# Recitation: Cache Lab and Blocking

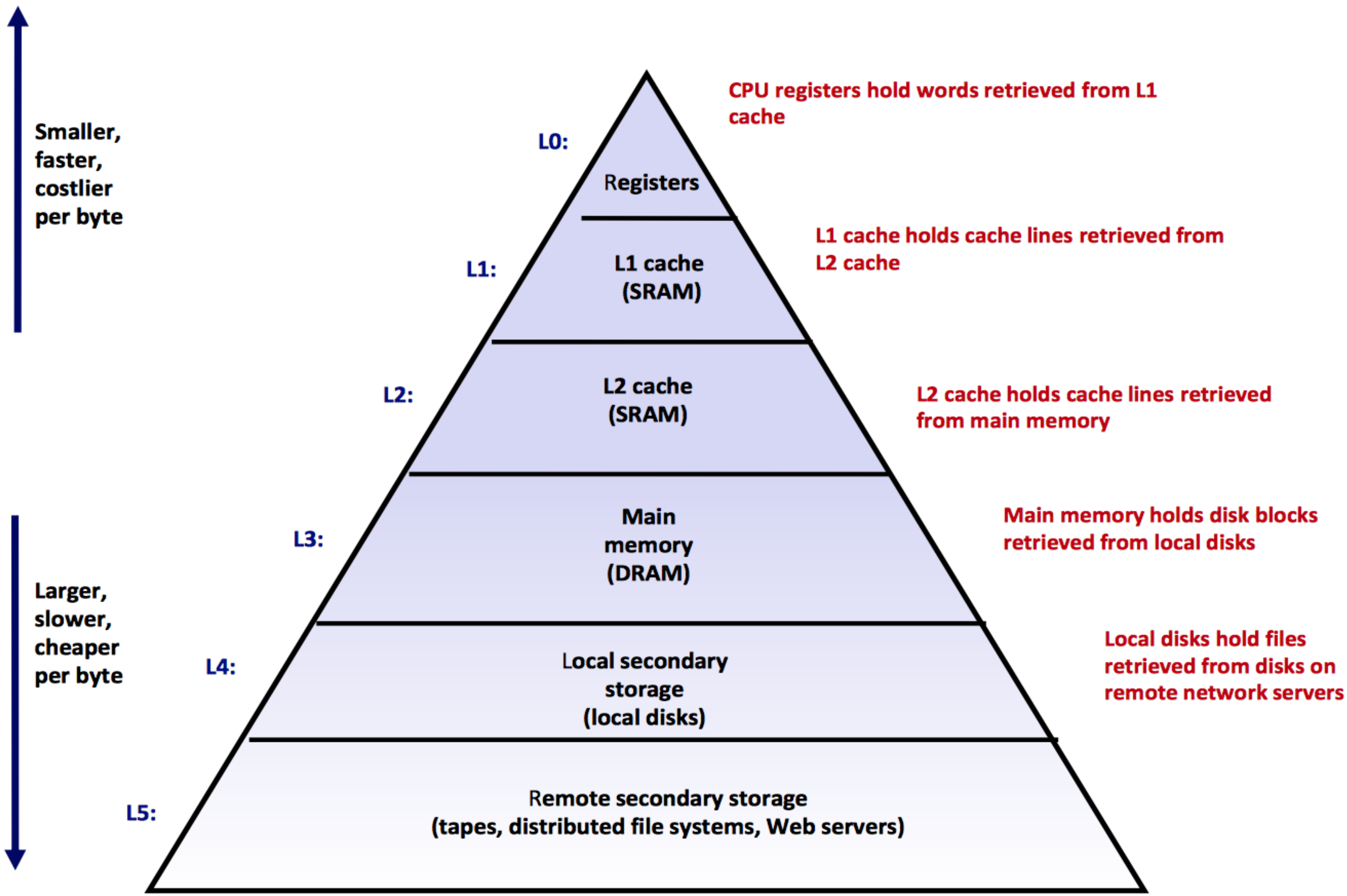
# IMPORTANT INFORMATION

- Due date for Cache Lab
- First lab with coding and points for styling.
- Start preparing for the mid-term exam(previous years question papers are a good place to start)

