18-600 Recitation #10

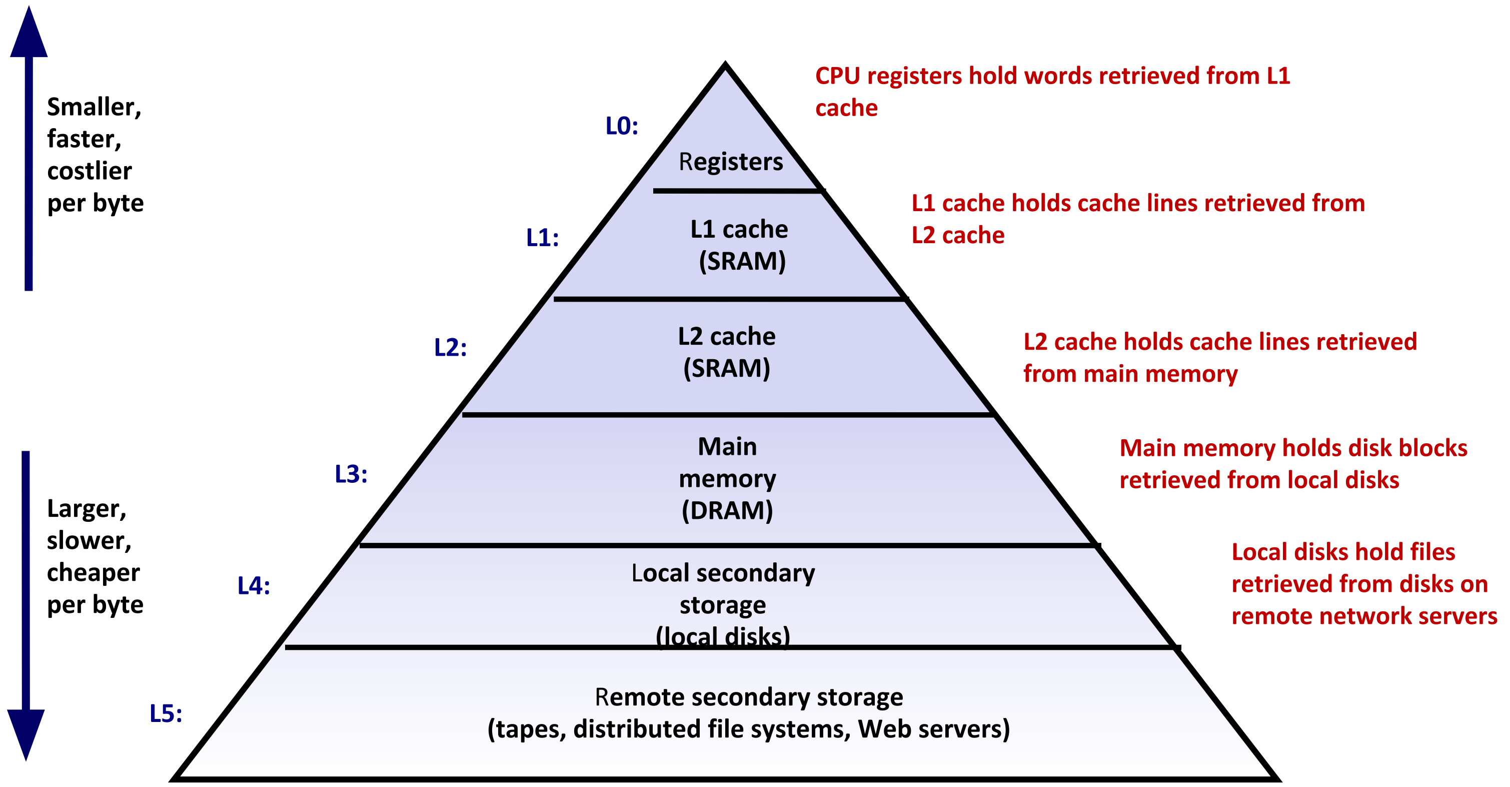
Cache Lab

October 31, 2017

Outline

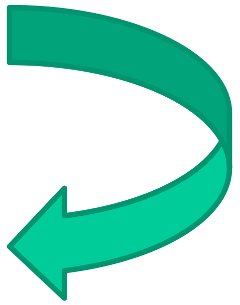
- **Memory organization**
- **Caching**
  - Different types of locality
  - Cache organization
- **Cache lab**
  - Part A Building Cache Simulator
  - Part B Building Cache Simulator for Multi-Core (MSI)
  - Part C Efficient Matrix Transpose
  - Blocking

