Project 2: Understanding Cache Memories

518030910022 杨涵章 linqinluli@sjtu.edu.cn

1. Introduction

This lab is mainly related to cache memories. The main purpose is to help us understand the impact that cache memories can have on the performance of my C programs.

The lab consists of two parts. In part A, the task is to write a C program to simulate the behavior of a cache memory. The program takes a `valgrind` memory trace as input, simulates the hit/miss behavior of a cache memory on this trace, and outputs the total number of hits, misses, and evictions. There is a reference cache simulator and our simulator ought to take the same parameters as the reference simulator. The job I've done in this project is that I wrote a C program as it required. The main program will be introduced in the following part in detail. Then I executed `test-csim` to test the correctness of my cache simulator on the reference traces and got full marks.

In part B, the task is to optimize a small matrix transpose function, with the goal of minimizing the number of cache misses. There are three size of matrix and the task is to find the suitable methods to minimizing the number of cache misses. The main idea is matrix blocking. However, it can only optimize a small part. Thus, for different matrix, it needs different optimization. The job I've done in this part is that I wrote three different matrix transpose functions. However, the knowledge cannot support me finish all job. I learning some other optimization method on the internet. Then I tested the three functions and got full marks.

2. Experiments

2.1 Part A

2.1.1 Analysis

This part asks us to write a C program to simulate the performance of cache memories. It needs to take a command line and read the data line in traces files. Then simulate the cache memories and print the number of hit, miss and evict.

At first, it needs to take a command line to execute. Thus, I learned the use of the `getopt` function to parse the command line arguments. This function uses `optstring` to scan the command line and get the parameter s, E, b and t.

Then for the simulator must work correctly for arbitrary s (the number of sets), E (the associativity), and b (number of block bits), I used

`malloc` function to allocate memories for caches.

Next is the read of instructions. I get a whole line and do further works. For I instruction, there is nothing to do with. And for both L and M, it needs to update the situations of caches once. And for M, it needs to update twice, because it means a data load followed by a data store. After updating the situations, the time stamp of each line needs to be update as well.

For the data structure of cache line, it needs three data members. Because it uses the LRU (least-recently used) replacement policy when choosing which cache line to evict, it needs a time stamp. Valid bit and address tag are must as well.

For the update function, there are three cases. First case is that there is a hit. And it only needs to restart the time stamp. Second case is that there isn't a hit and there is an empty line. And it needs to change the valid bit, change address and restart the time stamp. Third case is that there is an eviction. And it needs to find the cache line with the maximum time stamp. Then change valid bit, change address and restart the time stamp.

