grey = currently accessing

Red border = in cache



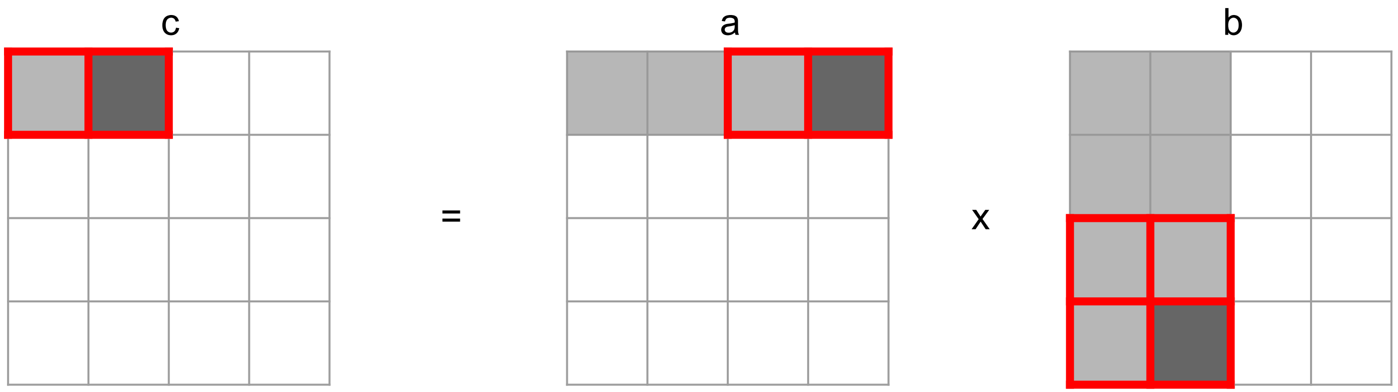
iter	i	j	k	operation	miss?
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	(m, m)
1	0	0	1	c[0][0] += a[0][1] * b[1][0]	(h, m)
2	0	0	2	c[0][0] += a[0][2] * b[2][0]	(m, m)
3	0	0	3	c[0][0] += a[0][3] * b[3][0]	(h, m)
4	0	1	0	c[0][1] += a[0][0] * b[0][1]	(m, m)
5	0	1	1	c[0][1] += a[0][1] * b[1][1]	(h, m)

Key:
Grey = accessed
Dark grey = currently accessing
Red border = in cache



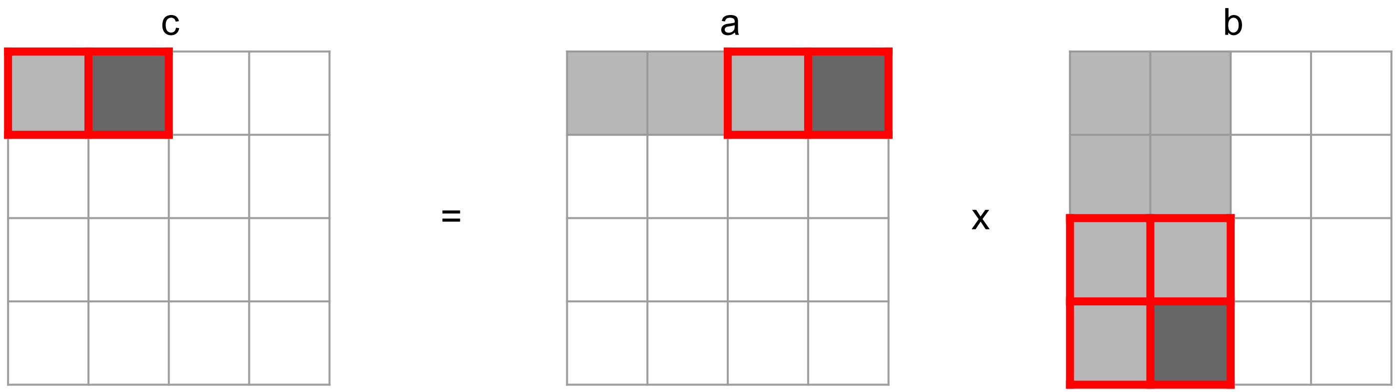
iter	i	j	k	operation	miss?
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	(m, m)
1	0	0	1	c[0][0] += a[0][1] * b[1][0]	(h, m)
2	0	0	2	c[0][0] += a[0][2] * b[2][0]	(m, m)
3	0	0	3	c[0][0] += a[0][3] * b[3][0]	(h, m)
4	0	1	0	c[0][1] += a[0][0] * b[0][1]	(m, m)
5	0	1	1	c[0][1] += a[0][1] * b[1][1]	(h, m)
6	0	1	2	c[0][1] += a[0][2] * b[2][1]	(m, m)

Key:
Grey = accessed
Dark grey = currently accessing
Red border = in cache



iter	i	j	k	operation	miss?
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	(m, m)
1	0	0	1	c[0][0] += a[0][1] * b[1][0]	(h, m)
2	0	0	2	c[0][0] += a[0][2] * b[2][0]	(m, m)
3	0	0	3	c[0][0] += a[0][3] * b[3][0]	(h, m)
4	0	1	0	c[0][1] += a[0][0] * b[0][1]	(m, m)
5	0	1	1	c[0][1] += a[0][1] * b[1][1]	(h, m)
6	0	1	2	c[0][1] += a[0][2] * b[2][1]	(m, m)
7	0	1	3	c[0][1] += a[0][3] * b[3][1]	(h, m)

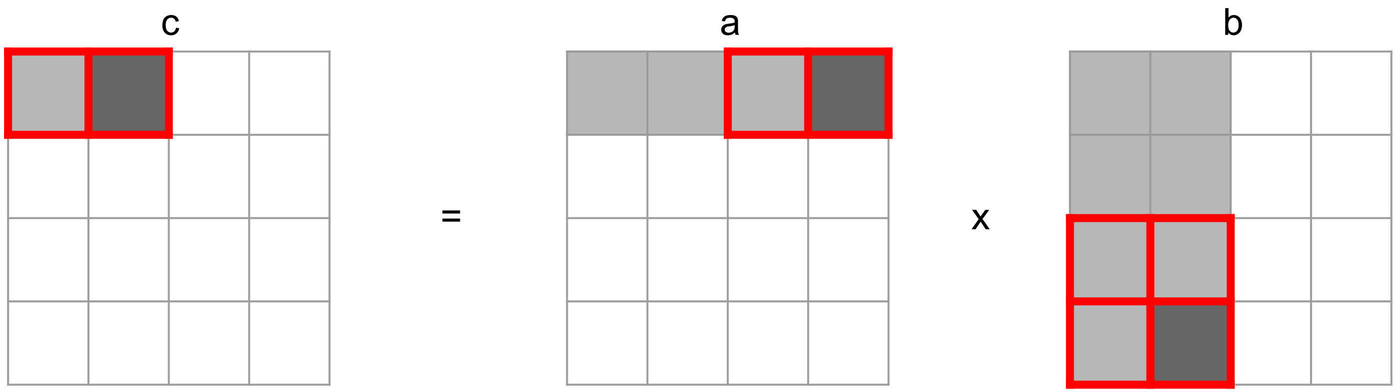
Key:
Grey = accessed
Dark grey = currently accessing
Red border = in cache



iter	i	j	k	operation	miss?
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	(m, m)
1	0	0	1	c[0][0] += a[0][1] * b[1][0]	(h, m)
2	0	0	2	c[0][0] += a[0][2] * b[2][0]	(m, m)
3	0	0	3	c[0][0] += a[0][3] * b[3][0]	(h, m)
4	0	1	0	c[0][1] += a[0][0] * b[0][1]	(m, m)
5	0	1	1	c[0][1] += a[0][1] * b[1][1]	(h, m)
6	0	1	2	c[0][1] += a[0][2] * b[2][1]	(m, m)
7	0	1	3	c[0][1] += a[0][3] * b[3][1]	(h, m)

