[CSED211] Introduction to Computer Software Systems

Lab 6: Cache Lab

Dowon Son

COMPUTER ARCHITECTURE & OPERATING SYSTEMS LABORATORY

2024.11.14

Today's Agenda

- Background

- Cache Lab Part A: Building Cache Simulator
  - Valgrind
  - File I/O APIs
  - Dynamic Allocation & Deallocation
  - Parsing Command Line Options

- Cache Lab Part B: Efficient Matrix Transpose
  - Hit Ratio
  - Matrix Multiplication

Today’s Agenda

- Background

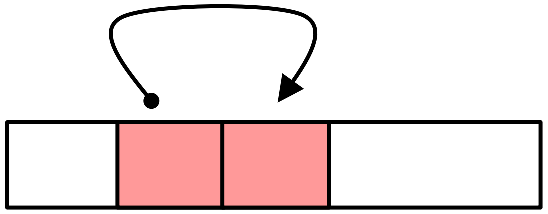
- Cache Lab Part A: Building Cache Simulator
  - Valgrind
  - File I/O APIs
  - Dynamic Allocation & Deallocation
  - Parsing Command Line Options

- Cache Lab Part B: Efficient Matrix Transpose
  - Hit Ratio
  - Matrix Multiplication

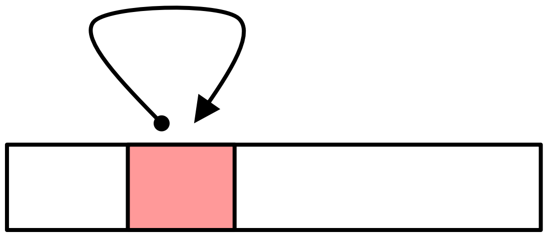
Background: Locality

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

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



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



Background: Cache Memory

- Cache memory is small, fast SRAM-based memory
  - Automatically managed in hardware
  - Holds frequently accessed blocks of main memory

- CPU first looks for data in caches (e.g., L1, L2, and L3), then in main memory

Background: Cache Memory (Cont.)

- Cache set
  - Cache memory consists of $S\ (= 2^s)$ cache sets

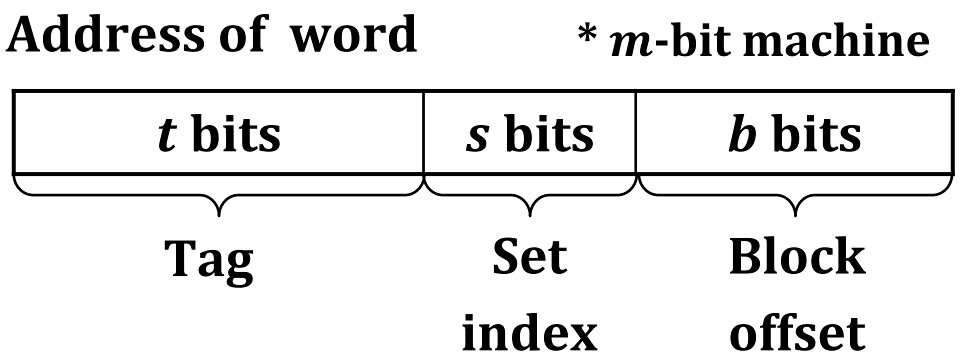
- Cache line
  - Each cache set consists of E (associativity) cache lines
  - Each cache line consists of valid bit, tag, and (2^b-byte) data block

- Valid bit: Shows the cache line is valid or not

- Tag: Compares the tag of the cache line with the currently accessed memory address to check for a match