Memory Hierarchy


Smaller, faster, costlier per byte
Larger, slower, cheaper per byte
L0:
Registers
CPU registers hold words retrieved from L1 cache
L1:
L1 cache (SRAM)
L1 cache holds cache lines retrieved from L2 cache
L2:
L2 cache (SRAM)
L2 cache holds cache lines retrieved from main memory
L3:
Main memory (DRAM)
Main memory holds disk blocks retrieved from local disks
L4:
Local secondary storage (local disks)
Local disks hold files retrieved from disks on remote network servers
L5:
Remote secondary storage (tapes, distributed file systems, Web servers)

Memory Hierarchy

- Registers
- SRAM
- DRAM
- Local Secondary storage
- Remote Secondary storage



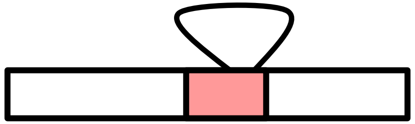
We will discuss this interaction

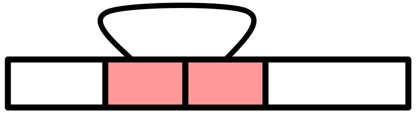
SRAM vs DRAM tradeoff

- **SRAM (cache)**
  - Faster (L1 cache: 1 CPU cycle)
  - Smaller (Kilobytes (L1) or Megabytes (L2))
  - More expensive and “energy-hungry”
- **DRAM (main memory)**
  - Relatively slower (hundreds of CPU cycles)
  - Larger (Gigabytes)
  - Cheaper

Locality

- **Temporal locality**
  - Recently referenced items are likely to be referenced again in the near future
  - After accessing address X in memory, save the bytes in cache for future access





- **Spatial locality**
  - Items with nearby addresses tend to be referenced close together in time
  - After accessing address X, save the block of memory around X in cache for future access

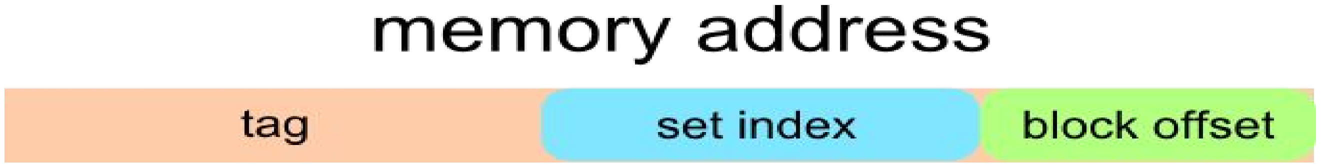
Find the type of locality?

```
for (int i = 0; i < size - 1; i++) {
    arr[i] = arr[i+1];
}
```

A.	Spatial
B.	Temporal
[✓ C.]	Both A & B
D.	Neither

Memory Address

- 64-bit on shark machines



- Block offset: b bits
- Set index: s bits
- Tag Bits: (Address Size – b – s)