Key:
Grey = accessed
Dark grey = currently accessing
Red border = in cache

What is the miss rate of a?



iter	i	j	k	operation	miss?
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	(m, m)
1	0	0	1	c[0][0] += a[0][1] * b[1][0]	(h, m)
2	0	0	2	c[0][0] += a[0][2] * b[2][0]	(m, m)
3	0	0	3	c[0][0] += a[0][3] * b[3][0]	(h, m)
4	0	1	0	c[0][1] += a[0][0] * b[0][1]	(m, m)
5	0	1	1	c[0][1] += a[0][1] * b[1][1]	(h, m)
6	0	1	2	c[0][1] += a[0][2] * b[2][1]	(m, m)
7	0	1	3	c[0][1] += a[0][3] * b[3][1]	(h, m)

Key:

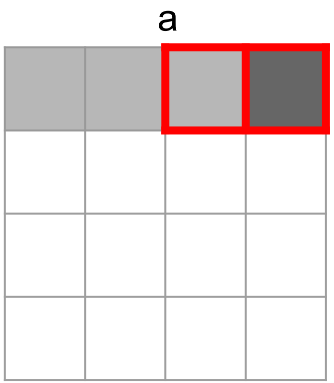
Grey = accessed
Dark grey = currently accessing
Red border = in cache

What is the miss rate of a?

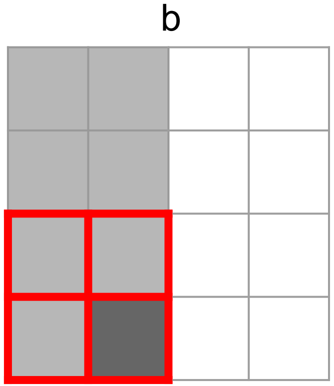
What is the miss rate of b?

What went wrong?

- Bad temporal locality!
- Blocks are used multiple times, but are never in cache when we need them.



Miss Rate: 50%



Miss Rate: 100%

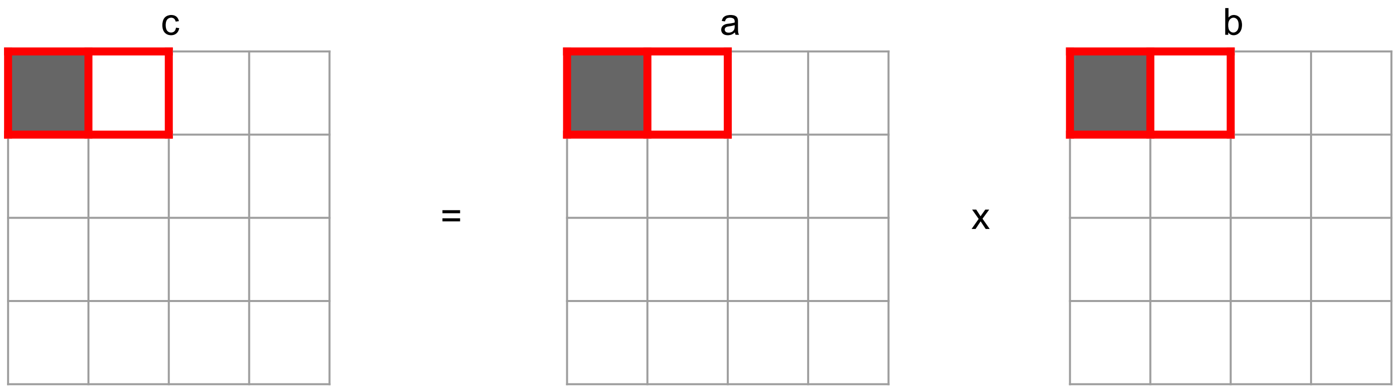
Example: Matrix Multiplication (blocking)

```
/* multiply 4x4 matrices using blocks of size 2 */
void mm_blocking(int a[4][4], int b[4][4], int c[4][4]) {
    int i, j, k;
    int i_c, j_c, k_c;
    int B = 2;
    // control loops
    for (i_c = 0; i_c < 4; i_c += B)
        for (j_c = 0; j_c < 4; j_c += B)
            for (k_c = 0; k_c < 4; k_c += B)
                // block multiplications
                for (i = i_c; i < i_c + B; i++)
                    for (j = j_c; j < j_c + B; j++)
                        for (k = k_c; k < k_c + B; k++)
                            c[i][j] += a[i][k] * b[k][j];
```

Let's step through this to see what's actually happening

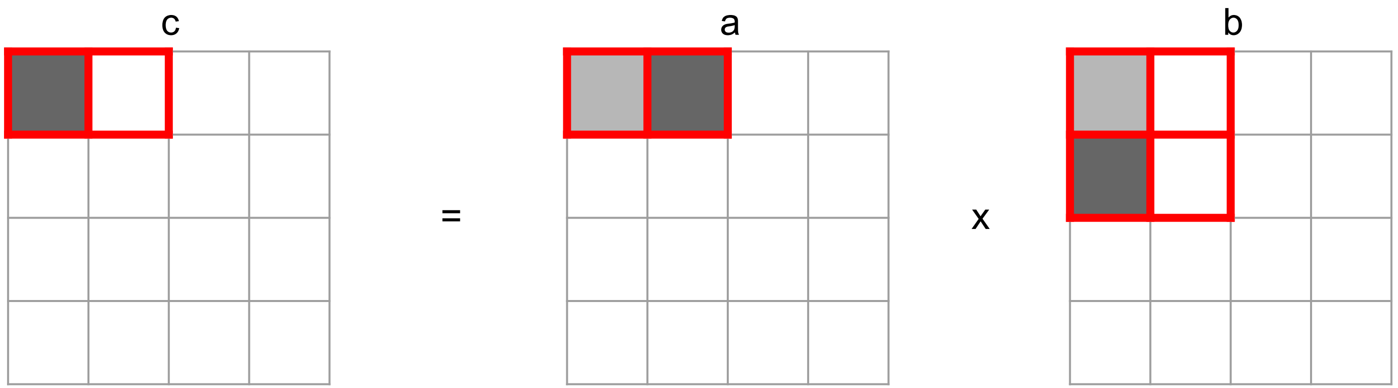
Example: Matrix Multiplication (blocking)

- Assume a tiny cache with 4 lines of 8 bytes (2 ints)
  - S = 1, E = 4, B = 8
- Let's see what happens if we now use blocking



iter	i	j	k	operation	miss?
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	(m, m)

Key:
Grey = accessed
Dark grey = currently accessing
Red border = in cache



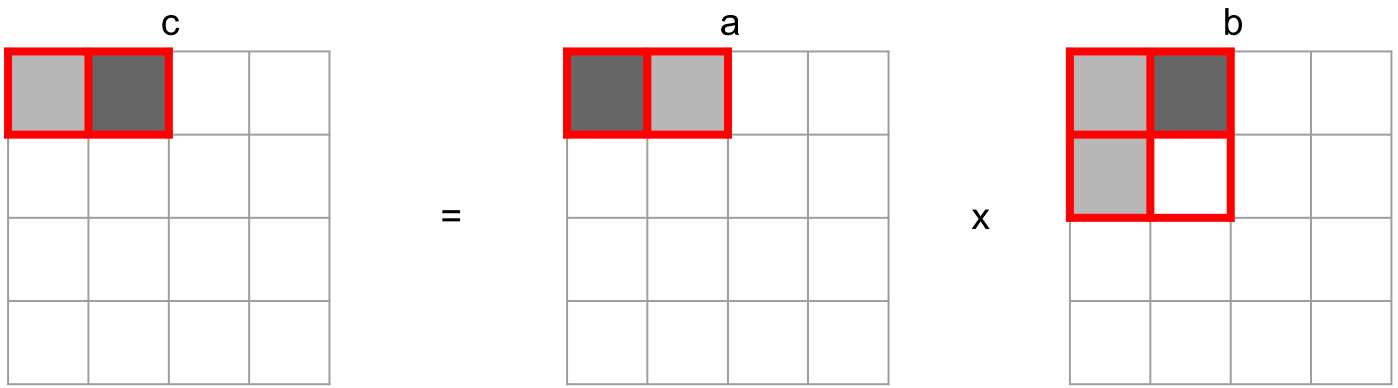
iter	i	j	k	operation	miss?
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	(m, m)
1	0	0	1	c[0][0] += a[0][1] * b[1][0]	(h, m)