2.1.2 Code

```
#include "cachelab.h"
#include<stdlib.h>
#include<unistd.h>
#include<stdio.h>
#include<limits.h>
#include<getopt.h>
#include<string.h>
int s, E, b, S;
int number_hits=0, number_miss=0, number_eviction=0;
//S is the number of sets, E is the associativity, b is number of block bits
char filename[1000];//The file name
char buffer[1000];//The buffer of input
typedef struct
{
    int valid_bit, tag, stamp;//data structure of the cache
}cache_line;
cache_line **cache = NULL;
void update(unsigned int address)
{
    int max_stamp = INT_MIN, max_stamp_id = -1;
    int  t_address, s_address;// The t value and s value of address
    s_address = (address >> b) & ((1U) >> (32 - s));
     //use bit manipulation to get s_address, -1U equals to INT_MAX
    t_address = address >> (s + b);
```

```
    //check whether there is a hit
    for(int i = 0; i < E; i++)
        if(cache[s_address][i].tag == t_address)// a hit
        {
            //is_placed = 1;
            cache[s_address][i].stamp = 0;
     //restart the time stamp control unit
            number_hits++;
            //printf("hit\n");
            return ;//just return now
        }
    //to check whether is an empty line
    for(int i = 0; i < E; i++)
        if(cache[s_address][i].valid_bit == 0)
        {
            cache[s_address][i].valid_bit = 1;
            cache[s_address][i].tag = t_address;
            cache[s_address][i].stamp = 0;
            number_miss++;//compulsory miss
            //printf("miss\n");
            return;
        }
    //If there is not any empty line, then an eviction will occur
    number_eviction++;
    number_miss++;
    for(int i = 0; i < E; i++)
        if(cache[s_address][i].stamp > max_stamp)
        {
            max_stamp = cache[s_address][i].stamp;
            max_stamp_id = i;
        }
    cache[s_address][max_stamp_id].tag = t_address;
    cache[s_address][max_stamp_id].stamp = 0;
    return;
}
void update_time(void)//update the time stamp
{
    for(int i = 0; i < S; i++)
        for(int j = 0; j < E; j++)
            if(cache[i][j].valid_bit == 1)//if valid
                cache[i][j].stamp++;
}
int main(int argc,char* argv[])
{
```

```
69.     int opt=0, temp;//The getopt return value
70.     char type;//type of a single trace record
71.     unsigned int address;//address of memory
72.     while(-1 != opt)
73.     {
74.         opt = getopt(argc, argv, "s:E:b:t:");
75.         switch(opt)
76.         {
77.             case 's':s = atoi(optarg);
78.                      break;
79.             case 'E':E = atoi(optarg);
80.                      break;
81.             case 'b':b = atoi(optarg);
82.                      break;
83.             case 't':strcpy(filename, optarg);
84.                      break;
85.         }
86.     }
87.     FILE* fp = fopen(filename,"r");
88.     S = (1 << s); // S equals to 2^s
89.     cache = (cache_line**)malloc(sizeof(cache_line*) * S);
90.     for(int i = 0; i < S; i++)
91.         cache[i] = (cache_line*)malloc(sizeof(cache_line) * E);
92.      //malloc each row of cache
93.     for(int i = 0; i < S; i++)
94.         for(int j = 0; j < E; j++)
95.         {
96.             cache[i][j].valid_bit = 0;
97.             cache[i][j].tag = cache[i][j].stamp = -1;
98.         }//initialization
99.     while(fgets(buffer,1000,fp))//get a whole line
100.     {
101.         sscanf(buffer," %c %xu,%d", &type, &address, &temp);//hexdecimal
102.         switch(type)
103.         {
104.             case 'L':update(address);
105.                      break;
106.             case 'M':update(address);//just let it fall through, do twice
107.             case 'S':update(address);
108.                      break;
109.         }
110.         update_time();
111.     }
112.     for(int i = 0; i < S; i++)
```

```
113.         free(cache[i]);//free allocated space first
114.     free(cache);
115.     fclose(fp);
116.     printSummary(number_hits, number_miss, number_eviction);
117.     return 0;
118. }
```

2.1.3 Evaluation

```
linqinluli@ubuntu:~/arclab2/project2-handout$ ./csim -s 1-E 1-b 1-t traces/yi2.trace
Segmentation fault (core dumped)
linqinluli@ubuntu:~/arclab2/project2-handout$ ./csim -s 1 -E 1 -b 1 -t traces/yi2.trace
hits:9 misses:8 evictions:6
linqinluli@ubuntu:~/arclab2/project2-handout$ ./csim -s 4 -E 2 -b 4 -t traces/yi.trace
hits:4 misses:5 evictions:2
linqinluli@ubuntu:~/arclab2/project2-handout$ ./csim -s 2 -E 1 -b 4 -t traces/dave.trace
hits:2 misses:3 evictions:1
linqinluli@ubuntu:~/arclab2/project2-handout$ ./csim-s 2 -E 1 -b 3 -t traces/trans.trace
bash: ./csim-s: No such file or directory
linqinluli@ubuntu:~/arclab2/project2-handout$ ./csim -s 2-E 1 -b 3 -t traces/trans.trace
Segmentation fault (core dumped)
linqinluli@ubuntu:~/arclab2/project2-handout$ ./csim -s 2 -E 1 -b 3 -t traces/trans.trace
hits:167 misses:71 evictions:67
linqinluli@ubuntu:~/arclab2/project2-handout$ ./csim -s 2 -E 2 -b 3 -t traces/trans.trace
hits:201 misses:37 evictions:29
```

Figur1: test of simulator (command line)

```
linqinluli@ubuntu:~/arclab2/project2-handout$ ./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

TEST_CSIM_RESULTS=27
linqinluli@ubuntu:~/arclab2/project2-handout$
```

Figur2: test of simulator (`test-csim`)

It's clear that after execute the `test-csim` program, it can pass all the test traces and print correct numbers of hit, miss and evict. And the score is 27 which is full marks.