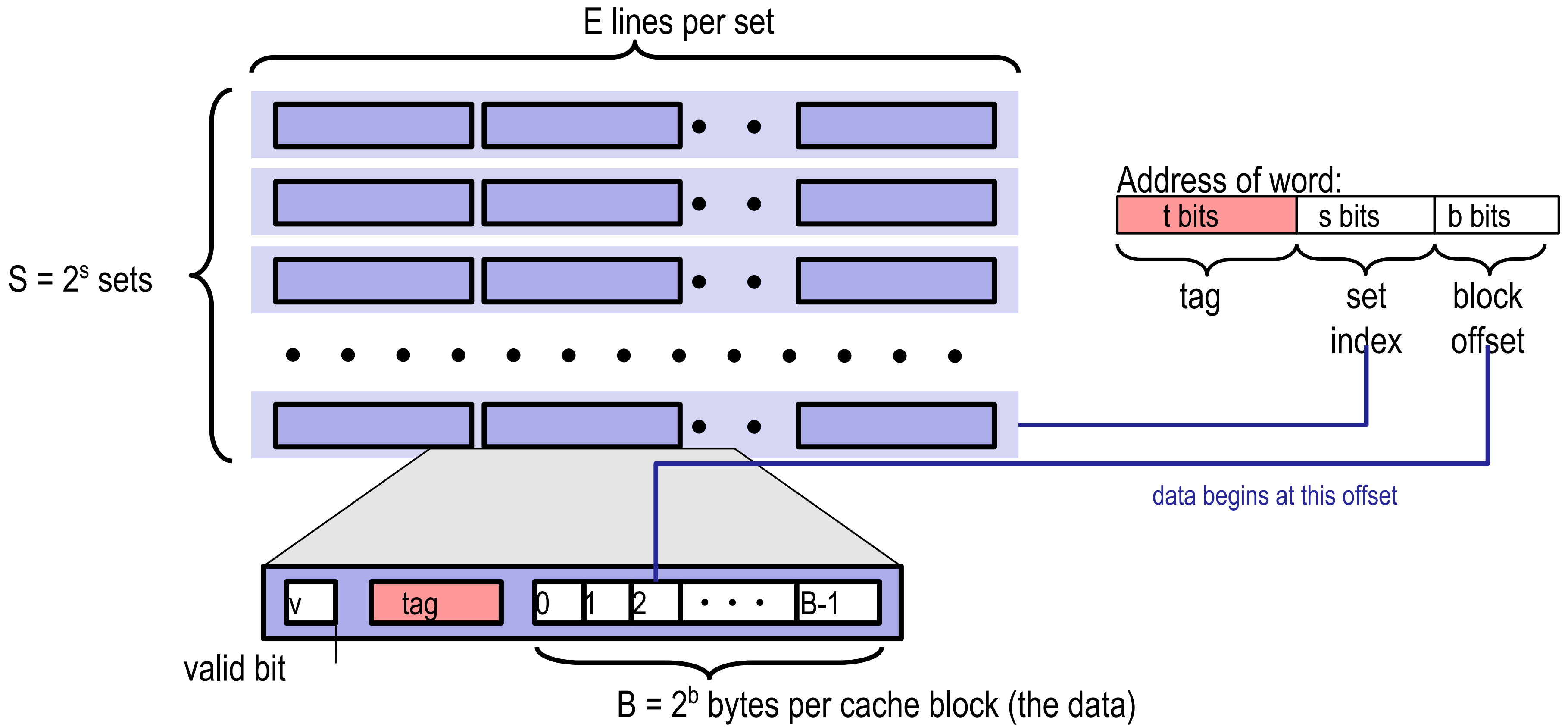
Cache

- **A cache is a set of S = 2^s cache sets**

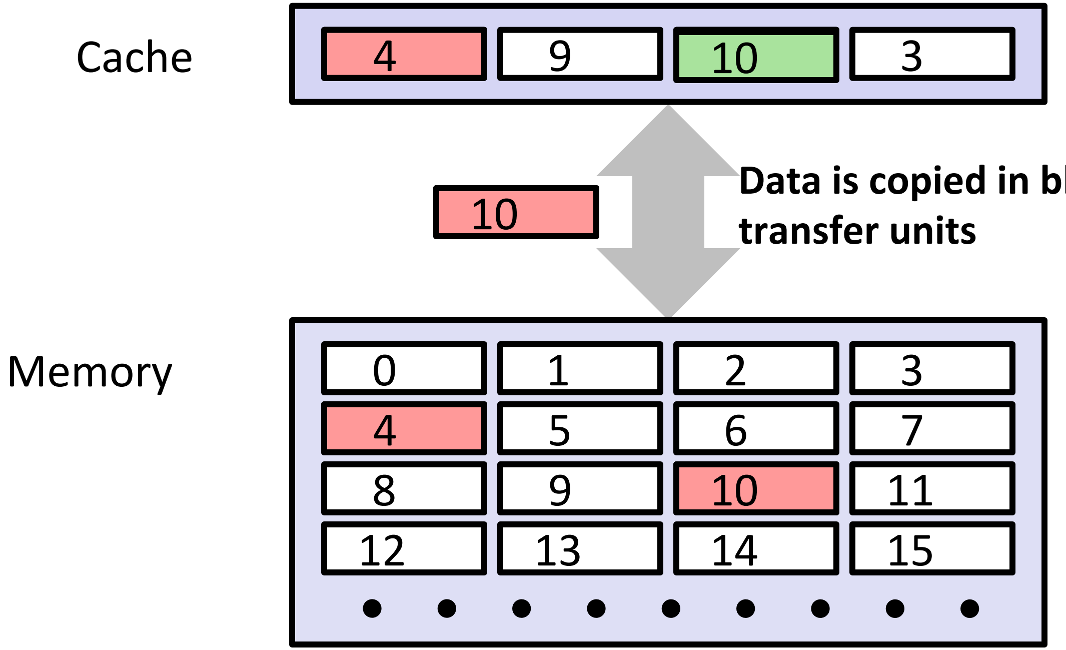
- **A cache set is a set of E cache lines**
  - E is called associativity
  - If E=1, it is called “direct-mapped”
- **Each cache line stores a block of size B = 2^b bytes**