Key:

Grey = accessed

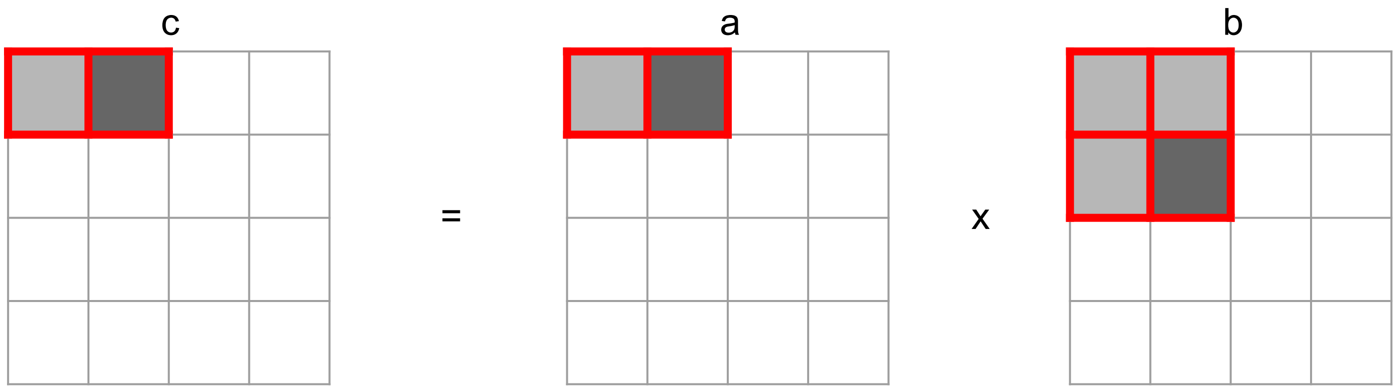
Dark grey = currently accessing

Red border = in cache



iter	i	j	k	operation	miss?
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	(m, m)
1	0	0	1	c[0][0] += a[0][1] * b[1][0]	(h, m)
2	0	1	0	c[0][1] += a[0][0] * b[0][1]	(h, h)

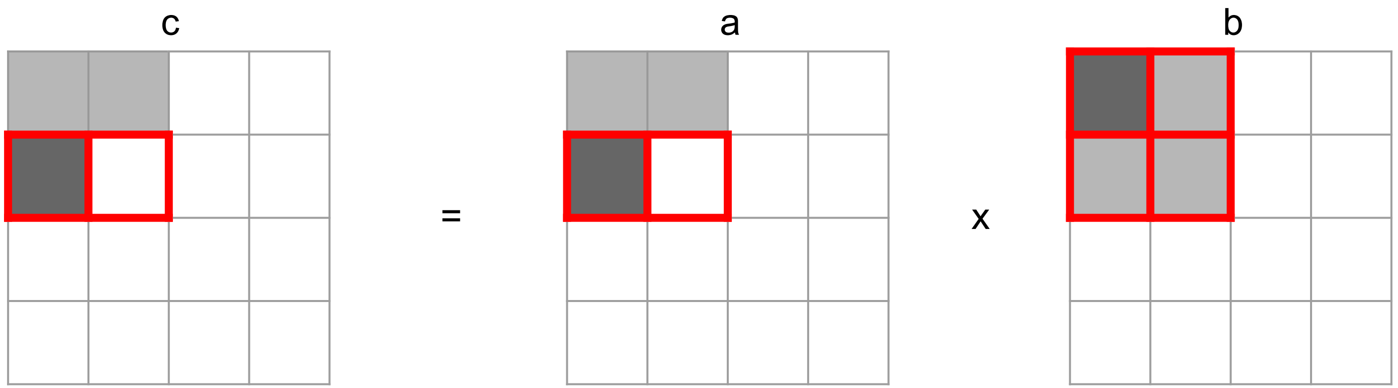
Key:
Grey = accessed
Dark grey = currently accessing
Red border = in cache



iter	i	j	k	operation	miss?
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	(m, m)
1	0	0	1	c[0][0] += a[0][1] * b[1][0]	(h, m)
2	0	1	0	c[0][1] += a[0][0] * b[0][1]	(h, h)
3	0	1	1	c[0][1] += a[0][1] * b[1][1]	(h, h)

Key:

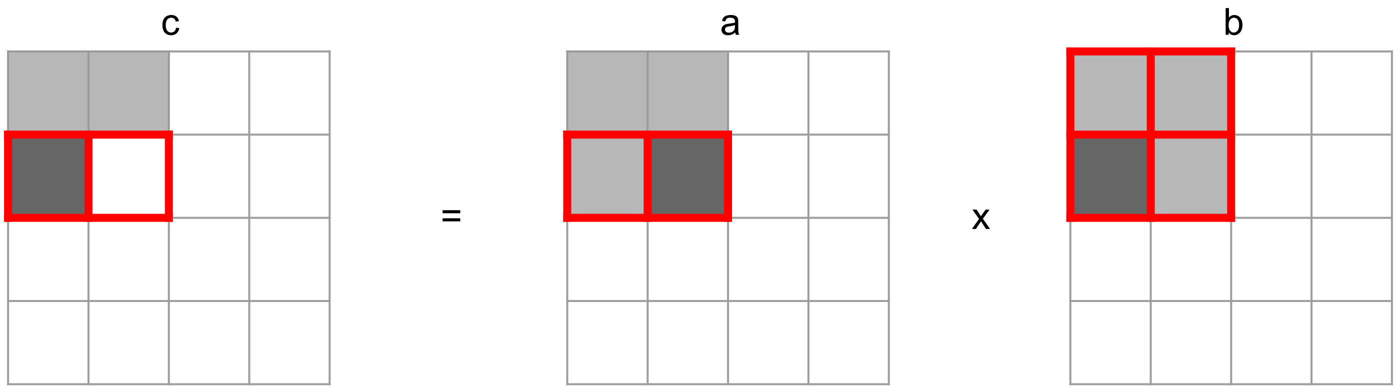
Grey = accessed
Dark grey = currently accessing
Red border = in cache



iter	i	j	k	operation	miss?
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	(m, m)
1	0	0	1	c[0][0] += a[0][1] * b[1][0]	(h, m)
2	0	1	0	c[0][1] += a[0][0] * b[0][1]	(h, h)
3	0	1	1	c[0][1] += a[0][1] * b[1][1]	(h, h)
4	1	0	0	c[1][0] += a[1][0] * b[0][0]	(m, h)

Key:

