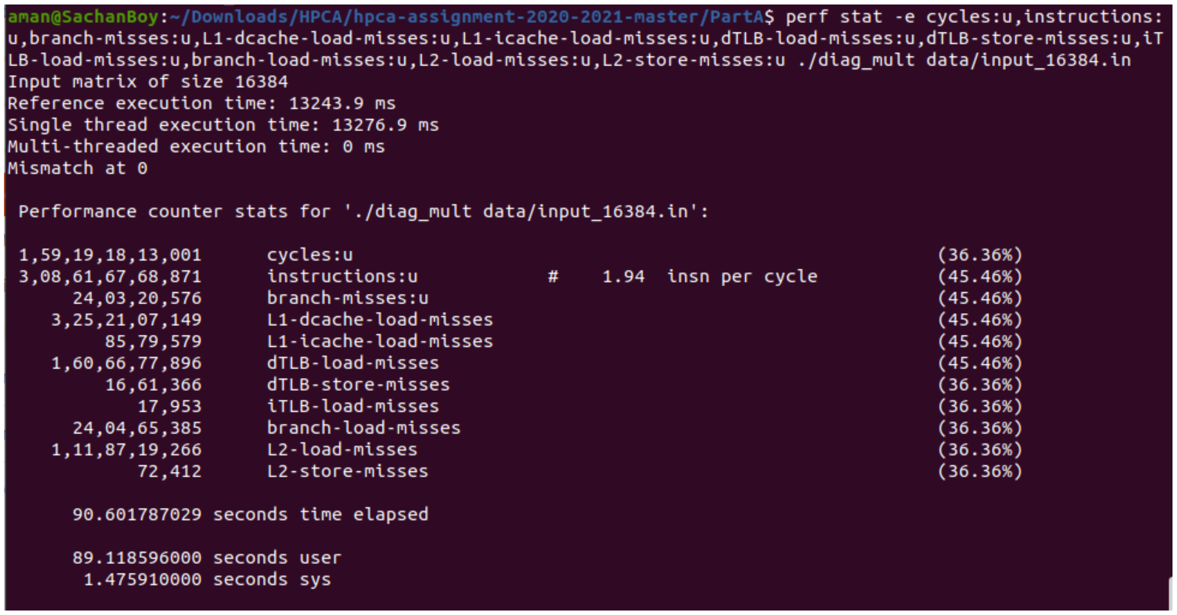
E0 243: Programming assignment: Part 1

Name: Aman Sachan **Due Date:** 17th Jan
S.R No.: 18094 **Program:** MTech-D

Part A-I (Single Threading)

I have done all the computations in the `Intel(R) Core(TM) i3-6006U CPU @ 2.00GHz processor`. To check the number of different misses occurred due to the given code, I used the command for input size 16384 :

```
perf stat -e cycles:u,instructions:u,branch-misses:u,L1-dcache-load-misses:u,
L1-icache-load-misses:u,dTLB-load-misses:u,dTLB-store-misses:u,iTLB-load-miss
es:u,branch-load-misses:u,L2-load-miss es:u,L2-store-misses:u ./diag_mult dat
a/input_16384.in
```



```
aman@SachanBoy:~/Downloads/HPCA/hpca-assignment-2020-2021-master/PartA$ perf stat -e cycles:u,instructions:
u,branch-misses:u,L1-dcache-load-misses:u,L1-icache-load-misses:u,dTLB-load-misses:u,dTLB-store-misses:u,iT
LB-load-misses:u,branch-load-misses:u,L2-load-misses:u,L2-store-misses:u ./diag_mult data/input_16384.in
Input matrix of size 16384
Reference execution time: 13243.9 ms
Single thread execution time: 13276.9 ms
Multi-threaded execution time: 0 ms
Mismatch at 0

 Performance counter stats for './diag_mult data/input_16384.in':

 1,59,19,18,13,001      cycles:u                                                      (36.36%)
 3,08,61,67,68,871      instructions:u            #    1.94  insn per cycle           (45.46%)
       24,03,20,576      branch-misses:u                                              (45.46%)
    3,25,21,07,149      L1-dcache-load-misses                                         (45.46%)
         85,79,579      L1-icache-load-misses                                         (45.46%)
    1,60,66,77,896      dTLB-load-misses                                              (45.46%)
         16,61,366      dTLB-store-misses                                             (36.36%)
            17,953      iTLB-load-misses                                              (36.36%)
       24,04,65,385      branch-load-misses                                           (36.36%)
    1,11,87,19,266      L2-load-misses                                                (36.36%)
            72,412      L2-store-misses                                               (36.36%)

      90.601787029 seconds time elapsed

      89.118596000 seconds user
       1.475910000 seconds sys
```

