565HW2 Writeup

Yisong Zou
yz558

Problem1

First, convert the addresses to binary.
As the size of a set is 2^6 * 2 = 2^7 bytes, and there are total 2^9 / 2 ^ 7 = 4 sets, so the block offset is 6 bits, set index is 2 bits.
The set index is shown as green in the following addresses and **the right show if it is a hit or a miss**

ABCDE : 1010101111001**11**011110 **(Tag ABC) (Miss)**

14327 : 1010000110**00**100111**(Tag 143) (Miss)**

DF148 : 1101111100010**1**001000**(Tag DF1) (Miss)**

8F220 : 1000111100100**0**100000**(Tag 8F2) (Miss)**

CDE4A : 1100110111101**0**001010**(Tag CDE) (Miss)**

1432F : 1010000110**00**101111**(Tag 143) (Hit)**

52C22 : 1010010110**00**100010**(Tag 52C) (Miss)**

ABCF2 : 1010101111001**11**110010**(Tag ABC) (Hit)**

92DA3 : 1001001011011**0**100011**(Tag 92D) (Miss)**

F125C : 1111000100101**0**011100**(Tag F12) (Miss)**

The final results are as follows:

	Way 0	Way 1
Set 0	1432F	52C22
Set 1	F125C	CDE4A
Set 2	92DA3	
Set 3	ABCF2	

Problem2

a) AAT = 1 + 0.03*(15 + 0.3*300) = 4.15 CPU cycles
b) AAT = 1 + 0.1*(15 + 0.05*300) = 4 CPU cycles

Problem3

(c)

From the file /sys/devices/system/cpu/cpu0/cache/index* the L1 data cache is 32k in size.

So I decided to use an array of **4096 uint64,** which in total will add up to **32k Bytes.**

For the write traffic only one, the code critical part is as follows:

```
clock_gettime(CLOCK_MONOTONIC, &start_time);
 int temp = num_traversals;
 int read = 0;
 while (temp > 0) {
   for (i = 0; i < num_elements; i++) {
     array[i] = 1;
   }
   --temp;
 }
clock_gettime(CLOCK_MONOTONIC, &end_time);
```

The bandwidth is calculated by this:

```
((((uint64_t)num_elements * (uint64_t)num_traversals) * 8) /
          (elapsed_ns))
```

I initialize the array and use traversals to compute the average. The **num_elements** is fixed to 4096 in order to compare to the cache size. The program will run the inner loop for **traversal times** which is set in the build.sh to **10000000**, and this can be easily changed in the script.

For the 1:1 read-to-write ratio one, the code critical part is as follows:

```
clock_gettime(CLOCK_MONOTONIC, &start_time);
  int temp = num_traversals;
  int read = 0;
  while (temp > 0) {
    for (i = 0; i < num_elements; i++) {
```

```
      read = array[i];
      array[i] = 1;
    }
    --temp;
  }
  clock_gettime(CLOCK_MONOTONIC, &end_time);
```

The bandwidth is calculated by this:

```
((((uint64_t)num_elements * (uint64_t)num_traversals) * 8 * 2) /
          (elapsed_ns)))
```

As the instructions in the loop for read and write has the same number of instructions, they have the same ratio.

For the 2:1 read-to-write ratio one, the code critical part is as follows:

```
clock_gettime(CLOCK_MONOTONIC, &start_time);
  int temp = num_traversals;
  int read = 0;
  while (temp > 0) {
    for (i = 0; i < num_elements; i++) {
      read = array[i];
      read = array[i];
      array[i] = 1;
    }
    --temp;
  }
  clock_gettime(CLOCK_MONOTONIC, &end_time);
```

The bandwidth is calculated by this:

```
((((uint64_t)num_elements * (uint64_t)num_traversals) * 8 * 3) /
          (elapsed_ns))
```

As the instructions in the loop for read and write has 2:1 number of instructions, they have the 2:1 ratio.

To run the program, just run ./build.sh

This will build all the three programs and output the result for them, the default traversal number is set in the script to 10000000.