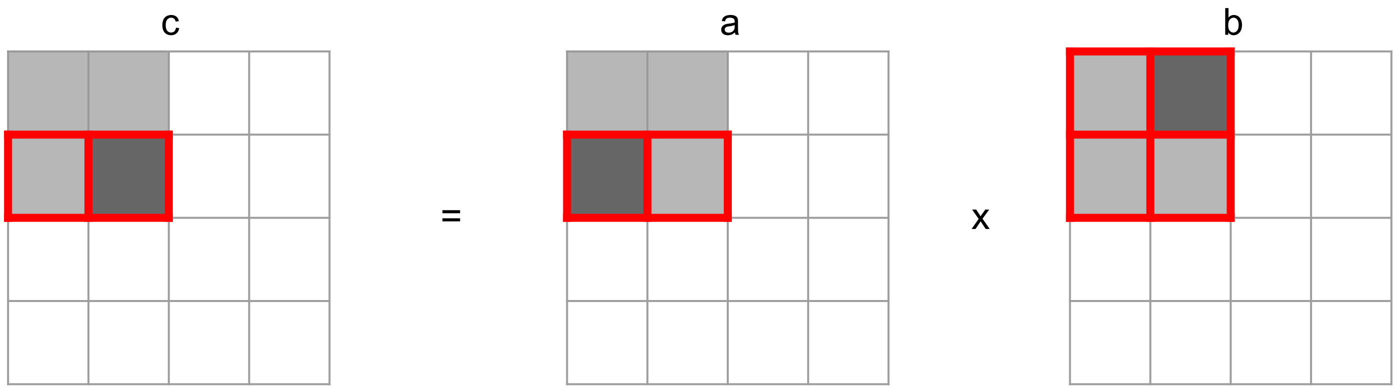
Grey = accessed
Dark grey = currently accessing
Red border = in cache



iter	i	j	k	operation	miss?
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	(m, m)
1	0	0	1	c[0][0] += a[0][1] * b[1][0]	(h, m)
2	0	1	0	c[0][1] += a[0][0] * b[0][1]	(h, h)
3	0	1	1	c[0][1] += a[0][1] * b[1][1]	(h, h)
4	1	0	0	c[1][0] += a[1][0] * b[0][0]	(m, h)
5	1	0	1	c[1][0] += a[1][1] * b[1][0]	(h, h)

Key:

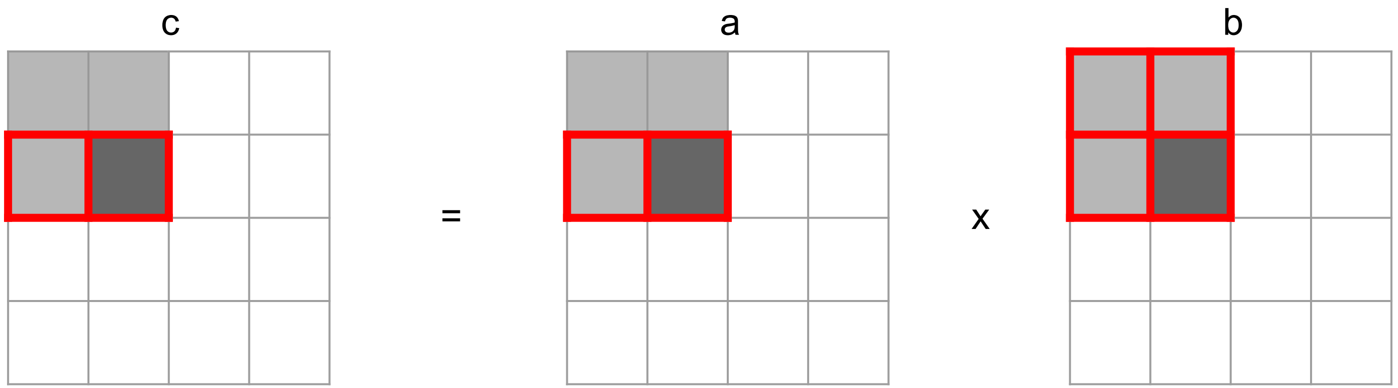
Grey = accessed
Dark grey = currently accessing
Red border = in cache



iter	i	j	k	operation	miss?
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	(m, m)
1	0	0	1	c[0][0] += a[0][1] * b[1][0]	(h, m)
2	0	1	0	c[0][1] += a[0][0] * b[0][1]	(h, h)
3	0	1	1	c[0][1] += a[0][1] * b[1][1]	(h, h)
4	1	0	0	c[1][0] += a[1][0] * b[0][0]	(m, h)
5	1	0	1	c[1][0] += a[1][1] * b[1][0]	(h, h)
6	1	1	0	c[1][1] += a[1][0] * b[0][1]	(h, h)

Key:

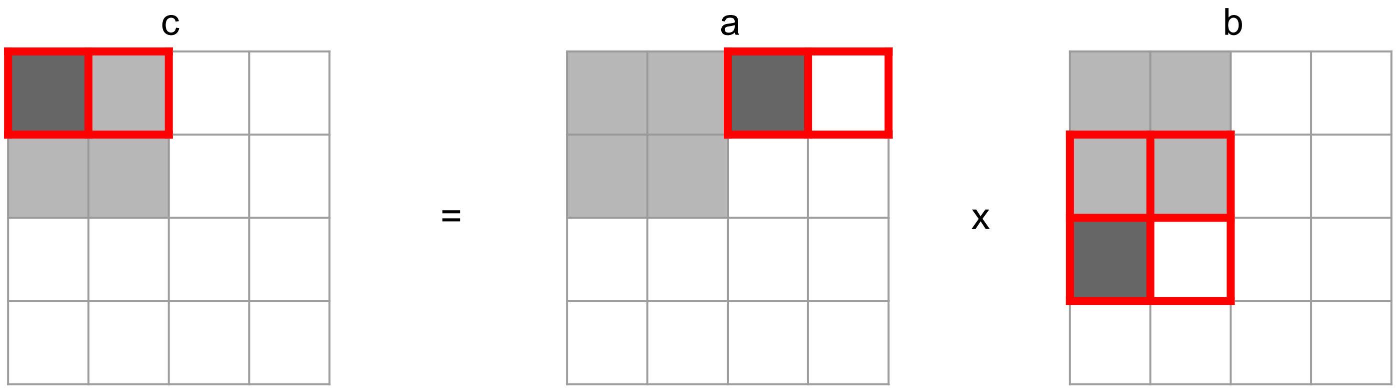
Grey = accessed
Dark grey = currently accessing
Red border = in cache



iter	i	j	k	operation	miss?
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	(m, m)
1	0	0	1	c[0][0] += a[0][1] * b[1][0]	(h, m)
2	0	1	0	c[0][1] += a[0][0] * b[0][1]	(h, h)
3	0	1	1	c[0][1] += a[0][1] * b[1][1]	(h, h)
4	1	0	0	c[1][0] += a[1][0] * b[0][0]	(m, h)
5	1	0	1	c[1][0] += a[1][1] * b[1][0]	(h, h)
6	1	1	0	c[1][1] += a[1][0] * b[0][1]	(h, h)
7	1	1	1	c[1][1] += a[1][1] * b[1][1]	(h, h)