2.2 Part B

2.2.1 Analysis

This part asks us to modify matrix transpose functions, with the goal of minimizing the number of cache misses. The normal ways of optimizing matrix transpose is matrix blocking, which divides matrix into smaller parts and to reduce cache misses. However, there are three kinds of matrix: 32x32, 64x64 and 61x67. For different size of matrix, it needs different blocking ways, or even more effective ways.

The reference simulator simulates a cache with parameters ($s = 5$, $E = 1$, $b = 5$). The block size is 32 byte and int size is 4 bytes, so a block can contain 8 int. There are 5 sets, so cache size is 32x32 bytes which is 32

blocks. And the manage way is direct mapped.

Matrix blocking:

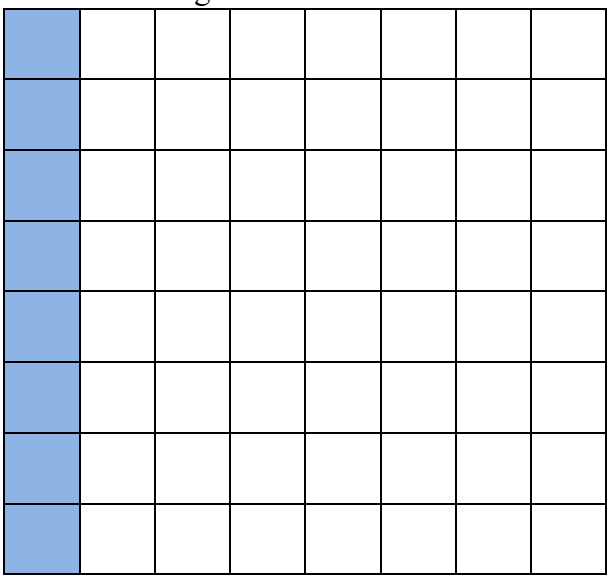
It's the main optimizing way. It divides the matrix into smaller blocks to take full use of cache and reduce cache misses.

Prefetch:

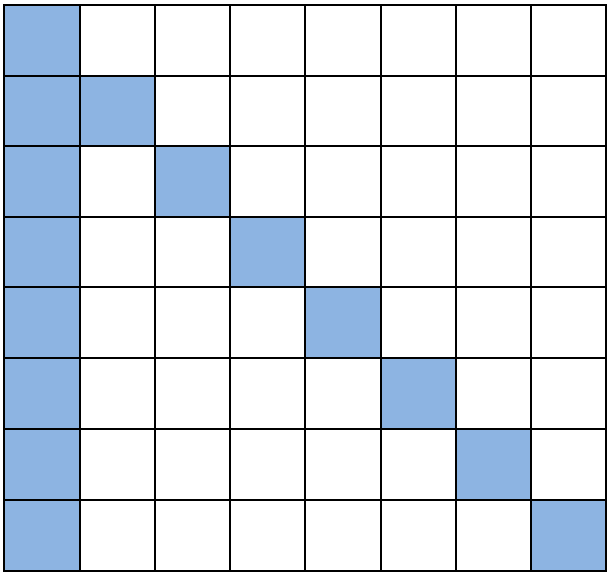
Prefetch the elements that will be used soon in the cache block to reduce the misses.