The results is as the follows:

```
yz558@vcm-15935:~/ECE565/ECE565_Parallel/hw2$ ./build.sh
Time=7318351068.000000
Write Traffic Only Bandwidth(GB/second)=44.775114
Time=7332541143.000000
1:1 read-to-write ratio Bandwidth(GB/second)=89.376928
Time=7347048065.000000
2:1 read-to-write ratio Bandwidth(GB/second)=133.800676
```

Write Traffic Only Bandwidth(GB/second)	44.775114
1:1 read-to-write ratio Bandwidth(GB/second)	89.376928
2:1 read-to-write ratio Bandwidth(GB/second)	133.800676

From the results above, we can come into a conclusion that it is much faster to do read operations compared to write. This matches my knowledge because write will result in more CPU instructions and make it longer to execute. So more read ratio will make the bandwidth larger.

(d)

The largest cash is the L3 cach wich is 25344k Bytes. And thus to store the 8-Byte uint64, the array should contain 25344k/8 = 3244032 number of elements.

After modifying the element number to more than 3244032, which is 3400000, the results are as the follows:

```
yz558@vcm-15935:~/ECE565/ECE565_Parallel/hw2$ ./build.sh
Time=244554209.000000
Write Traffic Only Bandwidth(GB/second)=11.122278
Time=224750260.000000
1:1 read-to-write ratio Bandwidth(GB/second)=24.204644
Time=2165046727.000000
2:1 read-to-write ratio Bandwidth(GB/second)=37.689718
```

Write Traffic Only Bandwidth(GB/second)	11.122278
1:1 read-to-write ratio Bandwidth(GB/second)	24.204644
2:1 read-to-write ratio Bandwidth(GB/second)	37.689718

The results shows that the bandwidth becomes nearly 4 times smaller, this matches my expectation because the current array size has exhausted all the cash size, and the CPU may need to get to main memory for the data, that will cause the process much slower.

Problem4

(b)

Note: using ./testIJK <1/2/3> to run the code.

1:IJK 2:JKI 3:IKJ

	Test1	Test2	Test3
IJK	18.07	17.12	15.63
JKI	27.22	25.54	23.28
IKJ	10.59	10.46	10.48

Test1 screenshot:

```
yz558@vcm-15935:~/ECE565/ECE565_Parallel/hw2/hw2Q4$ ./testIJK 1
IJK_Time=18.071656 Seconds
yz558@vcm-15935:~/ECE565/ECE565_Parallel/hw2/hw2Q4$ ./testIJK 2
JKI_Time=27.219600 Seconds
yz558@vcm-15935:~/ECE565/ECE565_Parallel/hw2/hw2Q4$ ./testIJK 3
IKJ_Time=10.589702 Seconds
```

This match the class results. As the miss rate per iteration for the 3 cases are

ijk:1.125

jki:2

ikj:0.25

This match the ratio of the experiment result that jki is the slowest and ikj is the fastest.

(d)

Note: using ./testIJK <1/2/3/4> to run the code.

1:IJK 2:JKI 3:IKJ 4:IJK with tiling

As the L2 cash is 1024kB

Block Size should <= 1024kb

So 3 * 8 * N^2 <= 1024k

N <=209

So I choose N = 128, block size is 128^2*8 = 128KB

The code for tiling is as follows:

```
1.  void IJK_Tiling(vector<vector<double>> &A, vector<vector<double>> &B,
2.                  vector<vector<double>> &C) {
3.    double sum;
4.    for (int i = 0; i < 1024; i += 128) {
5.      for (int j = 0; j < 1024; j += 128) {
6.        for (int ii = i; ii < i + 128; ii++) {
7.          for (int jj = j; jj < j + 128; jj++) {
8.            sum = 0;
9.            for (int k = 0; k < 1024; k += 128) {
10.             for (int kk = k; kk < k + 128; kk++) {
11.               sum += A[ii][kk] * B[kk][jj];
12.             }
13.           }
14.           C[ii][jj] = sum;
15.         }
16.       }
17.     }
18.   }
19. }
```

```
yz558@vcm-15935:~/ECE565/ECE565_Parallel/hw2/hw2Q4$ ./testIJK 4
IJK_Tiling_Time=13.831780 Seconds
```

From the result above, the tiled version is much faster than the JKI version as well as its origin IJK version which is reasonable because loop tiling will make vectorization easier and make program run faster, but slower than IKJ version because I think here miss rate plays the main role.