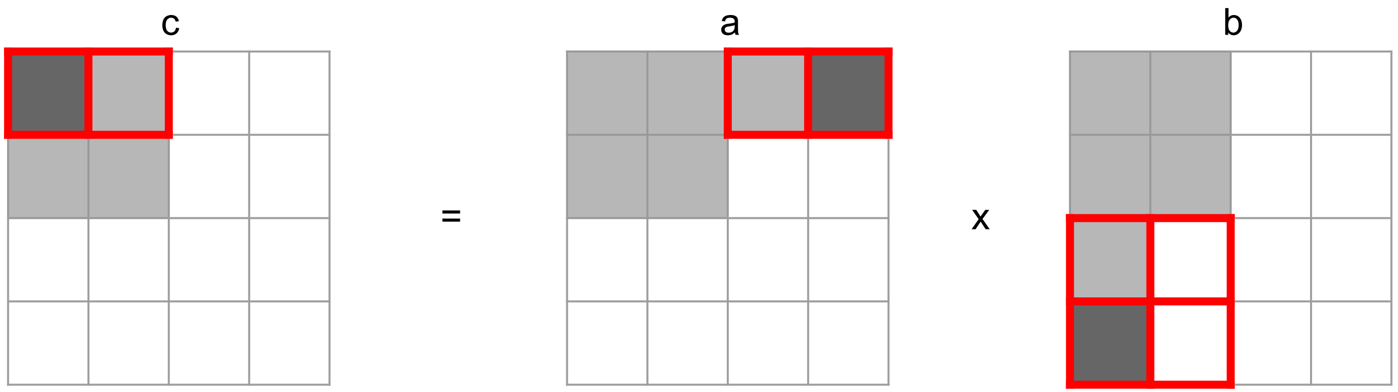
<u>Key:</u>

Grey = accessed
Dark grey = currently accessing
Red border = in cache



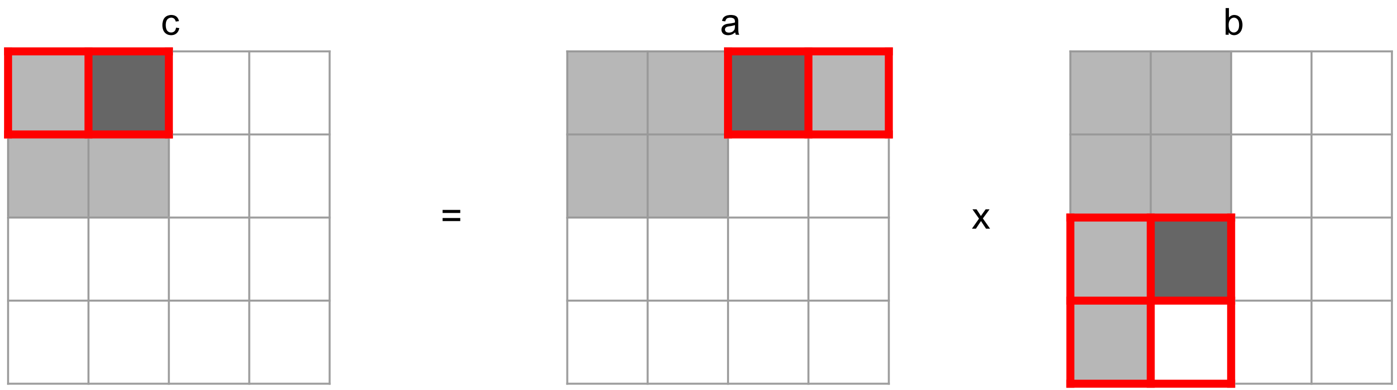
iter	i	j	k	operation	miss?
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	(m, m)
1	0	0	1	c[0][0] += a[0][1] * b[1][0]	(h, m)
2	0	1	0	c[0][1] += a[0][0] * b[0][1]	(h, h)
3	0	1	1	c[0][1] += a[0][1] * b[1][1]	(h, h)
4	1	0	0	c[1][0] += a[1][0] * b[0][0]	(m, h)
5	1	0	1	c[1][0] += a[1][1] * b[1][0]	(h, h)
6	1	1	0	c[1][1] += a[1][0] * b[0][1]	(h, h)
7	1	1	1	c[1][1] += a[1][1] * b[1][1]	(h, h)

iter	i	j	k	operation	miss?
8	0	0	2	c[0][0] += a[0][2] * b[2][0]	(m, m)



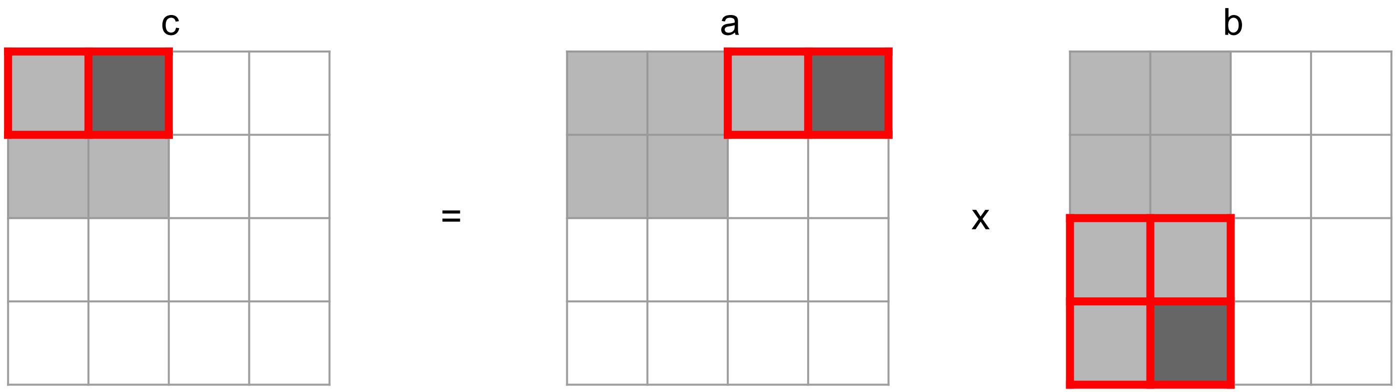
iter	i	j	k	operation	miss?
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	(m, m)
1	0	0	1	c[0][0] += a[0][1] * b[1][0]	(h, m)
2	0	1	0	c[0][1] += a[0][0] * b[0][1]	(h, h)
3	0	1	1	c[0][1] += a[0][1] * b[1][1]	(h, h)
4	1	0	0	c[1][0] += a[1][0] * b[0][0]	(m, h)
5	1	0	1	c[1][0] += a[1][1] * b[1][0]	(h, h)
6	1	1	0	c[1][1] += a[1][0] * b[0][1]	(h, h)
7	1	1	1	c[1][1] += a[1][1] * b[1][1]	(h, h)

iter	i	j	k	operation	miss?
8	0	0	2	c[0][0] += a[0][2] * b[2][0]	(m, m)
9	0	0	3	c[0][0] += a[0][3] * b[3][0]	(h, m)



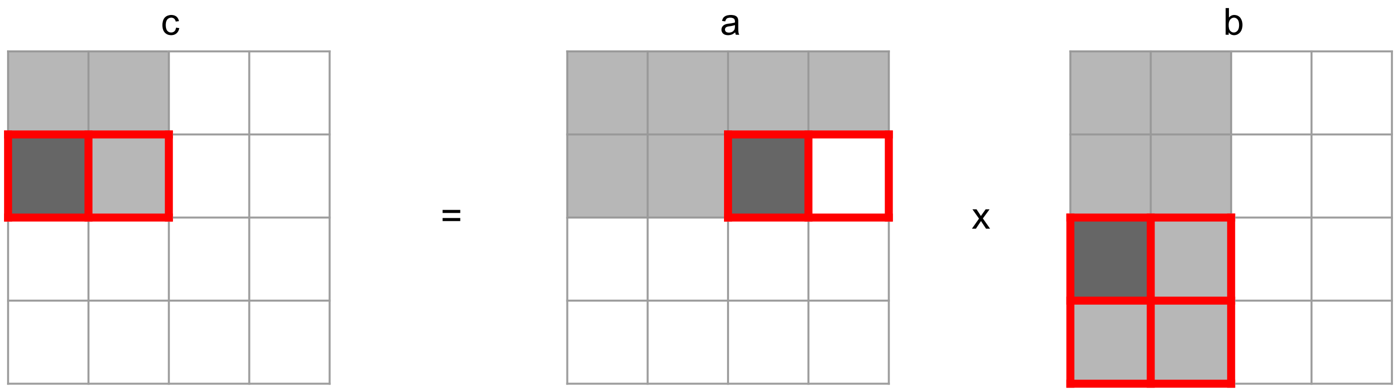
iter	i	j	k	operation	miss?
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	(m, m)
1	0	0	1	c[0][0] += a[0][1] * b[1][0]	(h, m)
2	0	1	0	c[0][1] += a[0][0] * b[0][1]	(h, h)
3	0	1	1	c[0][1] += a[0][1] * b[1][1]	(h, h)
4	1	0	0	c[1][0] += a[1][0] * b[0][0]	(m, h)
5	1	0	1	c[1][0] += a[1][1] * b[1][0]	(h, h)
6	1	1	0	c[1][1] += a[1][0] * b[0][1]	(h, h)
7	1	1	1	c[1][1] += a[1][1] * b[1][1]	(h, h)

iter	i	j	k	operation	miss?
8	0	0	2	c[0][0] += a[0][2] * b[2][0]	(m, m)
9	0	0	3	c[0][0] += a[0][3] * b[3][0]	(h, m)
10	0	1	2	c[0][1] += a[0][2] * b[2][1]	(h, h)