- Data block: Saves the loaded data

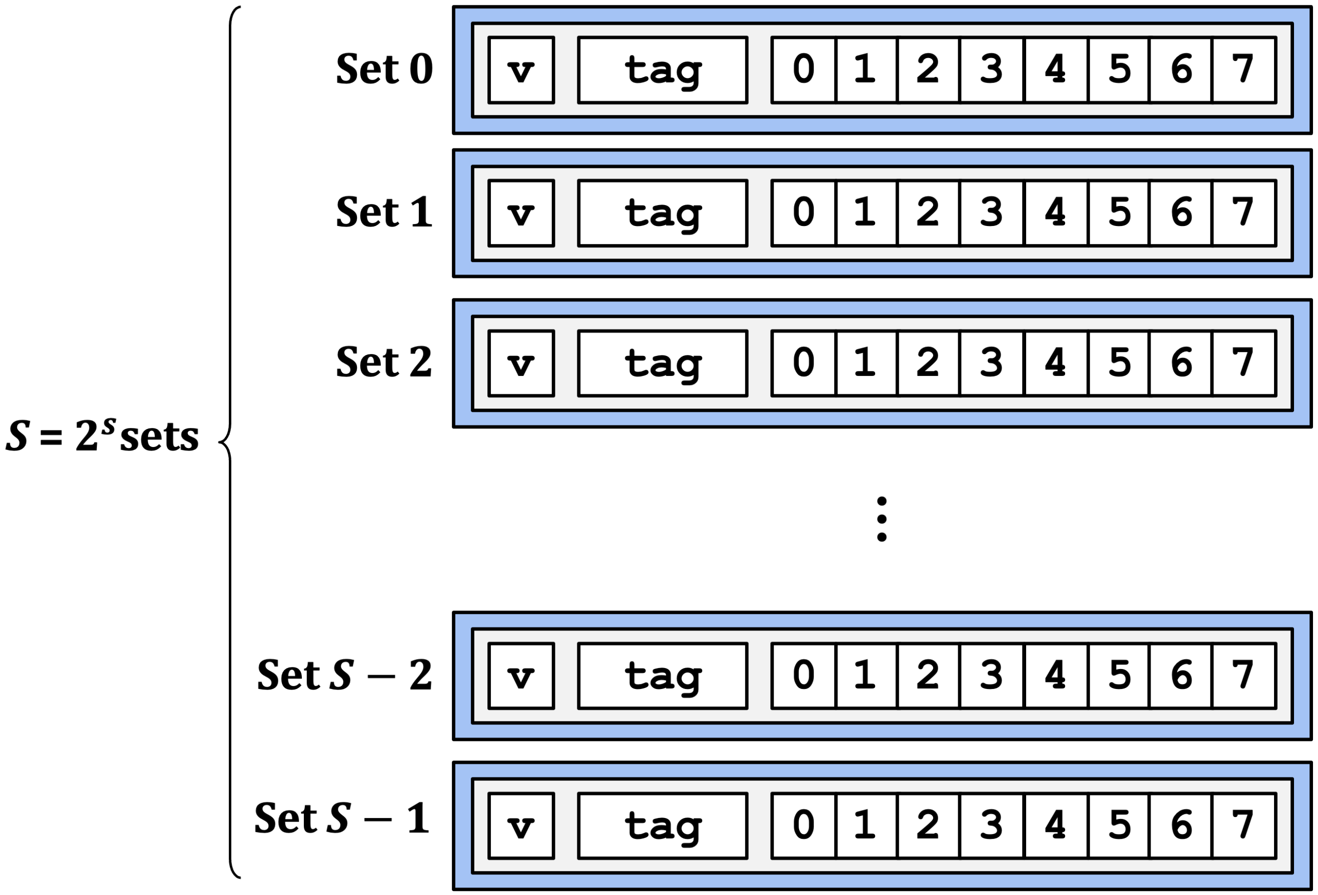
Background: Memory Address Format

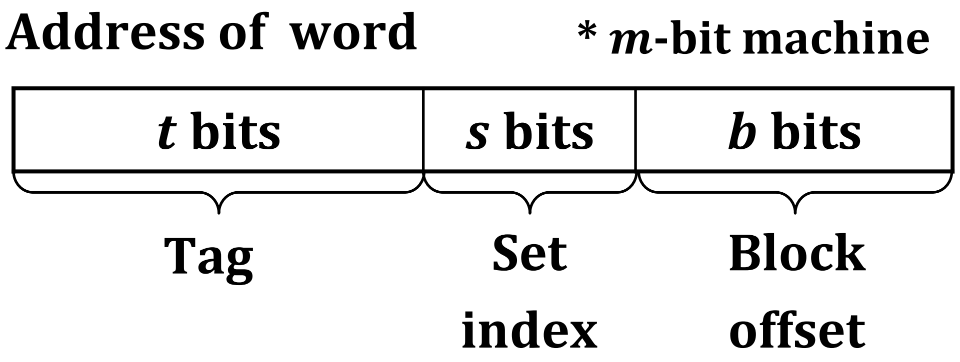


- Block offset: b bits
- Set index: s bits
- Tag: $m - (s + b)$ bits

Background: Direct Mapped Cache

- Each cache set consists of a single cache line



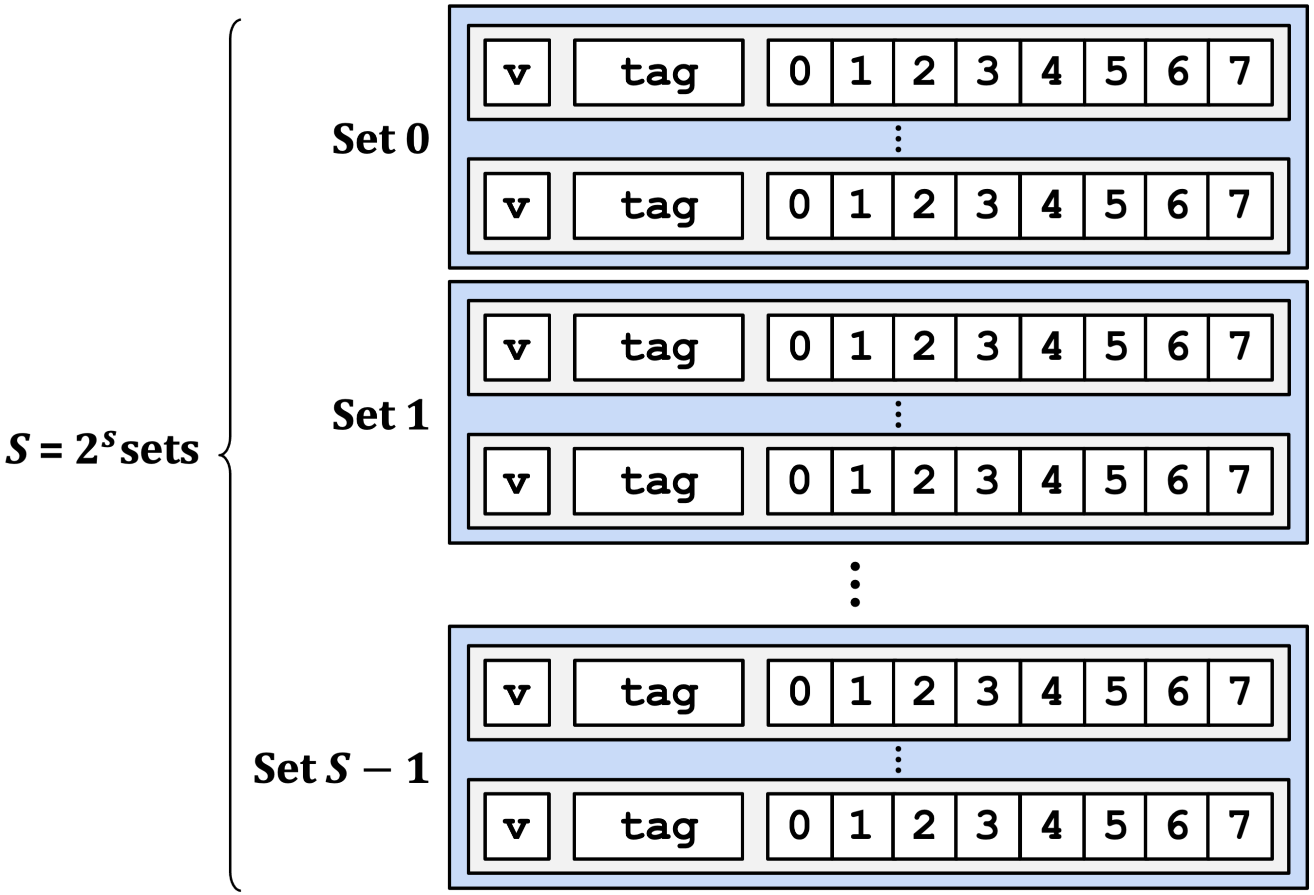


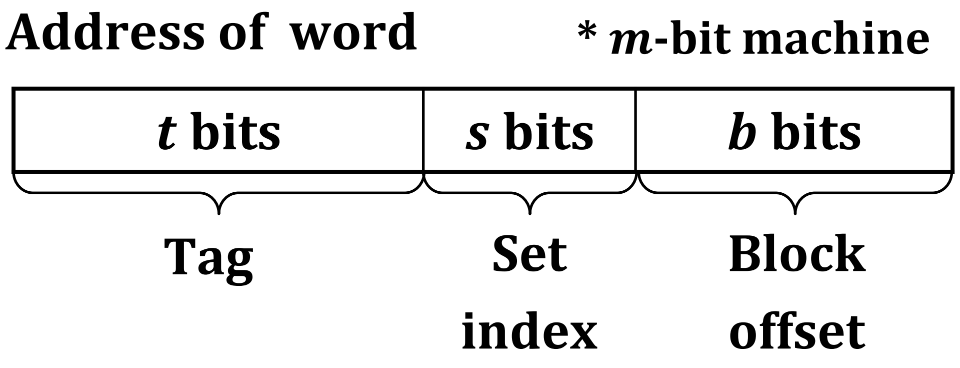
Associativity (E)	Mapping
1	Direct mapping
$1 < E < C/2^b$	Set associative mapping
$C/2^b$	Fully associative mapping