32x32:

For every cache block can contain 8 int, I divided the matrix into 16 blocks with 8x8 size to take full use of each block size. However, the element in the diagonal block will cause many misses. Because both A and B mapped to the same cache blocks and it caused cache misses. Therefore, I prefetch the 8 elements will be copied in matrix A and store them in local variables. Then copy them to matrix B. The total number of local variables is 11 which is less than the limitation. Thus, for blocks in the diagonal, the total misses are 23 times (some pictures can help understand). For other blocks, the total misses are 16 times. Therefore, the number of misses is 23x4+16x12=284 times which is almost the same to the result I got in the lab.



Matrix block on the diagonal of A (blue color means a hit)



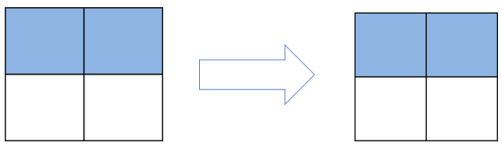
Matrix block on the diagonal of A (blue color means a hit)

64x64:

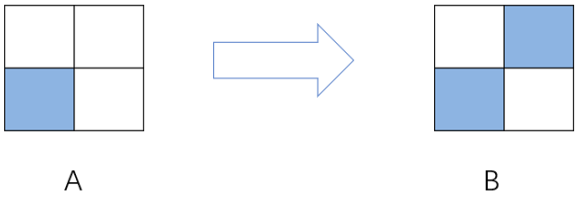
This part is more difficult than 32x32. If we still use 8x8 blocking, the first 4 rows and the last 4 rows will be mapped to the same cache blocks which causes a lot cache misses. If we use 4x4 blocking, the cache block will be wasted, because cache blocks can contain 8 elements. The result of 4x4 blocking is 1843 misses which is still cannot get full marks.

It's hard for me to finish 64x64 blocks, so I searched on the internet and found an effective way to reduce misses. The main idea is to copy elements of blocks in A to blocks in B first. Then transpose it in matrix B. It can be explained by following figures.

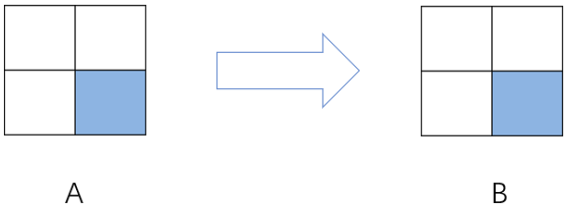
a. copy the first 4 rows of matrix blocks of A to the same place of matrix blocks in B.



b. move the upper right corner of blocks in B to correct places and transpose the lower left corner of blocks in A to B.



c. transpose the last corner of blocks in A to blocks in B



61x64:

The limitation for this matrix transpose is more relaxed. So, I only used a 16x16 blocking (after trying different size of blocking) and prefetch the elements in A. Fortunately, the number of misses is under the limitation.

2.2.2 Code

```
1.  void transpose11(int M, int N, int A[N][M], int B[M][N])
2.  {
3.      int i,j;
4.      int i1;
5.      int val1,val2,val3,val4,val5,val6,val7,val8;
6.      for(i=0;i<32;i+=8){
7.          for(j=0;j<32;j+=8){
8.              for(i1=i;i1<i+8;i1++){
9.                      val1 = A[i1][j+0];
10.                     val2 = A[i1][j+1];
11.                     val3 = A[i1][j+2];
12.                     val4 = A[i1][j+3];
13.                     val5 = A[i1][j+4];
14.                     val6 = A[i1][j+5];
15.                     val7 = A[i1][j+6];
16.                     val8 = A[i1][j+7];
17.                     B[j+0][i1] = val1;
18.                     B[j+1][i1] = val2;
19.                     B[j+2][i1] = val3;
20.                     B[j+3][i1] = val4;
```

```
                    B[j+4][i1] = val5;
                    B[j+5][i1] = val6;
                    B[j+6][i1] = val7;
                    B[j+7][i1] = val8;
                }
            }
        }
}
```