- **Total Capacity = S*B*E**

Visual Cache Terminology



General Cache Concepts

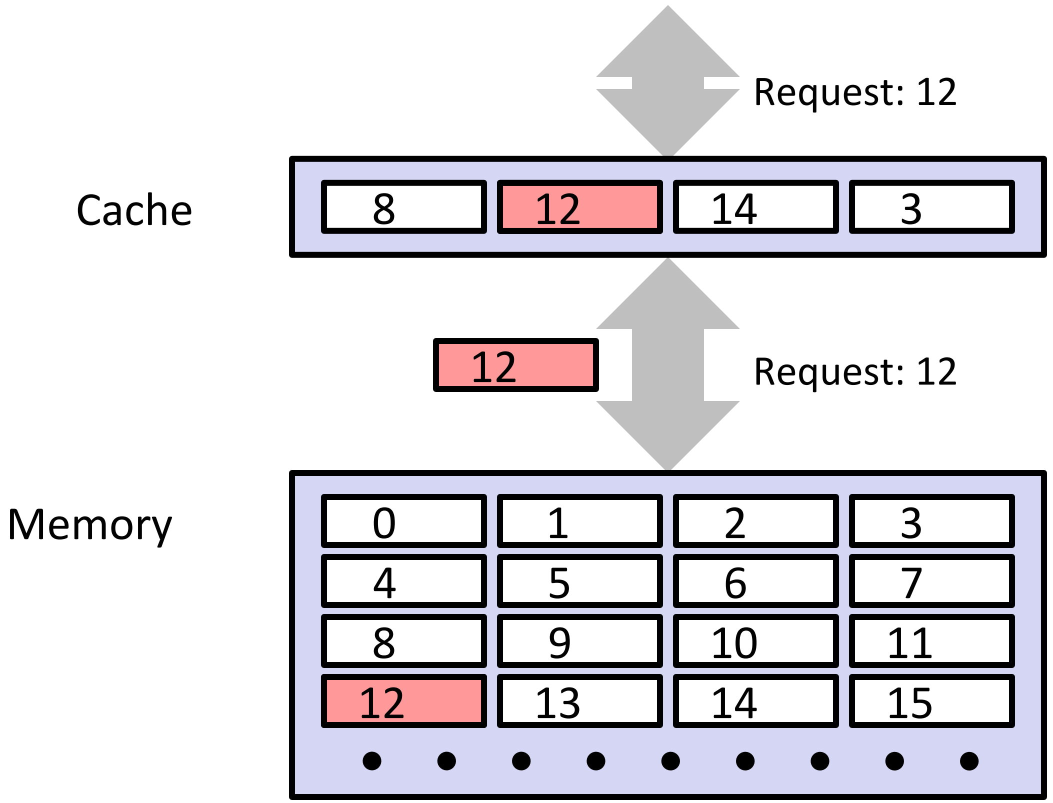


Smaller, faster, more expensive memory caches a subset of the blocks

Data is copied in block-sized transfer units

Larger, slower, cheaper memory viewed as partitioned into "blocks"

General Cache Concepts: Miss



Data in block b is needed

Block b is not in cache: Miss!

***Block b is fetched from** memory*

Block b is stored in cache

- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (victim)

Types of Cache Misses

- **Cold (compulsory) miss**
  - The first access to a block has to be a miss
- **Conflict miss**
  - Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block
    - E.g., Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time
  - You are required to minimize this in cache lab
- **Capacity miss**
  - Occurs when the set of active cache blocks (**working set**) is larger than the cache

Cache Lab

- Part A Building Cache Simulator
- Part B Building Cache Simulator for Multi-Core (MSI)
- Part C Efficient Matrix Transpose

Part A : Cache simulator

- **A cache simulator is NOT a cache!**
  - Memory contents NOT stored.
  - Block offsets are NOT used – the b bits in your address don't matter.
  - Simply **count** hits, misses, and evictions.
- **Your cache simulator needs to work for different s, b, E, given at run time.**
- **Use LRU – Least Recently Used replacement policy**
  - Evict the least recently used block from the cache to make room for the next block.
  - Counters? Queues ? Time Stamps?

Part A : Hints

- **A cache is just 2D array of *cache lines*:**
  - struct cache_line_t cache[S][E];
  - S = 2^s, is the number of sets
  - E is associativity

- **Each cache_line has:**
  - Valid bit
  - Tag
  - LRU counter (only if you are not using a queue)

Part A : Malloc/free

- **Use malloc to allocate memory on the heap.**

- **Always free what you malloc, otherwise may get memory leak**
    - some_pointer_you_malloced = malloc(sizeof(int));
    - free(some_pointer_you_malloced);