## TODAY'S AGENDA

- Memory Organization
- Cache Organization
- Locality in Caches
- Cache Lab

# Memory Hierarchy



# Memory Hierarchy

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

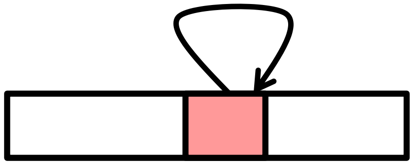
We will discuss this interaction

# SRAM vs DRAM

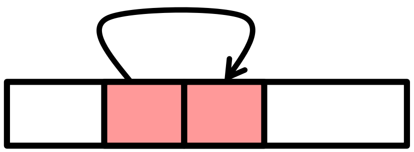
- SRAM (cache)
  - Faster (L1 cache: 1 CPU cycle)
  - Smaller (Kilobytes (L1) or Megabytes (L2))
  - More expensive and "energy-hungry"
  - Closer to processor
- DRAM (main memory)
  - Relatively slower (hundreds of CPU cycles)
  - Larger (Gigabytes)
  - Cheaper
  - Farther away from processor

# 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
  - Example: Accessing a variable over and over again for printing onto the screen

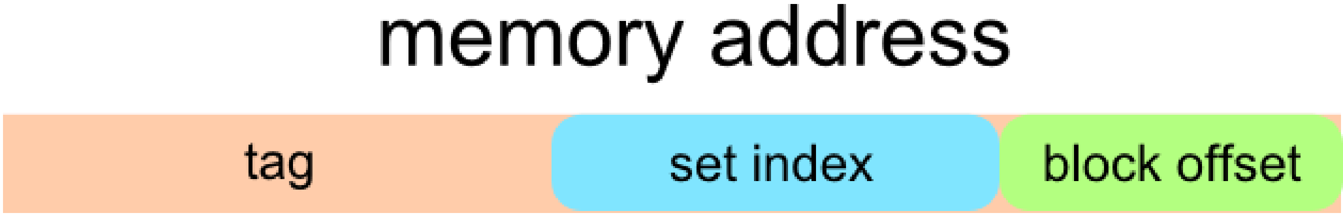


- Spatial locality
  - Items with nearby addresses tend to be referenced close together in time
  - After accessing address X, save the block of memory around X in cache for future access
  - Example: Array access(think how this is spatial locality)



# Memory Address

- Virtually or Physically addressed, the following is the format in which the address is broken down to get the information required to fetch a block of data from the cache




- Block offset: the least significant b bits
- Set index: s bits(follows the block bits)
- Tag Bits: Address Size – b – s (the most significant bits remaining after set bits and block bits)