* C: Cache size

Background: Set Associative Mapped Cache

- Each cache set consists of E (associativity) cache lines



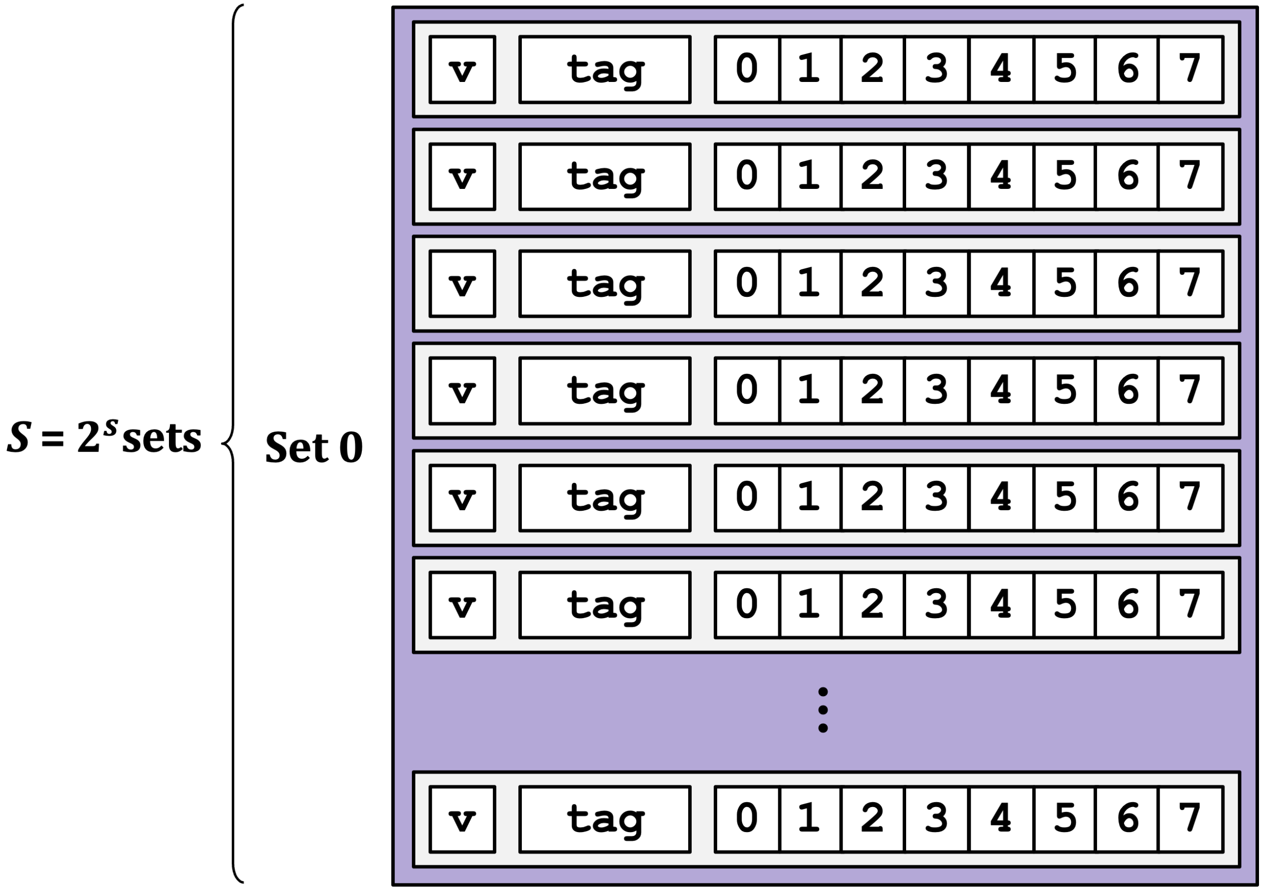


Associativity (E)	Mapping
1	Direct mapping
$1 < E < C/2^b$	Set associative mapping
$C/2^b$	Fully associative mapping

* C: Cache size

Background: Fully Associative Mapped Cache

- Cache has only one set that is composed of $C/2^b$ cache lines



Address of word

* m-bit machine

t bits	b bits
Tag	Block offset

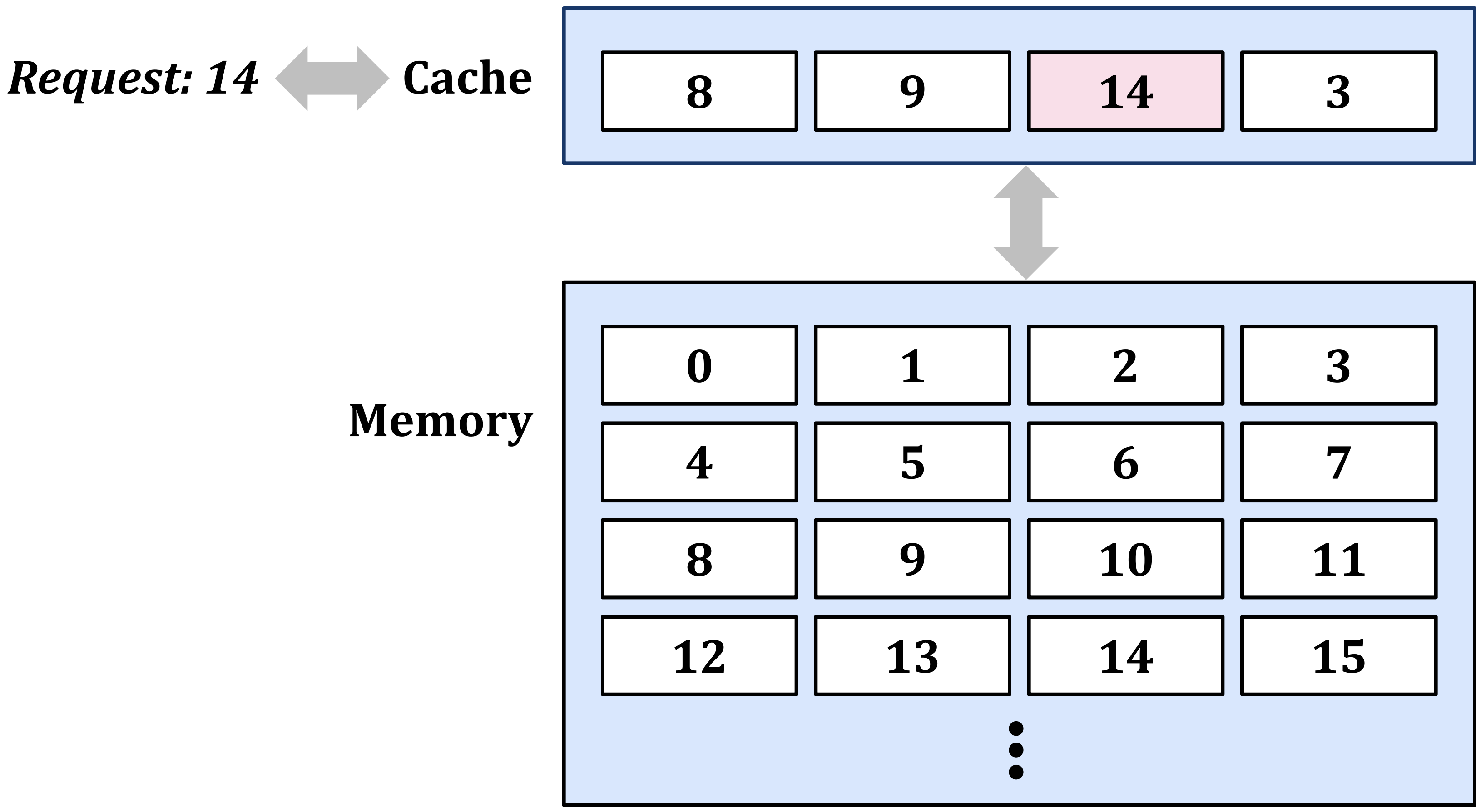
No Set Index ($s = 0$ bit)