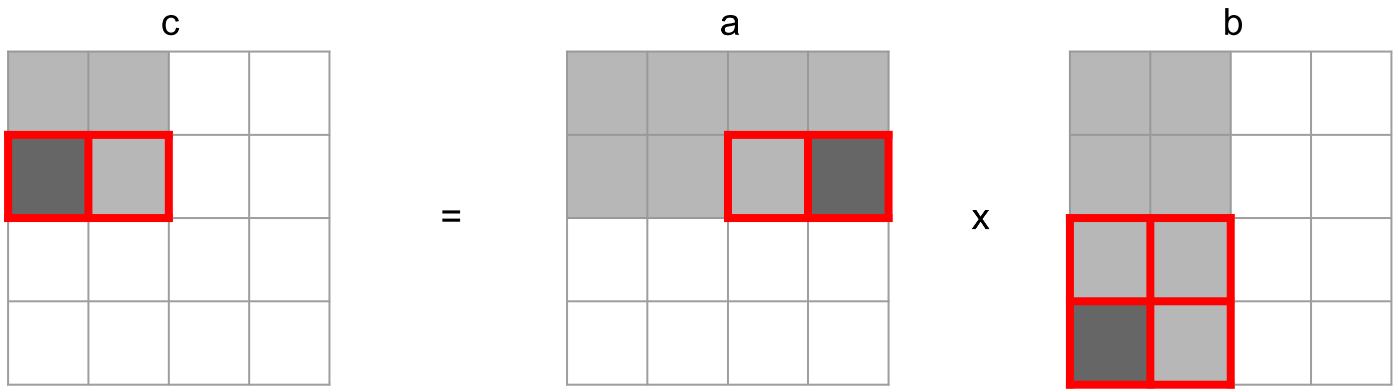
iter	i	j	k	operation	miss?
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	(m, m)
1	0	0	1	c[0][0] += a[0][1] * b[1][0]	(h, m)
2	0	1	0	c[0][1] += a[0][0] * b[0][1]	(h, h)
3	0	1	1	c[0][1] += a[0][1] * b[1][1]	(h, h)
4	1	0	0	c[1][0] += a[1][0] * b[0][0]	(m, h)
5	1	0	1	c[1][0] += a[1][1] * b[1][0]	(h, h)
6	1	1	0	c[1][1] += a[1][0] * b[0][1]	(h, h)
7	1	1	1	c[1][1] += a[1][1] * b[1][1]	(h, h)

iter	i	j	k	operation	miss?
8	0	0	2	c[0][0] += a[0][2] * b[2][0]	(m, m)
9	0	0	3	c[0][0] += a[0][3] * b[3][0]	(h, m)
10	0	1	2	c[0][1] += a[0][2] * b[2][1]	(h, h)
11	0	1	3	c[0][1] += a[0][3] * b[3][1]	(h, h)



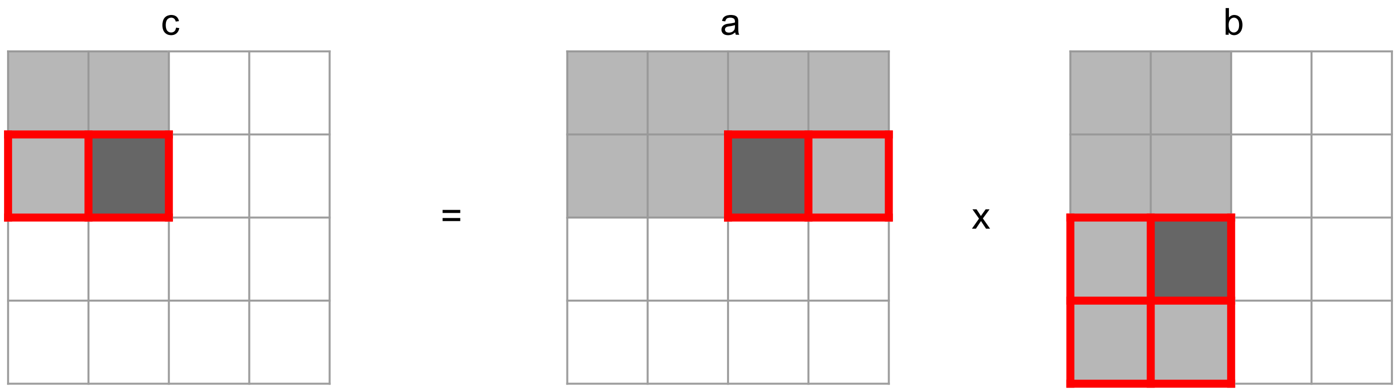
iter	i	j	k	operation	miss?
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	(m, m)
1	0	0	1	c[0][0] += a[0][1] * b[1][0]	(h, m)
2	0	1	0	c[0][1] += a[0][0] * b[0][1]	(h, h)
3	0	1	1	c[0][1] += a[0][1] * b[1][1]	(h, h)
4	1	0	0	c[1][0] += a[1][0] * b[0][0]	(m, h)
5	1	0	1	c[1][0] += a[1][1] * b[1][0]	(h, h)
6	1	1	0	c[1][1] += a[1][0] * b[0][1]	(h, h)
7	1	1	1	c[1][1] += a[1][1] * b[1][1]	(h, h)

iter	i	j	k	operation	miss?
8	0	0	2	c[0][0] += a[0][2] * b[2][0]	(m, m)
9	0	0	3	c[0][0] += a[0][3] * b[3][0]	(h, m)
10	0	1	2	c[0][1] += a[0][2] * b[2][1]	(h, h)
11	0	1	3	c[0][1] += a[0][3] * b[3][1]	(h, h)
12	1	0	2	c[1][0] += a[1][2] * b[2][0]	(m, h)

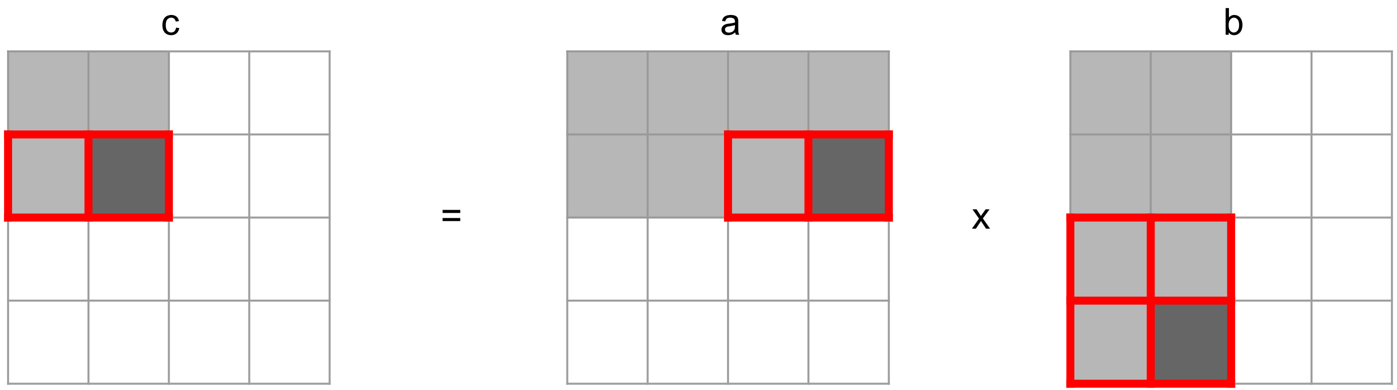


iter	i	j	k	operation	miss?
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	(m, m)
1	0	0	1	c[0][0] += a[0][1] * b[1][0]	(h, m)
2	0	1	0	c[0][1] += a[0][0] * b[0][1]	(h, h)
3	0	1	1	c[0][1] += a[0][1] * b[1][1]	(h, h)
4	1	0	0	c[1][0] += a[1][0] * b[0][0]	(m, h)
5	1	0	1	c[1][0] += a[1][1] * b[1][0]	(h, h)
6	1	1	0	c[1][1] += a[1][0] * b[0][1]	(h, h)
7	1	1	1	c[1][1] += a[1][1] * b[1][1]	(h, h)