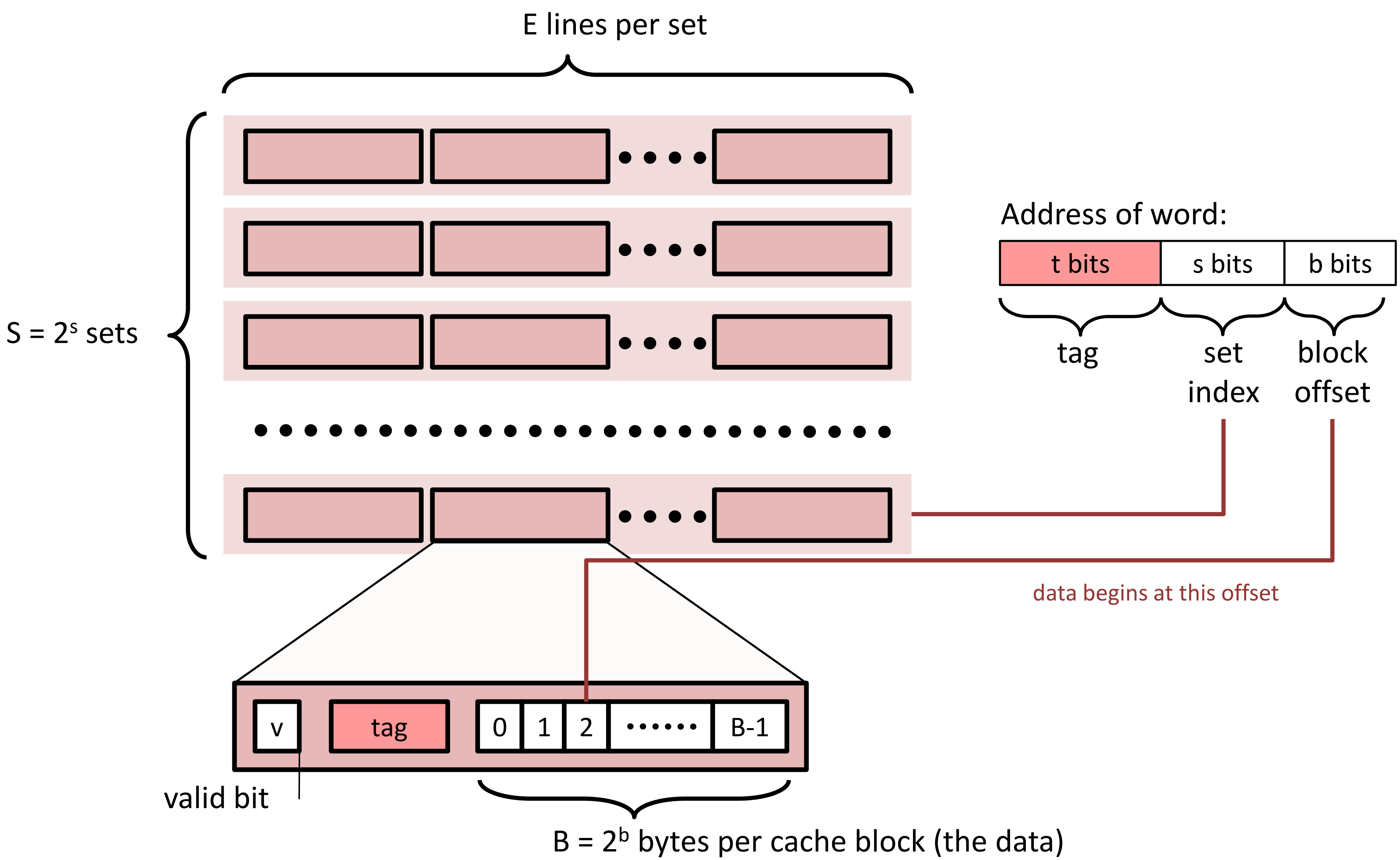
# Cache

- A cache is a set of 2^s cache sets(S=2^s)
- Where "S" is the number of sets and "s" is the number represented by the set bits.

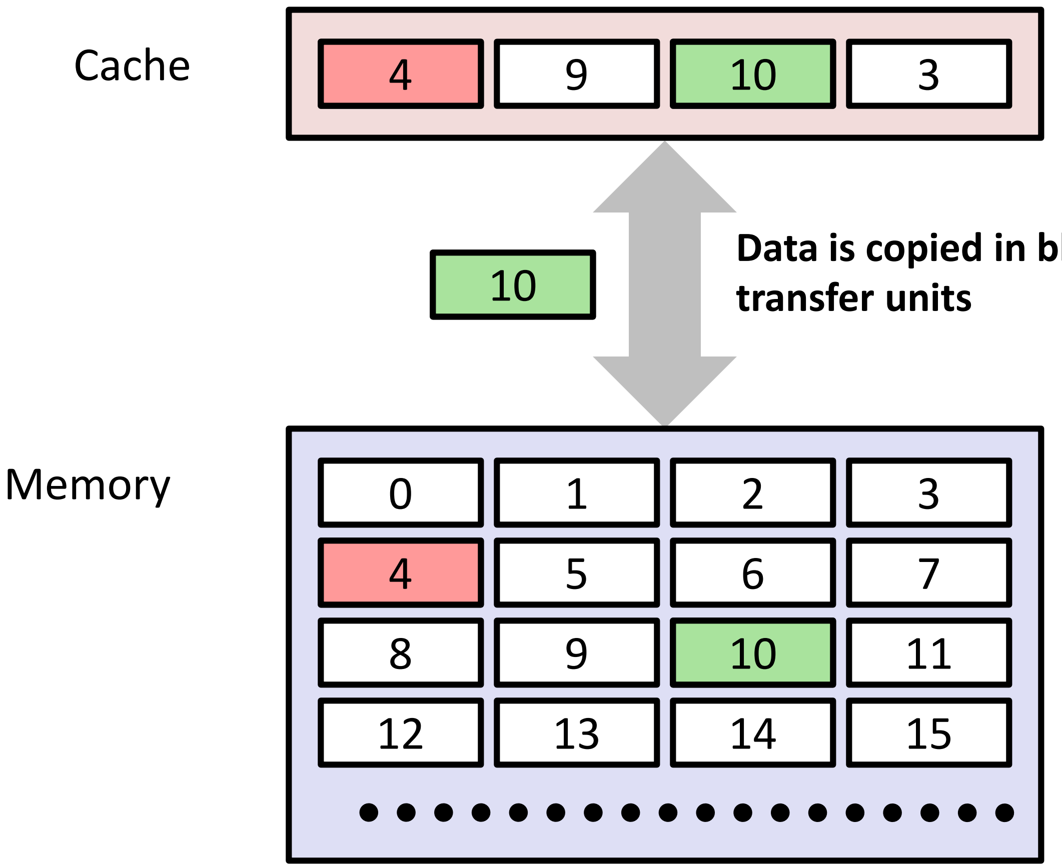
- A cache set is a set of E cache lines
  - E is called associativity
  - If E=1, it is called "direct-mapped"

