PAP - Assignment - 01

For this assignment, we were tasked to handle a cache related issue happening while transposing a given square matrix. There are multiple ways to implement a transpose function for a matrix which we will discuss in the following report.

1. The first approach to this problem is the naive solution. This solution is the simplest way to find the transpose of a matrix. However, it is not the smartest way to do it. The naive function consists of two nested loops which read every single element in source matrix and put them in the right place on **destination matrix**.

```
2  void naive(int *src, int *dst, int SIZE)
3  {
4      for(int i=0; i < SIZE; i++)
5          for(int j=0; j < SIZE; j++)
6              dst[(j * SIZE) + i] = src[(i * SIZE) + j];
7  }
8
```

   This solution has two major cache related problems which makes the function too slow to be practical.

   - The first problem is that this algorithm results too many capacity misses due to the **special locality** trait of cache.
     Let us take a look at the following example of a 4x4 matrix with a 2x4 cache with replacement method ***LRU***:

Source

| 1 | 2 | 3 | 4 |
|---|---|---|---|
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 |

Destination

| 1 | 2 | 3 | 4 |
|---|---|---|---|
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 |

The first element of the matrix is called (S1). Due to special locality (S2) is also called to cache.
(Compulsory Cache miss)
LRU -> 1

| S1 | S2 |
|---|---|
| | |
| | |
| | |

(S1) is going to be written in destination (D1). Therefore, first block of destination is Brought to the cache.
(Compulsory Cache miss)

LRU -> 2,1

| S1 | S2 |
|---|---|
| D1 | D2 |
| | |
| | |

Now (S2) is going to be written, it already exist in the cache
(Cache hit).
(S2) is going to be written in destination (D5). Therefore, (D5) is called from destination.
(Compulsory Cache miss happens)

LRU -> 3,1,2

| S1 | S2 |
|---|---|
| D1 | D2 |
| D5 | D6 |
| | |

Next, (S3) is called to the cache.
(Compulsory Cache miss)

LRU -> 4,3,1,2

| S1 | S2 |
|---|---|
| D1 | D2 |
| D5 | D6 |
| S3 | S4 |

(S3) is going to be written in (D9).
Using LRU method, (D9) replaces the second row of our cache.
(Compulsory Cache miss)

LRU -> 2,4,3,1

| S1 | S2 |
|---|---|
| D9 | D10 |
| D5 | D6 |
| S3 | S4 |

(S4) already exists in the cache
(Cache hit).
It is going to be written in (D13). Using LRU method, (D13) replaces the first row. (Compulsory Cache miss)

LRU -> 1,2,4,3

| D13 | D14 |
|---|---|
| D9 | D10 |
| D5 | D6 |
| S3 | S4 |

This is time to transpose the second row of source matrix. (S5) is called from memory. Using LRU method, (S5) replaces the third row.
(Compulsory Cache miss)

LRU -> 3,1,2,4

| D9 | D10 |
|---|---|
| D13 | D14 |
| S5 | S6 |
| S3 | S4 |

(S5) is about to be written in (D2). We already called D2 to cache once before. We are recalling it once again from memory. Using LRU method (D2) will replace the forth row of the cache. (Capacity miss).

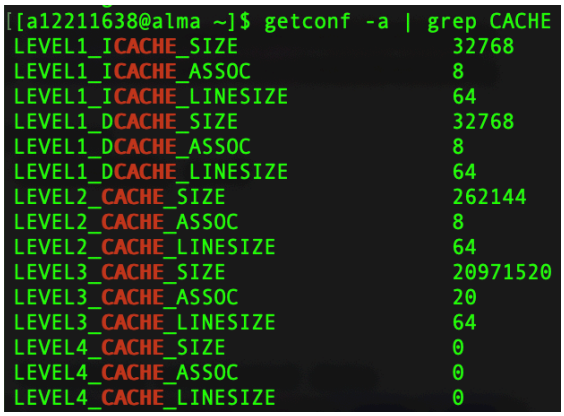
LRU -> 4,3,1,2

| D9 | D10 |
|---|---|
| D13 | D14 |
| D5 | D6 |
| D2 | D3 |

From here, on even columns (every second column) of destination matrix on every single call on destination matrix a capacity miss happens because we already read these blocks before (due to special locality).

- The second problem is an increment to the first problem. Since we are using a 8way set-associative cache in **ALMA,** we end up getting more conflict misses because each memory block is mapped to a certain cache set.

2. The second approach is called Cache-Aware solution. It is called Cache-Aware because we already know the size of our cache and we use it in our algorithm.
   For this assignment, we simply pull the information on **ALMA** cache using "getconf -a | grep CACHE" command in **ALMA**'s terminal.