Associativity (E)	Mapping
1	Direct mapping
$1 < E < C/2^b$	Set associative mapping
$C/2^b$	Fully associative mapping

* C: Cache size

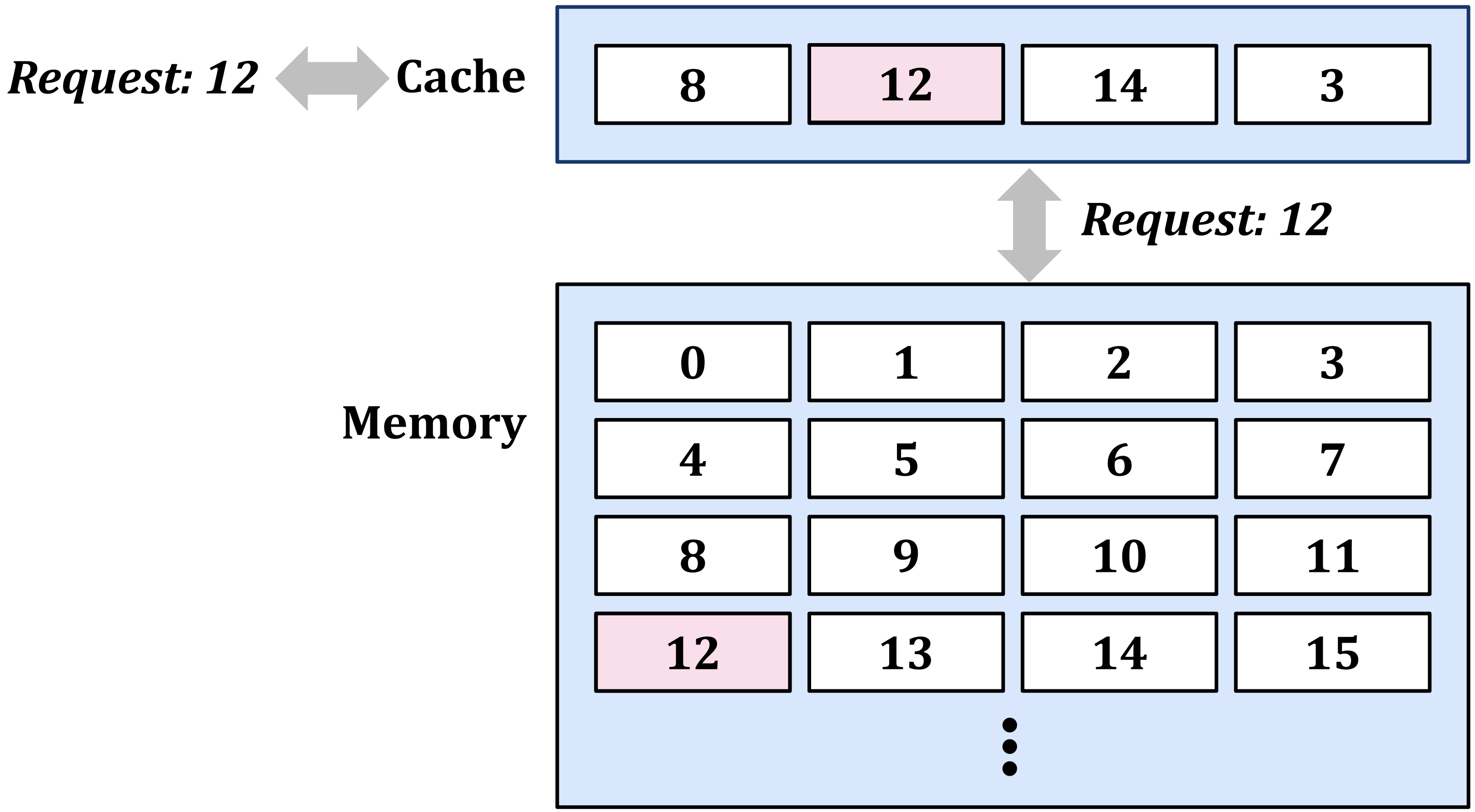
Background: Cache Hit

- Hit: An access by program to a block that is in cache



Background: Cache Miss

- Miss: An access by program to a block that is not in cache



Background: Cache Replacement Policy

- Size of cache memory is limited
  - Not all blocks from main memory can be stored in a cache

- Cache memory needs to replace old block with a new block
  - Replacement policy: Determines which block gets evicted

- Replacement policy algorithms
  - LRU: Replace least recently used line with the new one
    - Use counter to trace recently accessed line
    - Reference cache simulator use LRU replacement policy
  - FIFO, LIFO, etc.

