System Programming
Lab #4

2021-04-28

sp-tas

Lab Assignment #4 : Cache Lab

- Download skeleton code & pdf from eTL
  - cachelab-handout.tar, malloclab-handout.pdf
- Hand In
  - Upload your files eTL
    - 압축파일 양식 : [학번]_[이름]_cachelab.zip
    - Ex) 2021-12345_홍길동_cachelab.zip
  - A zip file should include:
  - (1) csim.c (2) trans.c (3) Report
    - .c 양식 : [Filename]-[학번].c ex) csim-2021-12345.c (제출할 때만 바꿔서)
    - Report 양식 : [학번]_[이름]_cachelab_report.pdf (or .hwp, .docx etc.)
- Please, READ the Hand-out thoroughly!
- Assigned : Apr. 28th
- Deadline : May. 19th , 23:59:59
- Delay policy : Same as before
- Next time (May. 12th)
  - Kernel LAB assign

Cache LAB Preview

- There are two parts in this LAB

  - implementing your own…

  A. Cache simulator – csim.c

  B. Matrix transpose function – trans.c

A. Cache simulator

- Simulates how cache works
- By replaying trace files

B. Matrix transpose

- Implement matrix transpose function
- Cache-friendly

A. Cache simulator

- A cache simulator is NOT a real cache
  - Memory contents are not stored, Block offsets are not used
  - Simply count hits, misses, and evictions

- Your cache simulator needs to work for different:
  - Number of set
  - Block size
  - Associativity

- Use LRU for replacement policy

Cache simulator – Trace files

- Trace files are generated from ‘Valgrind’
  - Linux program, for summarize memory management, threading, etc.

- Valgrind memory traces have the following form:

```
I 0400d7d4,8
 M 0421c7f0,4
 L 04f6b868,8
 S 7ff0005c8,8
```

- Format of each line

```
[space]operation address,size
```

Cache simulator – Trace files

- Operations:
  - I: Instruction load
  - L: Data load
  - S: Data store
  - M: Data Modify

```
I 0400d7d4,8
 M 0421c7f0,4
 L 04f6b868,8
 S 7ff0005c8,8
```

- No space before each ‘I’
- There’s always a space(‘ ’) before each M, L, and S

Cache simulator

- Write a simulator in `csim.c`
  - Simulates the hit/miss behavior of a cache memory with memory trace file
  - Outputs total count of hit, miss, and evict
- We provide `csim-ref`
  - The binary executable of a reference cache simulator

  Usage: `./csim-ref [-hv] -s <s> -E <E> -b <b> -t <tracefile>`

    - `-h`: Optional help flag that prints usage info
    - `-v`: Optional verbose flag that displays trace info
    - `-s <s>`: Number of set index bits ($S = 2^s$ is the number of sets)
    - `-E <E>`: Associativity (number of lines per set)
    - `-b <b>`: Number of block bits ($B = 2^b$ is the block size)
    - `-t <tracefile>`: Name of the valgrind trace to replay

Cache simulator – csim-ref example

- For example:

```
ta@ubuntu:~/cachelab$ ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace
hits:4 misses:5 evictions:3
```

- The same example in verbose mode:

```
ta@ubuntu:~/cachelab$ ./csim-ref -v -s 4 -E 1 -b 4 -t traces/yi.trace
L 10,1 miss
M 20,1 miss hit
L 22,1 hit
S 18,1 hit
L 110,1 miss eviction
L 210,1 miss eviction
M 12,1 miss eviction hit
hits:4 misses:5 evictions:3
```

Keep in mind

Programming rules for part A:

- Your `csim.c` file must compile without warnings in order to receive credit.
- Your simulator must work correctly for arbitrary s, E, and b. This means that you will need to allocate storage for your simulator's data structures using the `malloc` function. Type "man malloc" for information about this function.
- For this lab, we are interested only in data cache performance, so your simulator should ignore all instruction cache accesses (lines starting with "I"). Recall that `valgrind` always puts "I" in the first column (with no preceding space), and "M", "L", and "S" in the second column (with a preceding space). This may help you parse the trace.
- To receive credit for Part I, you must call the function `printSummary`, with the total number of hits, misses, and evictions, at the end of your `main` function:

```
printSummary(hit_count, miss_count, eviction_count);
```

- For this this lab, you should assume that memory accesses are aligned properly, such that a single memory access never crosses block boundaries. By making this assumption, you can ignore the request sizes in the `valgrind` traces.

