Assignment 1

Siddhant S. Aphale,
Department of Mechanical and Aerospace Engineering,
University at Buffalo,
Person # - 50164327

October 4, 2016

Problem 1

1. User having most Pending jobs with number of pending jobs and total nodes this user using

Obtaining the users with the most pending jobs will require setting a filter to sort jobs flagged "PD". This can be achieved using `grep "PD"`. Once the list is filtered for only Pending Jobs, we next translate it and cut the all unnecessary columns using delimiters. Once the final list is ready with just the usernames, we sort it and then find out all unique usernames using `uniq`. Further we calculate the total duplicates of each username using `uniq -c` and sort it in descending order to find out the most pending jobs and username. This can be done as given in section 2 below.
To find out the total number of nodes that user is using, we need to keep the column containing the number of nodes in use. Once we have it, we use the `awk` command to calculate the subtotal for each unique username. We use a for loop to print the subtotal for each unique username. The command for printing the total number of nodes with the username is given below

```
cat queue.txt | grep "PD" | tr -s ' '| cut -d ' ' -f 5,8 | awk '{a[$1]+=$2}END
    {for (k in a) print k, a[k]}' | sort -k2 -nr | head -1
```

The output of the above command can be given as

```
siddhant@kryptonite:/mnt/UB LEARNS/FALL-2016/HPC-1/Homework$ cat queue.txt |
   grep "PD" | tr -s ' '| cut -d ' ' -f 5,8 | awk '{a[$1]+=$2}END {for (k in a
   ) print k, a[k]}' | sort -k2 -nr | head -1
arpanmuk 1426
```

2. User having the most pending jobs and number of pending jobs

Obtaining the users with the most pending jobs will require setting a filter to sort jobs flagged "PD". This can be achieved using `grep "PD"`. Once the list is filtered for only Pending Jobs, we next translate it and cut the all unnecessary columns using delimiters. Once the final list is ready with just the usernames, we sort it and then find out all unique usernames using `uniq`. Further we calculate the total duplicates of each username using `uniq -c` and sort it in descending order to find out the most pending jobs and username. The detail one line command is given below

```
cat queue.txt | grep "PD" | tr -s ' '| cut -d ' ' -f 5 | sort | uniq -c | sort
    -nr | head -1
```

The output of the above command can be given as

```
siddhant@kryptonite:/mnt/UB LEARNS/FALL-2016/HPC-1/Homework$ cat queue.txt |
    grep "PD" | tr -s ' '| cut -d ' ' -f 5 | sort | uniq -c | sort -nr | head -1
     218 arpanmuk
```

3. User with a running job having the longest username

To obtain the user with a running job and having the longest username, initially `grep` was used to find out all the jobs that were running. Further the filtered list was sorted according to the username. In addition it was translated and the data except the username was deleted using `cut`. This data was further replaced by only unique entries and then the usernames having longest username was filtered. All the usernames having the longest length were printed using `awk`. And at last, the total number of characters in the username was added before each username and was sorted alphabetically. The one line command used to achieve the result is given below

```
cat queue.txt | grep "R" | sort -k4 | tr -s ' '| cut -d ' ' -f 5 | uniq | awk
    'length > x {delete y; x=length} length==x{y[NR]=$0 } END {for (z in y)
    print y[z]}' | awk '{print length, $0}' | sort
```

The output obtained after running the above command is

```
siddhant@kryptonite:/mnt/UB LEARNS/FALL-2016/HPC-1/Homework$ cat queue.txt |
    grep "R" | sort -k4 | tr -s ' '| cut -d ' ' -f 5 | uniq | awk 'length > x {
    delete y; x=length} length==x{y[NR]=$0}END{for (z in y) print y[z]}' |
    awk '{print length, $0}' | sort
8 adamphil
8 alexmarc
8 anijamud
8 ciaranwi
8 haihuaya
8 hrajabza
8 jameshoo
8 jiajunli
8 karneshj
8 kasianwa
8 laipingw
8 ninatymi
8 rsubrama
8 sabrygad
8 sbfrench
8 shomakit
8 skhopkar
8 wenjingg
8 xiangyuj
8 yadongwa
8 yudhajit
8 zhixuanc
```

4. Largest number of nodes utilized and the user utilizing it

To obtain the largest number of nodes and the user using it, it is necessary that we delete all the data except the data in columns USER and NODES. First, we traslate the data and then cut the data with delimiters and keeping the column 5 and 8 of `queue.txt`. Then we sort the obtained result based on the

second column, which is the column representing number of nodes and arrange it in descending order. Further we print out the first result using `head -1`. The one liner command is given below

```
cat queue.txt | tr -s ' ' | cut -d ' ' -f 5,8 | sort -k2 -nr | head -1
```