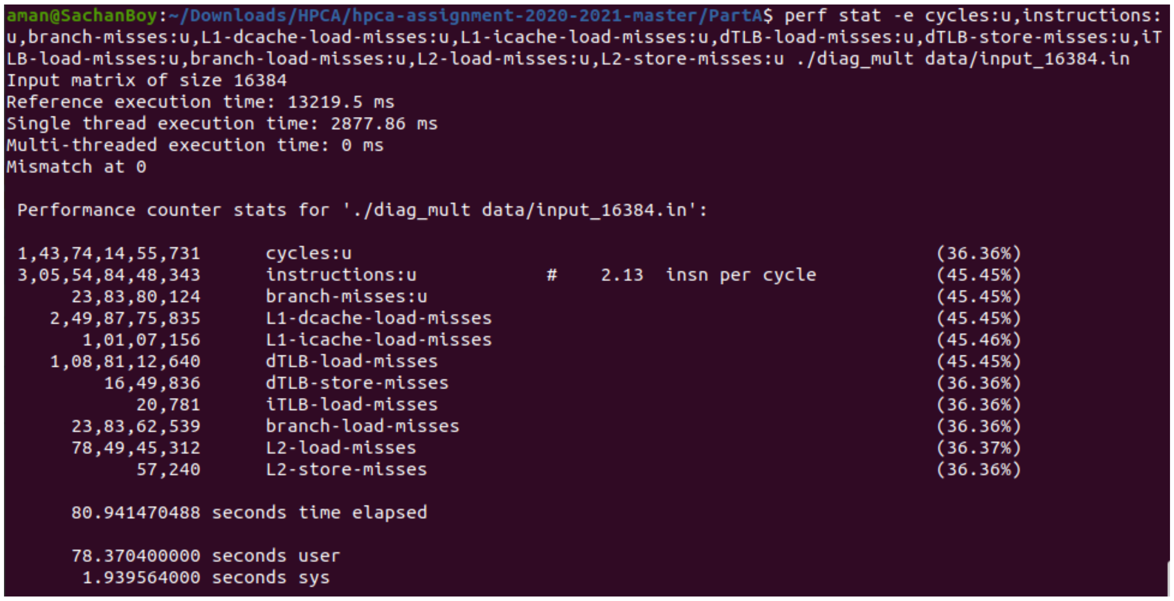
In this we have a lot of `L1-dcache-load-misses`. So, we will try to optimize this. For this we must know the size of L1 cache. To know the size of L1 cache `getconf -a | grep CACHE`. I found out the `LEVEL1_DCACHE_LINESIZE` as 64 Bytes and every integer takes 4 Bytes.

I used the concept of blocking and loop unfolding to optimize the code execution time. I accessed the **matrixA** in row order fashion but **matrixB** in column order. So, while accessing a element if it is not in the L1 cache it will bring from the lower level caches or memory. But, it will not bring only that element, it will fetch one block i.e, of 16 elements (because, cache line is of 64 bytes and 1 integer takes 4 bytes).

If we directly fetch 1 element from each column then due to constraint in CACHE SIZE, 16 cache lines will be accommodated in the cache but when 17^{th} cache block is accessed then first block is evicted from the cache and we wasted 15 elements, which are evicted without doing its work and we have to again fetch that cache block. So, it will increase the misses as well as execution time.

So, from the start of the execution of the program we will fetch 16 elements of matrixB each from different row. Then the 16*16 element will be fetched into the cache. And in the matrixA we will fetch in row-wise order. So, we will fetch the first element then process all the 16 elements of the fetched row (i.e, in a block) after that 16 elements from the next row (i.e, in a next block).

Due to this the number of hits will increase. In a 16*16 elements fetching we have 2*16*16 - 2*16 hits. The misses after this implementation :



```
aman@SachanBoy:~/Downloads/HPCA/hpca-assignment-2020-2021-master/PartA$ perf stat -e cycles:u,instructions:
u,branch-misses:u,L1-dcache-load-misses:u,L1-icache-load-misses:u,dTLB-load-misses:u,dTLB-store-misses:u,iT
LB-load-misses:u,branch-load-misses:u,L2-load-misses:u,L2-store-misses:u ./diag_mult data/input_16384.in
Input matrix of size 16384
Reference execution time: 13219.5 ms
Single thread execution time: 2877.86 ms
Multi-threaded execution time: 0 ms
Mismatch at 0

 Performance counter stats for './diag_mult data/input_16384.in':

 1,43,74,14,55,731      cycles:u                                                    (36.36%)
 3,05,54,84,48,343      instructions:u            #    2.13  insn per cycle         (45.45%)
      23,83,80,124      branch-misses:u                                             (45.45%)
    2,49,87,75,835      L1-dcache-load-misses                                       (45.45%)
       1,01,07,156      L1-icache-load-misses                                       (45.46%)
    1,08,81,12,640      dTLB-load-misses                                            (45.45%)
         16,49,836      dTLB-store-misses                                           (36.36%)
            20,781      iTLB-load-misses                                            (36.36%)
      23,83,62,539      branch-load-misses                                          (36.36%)
      78,49,45,312      L2-load-misses                                              (36.37%)
            57,240      L2-store-misses                                             (36.36%)

      80.941470488 seconds time elapsed

      78.370400000 seconds user
       1.939564000 seconds sys
```