Useful hints for part A

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

- Each `cache_line` has:
  - Valid bit
  - Tag
  - LRU counter

Part (a) : 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**
  #include <unistd.h>

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

Part (a) : 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) : fscanf

- **The fscanf() function is just like scanf() except it can specify a stream to read from (scanf always reads from stdin)**
  - parameters:
    - A stream pointer
    - format string with information on how to parse the file
    - the rest are pointers to variables to store the parsed data
  - You typically want to use this function in a loop. It returns -1 when it hits EOF or if the data doesn't match the format string
- **For more information,**
  - man fscanf
  - http://crasseux.com/books/ctutorial/fscanf.html
- **fscanf will be useful in reading lines from the trace files.**
  - L 10,1
  - M 20,1

Part (a) : fscanf example

```
FILE * pFile; //pointer to FILE object

pFile = fopen ("tracefile.txt",“r"); //open file for reading

char identifier;
unsigned address;
int size;
// Reading lines like " M 20,1" or "L 19,3"

while(fscanf(pFile,“ %c %x,%d”, &identifier, &address,
&size)>0)
{
   // Do stuff
}

fclose(pFile); //remember to close file when done
```

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**

B. Matrix Transpose Function

- Write a transpose function in `trans.c`
  - **Cache-friendly!**

- An example transpose function in `trans.c`

```
char trans_dest[] = “Simple row-wise scan transpose”;
void trans(int M, int N, int A[N][M], int B[M][N])
```

  - It correctly works, but inefficient

- Goal: **minimize** the number of cache misses across different sized matrices

Part (b) Efficient Matrix Transpose

- Matrix Transpose (A -> B)

Matrix A

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

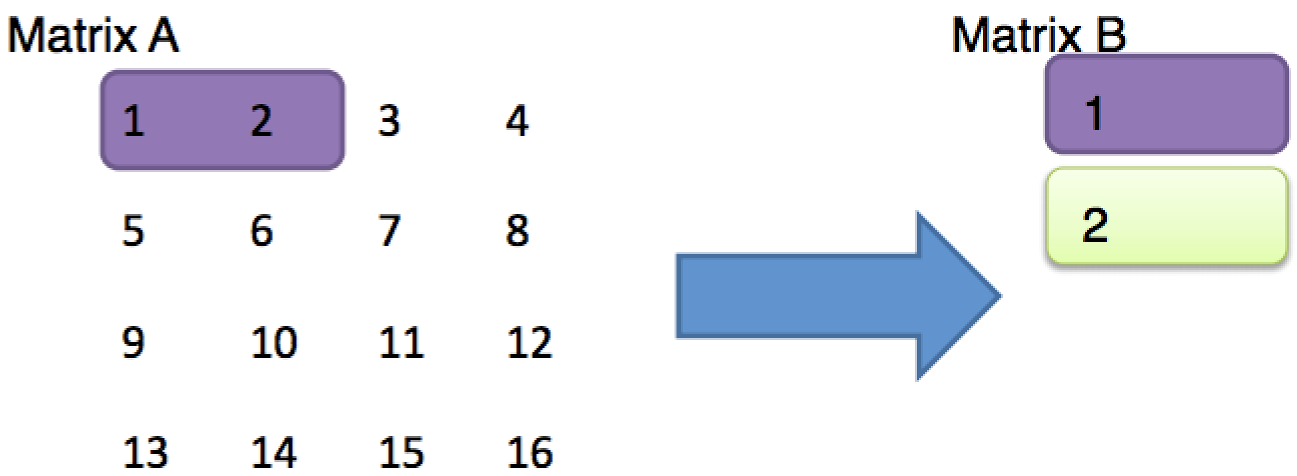
Matrix B

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?

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**

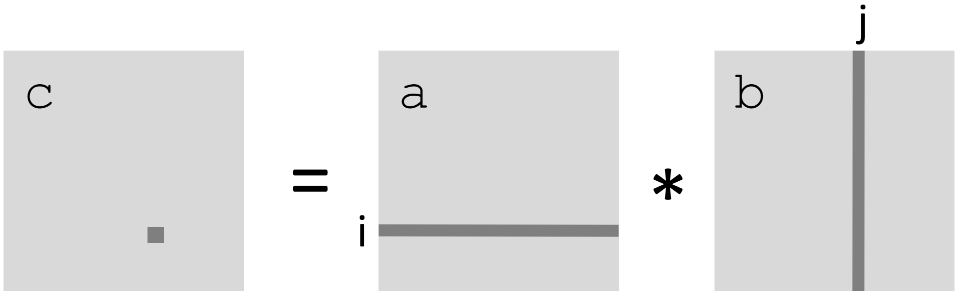
Should we handle 3 & 4 next or 5 & 6 ?