Cache Lab: Overview

- Cache lab consists of Part A & B
  - Part A: Cache simulator
    - Goal: Understand the mechanism of cache memory and design cache simulator
    - Todo: Write a program simulates cache memory access and counts hit/miss/eviction
  - Part B: Efficient matrix transpose
    - Goal: Optimize matrix transpose ($A \rightarrow A^T$)
    - Todo: Write an efficient code for the highest hit ratio (i.e., minimize the cache miss)
- Submit code files and your lab report (in pdf)
  - Source code name: `[student id]_csim.c`, `[student id]_trans.c`
    e.g., `20242057_csim.c`, `20242057_trans.c`
    (Other formats will not be accepted)
  - Report name: `[student id].pdf` (e.g., `20242057.pdf`)

Today's Agenda

- Background
- Cache Lab Part A: Building Cache Simulator
  - Valgrind
  - File I/O APIs
  - Dynamic Allocation & Deallocation
  - Parsing Command Line Options
- Cache Lab Part B: Efficient Matrix Transpose
  - Hit Ratio
  - Matrix Multiplication

Cache Lab Part A: Building Cache Simulator

- Todo
  - Before starting Part A, read `writeup_cachelab.pdf`
  - Write a program which simulates cache memory access and count hit/miss/eviction

- Goals
  - Understand the mechanism of cache memory and design cache simulator

- Input & Output
  - Input file is pre-generated by Valgrind tool
  - Output contains hit/miss/eviction for each instruction in input file

Valgrind

- Tool collection for debugging and profiling programs to identify memory management issues

- Usage: `$ valgrind --tool=[tool_name] [option] [program]`
  - e.g., `$ valgrind --tool=lackey ls`

- Installation (in Ubuntu)
  - `$ sudo apt install valgrind`

Tools	Function
`Memcheck`	Memory Proifiler
`Cachegrind`	Cache Profiler
`Callgrind`	Extension of Cachegrind
`Massif`	Heap Profiler
`Helgrind`	Multi-threaded Program Debugger
`Lackey`	Simple Profiler

Example: Valgrind Lackey

- Lackey: Simple Valgrind tool for profiling basic operation of a program
- Other tools (e.g., massif, cachegrind) can profile various activities, such as heap usage and cache performance
  - For more details, check valgrind manual
  - https://linux.die.net/man/1/valgrind

Program to execute

```
$ valgrind --log-fd=1 --tool=lackey -v --trace-mem=yes ./your_program
```

Output logs to specified file descriptor #

Tool name

Verbose option (Print detailed logs)

Memory trace option (Trace almost every memory access made by the program)

File Descriptor	#
standard input	0
standard output	1
standard error	2

Valgrind Output

- Format: [operation] [address], [size]

- Operation: I (instruction load), L (data load), S (data store), M (data modify)
  - In this lab, only care about data cache activity and ignore instruction load
  - M (data modify) means one load and one store

- Address: Virtual memory address that program accessed (in HEX format)

- Size: Number of bytes used in operation

```
 S 0064320c,4
 L 00603108,4
 S 00643318,4
 L 0060310c,4
 S 00643424,4
 L 00603110,4
 S 00643530,4
 L 00603114,4
 S 0064363c,4
```

File I/O APIs

- In this lab, file I/O APIs are used to read trace file (Valgrind output)

- High-level standard I/O library (e.g., `fopen()`, `fclose()`)
  - `fscanf()`: Formatted input (for reading line from trace file)
  - `fgets()`: Read a string of n bytes

- Typical usage
  - ```
    fp = open("text.txt", "r");
    // Do something with the file
    fclose(fp);
    ```
    ```

Dynamic Memory Allocation & Deallocation

- In this lab, cache memory configuration (e.g., # of cache sets, # of cache lines per set, block size) can be different by options
  - Designing simulator may need dynamic memory allocation and deallocation

- `malloc()`: Allocate consecutive dynamic memory space in heap
  - e.g., `int *p =(int*) malloc(sizeof(int) * NUM_ELEMS);`

- `free()`: De-allocates specified memory space allocated by `malloc`
  - e.g., `free(p)`

Parsing Command Line Options

- In this lab, parsing command line options is necessary
  - e.g., help flag, verbose flag, target trace file, cache configuration

- Short options: Single-character option preceded by a single hyphen (-)
  - e.g., "`ls -l`", "`df -h`", "`gcc -o [arg]`"

- Long options: Descriptive option preceded by two hyphens (--)
  - e.g., "`ls --help`", "`gcc --version`", "`gcc --std=c11`"

Parsing Command Line Options (Cont.)

- Option parsing functions: Included in `getopt.h` header file
  - `getopt()`: Parse short options
  - `getopt_long()`: Parse short + long options

- In this lab, only `getopt()` function is required
  - Function prototype

```
int getopt(int argc, char * const argv[], const char *optstring);
```

    - Returns ASCII value of parsed short option (single character)
    - `argc`: Number of arguments (passed through `main` function)
    - `argv[]`: Array of pointers to argument strings (passed through `main` function)
    - `optstring`: A string that specifies the options to process
      - Colon (:) means the left option needs option argument
      - e.g., “`ho:`”: “`-o`” option needs argument, but “`-h`” option does not need argument

Parsing Command Line Options (Cont.)

- Example usage