- Each cache line stores a block
  - Each block has B = 2^b bytes
- Total Capacity = S*B*E

# Visualization



# General Cache Concepts

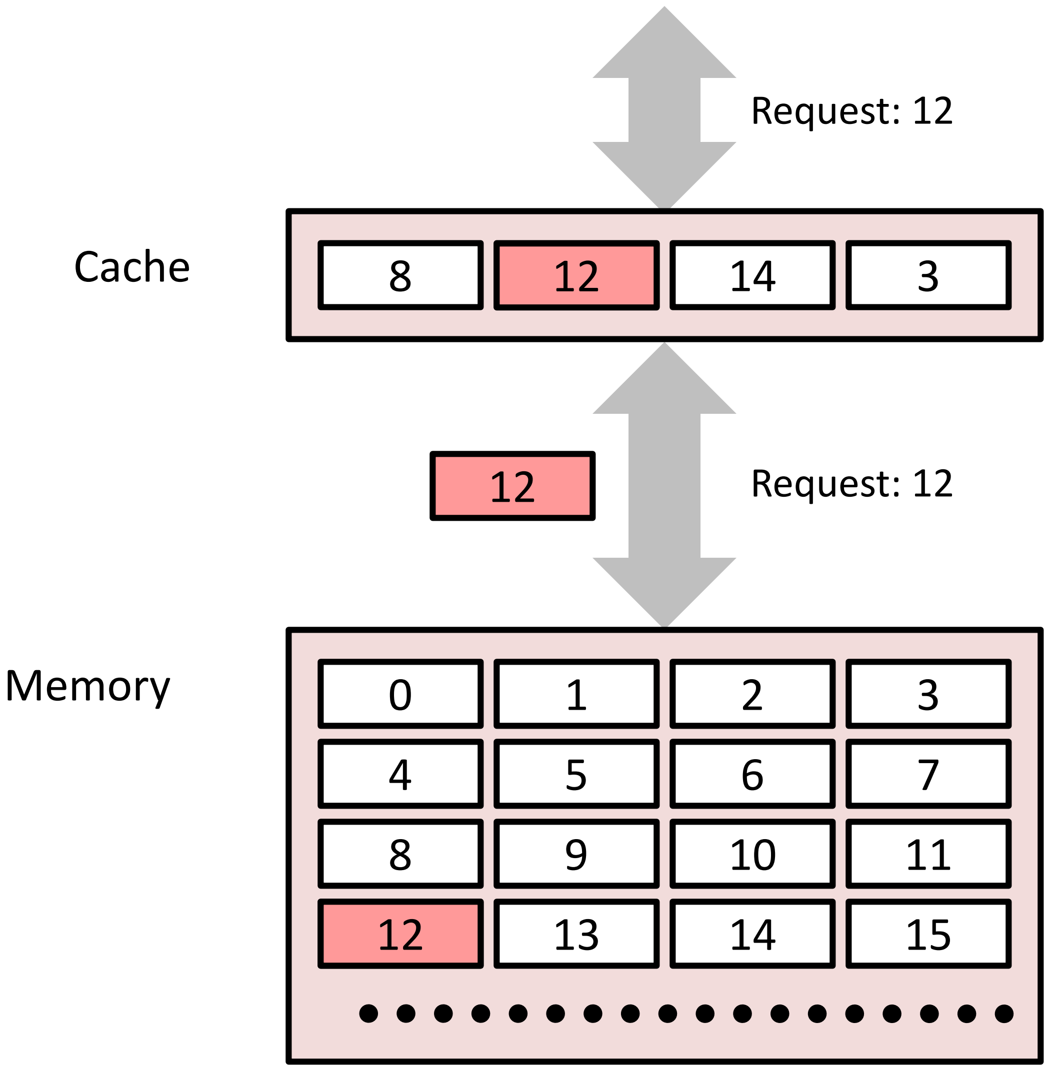



**Smaller, faster, more expensive memory caches a subset of the blocks**

**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)

# General Caching Concepts: Types of Cache Misses

- Cold (compulsory) miss
  - The first access to a block has to be a miss as the corresponding block would not have been cached yet.
- 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
- Capacity miss
  - Occurs when the set of active cache blocks (working set) is larger than the cache

# Cache Lab

- Part (a) Building a cache simulator:

  If you have not started this do it right away.

- Part (b) Optimizing matrix transpose

  This is where the concept of blocking comes into play

# 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
  - Basically use the meta-data to calculate the above parameters
- 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.
  - Pointer manipulations required for house keeping of these cache blocks.

# Getopt

- getopt() automates parsing elements on the unix command line If function declaration is missing
  - Typically called in a loop to retrieve arguments
  - Its return value is stored in a local variable
  - When getopt() returns -1, there are no more options

- To use getopt, your program must include the header file unistd.h

- If not running on the shark machines then you will need #include <getopt.h>.
  - Better Advice: Run on Shark Machines !

# Getopt

- A switch statement is used on the local variable holding the return value from getopt()
  - Each command line input case can be taken care of separately
  - “optarg” is an important variable – it will point to the value of the option argument
- Think about how to handle invalid inputs
- For more information,
  - look at man 3 getopt
  - http://www.gnu.org/software/libc/manual/html_node/Getopt.html

# Part (a) : getopt Example

```
int main(int argc, char** argv){
    int opt, x,y;
    /* looping over arguments */
    while(-1 != (opt = getopt(argc, argv, “x:y:"))){
        /* determine which argument it’s processing */
        switch(opt) {
            case 'x':
                x = atoi(optarg);
                break;
            case ‘y':
                y = atoi(optarg);
                break;
            default:
                printf(“wrong argument\n");
                break;
        }
    }
}
```

- Suppose the program executable was called “foo”. Then we would call “./foo -x 1 –y 3“ to pass the value 1 to variable x and 3 to y.

# 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 that you didn’t allocate
- Every allocated location is represented by a pointer, the meta-data for the allocated locations are managed by the memory allocator.

# Part (b) Matrix Transpose

- Matrix Transpose (A -> B)

Matrix A

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 |

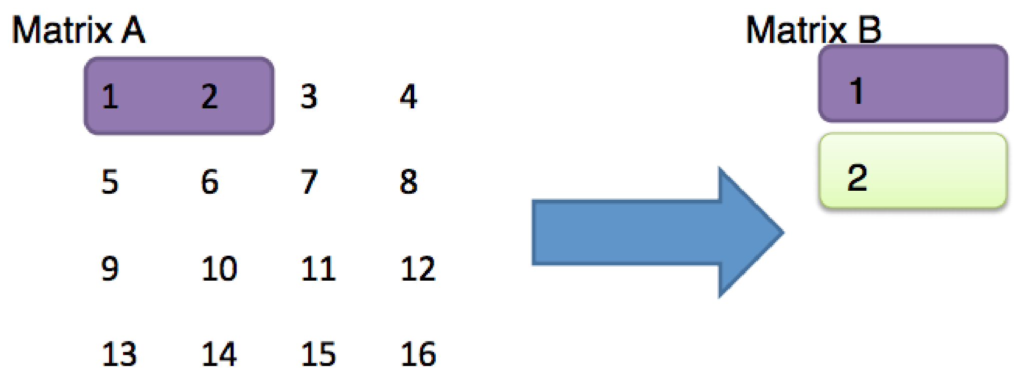
Matrix B

| | | | |
|---|---|---|---|
| 1 | 5 | 9 | 13 |
| 2 | 6 | 10 | 14 |
| 3 | 7 | 11 | 15 |
| 4 | 8 | 12 | 16 |

- How do we optimize this operation using the cache?
- Optimization in our case is to reduce the number of cache misses, while performing the matrix transpose.