Part (b) : Blocking

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

Example: Matrix Multiplication

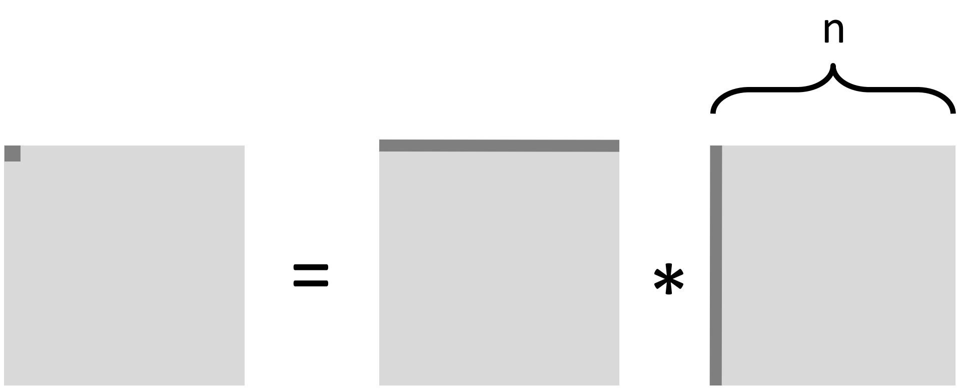
```
c = (double *) calloc(sizeof(double), n*n);

/* 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

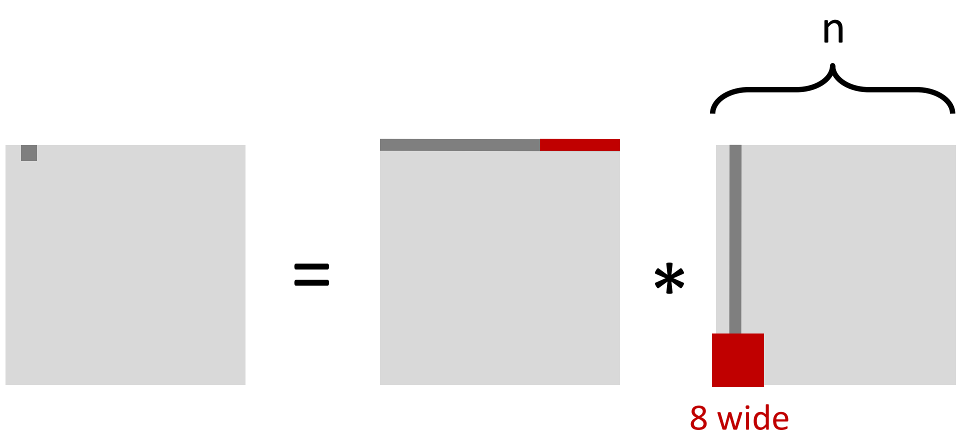


  - Afterwards in cache:
    (schematic)

Cache Miss Analysis

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

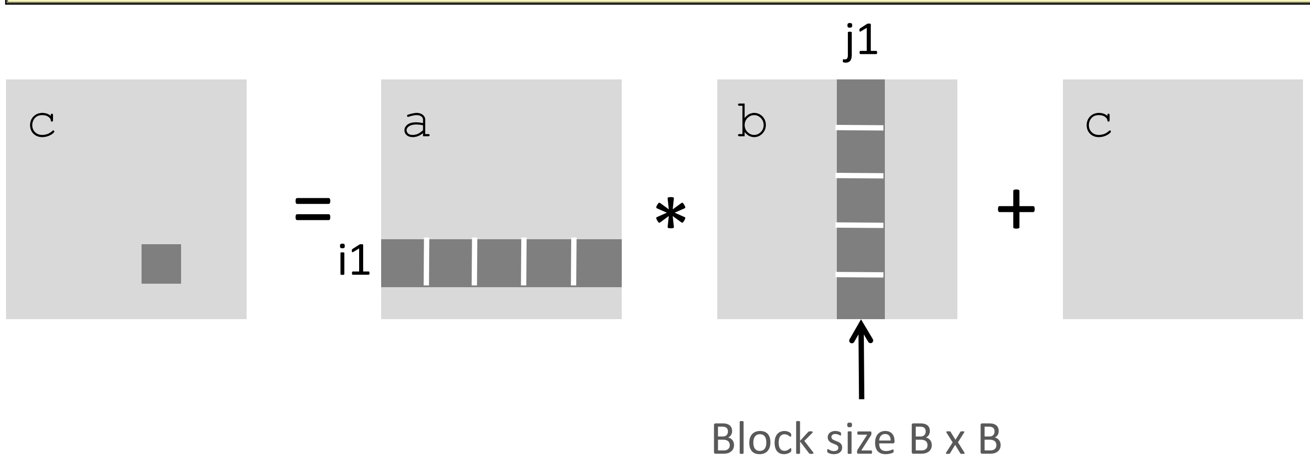
- Second iteration:
  - Again:
    n/8 + n = 9n/8 misses



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

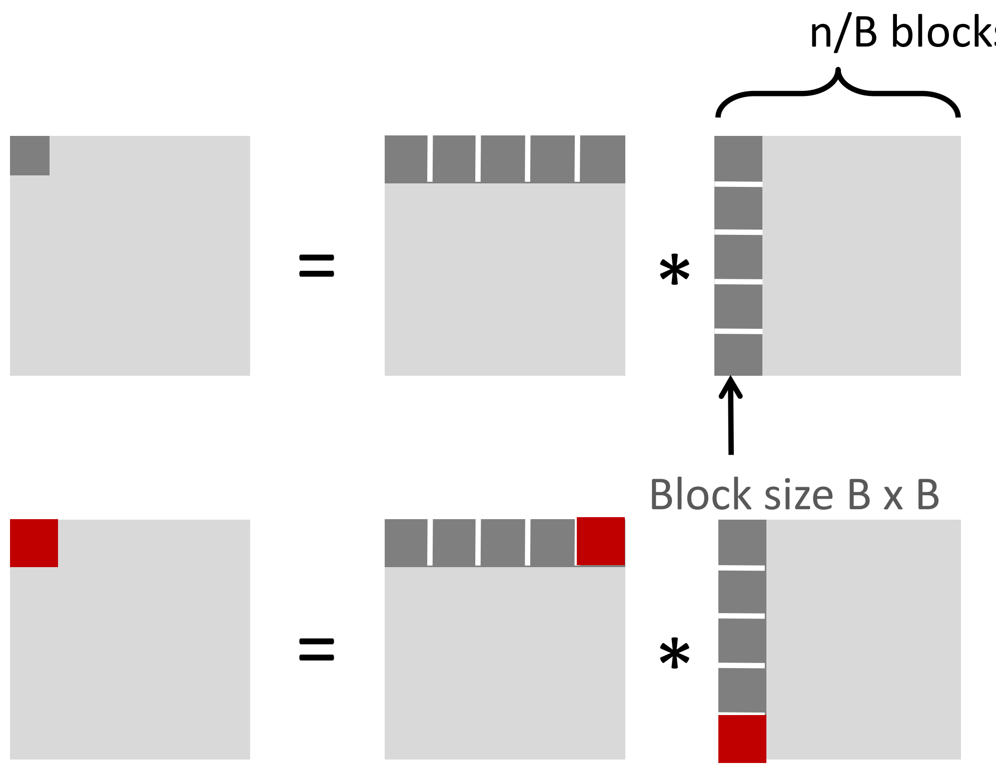
Blocked Matrix Multiplication