```
int main(int argc, char *argv[]){
    int opt;
    char *outputFile;
    while((opt = getopt(argc, argv, "ho:")!=-1){
        switch(opt){
            case 'h':
                print_help();
                break;
            case 'o':
                /* extern char *optarg
                   is defined in getopt.h */
                outputFile = optarg;
                print_output(outputFile);
                break;
        }
    }
    return 0;
}
```

Programming Rules for Part A

- Your `csim.c` file must compile without warnings to receive credit

- You must:
  1. Design a simulator that works correctly for arbitrary `s`, `E`, and `b`
  2. Ignore all instruction cache accesses (`"I"`)
  3. Call the function `printSummary()`, at the end of your main function
     - e.g., `printSummary(hit_count, miss_count, eviction_count);`
  4. Assume that a single memory access never crosses block boundaries
     - By making this assumption, you can ignore the request sizes in the Valgrind traces

Today’s Agenda

- Background
- Cache Lab Part A: Building Cache Simulator
  - Valgrind
  - File I/O APIs
  - Dynamic Allocation & Deallocation
  - Parsing Command Line Options
- Cache Lab Part B: Efficient Matrix Transpose
  - Hit Ratio
  - Matrix Multiplication

Cache Lab Part B: Efficient Matrix Transpose

- Todo
  - Before starting Part B, read `writeup_cachelab.pdf`
  - Write the efficient code for the highest hit ratio (i.e., minimize the cache miss)

- Goals
  - Optimize matrix transpose ($A \rightarrow A^T$)

- Cache Configuration
  - Direct mapped (E=1) cache
  - 1-kilobyte cache size
  - 32-byte (b=5) block size
  - 32 sets (s=5) in cache

Hit Ratio

- Hit Ratio
  - Ratio of accesses that result in cache hits (# of hit / # of total access)

- Example
  - 32-byte directed mapped cache with 16-byte block size
  - Row-major order accesses

`int A[4][4]`

M 0	H 1	H 2	H 3
M 4	H 5	H 6	H 7
M 8	H 9	H 10	H 11
M 12	H 13	H 14	H 15

Cache

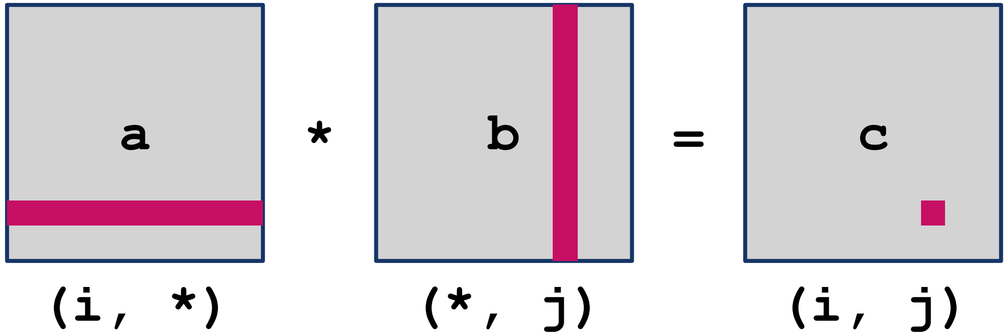
8	1	2	3
12	5	6	7

Hit Ratio = 3/4

Matrix Multiplication w/o Blocking

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

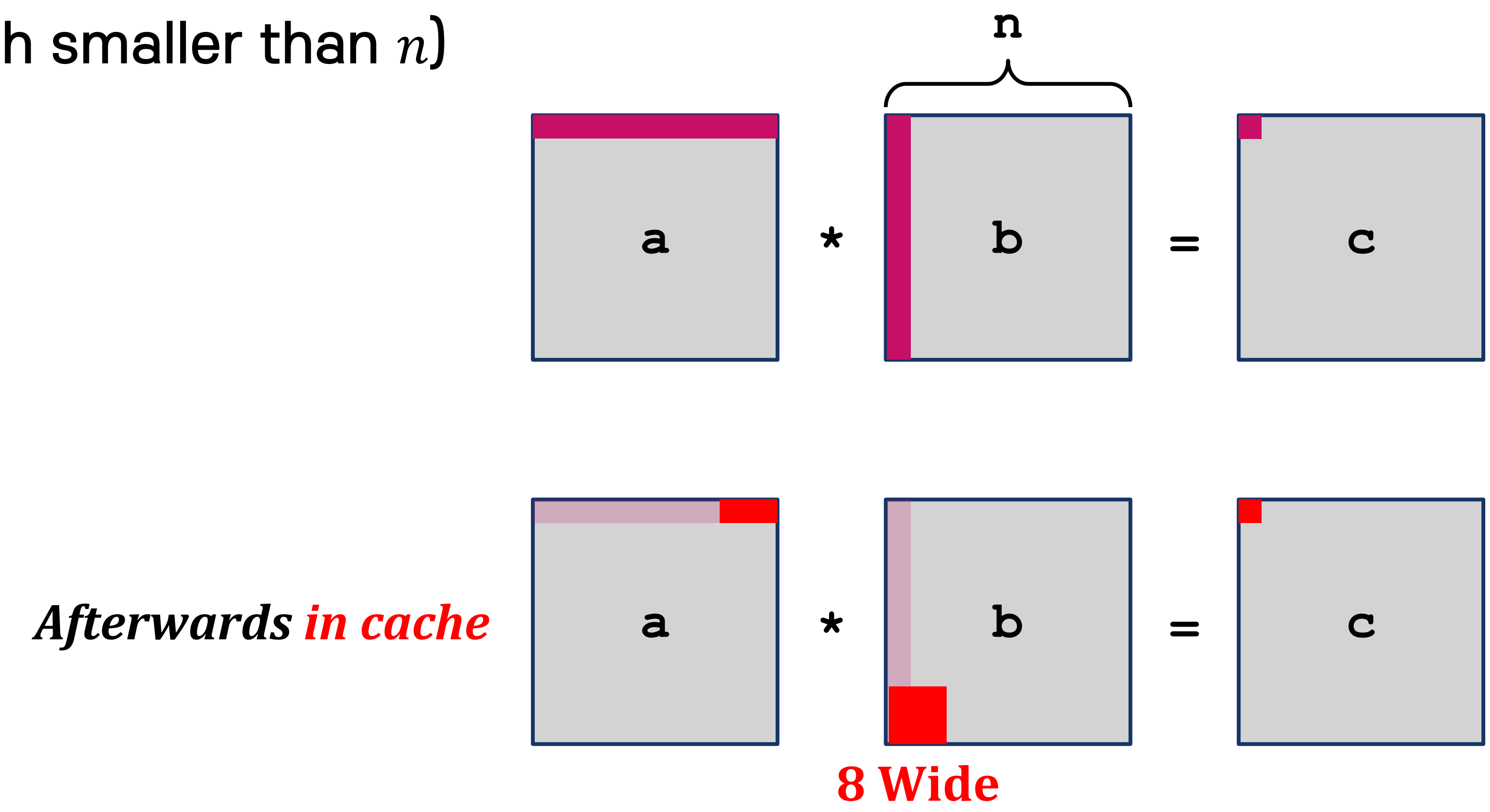
/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n){
  int i, j, k;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      for (k = 0; k < n; k++)
        c[i * n + j] += a[i * n + k] * b[k * n + j];
}
```



Cache Miss Analysis

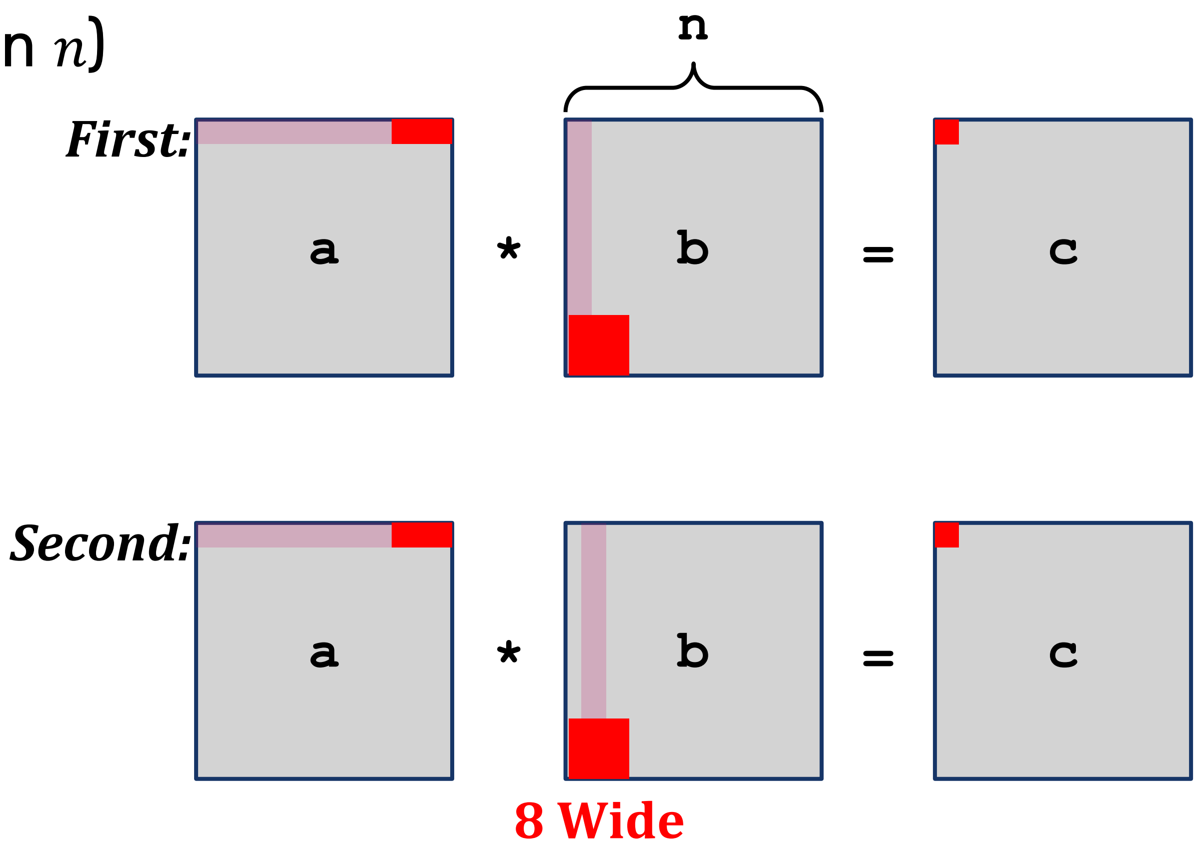
- Assumptions
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size $C \ll n$ (much smaller than n)

