Computer Organisation
Name: Johri Aniket Manish
Roll.no: CS22B028
IIT Tirupati

Lab: 01

This report answers the observation, explanation and runtime of problem set sequentially.

1. *Compile the attached program ast.cpp using the commands g++ -O0 -o astO0.out ast.cpp and g++ -O3 -o astO3.out ast.cpp. Compare the sizes of the resulting binaries and their runtimes..*

```
Runtime for 10 observations for
no optimization(microsec):  O3 optimization:        O3 (in nanosec):
267204                      0                       279
252088                      0                       245
252881                      0                       220
250794                      0                       216
252604                      0                       271
252288                      0                       281
251018                      0                       252
253841                      0                       223
252286                      0                       204
253421                      0                       272

Size of binary of O0 is 18152 bytes.
Size of binary of o3 is 16416 bytes.
The size of both the files are almost same because of same structure of code.
The only difference is of the different optimizations of the code.

We can observe the runtime for each of 10 observations. The O3 optimization in
microsec is very small and so it is showing 0 microsec but setting upon
correct precision in nanosec, we can see the above results. Since the time is
very small, it was showing 0 microseconds above.
```

2. *Add a statement to print the value of the sum (at the end of the computation) in the code. Compile and compare the runtimes.*

```
Runtime for 10 observations for
no optimization(microsec):          O3 optimization(microsec):
236844                              121246
0                                   0
216478                              88561
0                                   0
219338                              94320
0                                   0
```

```
219756                          103829
0                               0
232086                          103509
0                               0
218852                          106779
0                               0
217085                          92813
0                               0
218852                          87187
0                               0
226132                          92607
0                               0
217188                          93523
0                               0
```

The following numbers represent the time duration and sum respectively.

Since the matrix was never initialised, the answer of the sum for each case would be 0. But now this time O3 optimization shows significant time in microseconds which is in contrast to the previous case. Now this is because the printing statement involves i/o operation and it is slower than the computation operation process. Moreover, it also occupies some additional space in memory which makes is slower as compared to previous case where only computation was involved.

3. *Interchange i and j when accessing the array and compile with -O0. Run each version of the code atleast ten times to calculate the average runtime. Compare the runtimes between the two versions, change the value of N, and repeat the experiment. Finally, plot a graph with N on the x-axis and Time on the y-axis.*

   The graph for Iteration(N) v/s Average Rumtime(in microsec) is shown below.

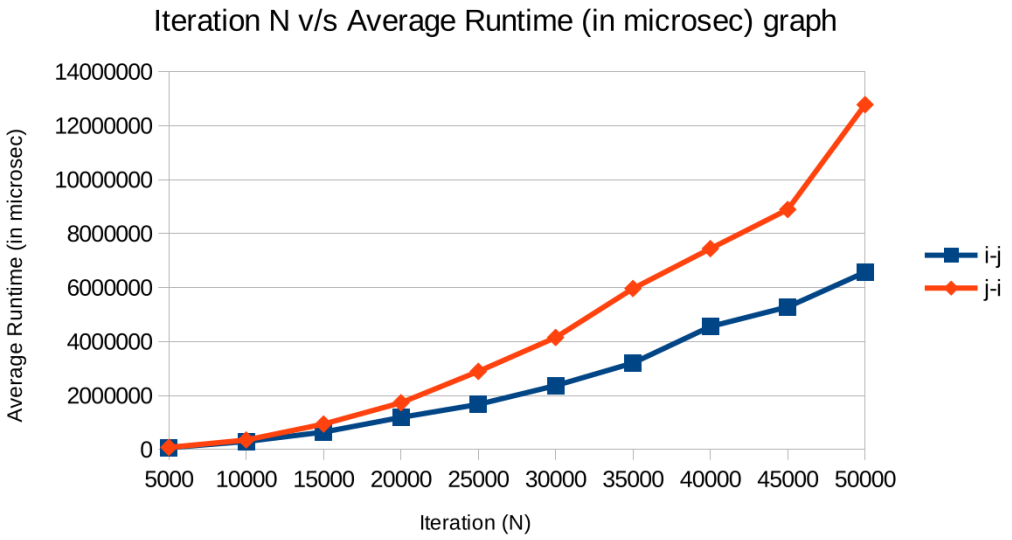

Figure 1: Iteration(N) v/s Average Rumtime(in microsec)

From the trend in the graph, it can be seen that it is an increasing graph which shows that the average runtime increases as the value of N increases which is indeed true as the number of computations increases.

Now i-j represents the row-wise traversal of 2D matrix and
j-i represents the column-wise traversal of 2D matrix.

It can be seen from the graph that row-wise traversal of 2D matrix always takes less time as compared to column-wise traversal of 2D matrix.
This is because of the "spatial-locality" which is generally used by caches in modern architecture. In the memory, any multidimensional matrix is simply encoded as a 1D array, row by row.

When we start reading from first cell, the CPU automatically caches the cells that are close by, which start by the first row (and if there is enough cache, may also go to the next row, etc). If your algorithm works row-by-row, then the next call will be to an element still in this row, which is cached, so you will get a fast response. If, however, you call an element in a different row (albeit in the same column), you are more likely to get a cache miss, and you will need to fetch the correct cell from a higher memory.