```
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];
}
```



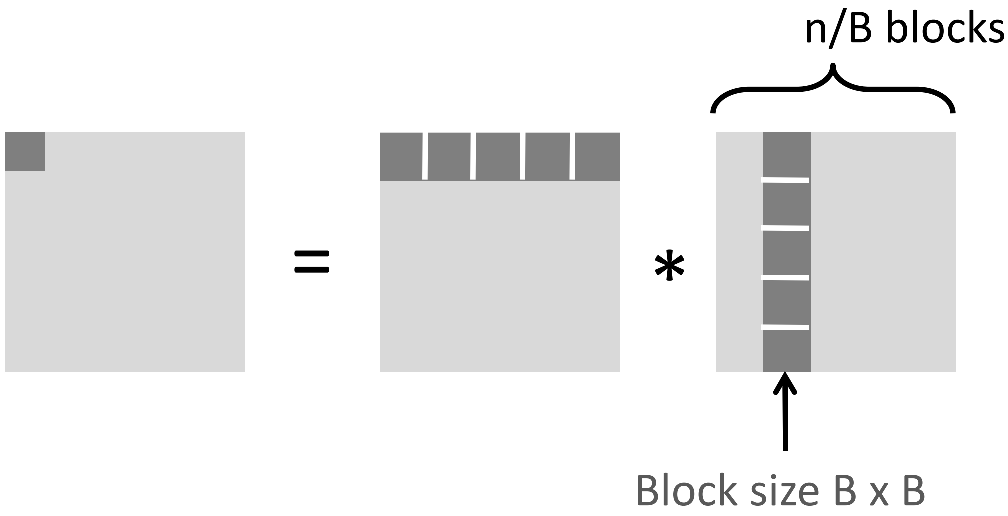
Cache Miss Analysis

- 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
  - Eviction policies in the cache

Keep in mind!

Programming rules for part B **(MUST READ CAREFULLY)**

*** You'll get 0 if you violate any of these rules ***

- Your code in `trans.c` must compile without warnings to receive credit.
- You are allowed to define at most 12 local variables of type `int` per transpose function.[1]
- You are not allowed to side-step the previous rule by using any variables of type `long` or by using any bit tricks to store more than one value to a single variable.
- Your transpose function may not use recursion.
- If you choose to use helper functions, you may not have more than 12 local variables on the stack at a time between your helper functions and your top level transpose function. For example, if your transpose declares 8 variables, and then you call a function which uses 4 variables, which calls another function which uses 2, you will have 14 variables on the stack, and you will be in violation of the rule.
- Your transpose function may not modify array A. You may, however, do whatever you want with the contents of array B.
- You are NOT allowed to define any arrays in your code or to use any variant of `malloc`.

[1]The reason for this restriction is that our testing code is not able to count references to the stack. We want you to limit your references to the stack and focus on the access patterns of the source and destination arrays.

How to work on Part B

- Write your functions in `trans.c` :