   Size of Cache : 32768
   Line Size : 64
   8way set-associative

   (Note: I think that since the cache line size is 64 and we are using 8way set-associative architecture in **ALMA**, 8 would be the best tile size since there will be 16 integer in each line and each set consists of 8 lines. We have to call both source and destination matrices from the main memory. Smaller tile size means more access to the main memory which is expensive and bigger tile size means more cache miss occurrences.)

   Speed in Cache-Aware approach:

| t=2 | t=4 | t=8 | t=16 | t=32 | t=64 | t=128 | t=256 | t=512 |
|---|---|---|---|---|---|---|---|---|
| 13.2 | 6.57 | 5.26 | 6.34 | 6.25 | 5.86 | 5.86 | 5.86 | 24.19 |

   To find the transpose matrix with Cache-Aware all we have to do is use this information. The problem with naive approach is that usually we get large matrices and there is not enough space in our cache. Therefore, we have to keep calling and replacing data blocks in our cache. In this approach on the other hand, we call data blocks with the size of our cache lines, use all the data available in the cache, then replace it with a new group of data blocks.

```
10  void aware(int *src, int *dst, int SIZE)
11  {
12      int t = 8;
13
14      for (int i = 0; i < SIZE; i += t) {
15          for (int j = 0; j < SIZE; j += t) {
16              for (int k = i; k < i + t; ++k) {
17                  for (int l = j; l < j + t; ++l) {
18                      dst[k + l*SIZE] = src[l + k*SIZE];
19                  }
20              }
21          }
22      }
23  }
```

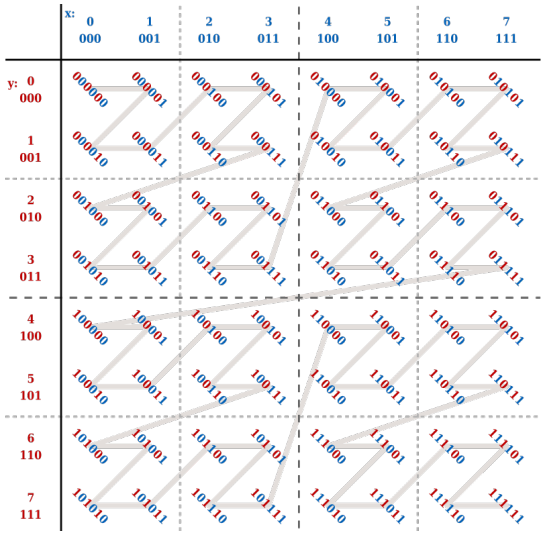
3. The third approach is Oblivious solution. The Intuition behind this approach is to find a way to avoid the cache problem without knowing any prior information about our cache. We can implement this solution with using the Z-Curve (Morton Code). Z-Curve will allow us to read the matrix in way that the minimum amount of cache misses could occur. This also fixes the column-major problem in the destination matrix. To do this we will use Morton Decoding. For this assignment Magic Number Morton Decoding was used.

```
26  uint32_t morton_decode(uint32_t z)
27  {
28      z = z & 0x55555555;
29      z = (z | (z >> 1)) & 0x33333333;
30      z = (z | (z >> 2)) & 0x0F0F0F0F;
31      z = (z | (z >> 4)) & 0x00FF00FF;
32      z = (z | (z >> 8)) & 0x0000FFFF;
33      return z;
34  }
```

Now, we simply use the Morton-Decode to read the matrix with Z-Curve instead of Row-major like in the naive solution.

```
38  void oblivious(int *src, int *dst, int SIZE)
39  {
40      for(uint32_t z=0; z < SIZE*SIZE; z++)
41      {
42          uint32_t x = morton_decode(z);
43          uint32_t y = morton_decode(z >> 1);
44
45          dst[(y * SIZE) + x] = src[(x * SIZE) + y];
46
47      }
48  }
```

Extra explanation on Z-Curve: Z-Curve algorithm uses a third variable called Z. This Variable stores both I and j in one binary number. In the picture you can see that each red number (even counters) are constructing j (y) and each blue number (odd counters) are constructing I (x) from variable z. This also helps us to traverse the matrix with only one loop since we only have one variable to increment. Also, because of the way we are traversing the matrix, we get less cache misses both in source and destination matrices.