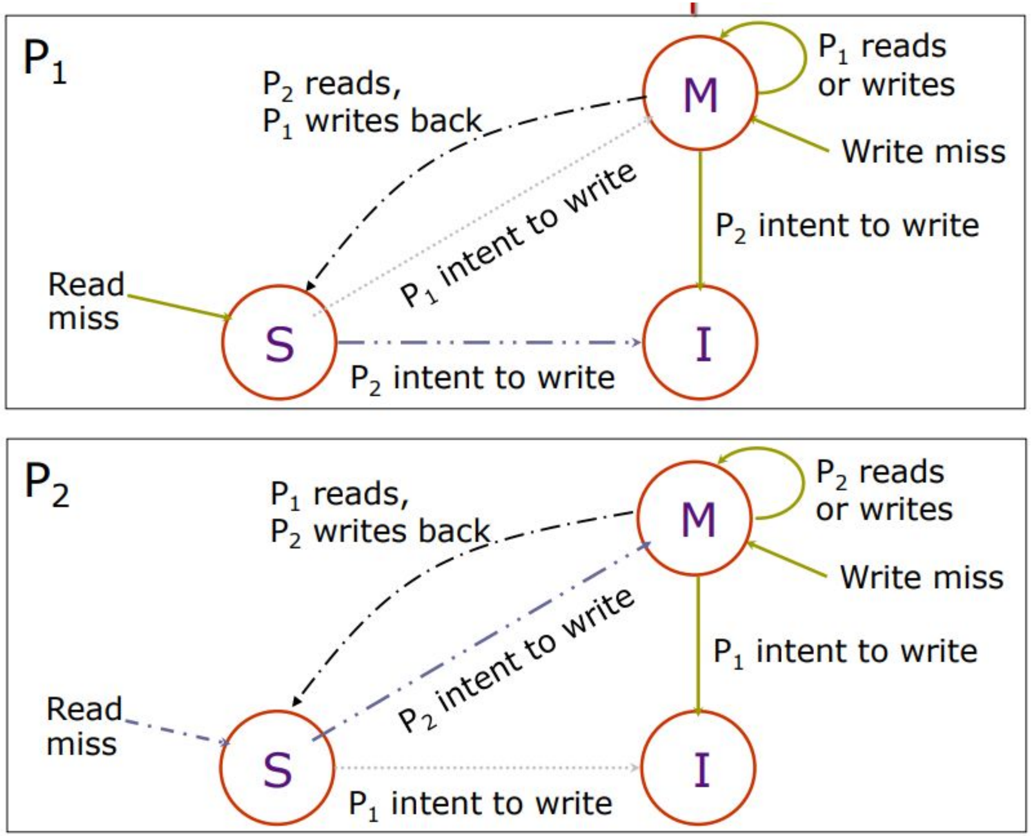
- **Don’t free memory you didn’t allocate.**

Part B : MSI

- Each core has it’s own cache.

- Each core needs to communicate with other cores for every access (snooping).

Part B : MSI



Key point - Notice the changes in the state when same memory locations are accessed or written to by other cores.

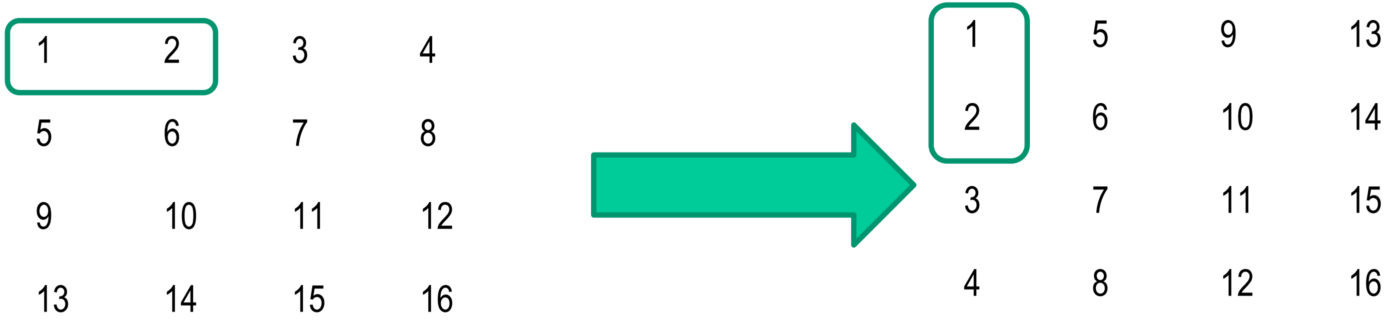
Code Resusability

- Look into cache.h, csim.h, and msim.h

- A lot of code can be reused for Part A and Part B

Part C: Efficient Matrix Transpose

- **Matrix Transpose (A -> B)**



$$A = \begin{matrix} \boxed{1 \quad 2} & 3 & 4 \\ 5 \quad 6 & 7 & 8 \\ 9 \quad 10 & 11 & 12 \\ 13 \quad 14 & 15 & 16 \end{matrix} \Longrightarrow B = \begin{matrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{matrix}$$