```
/* Header comment */
char trans_simple_desc[] = "A simple transpose";
void trans_simple(int M, int N, int A[N][M], int B[M][N]){
/* your transpose code here */
}
```

  - Do not modify `transpose_submit_desc[]`!

  => Just copy your submission code in `transpose_submit()`

- Register your function with the autograder

```
registerTransFunction(trans_simple, trans_simple_desc);
```

Testing your code (Part B)

- Re-build with `make`
- Run `test-trans`
  - For example, testing on a 32 x 32 matrix

```
ta@ubuntu:~/cachelab$ make
make: Nothing to be done for 'all'.
ta@ubuntu:~/cachelab$ ./test-trans -M 32 -N 32

Function 0 (4 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:1766, misses:287, evictions:255

Function 1 (4 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (Simple row-wise scan transpose): hits:870, misses:1183, evictions:1151

Function 2 (4 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 2 (column-wise scan transpose): hits:870, misses:1183, evictions:1151

Function 3 (4 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 3 (using a zig-zag access pattern): hits:1076, misses:977, evictions:945

Summary for official submission (func 0): correctness=1 misses=287

TEST_TRANS_RESULTS=1:287
```

Testing your code (Both A/B)

- Run ./driver.py

```
ta@ubuntu:~/cachelab$ ./driver.py
Part A: Testing cache simulator
Running ./test-csim
                        Your simulator     Reference simulator
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
     3 (1,1,1)       9       8       6       9       8       6  traces/yi2.trace
     3 (4,2,4)       4       5       2       4       5       2  traces/yi.trace
     3 (2,1,4)       2       3       1       2       3       1  traces/dave.trace
     3 (2,1,3)     167      71      67     167      71      67  traces/trans.trace
     3 (2,2,3)     201      37      29     201      37      29  traces/trans.trace
     3 (2,4,3)     212      26      10     212      26      10  traces/trans.trace
     3 (5,1,5)     231       7       0     231       7       0  traces/trans.trace
     6 (5,1,5)  265189   21775   21743  265189   21775   21743  traces/long.trace
    27


Part B: Testing transpose function
Running ./test-trans -M 32 -N 32
Running ./test-trans -M 64 -N 64
Running ./test-trans -M 61 -N 67

Cache Lab summary:
                        Points   Max pts      Misses
Csim correctness          27.0        27
Trans perf 32x32           8.0         8         287
Trans perf 64x64           8.0         8        1107
Trans perf 61x67          10.0        10        1913
          Total points    53.0        53
```

Evaluation

- Total (70 points)
  - Part A (27 points)
  - Part B (26 points)
  - Style (7 points)
  - Report (10 points)

- Warnings are errors!
  - Should fix all warnings on compile-time
  - To get credit on style

Evaluation

[Part A]

- Correct # of hits, misses, and evictions
  - For each test case

[Part B]

- For each test case, miss count = m
  - 32 x 32: n points if m < 300, 0 points if m > 600
  - 64 x 64: n points if m < 1300, 0 points if m > 2000
  - 61 x 67: m points if m < 2000, 0 points if m > 3000
- Of course, result of transpose must be correct

End & Notification

- Due: May. 19th 23:59:59
- Questions
  - eTL Q&A Board
  - eMail: sp_tas@dcslab.snu.ac.kr
- Please, read the handout & start early!

- Next time (May. 12th)
  - No Q&A Session
  - **Kernel LAB (LAB#5) will be assigned (Due: Jun. 2nd)**