HW 2
Rijish Ganguly
Performance and Parallelism
ECE 565

1. As we have 64B blocks, the offset requires 6 bits. Again, with 64B cache blocks with 512B total cache size, that means 8 blocks. As we have 8 blocks with 2-way set association, there will be 4 sets, thus requiring 2 set bits for the index. We have a 20 bit address minus and hence we would require 12 bits for the tag.

Hex Address	Tag	Set	Offset	Hit/Miss
0xABCDE.	101010111100	11	011110	M
0x14327.	000101000011	00	100111	M
0xDF148.	110111110001	01	001000	M
0x8F220.	100011110010	00	100000	M
0xCDE4A.	110011011110	01	001010	M
0x1432F.	000101000011	00	101111	H
0x52C22.	010100101100	00	100010	M
0xABCF2	101010111100	11	110010	H
0x92DA3	100100101101	10	100011	M
0xF125C	111100010010	01	011100	M

			Way 1		Way 0
	LRU	V	Tag	V	Tag
Set 0	0	1	010100101100 (0x52C22)	1	000101000011 (0x14327)
Set 1	1	1	110011011110 (0xCDE4A)	1	111100010010 (0xF125C)
Set 2	1	0		1	100100101101 (0x92DA3)
Set 3	1	0		1	101010111100 (0XABCF2)

2. AMAT = Hit Time_{L1} + Miss-Rate_{L1} X Miss Penalty_{L1}
   Miss-Penalty_{L1} = Hit-Time_{L2} + Miss-Rate_{L2} X Miss-Penalty_{L2}

   L2 cache hit time of 20 CPU cycle and a miss penalty of 250 CPU cycles
   L1 hit time is 1 CPU cycle

   a. L1 miss rate = 0.02 and L2 miss rate = 0.40
      AAT = 1 + 0.02*(20+250*0.40) = 3.4 CPU cycles

   b. L1 miss rate = 0.10 and L2 miss rate = 0.05
      AAT = 1+0.10*(20+250*0.05) = 4.25 CPU cycles

4.

   Using **MacOS**
   a. Code has been attached.

   b.

      Time taken for i-j-k: 10.8979 seconds
      Time taken for i-k-j: 0.642497 seconds
      Time taken for j-k-i: 24.1674 seconds

   Hence, according to my code i-k-j had the best performance and the arrangement j-k-i had the worst performance which matches our expectation. Considering cache block size to be 64 bytes, for i-j-k we would have 1.125 misses per iteration and for the arrangement j-k-i we would have 2 misses per iteration. However, if j is the inner loop such as in case of i-k-j we would have 0.25 misses per iteration. Hence, for i-k-j we observe that code performance is the best because we get good temporal locality for matrix A and good spatial locality for matrices B and C

   c. syscrtl -a | grep l2 revealed:
      hw.l2cachesize: 262144 which is 262.144 KB
      Let us assume the block size to be m x m
      Then $3m^2 * 8 <= 262.144KB$
      m ~= 104.5
      Code has been attached.

d. Execution time of the tiled version: 6.48707 seconds. When we compare it to the i-j-k arrangement without loop tiling, we observe that our code became significantly faster. The performance improvement is almost 40%. This is because loop tiling improves cache reuse and makes vectorization easier. However, i-k-j arrangement still gives me the best performance among all the versions. The tiled version was the second fastest. The reason I believe i-k-j has the best performance is because having j as the inner loop improves both kind of localities and the number of misses is drastically reduced compared to other cases. The block size I chose was 64X64 in order to accommodate the L2 cache.

3.

Using **MacOS**

a. L1 cache size was found to be 32768 KB. Code has been attached.

b.