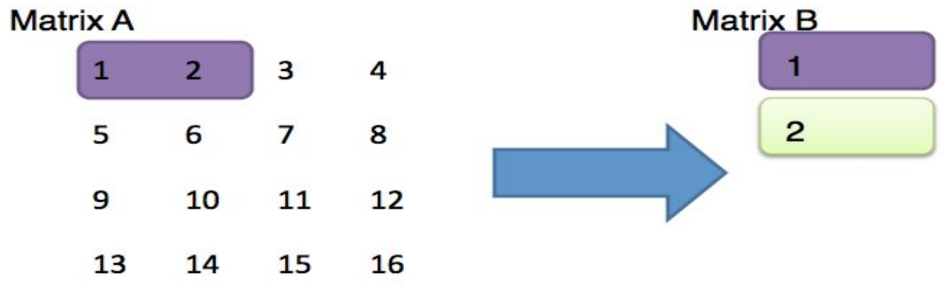
- **How do we optimize this operation for the cache?**

Part C : Blocking

- **Blocking: divide matrix into submatrices.**
- **Size of sub-matrix depends on cache block size, cache size, input matrix size.**
- **Try different sub-matrix sizes.**

Part (b) : Efficient Matrix Transpose

- Suppose Block size is 8 bytes, S is large, and E = 2

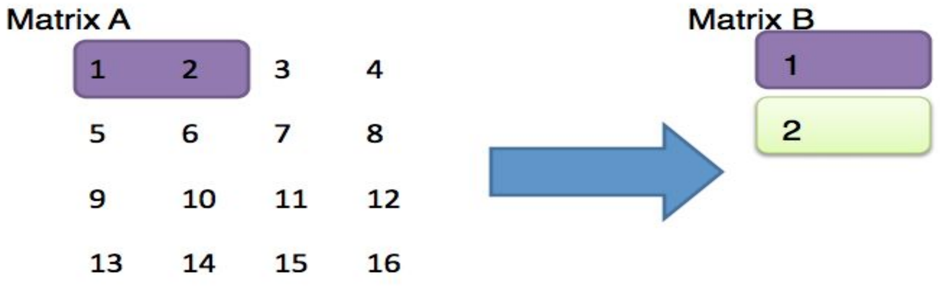


Should we handle 3 & 4 next or 5 & 6 ?

- Access A[0][0] cache miss
- Access B[0][0] cache miss
- Access A[0][1] cache hit
- Access B[1][0] cache miss
- Access A[1][0] cache miss
- Access B[0][1] cache hit
- Access A[1][1] cache hit
- Access B[1][1] cache hit

Part (b) : Efficient Matrix Transpose

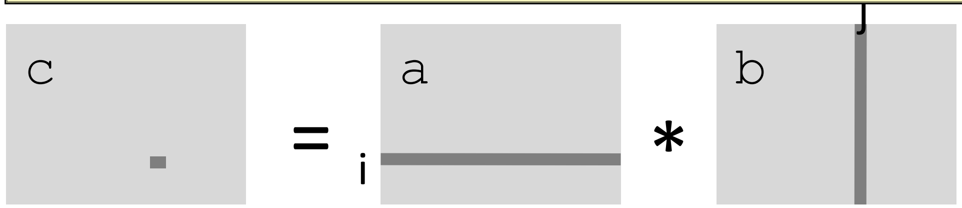
- Now lets try for E = 1 (direct mapped), assume matrices are aligned such that index 1 from both maps to same set



- Access A[0][0] cache miss
- Access B[0][0] cache miss evict
- Access A[0][1] cache miss evict
- Access B[1][0] cache miss
- Access A[1][0] cache miss evict
- Access B[0][1] cache miss evict
- Access A[1][1] cache hit
- Access B[1][1] cache miss evict

Example: Matrix Multiplication

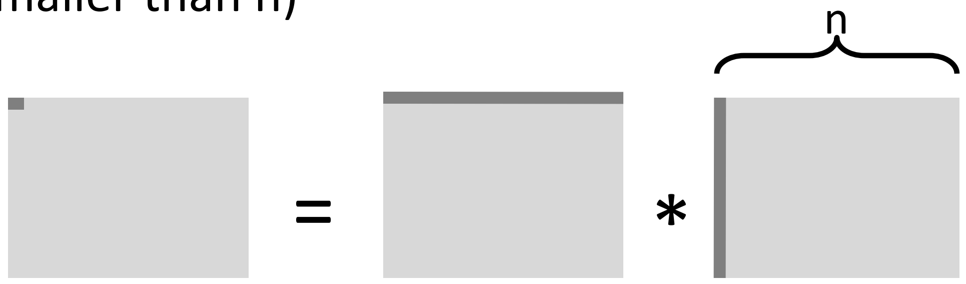
```
c = (double *) calloc(sizeof(double), n*n);

/* 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];
}
```



