ECE565 HW2

Problem 1

address	Tag	Index	Set 0		Set 1		Set 2		Set 3		Status
			way0	way1	way0	way1	way0	way1	way0	way1	
1ABCDE	1ABC	3							1ABC		Miss
14327	143	0	143						1ABC		Miss
DF148	DF1	1	143		DF1				1ABC		Miss
8F220	8F2	0	143	8F2	DF1				1ABC		Miss
CDE488	CDE	1	143	8F2	DF1	CDE			1ABC		Miss
1432F	143	0	143	8F2	DF1	CDE			1ABC		Hit
5C22	5C	0	143	5C	DF1	CDE			1ABC		Miss
1ABCF2	1ABC	3	143	5C	DF1	CDE			1ABC		Hit
92D183	92D	2	143	5C	DF1	CDE	92D		1ABC		Miss
F125C	F12	1	143	5C	F12	CDE	92D		1ABC		Miss

final cache content: 1432F, 5C22, F125C, CDE488, 92D183, 1ABCF2

Problem 2

L_2 : $t_{avg_2} = 15 + 0.3 * 300 = 105$ cpu cycle

L_1 : $t_{avg_1} = 1 + 0.03 * t_{avg_2} = 4.15$ cpu cycle

L_2 : $t_{avg_2} = 15 + 0.05 * 300 = 30$ cpu cycle

L_1 : $t_{avg_1} = 1 + 0.1 * t_{avg_2} = 4$ cpu cycle

Problem 3

(c)

```
void init_array() {
  int i = 0;
  for(i = 0; i < num_elements; ++i) {
    array[i] = rand();
  }
}

int main(int argc, char *argv[]) {
  // ...
  // array initialization (size is given as input argument)
  array = (uint32_t*)malloc(num_elements * sizeof(uint32_t));
  if(mode == 2 || mode == 3) {
    init_array();
  }

  // operations of three modes
  if(mode == 0) {
      clock_gettime(CLOCK_MONOTONIC, &start_time);
      for(i = 0; i < num_traversals; ++i) {
        for(j = 0; j < num_elements; j+=1) {
          array[j] = 500;   // just pick a number
        }
      }
      clock_gettime(CLOCK_MONOTONIC, &end_time);
    }
    else if(mode == 1) {
      int val1;
      clock_gettime(CLOCK_MONOTONIC, &start_time);
      for(i = 0; i < num_traversals; ++i) {
```

```
        for(j = 0; j < num_elements-1; j+=1) {
          array[j] = 500;
          val1 = array[j+1];
        }
      }
      clock_gettime(CLOCK_MONOTONIC, &end_time);
    }
    else if(mode == 2) {
      int val1, val2;
      clock_gettime(CLOCK_MONOTONIC, &start_time);
      for(i = 0; i < num_traversals; ++i) {
        for(j = 0; j < num_elements-2; j+=1) {
          array[j] = 500;
          val1 = array[j+1];
          val2 = array[j+2];
        }
      }
      clock_gettime(CLOCK_MONOTONIC, &end_time);
    }

  // calculate bandwith
  double elapsed_ns = calc_time(start_time, end_time);
    printf("Time=%f\n", elapsed_ns);
    long access_count = 0;
    printf("num_traversals: %ld\n", num_traversals);
    printf("num_elements: %ld\n", num_elements);
    switch(mode) {
      case 0:
        access_count = num_traversals * num_elements;
        break;
      case 1:
        access_count = num_traversals * num_elements * 2;
        break;
      case 2:
        access_count = num_traversals * num_elements * 3;
        break;
    }
    printf("access_count: %ld\n", access_count);
    printf("Bandwith = %fGB/s\n", (float)(access_count * 4 / pow(2, 30)) /
(elapsed_ns / pow(10, 9)));
}
```

In order to measure bandwidth, we need to execute independent operations so that as much as data read/write can be on flight. For **write only mode**, I simply write a fixed number to each element of array. For **mode 2**, I put one read operation and one write operation inside the for loop. For **mode 3**, I put two read operations and one write operation inside the for loop. All operations inside loop are

independent.