Traffic Type	Total Time (ns)	Time per operation (ns)	Throughput (GBPS)
Write	3902000	0.975500	8.200923
1:1 Read/Write	4303000	0.537875	14.873344
2:1 Read/Write	7453000	0.621083	12.880719

Traffic Type	Total Time (ns)	Time per operation (ns)	Throughput (GBPS)
Write	4735000	1.183750	6.758184
1:1 Read/Write	4277000	0.534625	14.963760
2:1 Read/Write	6907000	0.575583	13.898943 GBPS

Traffic Type	Total Time (ns)	Time per operation (ns)	Throughput (GBPS)
Write	4687000	1.171750	6.827395
1:1 Read/Write	4356000	0.544500	14.692378
2:1 Read/Write	7418000	0.618167	12.941494

c.

```
Write traffic only
Time = 3902000.000000 ns
Average ns per write = 0.975500
Bandwidth for write is = 8.200923 GBPS

1:1 Read and Write traffic
Time = 4303000.000000 ns
Average ns per operation for 1:1 read and write = 0.537875
Bandwidth for 1:1 read and write is = 14.873344 GBPS

2:1 Read and Write traffic
Time = 7453000.000000 ns
Averager ns per operation for 2:1 read and write = 0.621083
Bandwidth for 2:1 read and write is = 12.880719 GBPS
```

To run the program do the following:

bash ./build

./bandwidth

The cache size for my L1 cache was observed to be 32,768 KB. Hence, I chose a uint64 array with number of elements = 4,000,000 to be the primary data structure for this question. The size of an uint64 element is 8 bytes (4,000,000*8 = 32,000,000). Hence, data can be supplied by the L1 cache. For the write traffic bandwidth calculation, I updated each element of my array to a constant number with the help of a loop. For the 2:1 read and write band-width calculation I declared another array with 4,000,000 elements and re-wrote the i^{th} element of the first array by reading in and adding the i^{th} elements of the first and second array with the help of a loop. For 1:1 read and write bandwidth calculation I re-wrote the value of i^{th} element of the array by reading the $i+1^{th}$ element from the array itself with the help of a loop. My program stresses the bandwidth with the help of an iterating loop. According to my observation, my program achieved highest bandwidth for 1:1 read and write access pattern and lowest bandwidth for write only traffic. One interesting thing I observed was the fact that 1:1 read and write took a shorter time per operation compared to write operation. A reason behind this might be the fact that caches support read and write accesses via ports into the cache. A cache might have one port for read and one port for writes (or 2 for reads and 1 for writes). Hence, with write-only traffic, there can be up to a write every cycle into the cache. But with read-write traffic, there can be up to a read and a write every cycle. Hence, the average time per operation decreases for the latter case. 2:1 read and write took slightly more time compared to 1:1 read and write and was significantly faster than write operation.

d. The largest cache on my system is the L3 cache. The L3 cache size is 3145728 KB. In order to exceed the L3 cache I used the number of elements to be **400000000** so the cache cannot fit my entire array.

Traffic Type	Total Time (ns)	Time per operation (ns)	Throughput (GBPS)
Write	5214968000	13.037420	0.613618
1:1 Read/Write	2872175000	3.590219	2.228276
2:1 Read/Write	10671922000	8.893268	0.899557

Traffic Type	Total Time (ns)	Time per operation (ns)	Throughput (GBPS)
Write	2619044000	6.547610	1.221820
1:1 Read/Write	450564000	0.563205	14.204419
2:1 Read/Write	10658549000	8.882124	0.900685

Traffic Type	Total Time (ns)	Time per operation (ns)	Throughput (GBPS)
Write	2154599000	5.386497	1.485195
1:1 Read/Write	446548000	0.558185	14.332166
2:1 Read/Write	8215475000	6.846229	1.168526

As we increase the size of our data structure we observe that the total time taken for each access pattern increases significantly, the average time per operation increases significantly for 2:1 read and write and write access pattern and moderately for 1:1 read and write access pattern. Lastly, we observe that throughput deceases significantly for 2:1 read and write pattern and write traffic only. As cache limit is exceeded, processor has to fetch the data from the disk which is computationally expensive. The result was according to my expectations.