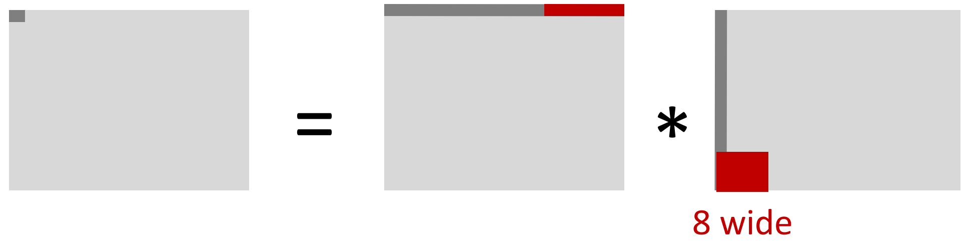
Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles, size of array n is arbitrary
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

- **First iteration:**
  - n/8 + n = 9n/8 misses

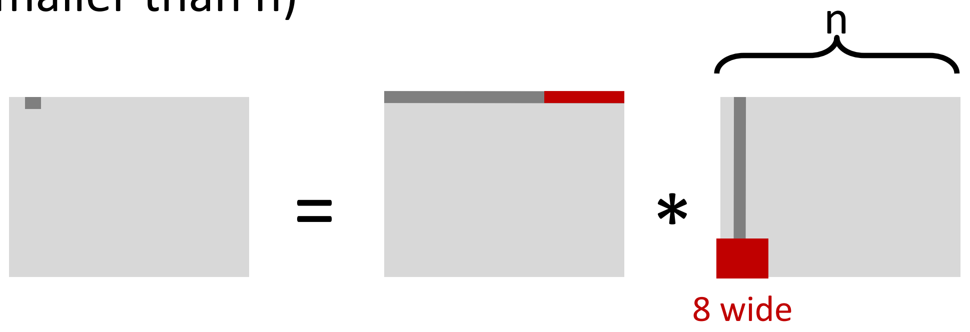


  - Afterwards in cache: (schematic)



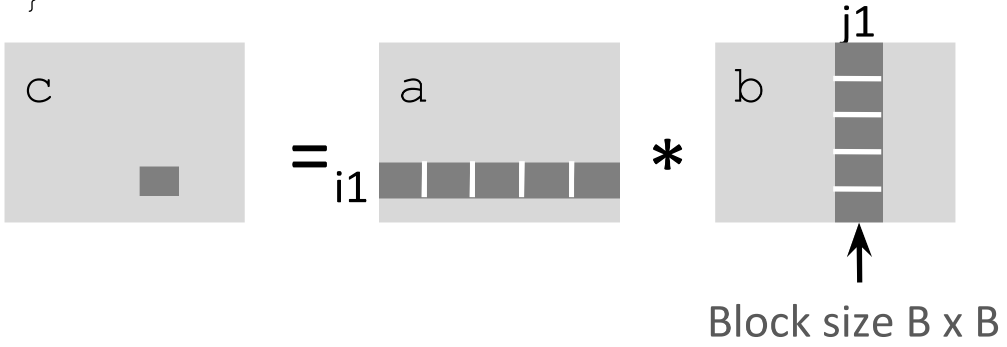
Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
- **Second iteration:**
  - Again:
    n/8 + n = 9n/8 misses
- **Total misses:**
  - $9n/8 * n^2 = (9/8) * n^3$



