Problem 1

Address	Tag	Index	Offset	Hit or Miss
ABCDE	101010111100	11	011110	M
14327	000101000011	00	100111	M
DF148	110111110001	01	001000	M
8F220	100011110010	00	100000	M
CDE4A	110011011110	01	001010	M
1432F	000101000011	00	101111	H
52C22	010100101100	00	100010	M
ABCF2	101010111100	11	110010	H
92DA3	100100101101	10	100011	M
F125C	111100010010	01	011100	M

		Way 0		Way 1	
	LRU	V	tag	V	tag
Set 0	0	1	143	1	52C
Set 1	1	1	F12	1	CDE
Set 2	1	1	92D	0	
Set 3	1	1	ABC	0	

64B cache blocks means 6 bits for offset.
512/64/2 = 4, so we have four sets which means 2 bits for index
20 – 6 – 2 = 12, so we have 12 bits for tag

Problem 2

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

Problem 3

```
js896@vcm-16679:~/ECE-565-Performance-Optimization-Parallelism/Project_2$ lscp
u
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              2
On-line CPU(s) list: 0,1
Thread(s) per core:  1
Core(s) per socket:  1
Socket(s):           2
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               15
Model name:          Intel(R) Xeon(R) Gold 6142 CPU @ 2.60GHz
Stepping:            1
CPU MHz:             2600.000
BogoMIPS:            5200.00
Hypervisor vendor:   VMware
Virtualization type: full
L1d cache:           32K
L1i cache:           32K
L2 cache:            1024K
L3 cache:            22528K
NUMA node0 CPU(s):   0,1
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
cmov pat pse36 clflush mmx fxsr sse sse2 ss syscall nx lm constant_tsc arch_pe
rfmon nopl tsc_reliable nonstop_tsc cpuid pni ssse3 cx16 tsc_deadline_timer hy
pervisor lahf_lm pti tsc_adjust arat
```

Using lscpu command, I got the size if L1 cache is 32KB which is 32768B. Also notice that the largest cache is L3 which is 22528KB, this is useful later.

How I design the test program:

1. I used 2 arrays; their sizes are both 2048 * 8B = 16384B so L1 cache can hold both of them.
2. To silence system noise, I added a variable called *num_traversal*. I let the test program execute repeatedly.
3. I used -O2 compile option

How to run the test program:

1. cd problem3
2. chmod +x run.sh
   (Might not need this step.)
3. sudo ./run.sh
   (The script will execute the test program for 5 times.)

Below is the result of executing test program:

```
js896@vcm-16679:~/ECE-565-Performance-Optimization-Parallelism/Project_2/problem3$ sudo ./run.sh
Write traffic only, time = 12501699.000000
Bandwidth = 21.471918 Gbps

1:1 read-to-write ratio, time = 20626378.000000
Bandwidth = 26.028366 Gbps

2:1 read-to-write ratio, time = 20677143.000000
Bandwidth = 38.946694 Gbps

Write traffic only, time = 10627480.000000
Bandwidth = 25.258618 Gbps

1:1 read-to-write ratio, time = 20887915.000000
Bandwidth = 25.702465 Gbps

2:1 read-to-write ratio, time = 21439917.000000
Bandwidth = 37.561077 Gbps

Write traffic only, time = 19780617.000000
Bandwidth = 13.570631 Gbps

1:1 read-to-write ratio, time = 20603908.000000
Bandwidth = 26.056752 Gbps

2:1 read-to-write ratio, time = 20606461.000000
Bandwidth = 39.080285 Gbps

Write traffic only, time = 18290882.000000
Bandwidth = 14.675916 Gbps

1:1 read-to-write ratio, time = 20764072.000000
Bandwidth = 25.855762 Gbps

2:1 read-to-write ratio, time = 20858138.000000
Bandwidth = 38.608737 Gbps

Write traffic only, time = 16845608.000000
Bandwidth = 15.935041 Gbps

1:1 read-to-write ratio, time = 20628425.000000
Bandwidth = 26.025783 Gbps

2:1 read-to-write ratio, time = 20786994.000000
Bandwidth = 38.740877 Gbps
```

My analysis:

1. *1:1 read to write ratio* has better bandwidth than *write traffic only*. Considering the L1 cache size, *1:1 read to write ratio* loads 2 arrays into the cache so it made better use of it and yet does not exceed the cache size. So, it makes sense *1:1 read to write ratio* has better bandwidth.
2. *2:1 read to write ratio* has better bandwidth than *1:1 read to write ratio,* I believe it is because I haven't reached the upper limit of CPU.