From wikipedia

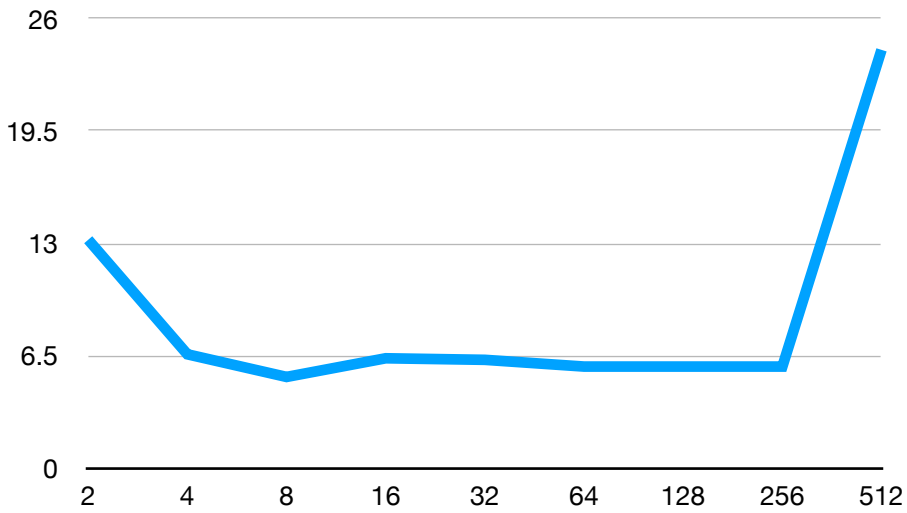
4. The forth and the final approach is to use a combination of both Cache-Aware and Oblivious solutions. The intuition behind this solution was to use the knowledge we have about our cache and the power of Z-Curve together. Since we are not reading the tiles in a column-major order anymore, number of cache misses go down considerably.

```
51  void optimized(int *src, int *dst, int SIZE)
52  {
53
54      int t = 8;
55
56      for (int z = 0; z < SIZE*SIZE; z += t*t) {
57
58          uint32_t i = morton_decode(z);
59          uint32_t j = morton_decode(z >> 1);
60
61          for (int k = i; k < i + t; ++k) {
62              for (int l = j; l < j + t; ++l) {
63                  dst[k + l*SIZE] = src[l + k*SIZE];
64              }
65          }
66      }
67  }
```

(Note: I think the reason we get a good time using this approach is the size of cache line and our tile size. Each tile has a size of 8 but in **ALMA** we can store 16 integers in each line. Since we are using Z-Curve to read the tiles, We are getting the tile we are using currently and the tile we are going to use next. Therefore, with each incrementation on z, we are reading two tiles and that makes the miss rate get even less that cache-aware approach. Instead of getting compulsory cache misses on every second column of tiles, we get cache hits.)

Speed Comparison:

1. Naive: 25.36 seconds
2. Cache-Aware: Almost 5x speedup with t=8 compared to Naive.




3. Oblivious: 7.87 seconds. 3x speedup compared to Naive.
4. Optimised: 2.88 seconds. Almost 9x speedup compared to Naive.

Note: With a small change in Oblivious and Optimised functions we can even achieve higher speedups. Z-Curve reads 2 data blocks. We can use this in a way to not repeat the memory calls even further.

new-Oblivious: 7.56 seconds.
new-Optimised: 2.72 seconds.

```
53  void oblivious(int *src, int *dst, int SIZE)
54  {
55      for(uint32_t z=0; z < SIZE*SIZE; z += 2)
56      {
57          uint32_t x = morton_decode(z);
58          uint32_t y = morton_decode(z >> 1);
59
60          dst[(y * SIZE) + x] = src[(x * SIZE) + y];
61
62          uint32_t xx = morton_decode(z+1);
63          uint32_t yy = morton_decode(z+1 >> 1);
64
65          dst[(yy * SIZE) + xx] = src[(xx * SIZE) + yy];
66
67      }
68  }
```

```
90  void optimized(int *src, int *dst, int SIZE)
91  {
92
93      int t = 8;
94
95      for (int z = 0; z < SIZE*SIZE; z += 2*(t*t)) {
96
97          uint32_t i = morton_decode(z);
98          uint32_t j = morton_decode(z >> 1);
99
100         for (int k = i; k < i + t; ++k) {
101             for (int l = j; l < j + t; ++l) {
102                 dst[k + l*SIZE] = src[l + k*SIZE];
103             }
104         }
105
106         uint32_t ii = morton_decode(z+t*t);
107         uint32_t jj = morton_decode(z+t*t >> 1);
108
109         for (int k = ii; k < ii + t; ++k) {
110             for (int l = jj; l < jj + t; ++l) {
111                 dst[k + l*SIZE] = src[l + k*SIZE];
112             }
113         }
114     }
115 }
```

NUMA-Aware: At current processor speeds, the signal path length from the processor to memory plays a significant role. Increased signal path length not only increases latency. To memory but also quickly becomes a bottleneck if the signal path is shared by multiple processors. Today's processors are so fast that they require memory to be directly attached to the socket that they are on. A main memory access has additional latency overhead. On the other hand, access from a single processor to local memory not only have lower latency but do not cause contention. It is good to avoid remote memory accesses. Proper placement of data improves the latency to main memory.
In this assignment we are focusing on optimising local memory usage. By reducing accesses to the main memory we managed to get 9x speedup. Also, it is worth mentioning that our program is using one core and one thread, so there is no need to worry about data sharing between cores.