Code1: 32x32

```
void transpose22(int M,int N,int A[N][M],int B[M][N])
{
    int i, j, ii, jj, val0, val1, val2, val3, val4, val5, val6, val7;
    for(ii = 0; ii < N; ii += 8)
    {
        for(jj = 0; jj < M; jj += 8)
        {
            //For each row in the 8*4 block
            for(i = 0; i < 4; i++)
            {
                val0 = A[ii + i][jj + 0];
                val1 = A[ii + i][jj + 1];
                val2 = A[ii + i][jj + 2];
                val3 = A[ii + i][jj + 3];
                val4 = A[ii + i][jj + 4];
                val5 = A[ii + i][jj + 5];
                val6 = A[ii + i][jj + 6];
                val7 = A[ii + i][jj + 7];
                B[jj + 0][ii + i] = val0;
                B[jj + 1][ii + i] = val1;
                B[jj + 2][ii + i] = val2;
                B[jj + 3][ii + i] = val3;
                B[jj + 0][ii + 4 + i] = val4;
                B[jj + 1][ii + 4 + i] = val5;
                B[jj + 2][ii + 4 + i] = val6;
                B[jj + 3][ii + 4 + i] = val7;
            }
            //First copy the first 4 rows
            for(i = 0; i < 4; i++)// transformation
            {
                //get this row of the right-upper 4*4 block
                val0 = B[jj + i][ii + 4];
                val1 = B[jj + i][ii + 5];
                val2 = B[jj + i][ii + 6];
```

```
                val3 = B[jj + i][ii + 7];
                //update this row to its correct value
                val4 = A[ii + 4][jj + i];
                val5 = A[ii + 5][jj + i];
                val6 = A[ii + 6][jj + i];
                val7 = A[ii + 7][jj + i];

                B[jj + i][ii + 4] = val4;
                B[jj + i][ii + 5] = val5;
                B[jj + i][ii + 6] = val6;
                B[jj + i][ii + 7] = val7;
                B[jj + 4 + i][ii + 0] = val0;
                B[jj + 4 + i][ii + 1] = val1;
                B[jj + 4 + i][ii + 2] = val2;
                B[jj + 4 + i][ii + 3] = val3;
            }
            //update the right lower 4*4 block
            for(i = 4; i < 8; i++)
                for(j = 4; j < 8; j++)
                    B[jj + j][ii + i] = A[ii + i][jj + j];
        }
    }
}
```

Code2: 64x64

```
void transpose33(int M,int N,int A[N][M],int B[M][N])
{
    int ii,jj,i,j,val0,val1,val2,val3,val4,val5,val6,val7;
    for(ii = 0; ii + 16 < N; ii += 16)
        for(jj = 0; jj + 16 < M; jj += 16)
        {
            for(i = ii; i < ii + 16; i++)
            {
                val0 = A[i][jj + 0];
                val1 = A[i][jj + 1];
                val2 = A[i][jj + 2];
                val3 = A[i][jj + 3];
                val4 = A[i][jj + 4];
                val5 = A[i][jj + 5];
                val6 = A[i][jj + 6];
                val7 = A[i][jj + 7];
                B[jj + 0][i] = val0;
                B[jj + 1][i] = val1;
                B[jj + 2][i] = val2;
```

```
                B[jj + 3][i] = val3;
                B[jj + 4][i] = val4;
                B[jj + 5][i] = val5;
                B[jj + 6][i] = val6;
                B[jj + 7][i] = val7;

                val0 = A[i][jj + 8];
                val1 = A[i][jj + 9];
                val2 = A[i][jj + 10];
                val3 = A[i][jj + 11];
                val4 = A[i][jj + 12];
                val5 = A[i][jj + 13];
                val6 = A[i][jj + 14];
                val7 = A[i][jj + 15];
                B[jj + 8][i] = val0;
                B[jj + 9][i] = val1;
                B[jj + 10][i] = val2;
                B[jj + 11][i] = val3;
                B[jj + 12][i] = val4;
                B[jj + 13][i] = val5;
                B[jj + 14][i] = val6;
                B[jj + 15][i] = val7;

            }
        }
    for(i = ii; i < N; i++)  //the rest matrix
        for(j = 0; j < M; j++)
            B[j][i] = A[i][j];
    for(i = 0; i < ii; i++)
        for(j = jj; j < M; j++)
            B[j][i] = A[i][j];
}
```