- First iteration
  - $n/8 + n = 9n/8$ misses



Cache Miss Analysis

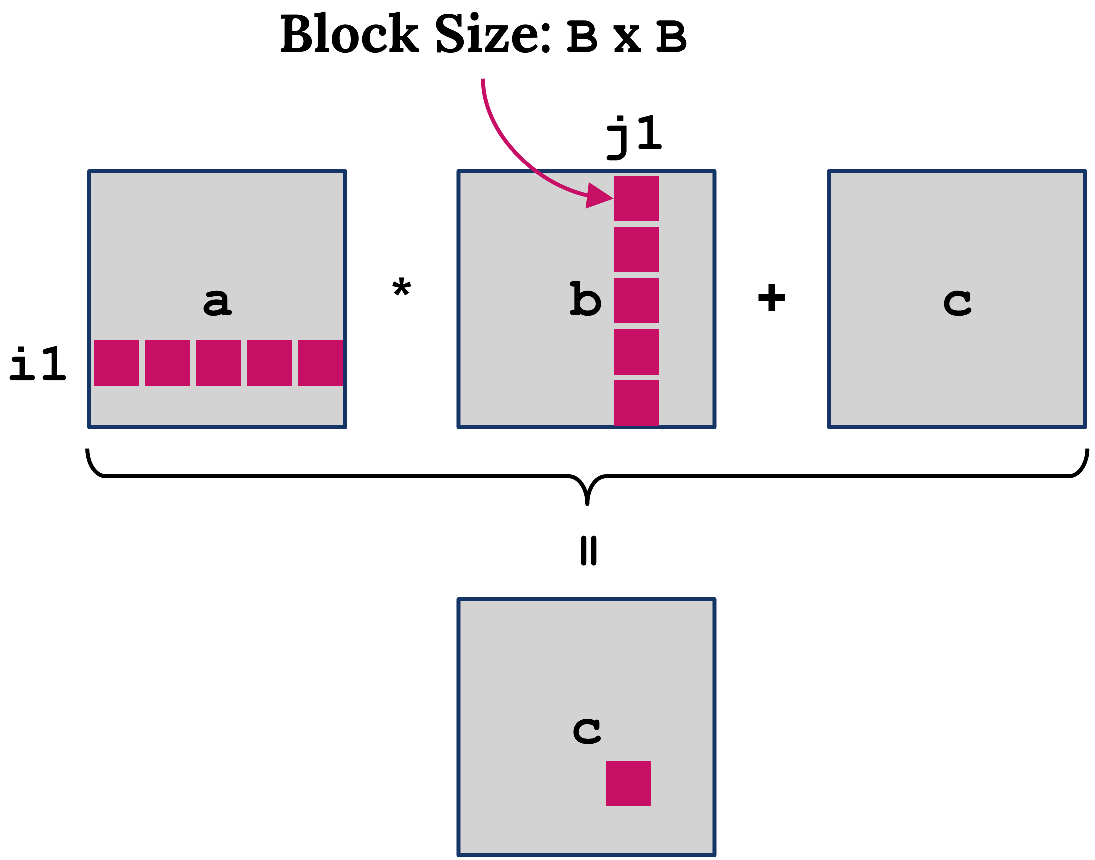
- Assumptions
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size $C \ll n$ (much smaller than n)
- First iteration
  - $n/8 + n = 9n/8$ misses
- Second iteration
  - Same as the first iteration
- Total misses
  - $9n/8 \times n^2 = 9/8 \times n^3$



Matrix Multiplication w/ Blocking

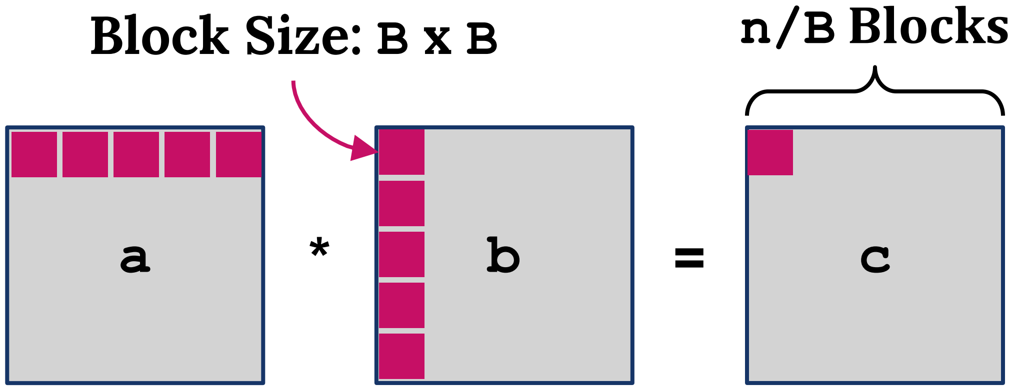
```
c = (double *) calloc(sizeof(double), n * n);

/* 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];
}
```