4. *Implement a program to store 10,000 elements using two methods. In Version 1, employ a static array: int arr[10000]. In Version 2, utilize a linked list. Fill each structure with random numbers, then traverse the structures to measure and compare only the traversal times.*

   The runtime to access the elements from a linked list is slower than the runtime to access from the array.

   This is because the array allocates data in a contiguous memory which makes it faster for the RAM to access elements. Whereas, in linked list, the pointers are used to search for the memory location of the next cell, which makes it slower to access elements.

5. *Explain the following code snippet i.e. add comments to each line of code.*

   The following code takes the unsigned 32-bit integer and performs a series of bitwise operations to manipulate its bits. It returns the popcount of binary representation input and stores back in n.

```
uint32_t function_x(uint64_t i) {
  /* 0x5555555555555555: sets alternate bit to 0
     0x3333333333333333: sets two bits from 4 bits to 0.
     0x0f0f0f0f0f0f0f0f: sets 4 bits from a byte to 0. */

  /* Takes every 2nd bit and does AND operation with hexadecimal number
     and subtract it from original number. For every duo pair, this operation
```

```
    is performed which gives the number of set bits in each set of 2 bits. */
  i = i - (( i >> 1) & 0x5555555555555555 ) ;

  /* Grouping the bits in pairs and takes every bit and does AND
     operation with hex number and atlast it sums with shifted pair.
     Taking every 4 bits, sum every 2 bits with their adjacent 2 bits after
     shifting and store it in 4 bits*/
  i = ( i & 0x3333333333333333 ) + (( i >> 2) & 0x3333333333333333 ) ;

  /* Summing up every 4 bits with their adjacent 4 bits after shifting,
     and then clear higher bits from the input binary representation.
     and subtract it from original number ie Taking every 8 bits, sum every
     4 bits with their adjacent 4 bits after shifting and
     store it in 8 bits */
  i = ( i + ( i >> 4) ) & 0x0f0f0f0f0f0f0f0f ;

  /* Taking every 16 bits, sum every 8 bits with their adjacent 8 bits
     after shifting and store it in 16 bits */
  i = i + ( i >> 8) ;

  /* Taking every 32 bits, sum every 16 bits with their adjacent 16 bits
     after shifting and store it in 32 bits */
  i = i + ( i >> 16) ;

  /* Two 32 bits number are added and we get a 64 bit unsigned int */
  i = i + ( i >> 32) ;

  return ( uint32_t ) i ;
}
```

So this function uses bitwise manipulation to summing bits in groups of 1, 2, 4, 8, 16, and 32. The final answer is popcount in the binary representation input.

6. *Compare the runtimes.*

The given code compares the runtime for the popcount of an integer by two different functions ie the one mentioned above and other using hardware counter.

The following are the runtime for 10 observations:

```
function_x : 1173.99 microseconds
 Hardware popcount duration : 1086.83 microseconds
hardware popcount is faster
-----------------------------------------------------
function_x : 754.248 microseconds
 Hardware popcount duration : 683.259 microseconds
hardware popcount is faster
```

```
-----------------------------------------------------
function_x : 351.249 microseconds
 Hardware popcount duration : 327.795 microseconds
hardware popcount is faster
-----------------------------------------------------
function_x : 375.607 microseconds
 Hardware popcount duration : 398.192 microseconds
function_x is faster
-----------------------------------------------------
function_x : 349.855 microseconds
 Hardware popcount duration : 354.192 microseconds
function_x is faster
-----------------------------------------------------
function_x : 320.252 microseconds
 Hardware popcount duration : 328.61 microseconds
function_x is faster
-----------------------------------------------------
function_x : 434.691 microseconds
 Hardware popcount duration : 420.725 microseconds
hardware popcount is faster
-----------------------------------------------------
function_x : 335.099 microseconds
 Hardware popcount duration : 328.372 microseconds
hardware popcount is faster
-----------------------------------------------------
function_x : 333.068 microseconds
 Hardware popcount duration : 351.107 microseconds
function_x is faster
-----------------------------------------------------
function_x : 429.345 microseconds
 Hardware popcount duration : 454.484 microseconds
function_x is faster
-----------------------------------------------------
The average runtime for function_x is: 485.7404 microseconds.
The average runtime for hardware counter is: 473.3566 microseconds.

From above data, we can observe that hardware counter takes less
time to compute than the function_x on an average. This is because
instructions in hardware popcount are specifically designed and
optimized for counting bits at hardware level of abstraction.
The _mm_popcnt_u32(i) is reference to Intel's Intrinsics Library
for counting the number of set bits (bits with a value of 1) in a
32-bit unsigned integer using SIMD (Single Instruction, Multiple Data)
instructions for improved performance.
The function_x involves the bit masking and manipulation to do the
same task, in which it takes more time to compute, whereas in
hardware popcount,the instructions are fed parallely to make the
computations faster.
```