`L1-dcache-load-misses` is reduced drastically and our other optimization is unfolding.
The instructions be like $x += y * z$, and if we don't do unfolding then addition functional unit will have to wait for the completion of multiplication. So, we have unfolded the last loop completely. And due to OoO(Out of Order) execution. It will decrease the execution time.

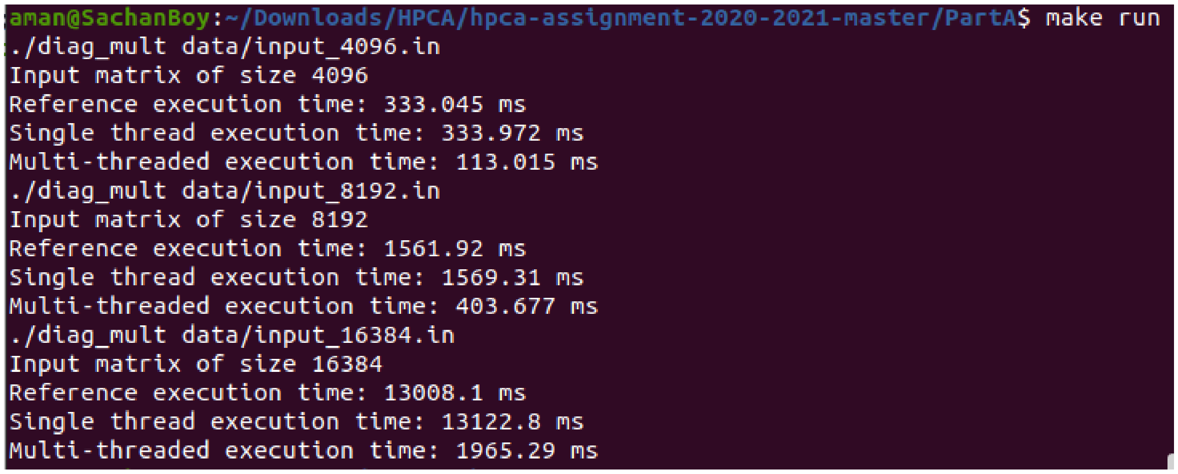
Part A-II (Multi Threading)

I have done all the computations in the `Intel(R) Core(TM) i3-6006U CPU @ 2.00GHz processor`. Number of threads per cores = 2 and I have tested the execution time in different combination of *number of threads* and *block size*. I got the best result in number of threads = 256 and block size = 64.

If the number of threads per core and the cores are increased then the execution time will decrease. This means that the program is scalable. As well as we have done loop unrolling (unfolding) which reduces the execution time.

We have made a personalised array for each thread of the size of the output array. If it computes the result for some of the index of the output array. It will store in their personalised array. So, that there is no need of synchronization. At the last we will wait for all the threads to complete its execution and will sum up the computed values for the same index value of output.

Below are the results of the implementation of multi-threading i.e, the execution time for different size of input.



```
aman@SachanBoy:~/Downloads/HPCA/hpca-assignment-2020-2021-master/PartA$ make run
./diag_mult data/input_4096.in
Input matrix of size 4096
Reference execution time: 333.045 ms
Single thread execution time: 333.972 ms
Multi-threaded execution time: 113.015 ms
./diag_mult data/input_8192.in
Input matrix of size 8192
Reference execution time: 1561.92 ms
Single thread execution time: 1569.31 ms
Multi-threaded execution time: 403.677 ms
./diag_mult data/input_16384.in
Input matrix of size 16384
Reference execution time: 13008.1 ms
Single thread execution time: 13122.8 ms
Multi-threaded execution time: 1965.29 ms
```

Conclusion : Speed-up for the single and multi threading implementation in the specified system configuration.

- Speed Up for single threading is around 5 times.
- Speed Up for single threading is around 7 times.