# Part (b) : Efficient Matrix Transpose

- **Suppose Block size is 8 bytes ?**




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

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

# Part (b) : Blocking

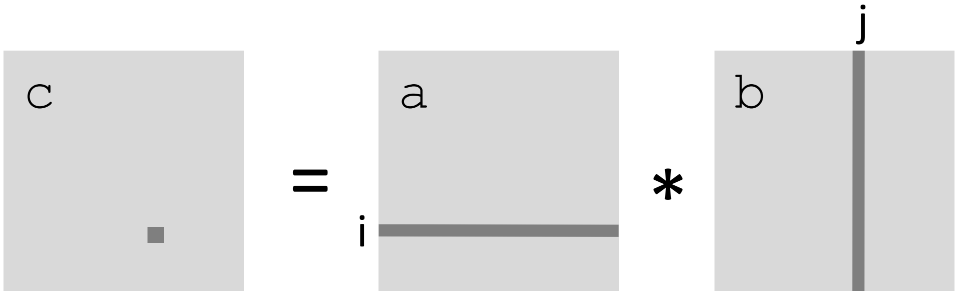
- Blocking: divide matrix into sub-matrices, such that it is feasible to have a row of the sub-matrix in a cache line, and access them such that locality factor is taken advantage of.

- Size of sub-matrix depends on cache block size, cache size, input matrix size.

- Try different sub-matrix sizes.