The output obtained by running this command is given below

```
siddhant@kryptonite:/mnt/UB LEARNS/FALL-2016/HPC-1/Homework$ cat queue.txt |
    tr -s ' ' | cut -d ' ' -f 5,8 | sort -k2 -nr | head -1
shomakit 32
```

5. Number of unique users who have pending jobs because of Priority

To obtain the number of unique users who have pending jobs because of priority is 21. To obtain this result, initially by using `grep` the list is filtered for "PD" and "Priority". It is sorted according to the username. And then The obtained list is then transalted and cut to obtain just the list of usernames. The obtained usernames are filtered to include only unique entries and are the counted to find the required output. This can be obtained by querying the `queue.txt` file using the following line of code.

```
cat queue.txt | grep "PD" | sort -k4 | grep "Priority" | tr -s ' '| cut -d ' '
    -f 5 | uniq | wc -l
```

The output of the above one liner is as follows

```
siddhant@kryptonite:/mnt/UB LEARNS/FALL-2016/HPC-1/Homework$ cat queue.txt |
    grep "PD" | sort -k4 | grep "Priority" | tr -s ' '| cut -d ' ' -f 5 | uniq
    | wc -l
21
```

Problem 2

Problem 2 -

$$\frac{\text{average cycles}}{\text{word access}} = f_c \times \frac{\text{cache cycles}}{\text{word access}} + (1-f_c) \times \frac{\text{main memory cycles}}{\text{word access}}$$

$S_{WM} = 2$

Processor Speed = 2GHz = 2×10^9 cycles/sec

$f_c = \begin{cases} \text{(i) } 99\% \\ \text{(ii) } 1\% \end{cases}$

To Find - Predicted Performance

Soln -

Case 1 - $f_c = 99\%$

$$\frac{\text{average cycles}}{\text{word access}} = 0.99 \times 2 + (1-0.99) \times 100 = 2.98$$

$$\text{Performance} = \frac{\text{Processor Speed} \times S_{WM}}{\frac{\text{average cycles}}{\text{word access}}} = \frac{2 \times 10^9 \times 2}{2.98}$$

Performance = 1.3422×10^9 Flop/s = 1.3422 GFlop/s

Case 2 - $f_c = 1\%$

$$\frac{\text{average cycles}}{\text{word access}} = 0.01 \times 2 + (1-0.01) \times 100 = 99.02$$

$$\text{Performance} = \frac{2 \times 10^9 \times 2}{99.02} = 40.3958 \times 10^6 \text{ Flop/s}$$

Performance = 40.3958 MFlop/s

Problem 3

In this problem two matrices, A and B of size 20000×20000 were added to get matrix C. The matrices were generated of random numbers between 0 and 99. Initially, the addition was timed looping over rows first and then columns. Then, the addition was timed looping over columns first and then rows. The codes were compiled and executed on 1 node with 8 cores and 24000 MB memory request. Two compilers were used to compile the codes viz., GCC 4.9.2 and Intel 15. For the first run on both the compilers, no optimization techniques were utilized. Then the `-O1`, `-O2` and `O3` optimization flags of the compilers were utilized to time the addition. The timed results are reported below:

First row loop and then column

For the first run, the code is executed without any optimization on both GCC and Intel compilers. The time required to execute the addition of two matrices on GCC compiler is reported as $13.8726s$. Same code when executed using the Intel compiler reported time of $3.70625s$.
On the second run, the code was optimized by using the `-O1` optimization flag of the compilers and we definitely observe improvement. This flag optimizes the loops, provides inline optimization as well as vectorization. On GCC compiler, the addition time was reported as $12.6916s$ and on Intel compiler it was reported as $2.48487s$.
To further check the optimization, `-O2` flag was set for compiling the code. This flag aligns the loops and functions. GCC compiler completed the execution in $12.6393s$ whereas the Intel compiler completed it in $2.48859s$.
Finally, the `O3` optimization flag was applied which optimizes the code yet more. It sets the inline function flag on. GCC copiler completed the addition in $12.6528s$ whereas the Intel compiler was able to execute the code in $2.48534s$.