After looking up cpu information, I find the size of L1 cache is 32K, which can hold an int array with almost 8192 elements. So I experiment the bandwidth with increasing array size.

```
jm668@vcm-16494:~/ECE565/hw2/problem3$ lscpu
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_perfmon nopl tsc_reliable nonstop_tsc cpuid pn
```

cpu info

How to run the program:

Using make to create executable file. Running this program with format:

```
./bandwith_test <size of array> <iteration times> <mode>
```

```
jm668@vcm-16494:~/ECE565/hw2/problem3$ ./bandwith_test 1000 1000000 0
Time=84805332.000000
num_traversals: 1000000
num_elements: 1000
access_count: 1000000000
Bandwith = 43.927548GB/s
jm668@vcm-16494:~/ECE565/hw2/problem3$ ./bandwith_test 2000 1000000 0
Time=161932100.000000
num_traversals: 1000000
num_elements: 2000
access_count: 2000000000
Bandwith = 46.010523GB/s
jm668@vcm-16494:~/ECE565/hw2/problem3$ ./bandwith_test 3000 1000000 0
Time=260503671.000000
num_traversals: 1000000
num_elements: 3000
access_count: 3000000000
Bandwith = 42.901011GB/s
jm668@vcm-16494:~/ECE565/hw2/problem3$ ./bandwith_test 8000 1000000 0
Time=663945758.000000
num_traversals: 1000000
num_elements: 8000
access_count: 8000000000
Bandwith = 44.886682GB/s
jm668@vcm-16494:~/ECE565/hw2/problem3$ ./bandwith_test 8000 1000000 0
Time=628126056.000000
num_traversals: 1000000
num_elements: 8000
access_count: 8000000000
Bandwith = 47.446404GB/s
jm668@vcm-16494:~/ECE565/hw2/problem3$ ./bandwith_test 9000 1000000 0
Time=907118570.000000
num_traversals: 1000000
num_elements: 9000
access_count: 9000000000
Bandwith = 36.960560GB/s
```

Testing with Different Size

From the test result, it can be found that the bandwidth reaches peak value when array size is around 8000. When the size reaches 9000, some data must be put on L2 cache, which sharply decrease the bandwidth.

```
jm668@vcm-16494:~/ECE565/hw2/problem3$ ./bandwith_test 8000 1000000 0
\Time=670326682.000000
num_traversals: 1000000
num_elements: 8000
access_count: 8000000000
Bandwith = 44.459400GB/s
jm668@vcm-16494:~/ECE565/hw2/problem3$ ./bandwith_test 8000 1000000 1
Time=627375231.000000
num_traversals: 1000000
num_elements: 8000
access_count: 16000000000
Bandwith = 95.006372GB/s
jm668@vcm-16494:~/ECE565/hw2/problem3$ ./bandwith_test 8000 1000000 2
Time=633258859.000000
num_traversals: 1000000
num_elements: 8000
access_count: 24000000000
Bandwith = 141.185498GB/s
```

Test with Three Modes

Write operation is slower than read operation, because of more instruction involved. And the result of testing also approves this. **Mode3 is faster than mode2, and mode2 is faster than mode1. Mode with higher read proportion is faster.**

(d)

According to cpu information, size of L3 cache is 22528K. So I changed the size of array to be 6000000, which is larger than L3 cache. The test result is shown below. The bandwidth is much smaller than only accessing L1 cache, which matches my expectation.

```
jm668@vcm-16494:~/ECE565/hw2/problem3$ ./bandwith_test 6000000 10000 0
Time=21165763049.000000
num_traversals: 10000
num_elements: 6000000
access_count: 60000000000
Bandwith = 10.560329GB/s
jm668@vcm-16494:~/ECE565/hw2/problem3$ ./bandwith_test 6000000 10000 1
Time=22009313316.000000
num_traversals: 10000
num_elements: 6000000
access_count: 120000000000
Bandwith = 20.311168GB/s
jm668@vcm-16494:~/ECE565/hw2/problem3$ ./bandwith_test 6000000 10000 2
Time=22040767183.000000
num_traversals: 10000
num_elements: 6000000
access_count: 180000000000
Bandwith = 30.423272GB/s
```