Code3: 61x67

2.2.3 Evaluation

```
linqinluli@ubuntu:~/arclab2/project2-handout$ ./test-trans -M 32 -N 32

Function 0 (2 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 (2 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

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

TEST_TRANS_RESULTS=1:287
```

Figure1: 32x32 matrix

For blocks in the diagonal, the total misses are 23 times (some pictures can help understand). For other blocks, the total misses are 16 times. Therefore, the theoretical number of misses is 23x4+16x12=284 times. After testing the 32x32 matrix, get a result of 287 misses which is close to the theoretical value and get full marks.

```
linqinluli@ubuntu:~/arclab2/project2-handout$ ./test-trans -M 64 -N 64

Function 0 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:8970, misses:1275, evictions:1243

Function 1 (2 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:3474, misses:4723, evictions:4691

Summary for official submission (func 0): correctness=1 misses=1275

TEST_TRANS_RESULTS=1:1275
```

Figure2: 64x64 matrix

For blocks in the diagonal, the total misses are 48 times (some pictures can help understand). For other blocks, the total misses are 16 times. Therefore, the theoretical number of misses is 48x8+16x56=1280 times. After testing the 64x64 matrix, get a result of 1275 misses which is close to the theoretical value and get full marks.

```
linqinluli@ubuntu:~/arclab2/project2-handout$ ./test-trans -M 61 -N 67

Function 0 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:6216, misses:1963, evictions:1931

Function 1 (2 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:3756, misses:4423, evictions:4391

Summary for official submission (func 0): correctness=1 misses=1963

TEST_TRANS_RESULTS=1:1963
```

Figure3: 61x67 matrix

After trying different size of blocking, 16x16 blocking gets 1963 misses which can satisfy the limitation request.

```
linqinluli@ubuntu:~/arclab2/project2-handout$ ./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        1275
Trans perf 61x67          10.0        10        1963
          Total points    53.0        53
```

Firuge4: points test of part A and B

After executing `driver.py`, it tests my programs. Then print the details and total points.

3. Conclusion

3.1 Problems

Part A:

Part A is just simulating the cache memories, so it's not so difficult to finish it. However, I sill learnt some useful C functions.

a. For it needs to parse the command line arguments, it's difficult to analyze the command line. The lab guidance suggests to use `getopt` function. I read related documents and find it help a lot.

b. It's hard to debug my code. I try to print current situation and find what's the problem.

Part B:
Part B is quite difficult. And it took me lots of time to learn matrix transpose optimizing methods.
a. Because we only learnt change matrix visiting order and prefetch. I've tried both the two ways, but is can not satisfy the limitation. I searched on the internet and found matrix blocking. It took full use of cache space and I passed 32x32 matrix and 61x67 matrix.
b. For 64x64 matrix, I've tried all the methods mentioned before. However, the limit of 1300 is so strict. Therefore, I referred the solution that I searched on the internet and found another method: first copy and then transpose.

3.2 Achievements

This lab is much harder than previous lab. One reason is that the lab needs us to finish by ourselves with no partners. Another reason is that part B really needs sufficient knowledge accumulation to support finishing the lab. It took me almost 5 days to finish this lab. Actually, the code part is not so difficult, I'm familiar with C language. I finished part A during a short time. Part B is quite difficult for me, and I even cannot understand what's the purpose of this part. I've reviewed slid and text books. Then I look up something about matrix in the book CS:APP. Finally, I searched it on the internet. Most of my time is used to learning new skills such as new C program functions, and matrix blocking. The direct achievements in this lab is the new skills I've learnt and these can help a lot in the future. Furthermore, the way of learning new skills also matters. The skill searched on the internet is messy and sometimes it's wrong. The knowledge in text book is neat and orderly. However, it may take a lot of time to understand.