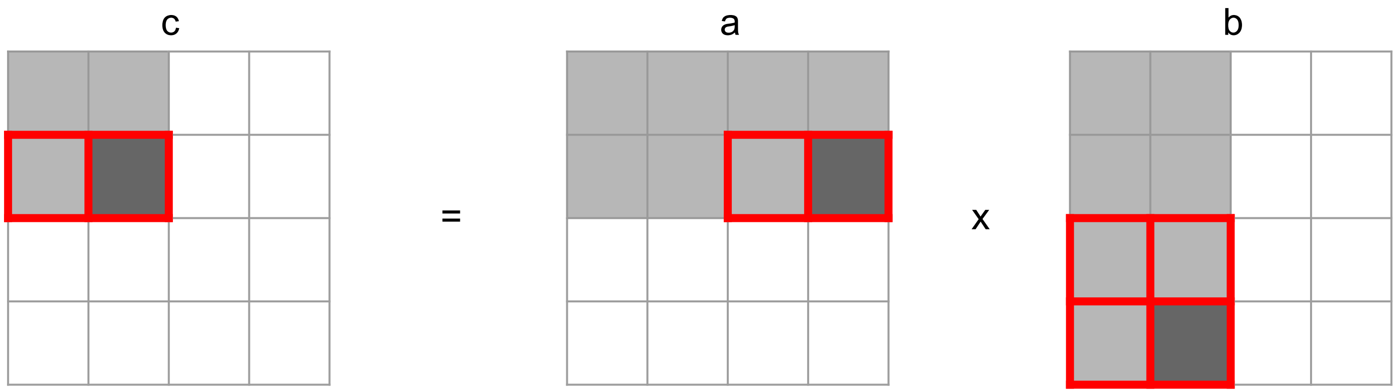
iter	i	j	k	operation	miss?
8	0	0	2	c[0][0] += a[0][2] * b[2][0]	(m, m)
9	0	0	3	c[0][0] += a[0][3] * b[3][0]	(h, m)
10	0	1	2	c[0][1] += a[0][2] * b[2][1]	(h, h)
11	0	1	3	c[0][1] += a[0][3] * b[3][1]	(h, h)
12	1	0	2	c[1][0] += a[1][2] * b[2][0]	(m, h)
13	1	0	3	c[1][0] += a[1][3] * b[3][0]	(h, h)



iter	i	j	k	operation	miss?	iter	i	j	k	operation	miss?
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	(m, m)	8	0	0	2	c[0][0] += a[0][2] * b[2][0]	(m, m)
1	0	0	1	c[0][0] += a[0][1] * b[1][0]	(h, m)	9	0	0	3	c[0][0] += a[0][3] * b[3][0]	(h, m)
2	0	1	0	c[0][1] += a[0][0] * b[0][1]	(h, h)	10	0	1	2	c[0][1] += a[0][2] * b[2][1]	(h, h)
3	0	1	1	c[0][1] += a[0][1] * b[1][1]	(h, h)	11	0	1	3	c[0][1] += a[0][3] * b[3][1]	(h, h)
4	1	0	0	c[1][0] += a[1][0] * b[0][0]	(m, h)	12	1	0	2	c[1][0] += a[1][2] * b[2][0]	(m, h)
5	1	0	1	c[1][0] += a[1][1] * b[1][0]	(h, h)	13	1	0	3	c[1][0] += a[1][3] * b[3][0]	(h, h)
6	1	1	0	c[1][1] += a[1][0] * b[0][1]	(h, h)	14	1	1	2	c[1][1] += a[1][2] * b[2][1]	(h, h)
7	1	1	1	c[1][1] += a[1][1] * b[1][1]	(h, h)						



iter	i	j	k	operation	miss?	iter	i	j	k	operation	miss?
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	(m, m)	8	0	0	2	c[0][0] += a[0][2] * b[2][0]	(m, m)
1	0	0	1	c[0][0] += a[0][1] * b[1][0]	(h, m)	9	0	0	3	c[0][0] += a[0][3] * b[3][0]	(h, m)
2	0	1	0	c[0][1] += a[0][0] * b[0][1]	(h, h)	10	0	1	2	c[0][1] += a[0][2] * b[2][1]	(h, h)
3	0	1	1	c[0][1] += a[0][1] * b[1][1]	(h, h)	11	0	1	3	c[0][1] += a[0][3] * b[3][1]	(h, h)
4	1	0	0	c[1][0] += a[1][0] * b[0][0]	(m, h)	12	1	0	2	c[1][0] += a[1][2] * b[2][0]	(m, h)
5	1	0	1	c[1][0] += a[1][1] * b[1][0]	(h, h)	13	1	0	3	c[1][0] += a[1][3] * b[3][0]	(h, h)
6	1	1	0	c[1][1] += a[1][0] * b[0][1]	(h, h)	14	1	1	2	c[1][1] += a[1][2] * b[2][1]	(h, h)
7	1	1	1	c[1][1] += a[1][1] * b[1][1]	(h, h)	15	1	1	3	c[1][1] += a[1][3] * b[3][1]	(h, h)



iter	i	j	k	operation	miss?
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	(m, m)
1	0	0	1	c[0][0] += a[0][1] * b[1][0]	(h, m)
2	0	1	0	c[0][1] += a[0][0] * b[0][1]	(h, h)
3	0	1	1	c[0][1] += a[0][1] * b[1][1]	(h, h)
4	1	0	0	c[1][0] += a[1][0] * b[0][0]	(m, h)
5	1	0	1	c[1][0] += a[1][1] * b[1][0]	(h, h)
6	1	1	0	c[1][1] += a[1][0] * b[0][1]	(h, h)
7	1	1	1	c[1][1] += a[1][1] * b[1][1]	(h, h)

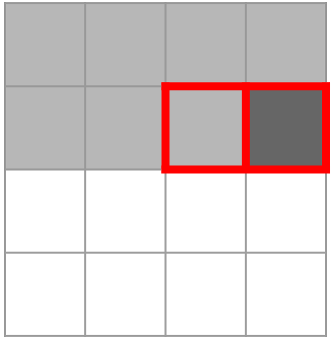
iter	i	j	k	operation	miss?
8	0	0	2	c[0][0] += a[0][2] * b[2][0]	(m, m)
9	0	0	3	c[0][0] += a[0][3] * b[3][0]	(h, m)
10	0	1	2	c[0][1] += a[0][2] * b[2][1]	(h, h)
11	0			* b[3][1]	(h, h)
12	1			* b[2][0]	(m, h)
13	1	0	3	c[1][0] += a[1][3] * b[3][0]	(h, h)
14	1			* b[2][1]	(h, h)
15	1			* b[3][1]	(h, h)

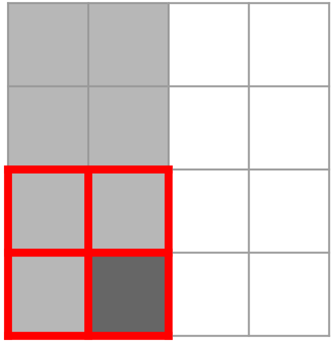
What is the miss rate of a?

What is the miss rate of b?

What happened?

- Good temporal locality!
- Blocks are reused while they are still in the cache



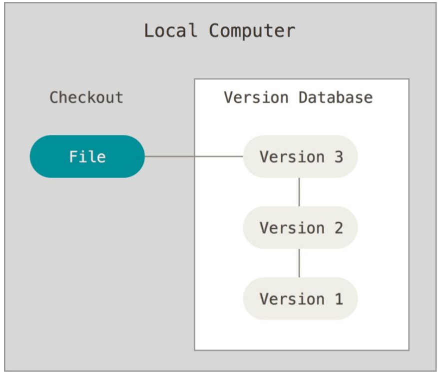


Version control is your friend

Introduction to Git

What is Git?