# 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];
}
```

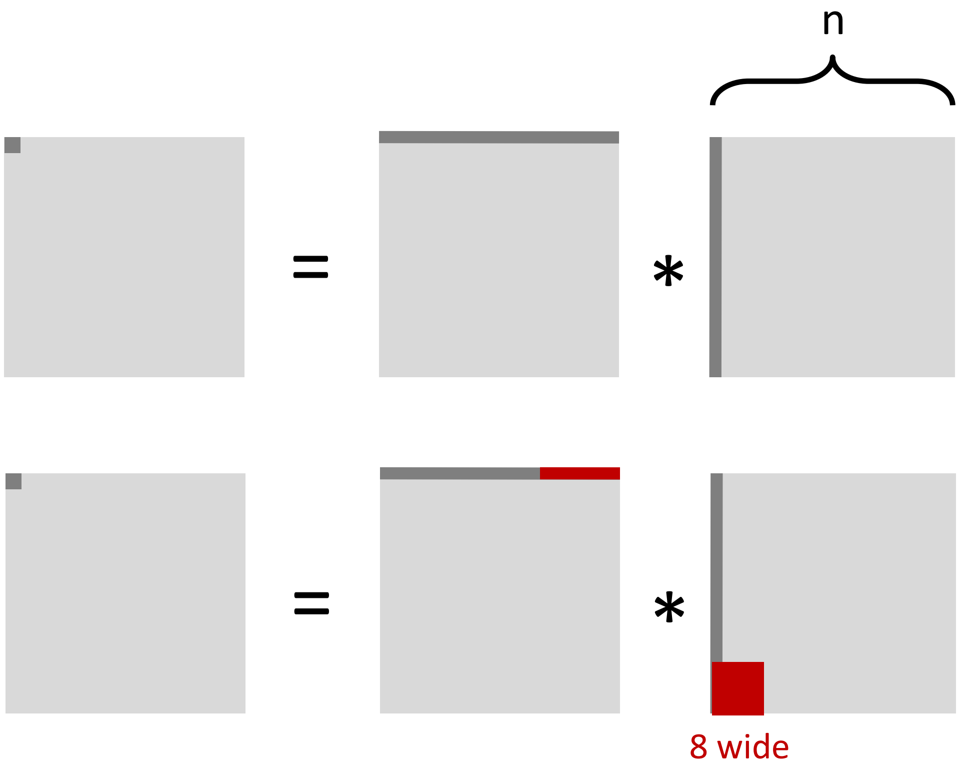


# Cache Miss Analysis

- Assume:
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

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

  - Afterwards in cache:
    (schematic)

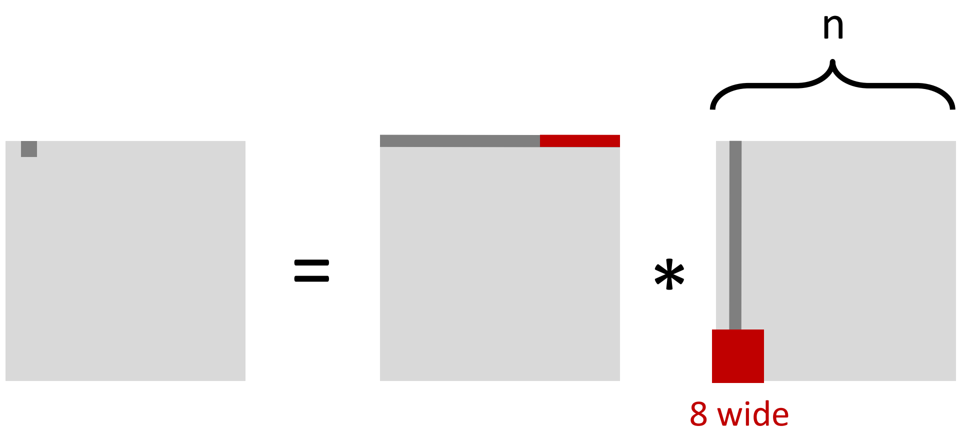