How I achieved the 3 patterns:

1. Write traffic only:

```
void write_only(int num_traversals) {
    for (int j = 0; j < num_traversals; ++j) {
        for (int i = 0; i < num_elements; ++i) {
            array[i] = 1;
        }
    }
}
```

2. 1:1 read write ratio

```
void one_read_one_write(int num_traversals) {
    for (int j = 0; j < num_traversals; ++j) {
        for (int i = 0; i < num_elements; ++ i) {
            array[i] = array_for_independence[i];
        }
    }
}
```

3. 2:1 read write ratio:

```
void two_read_one_write(int num_traversals) {
    for (int j = 0; j < num_traversals; ++j) {
        for (int i = 0; i < num_elements; ++ i) {
            array[i] = array[i] + array_for_independence[i];
        }
    }
}
```

Since the size of the largest cache is 22528KB, so I changed the sizes of arrays to 22528KB/8B = 2883584. To run the test program to this way, go to the file *run.sh*,

change *num_elements* to 2883584. Also, to avoid waiting too long for the execution of the program change *num_traversals* to 100, like below:

```
#!/bin/bash
for i in 1 2 3 4 5
do
   ./test_bandwith 2883584 100
done
```

Below is the result of execution:

```
js896@vcm-16679:~/ECE-565-Performance-Optimization-Parallelism/Project_2/problem3$ sudo ./run.sh
Write traffic only, time = 197997390.000000
Bandwidth = 11.650998 Gbps

1:1 read-to-write ratio, time = 419200297.000000
Bandwidth = 11.006038 Gbps

2:1 read-to-write ratio, time = 426202826.000000
Bandwidth = 16.237813 Gbps

Write traffic only, time = 194499425.000000
Bandwidth = 11.860535 Gbps

1:1 read-to-write ratio, time = 423545617.000000
Bandwidth = 10.893123 Gbps

2:1 read-to-write ratio, time = 420133176.000000
Bandwidth = 16.472400 Gbps

Write traffic only, time = 201204609.000000
Bandwidth = 11.465280 Gbps

1:1 read-to-write ratio, time = 417484598.000000
Bandwidth = 11.051269 Gbps

2:1 read-to-write ratio, time = 403775640.000000
Bandwidth = 17.139720 Gbps

Write traffic only, time = 183448913.000000
Bandwidth = 12.574984 Gbps

1:1 read-to-write ratio, time = 398844603.000000
Bandwidth = 11.567749 Gbps

2:1 read-to-write ratio, time = 403496151.000000
Bandwidth = 17.151593 Gbps

Write traffic only, time = 181126822.000000
Bandwidth = 12.736199 Gbps

1:1 read-to-write ratio, time = 396900165.000000
Bandwidth = 11.624420 Gbps

2:1 read-to-write ratio, time = 416143538.000000
Bandwidth = 16.630323 Gbps
```

Analysis:
The result matches my expectation. Since the array size is larger than the largest cache, the code has to fetch data from main memory which would worsen the bandwidth.

Problem 4

How to run my code:

./matrix_multiplication i-j-k
./matrix_multiplication j-k-i
./matrix_multiplication i-k-j
./matrix_multiplication loop-tilling

Three ordering

In file */proc/cpuinfo* I found that cache block size is 64B as below:

```
cache_alignment : 64
```

The element size is 8B so technically misses per iteration should be:
i-j-k: 1.125
j-k-i: 2
i-k-j: 0.25
And below is the result of execution:

```
js896@vcm-16679:~/ECE-565-Performance-Optimization-Parallelism/Project_2/problem4$ ./matrix_multiplication i-j-k
Ordering: i-j-k, time = 3.17047
js896@vcm-16679:~/ECE-565-Performance-Optimization-Parallelism/Project_2/problem4$ ./matrix_multiplication j-k-i
Ordering: j-k-i, time = 18.3944
js896@vcm-16679:~/ECE-565-Performance-Optimization-Parallelism/Project_2/problem4$ ./matrix_multiplication i-k-j
Ordering: i-k-j, time = 0.592516
```

The result matches my expectation.

Loop Tilling

By using *lscpu* command, I found that the size of L2 cache is 1024KB as below:

```
L2 cache:                1024K
```

So, I designed the sub-block size to be 1024. In this way 1024 * 8B = 8192B < 1024KB, it can fit within the L2 cache.

Below is the execution result:

```
js896@vcm-16679:~/ECE-565-Performance-Optimization-Parallelism/Project_2/problem4$ ./matrix_multiplication i-j-k
Ordering: i-j-k, time = 3.11283
js896@vcm-16679:~/ECE-565-Performance-Optimization-Parallelism/Project_2/problem4$ ./matrix_multiplication loop-tilling
Loop tilling, time = 3.33249
js896@vcm-16679:~/ECE-565-Performance-Optimization-Parallelism/Project_2/problem4$ ./matrix_multiplication i-j-k
Ordering: i-j-k, time = 3.45067
js896@vcm-16679:~/ECE-565-Performance-Optimization-Parallelism/Project_2/problem4$ ./matrix_multiplication loop-tilling
Loop tilling, time = 3.28973
js896@vcm-16679:~/ECE-565-Performance-Optimization-Parallelism/Project_2/problem4$ ./matrix_multiplication i-j-k
Ordering: i-j-k, time = 3.4164
js896@vcm-16679:~/ECE-565-Performance-Optimization-Parallelism/Project_2/problem4$ ./matrix_multiplication loop-tilling
Loop tilling, time = 3.30082
```

Analysis:

We can see there is almost no improvement by performing loop-tilling loop transformation. Here are my guesses:

It is because we limit N to 1024, which also sets the upper limit of sub-block size to 1024. If N could be bigger, then with a bigger chunk of sub-block maybe the code can make better use of the caches.

Please enlighten me with why this would happen, Yucheng or Yihao.