- Most widely used version control system out there
- Version control:
  - Help track changes to your source code over time
  - Help teams manage changes on shared code



Git Commands

- Clone: git clone <clone-repository-url>
- Add: git add <file-name> or git commit <file-name>
- Commit: git commit -m “your-commit-message”
  - Good commit messages are key!
  - Bad:“commit”, “change”, “fixed”
  - Good: “Fixed buffer overflow potential in AttackLab”
- Push / Pull: git push / git pull

If you get stuck…

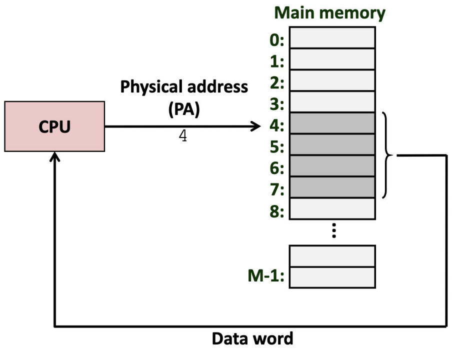
- Reread the writeup
- Look at CS:APP Chapter 6
- Review lecture notes (http://cs.cmu.edu/~213)
- Come to Office Hours
- Post private question on Piazza
- `man malloc`, `man valgrind`, `man gdb`

Further Content: Virtual Memory

Memory isn’t real

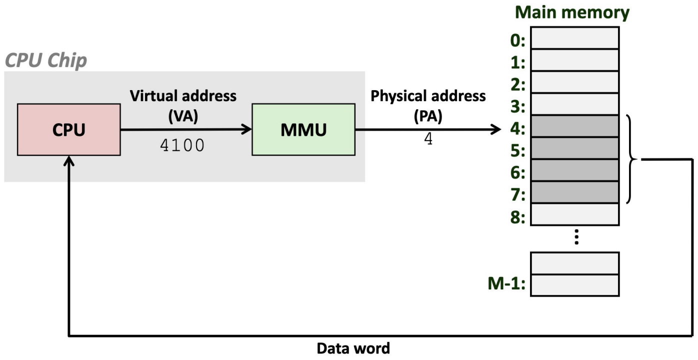
Review: What Is Virtual Memory?

Physical Addressing



Memory address refers to an exact location in memory—only used in simple systems

Virtual Addressing



Memory address refers to a process-specific address, mapped to physical memory via the hardware memory management unit.

One of the Great Ideas Of Computer Science™

What Is Virtual Memory and Why Should I Care?

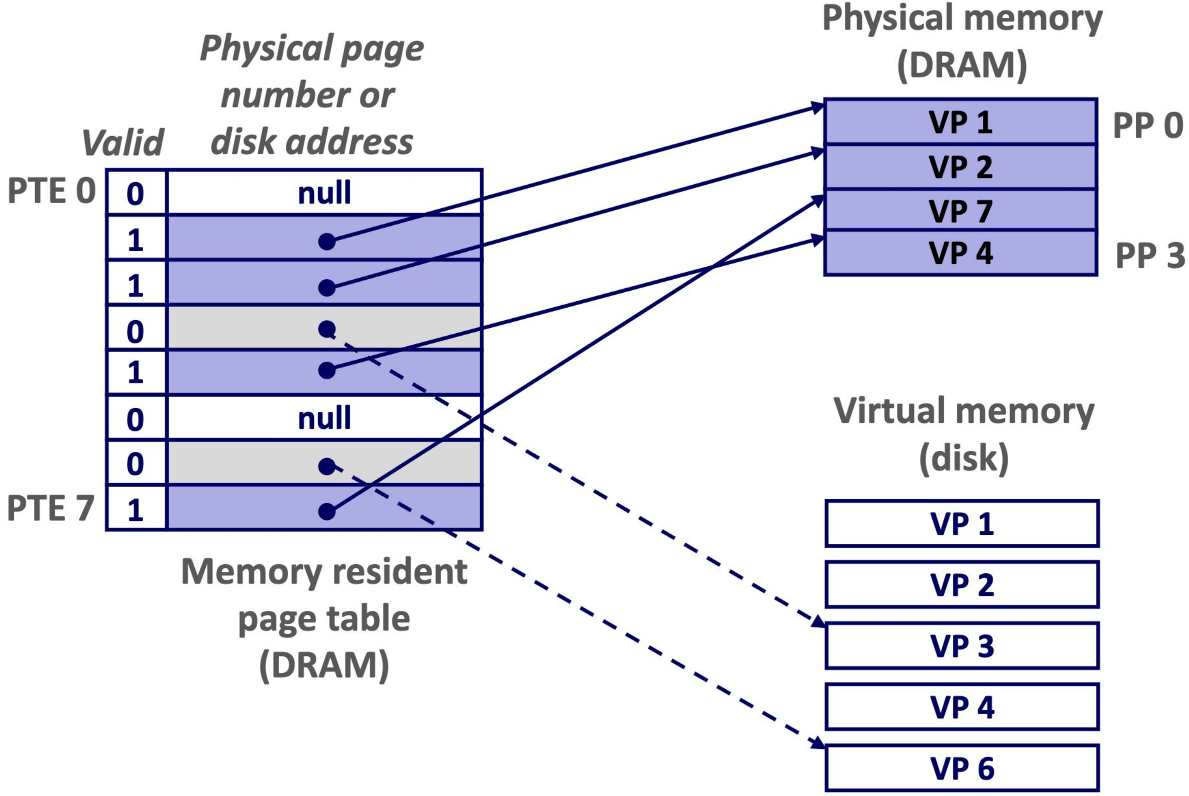
- Virtual memory is conceptually a byte array stored on disk, where the contents are cached in DRAM.
- Each process gets its own address space, mapped to memory or disk.
- This allows isolating memory per-process, improving security and allowing significantly more implementation flexibility.

Page Table

Virtual addresses are mapped to physical addresses in the page table. Each entry is called a page table entry.

Pages are in memory, like a cache. If they are not available in memory, we have a page miss.

A page miss causes a page fault, which causes the OS to fetch the page from disk and evict a page from DRAM.

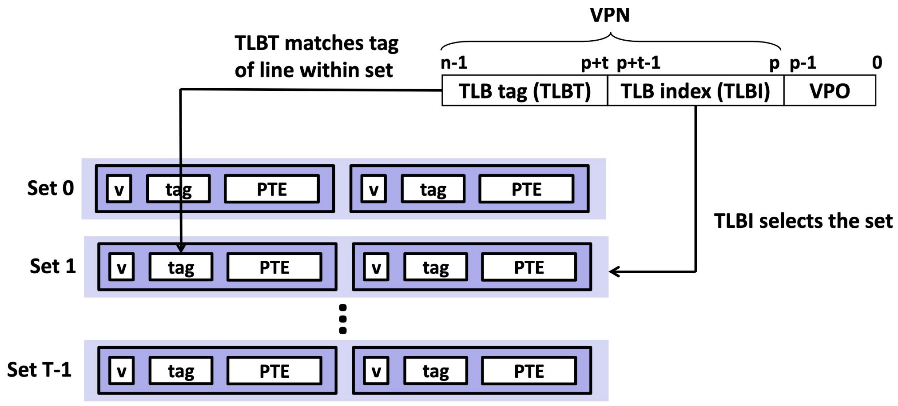


The TLB

Small cache within the MMU, caching page table entries to reduce accesses to the page table.

A portion of the address, the VPN, is used to index into the TLB. The tag and index for the internal sets are within the VPN. We can then get the PTE and the physical address, if we have a hit.

Otherwise, query the page table in memory as usual.



Multi-Level Page Tables

The size of a page table quickly gets out of control when we have to address large addresses space.

The solution is to nest page tables. The VPO/PPO acts as the pseudo-"block offset"