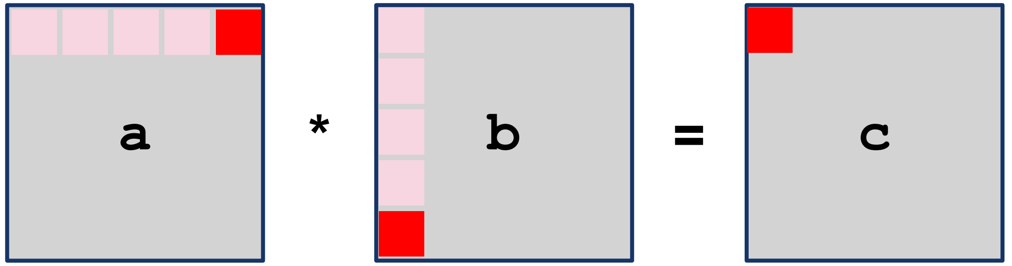


Cache Miss Analysis

- Assumptions
  - Cache block = 8 doubles
  - Cache size $C \ll n$ (much smaller than n)
  - Three blocks ■ fit into cache: $3B^2 < C$
- First (block) iteration
  - $B^2/8$ misses for each block
  - $2n/B \times B^2/8 = nB/4$ (omitting matrix `c`)

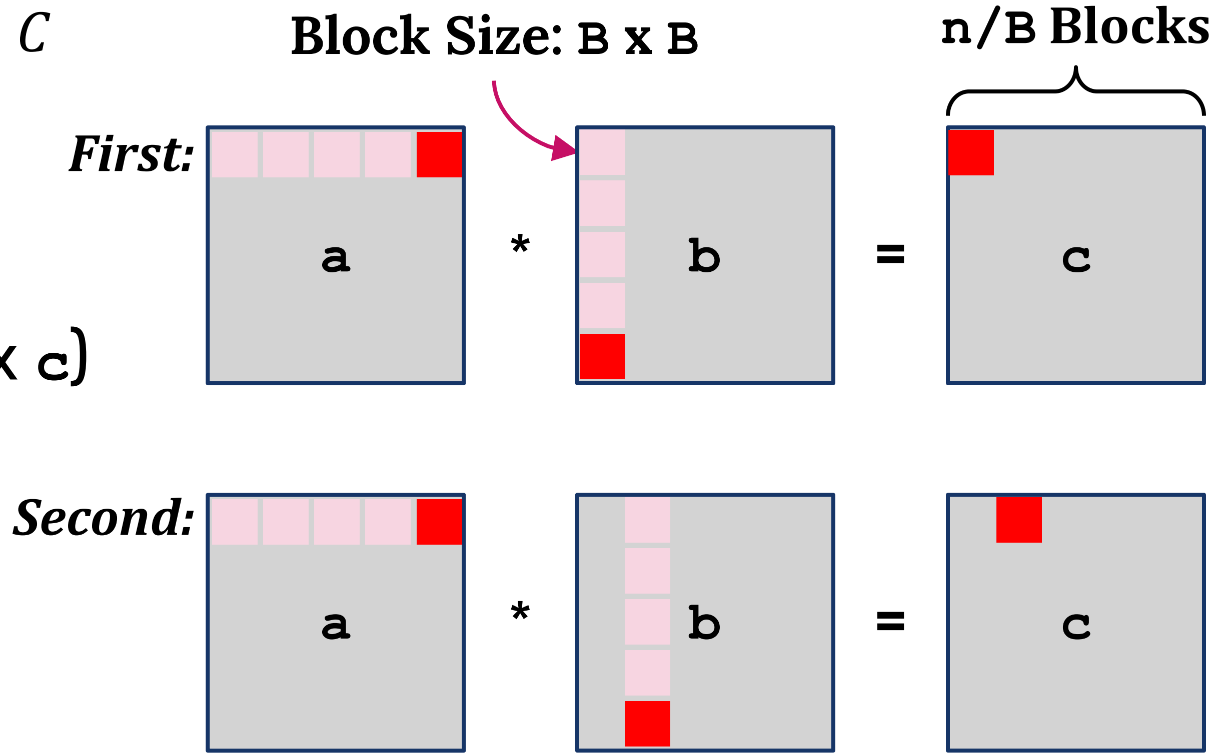


*Afterwards **in cache***



Cache Miss Analysis

- Assumptions
  - Cache block = 8 doubles
  - Cache size $C \ll n$ (much smaller than n)
  - Three blocks ■ fit into cache: $3B^2 < C$
- First (block) iteration
  - $B^2/8$ misses for each block
  - $2n/B \times B^2/8 = nB/4$ (omitting matrix `c`)
- Second (block) iteration
  - Same as the first iteration
- Total misses
  - $nB/4 \times (n/B)^2 = n^3/4B$



Blocking Summary

- No blocking: $9/8 \times n^3$
- Blocking: $(1/4B) \times n^3$
- Make the block size B as large as possible, but do not violate $3B^2 < C$ condition
- Reason for dramatic difference
  - Matrix multiplication has inherent temporal locality
    - Input data: $3n^2$, computation: $2n^3$
    - Every array elements are used $O(n)$ times
  - But program needs to be written properly

Programming Rules for Part B

- Your `trans.c` file must compile without warnings to receive credit

- You are not allowed to:
  1. Have more than 12 local variables on the stack at any time
     - Define at most 12 local variables per transpose function
     - You should also consider the case of helper functions
  2. Side-step rule #1 by using any tricks to store more values in a single variable
     - e.g., using variables of type `long`
  3. Define any arrays or any variant of `malloc` (e.g., linked list)
  4. Modify array `A` (modifying array `B` is fine)
  5. Use recursion

Cache Lab: Submission Guideline

- Due: 11/27 (Wed) 23:59 (Late submission will not be accepted)

- Submit code files and your lab report (in pdf)
  - Source code name: `[student id]_csim.c`, `[student id]_trans.c`
    e.g., `20242057_csim.c`, `20242057_trans.c`
    (Other formats will not be accepted)
  - Report name: `[student id].pdf` (e.g., `20242057.pdf`)
  - A correct submission is total three files

Cache Lab: Report Guideline

- Report
  - Attach the important parts of your code to your report
  - Explain how you built your cache simulator and optimized your matrix transpose
  - Report should not exceed 10 pages and use font Arial and font size 11pt
  - Include all references you refer to solve cache lab assignment in your report

Cheating Policy

- You can refer to
  - Cache lab writeup, lab slides, and lecture slides
  - Internet sources that do not include answers or code related to the cache lab
    - e.g., Valgrind manual

- You must not refer to
  - ChatGPT with direct query for answers or parts of a solution
  - Code and reports from seniors who have already taken this course
  - Blogs or github repositories that contain solution codes (`csim.c`, `trans.c`)

Quiz

- Go to PLMS, start the quiz
  - For fairness, quiz will be shut down after everyone leaves the classroom

[CSED211] Introduction to Computer Software Systems

Lab 6: Cache Lab

Dowon Son

COMPUTER ARCHITECTURE & OPERATING SYSTEMS LABORATORY

2024.11.14

Midterm Exam Claim

- We will do claim and quiz at the same time
  - Modifying your answer paper during the claim will be considered cheating

- Only 10 students will do claim at the same time
  - Please check the grading criteria and think which problems to claim for others
  - From 7:45, we will prioritize the students who have the schedule after 8:00