# Cache Miss Analysis

- Assume:
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

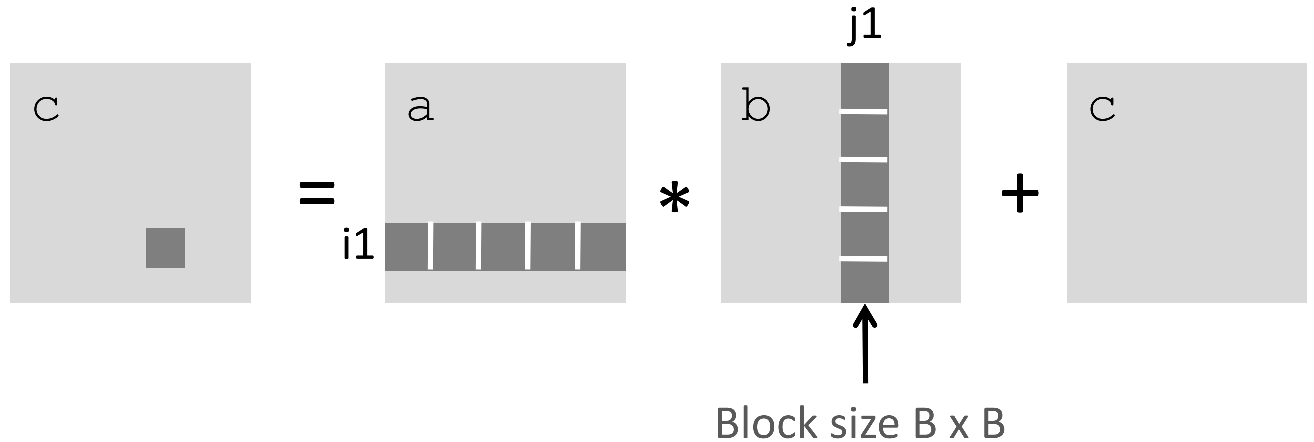
- Second iteration:
  - Again:
    n/8 + n = 9n/8 misses

- Total misses:
  - $9n/8 * n^2 = (9/8) * n^3$



# Blocked Matrix Multiplication

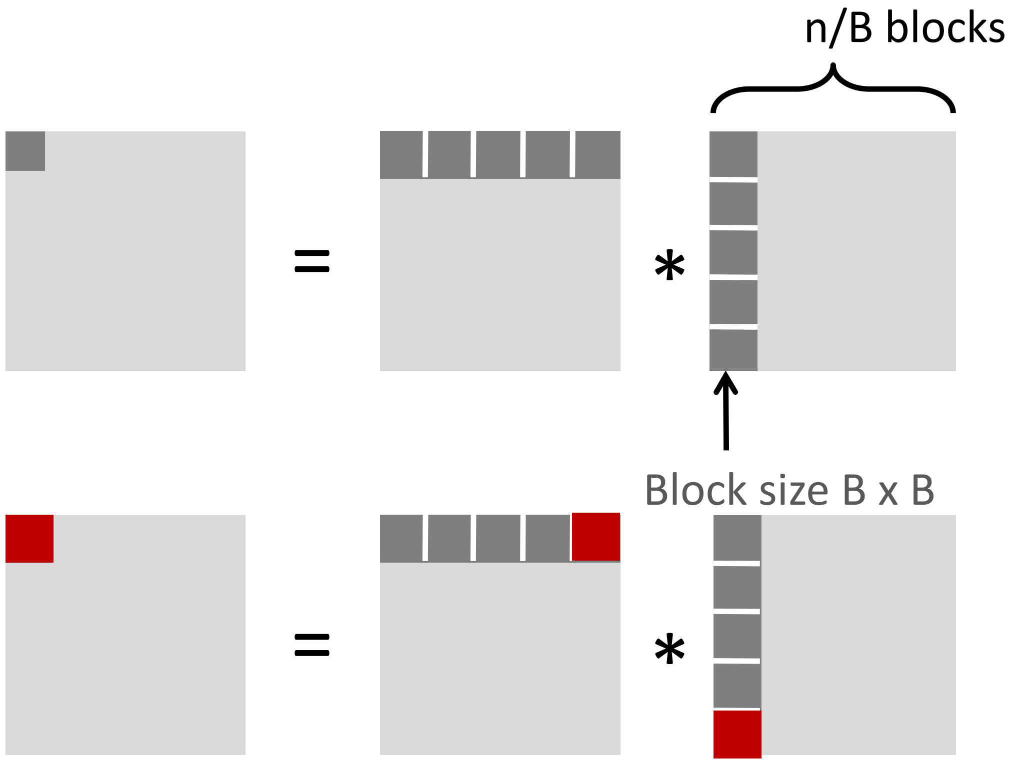
```
c = (double *) calloc(sizeof(double), n*n);