Blocked Matrix Multiplication

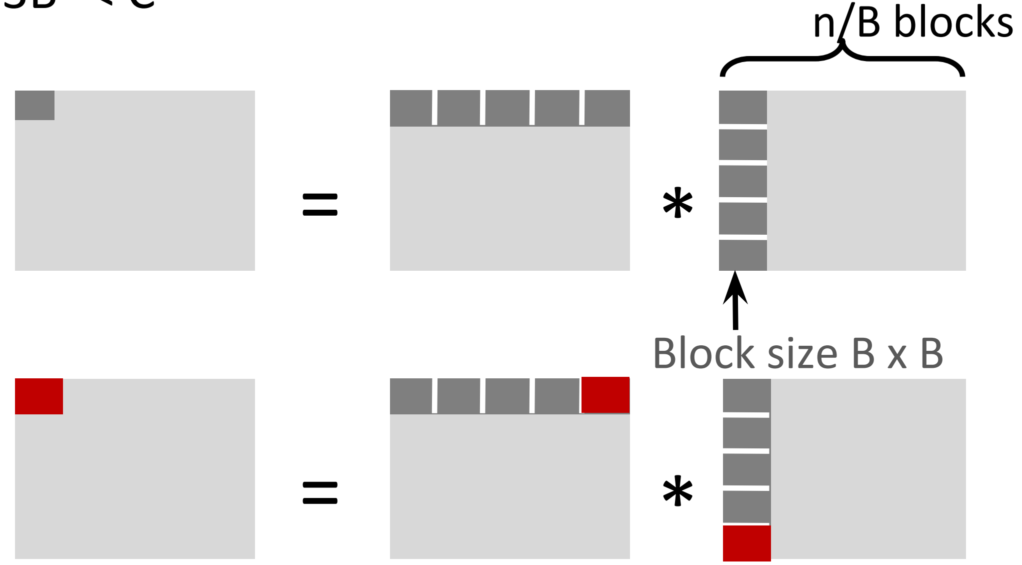
```
c = (double *) calloc(sizeof(double), n*n);

/* 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; i1++)
                      for (j1 = j; j1 < j+B; j1++)
                          for (k1 = k; k1 < k+B; k1++)
                           c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```



Cache Miss Analysis

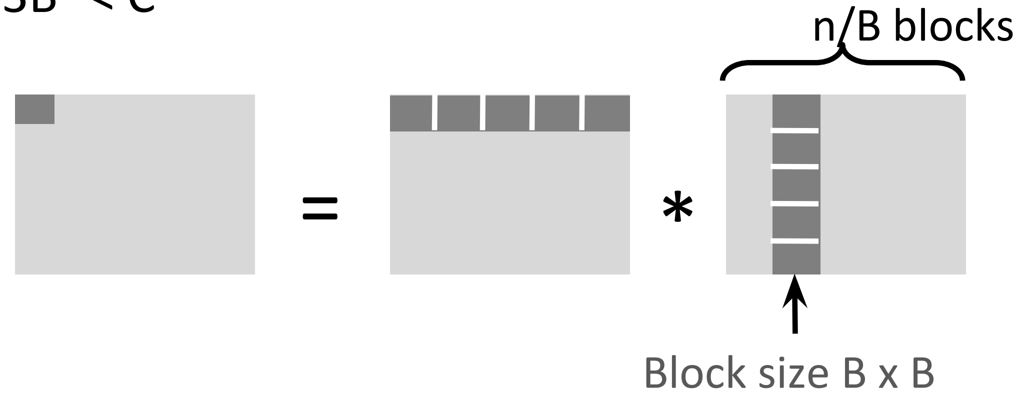
- **Assume:**
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks fit into cache: $3B^2 < C$

- **First (block) iteration:**
  - $B^2/8$ misses for each block
  - $2n/B * B^2/8 = nB/4$ (omitting matrix c)
  - Afterwards in cache (schematic)



Cache Miss Analysis

- **Assume:**
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks fit into cache: $3B^2 < C$
- **Second (block) iteration:**
  - Same as first iteration
  - $2n/B * B^2/8 = nB/4$
- **Total misses:**
  - $nB/4 * (n/B)^2 = n^3/(4B)$



Part C : Blocking Summary

- **No blocking: (9/8) * n^3**
- **Blocking: 1/(4B) * n^3**
- **Suggest largest possible block size B, but limit $3B^2 < C$!**
- **Reason for dramatic difference:**
  - Matrix multiplication has inherent temporal locality:
    - Input data: $3n^2$, computation $2n^3$
    - Every array elements used O(n) times!
  - But program has to be written properly
- **For a detailed discussion of blocking:**
  - http://csapp.cs.cmu.edu/public/waside.html

Part C : Specs

- **Cache:**
  - You get 1 kilobytes of cache
  - Directly mapped (E=1)
  - Block size is 32 bytes (b=5)
  - There are 32 sets (s=5)
- **Test Matrices:**
  - 32 by 32
  - 64 by 64

Part C

- **Things you'll need to know:**
    - Warnings are errors
    - Header files
    - Eviction policies in the cache

Eviction policies of Cache

- **The first row of Matrix A evicts the first row of Matrix B**
  - Caches are memory aligned.
  - Matrix A and B are stored in memory at addresses such that both the first elements align to the same place in cache!
  - Diagonal elements evict each other.
- **Matrices are stored in memory in a row major order.**
  - If the entire matrix can’t fit in the cache, then after the cache is full with all the elements it can load. The next elements will evict the existing elements of the cache.
  - Example:- 4x4 Matrix of integers and a 32 byte cache.
    - The third row will evict the first row!

Style

- **Document your code**
  - Header comments
  - High-level description of big chunks of code
- **Check for failures**
- **Write modular code**
- **80 characters per line**
- **Consistent braces and whitespace**
- **No memory or file descriptor leaks**

Questions?