Problem 4

(b)

Using make to create executable file. Running this program with format:

```
./bandwith_test <looptype (ijk or jki or ikj or ijk_tiling)>
```

Code snippet of I-J-K, J-K-I, and I-K-J loop.

```
void exec_ijk() {
  int i, j, k;
  double sum;
  clock_gettime(CLOCK_MONOTONIC, &start_time);
  for (i=0; i<N; i++) {
      for (j=0; j<N; j++) {
          sum = 0;
          for (k=0; k<N; k++) {
              sum += A[i][k] * B[k][j];
          }
          C[i][j] = sum;
    }
  }
  clock_gettime(CLOCK_MONOTONIC, &end_time);
}

void exec_jki() {
  int i, j, k;
  double tmp;
  clock_gettime(CLOCK_MONOTONIC, &start_time);
  for (j=0; j<N; j++) {
      for (k=0; k<N; k++) {
          tmp = B[k][j];
          for (i=0; i<N; i++) {
              C[i][j] += tmp * A[i][k];
      }
    }
  }
  clock_gettime(CLOCK_MONOTONIC, &end_time);
}

void exec_ikj() {
  int i, j, k;
  double tmp;
  clock_gettime(CLOCK_MONOTONIC, &start_time);
  for (i=0; i<N; i++) {
      for (k=0; k<N; k++) {
          tmp = A[i][k];
```

```
            for (j=0; j<N; j++) {
                C[i][j] += tmp * B[k][j];
        }
      }
    }
    clock_gettime(CLOCK_MONOTONIC, &end_time);
}
```

Test result of three nested loop:

```
jm668@vcm-16494:~/ECE565/hw2/problem4$ ./matrix_test ijk
Total Time = 1.877610 seconds
jm668@vcm-16494:~/ECE565/hw2/problem4$ ./matrix_test jki
Total Time = 19.167754 seconds
```

```
jm668@vcm-16494:~/ECE565/hw2/problem4$ ./matrix_test ikj
Total Time = 0.476255 seconds
```

Average miss rate of one iteration:

I-J-K: 1.125; J-K-I: 2.0; I-K-J: 0.25

The results are same as expectation, where I-K-J runs fastest.

(c)

Code snippet of Loop Tiling:

```
void exec_ijk_tiling() {
  int i, j, k, ii, jj, kk;
  double sum;
  clock_gettime(CLOCK_MONOTONIC, &start_time);
  for (i=0; i<N; i+=64) {
      for (j=0; j<N; j+=64) {
        for(ii = i; ii < i+64; ++ii) {
          for(jj = j; jj < j + 64; ++jj) {
            sum = 0;
              for (k=0; k<N; k+=64) {
                for(kk = k; kk < k+64; ++kk) {
                  sum += A[ii][kk] * B[k][jj];
                }
              }
              C[ii][jj] = sum;
          }
        }
    }
  }
  clock_gettime(CLOCK_MONOTONIC, &end_time);
```

```
}
```

Since we need to fit sub-block within L2 cache, the size of sub-block should be less than 1024KB, which is the size of L2 cache. The test result is shown below:

```
jm668@vcm-16494:~/ECE565/hw2/problem4$ ./matrix_test ijk_tiling
Total Time = 1.286943 seconds
```

Applying loop tiling on I-J-K is faster than original I-J-K and J-K-I, but is still slower than I-K-J. This loop transformation can first fetch blocks into L2 cache, which makes later calculation faster. However, this optimization still cannot remedy the time difference caused by miss rate, so it is still slower than I-K-J.