/* 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; 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

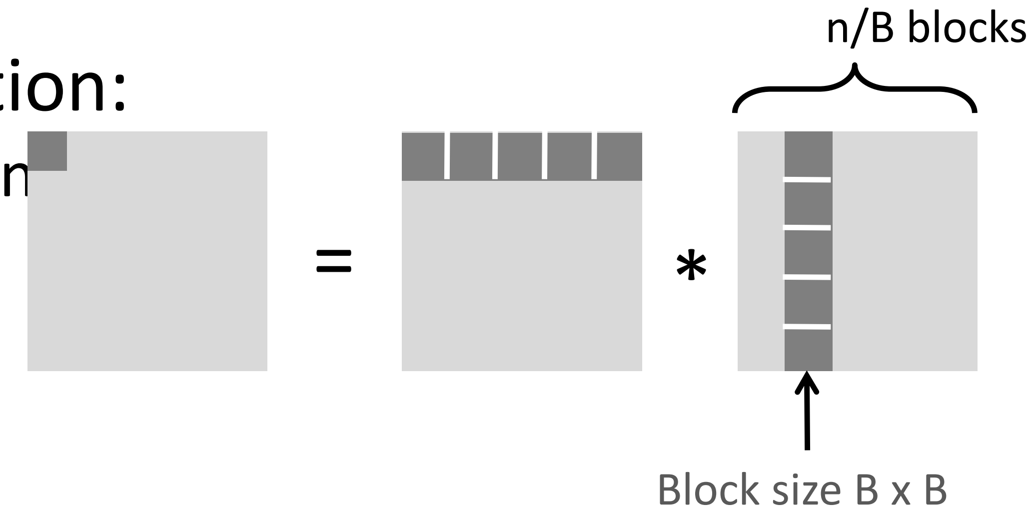
- Assume:
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks fit into cache: $3B^2 < C$

- 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(b) : 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 (b) : 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
  - 61 by 67

# Part (b)

- Things you'll need to know:
  - Warnings are errors
  - Header files
  - Useful functions

# Warnings are Errors

- Strict compilation flags
- Reasons:
  - Avoid potential errors that are hard to debug
  - Learn good habits from the beginning
- Add “-Werror” to your compilation flags

# Missing Header Files

- Remember to include files that we will be using functions from
- If function declaration is missing
  - Find corresponding header files
  - Use: man <function-name>
- Live example
  - man 3 getopt

# Style

- Read the style guideline
  - But I already read it!
  - Good, read it again.
  - Some important points- andrew id, Program description, function descriptions, freeing allocated memory, 80 character line limit.
  - There are many more of these in the style guideline.

- Pay special attention to Error checking
  - Functions don’t always work
  - What happens when a syscall fails??
  - Take a look at the wrappers provided in csapp.c
  - You are welcome to copy them over to csim.c

Read the Writeup over and over!

Questions?