First column loop and then row

Now the code is executed with first a column loop and then row loop. As earliers, for the first run, the code was executed without utilizing any optimization technique and the time was reported to complete the addition of the matrices generated by random numbers. For the GCC compiler, the time was reported as $73.9571s$ and that for Intel it was $54.4096s$.
The optimization techniques as used earlier were again implemented for this code. The `-O1` technique of optimization reported improvment in timing as compared to no-optimization. GCC compiler required $63.8852s$ to complete the addition. Whereas, the Intel compiler required $46.5471s$.
Now, the code was tested with `-O2` optimization technique. With this technique there was further improvement for GCC compiler. GCC compiler completed the addition in $65.0815s$. However, on the Intel compiler the time increased and took $48.2623s$.
Finally, the `O3` optimization flag was applied which optimizes the code yet more. It sets the inline function flag on. GCC copiler completed the addition in $64.3584s$ whereas the Intel compiler was able to execute the code in $47.5504s$. Thus, it can be seen that the optimization flag `O3` provided the best optimization results for GCC compiler. However, for Intel compiler, the best results were obtained with `O1` flag. Buth the difference was not much between the other two flags.

To summarize the above results, we can say that when the addition was timed with first looping rows, then columns the time required to execute the code is less compared to first looping columns, then rows. There is a drastic time difference in between these two as can be seen in the timings reported above. This drastic difference is observed because C++ is a row-major programming language. That means, the data is stored in the memory row-wise. Further it is observed that we get slightly improved timings when used optimization techniques. The `-O3` reported the best timings. Furthermore, it can also be concluded from above results that the Intel compiler is better compared to GCC in terms of performance as the timing required while using Intel compiler is much less than that required for GCC.

Problem 4

A program implementing vector dot product was written that was set to handle 64 MBytes vector size. This algorithm is known as L1 BLAS function which is the Level 1 Basic Linear Algorithm Subprogram which is used to check the performance of a processor. This program was specifically executed on IBM L5520 node at CCR. The IBM L5520 node has 8 cores/node and Clock rate of 2.27 GHz. The total number of operations per cycle handled by this particular CPU is 4 double precision. Thus the Theoretical Peak Performance was quantified as 2.27 GHz $\times 4 = 9.08$ GFlops per core. We have 8 cores per node, so the TPP for this architecture is 72.64 GFlops. The L1 BLAS function developed was executed using both GCC and Intel compilers. Furthermore, the O1, O2 and O3 optimization techniques were utilized to experiment the changes in the results. The plots for performance are as in the figures 1 - 4 below. The standard Netlib BLAS function ddot was developed to run on native CCR BLAS library. Finally, the results obtained from above two method was compared with the optimized version of `ddot` in the MKL library.

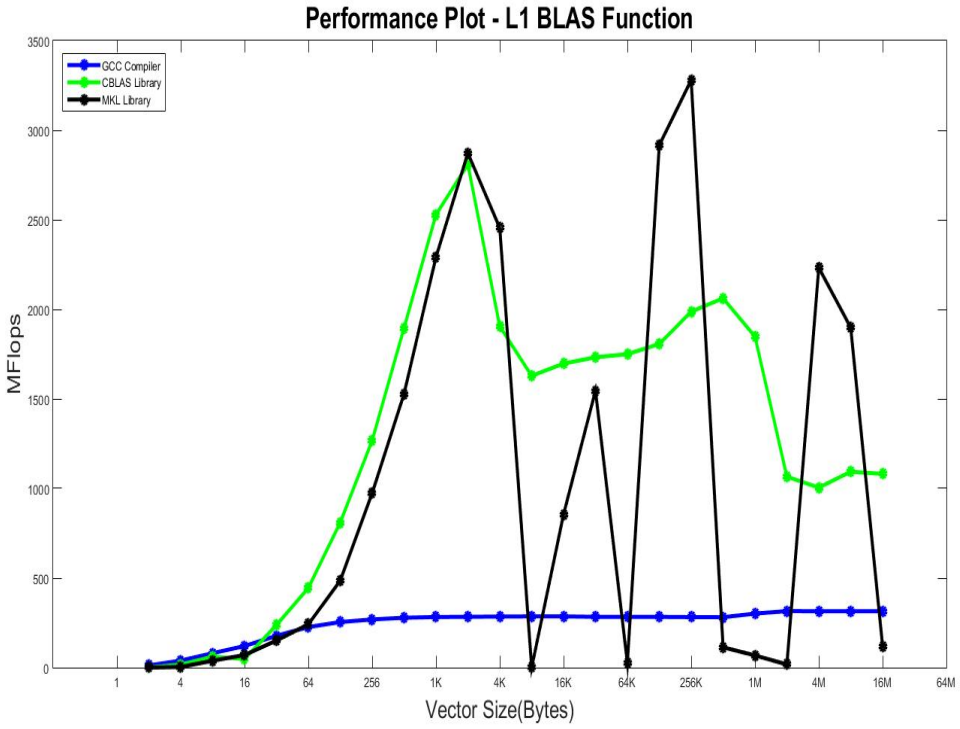

Figure 1: L1 BLAS Function without optimization

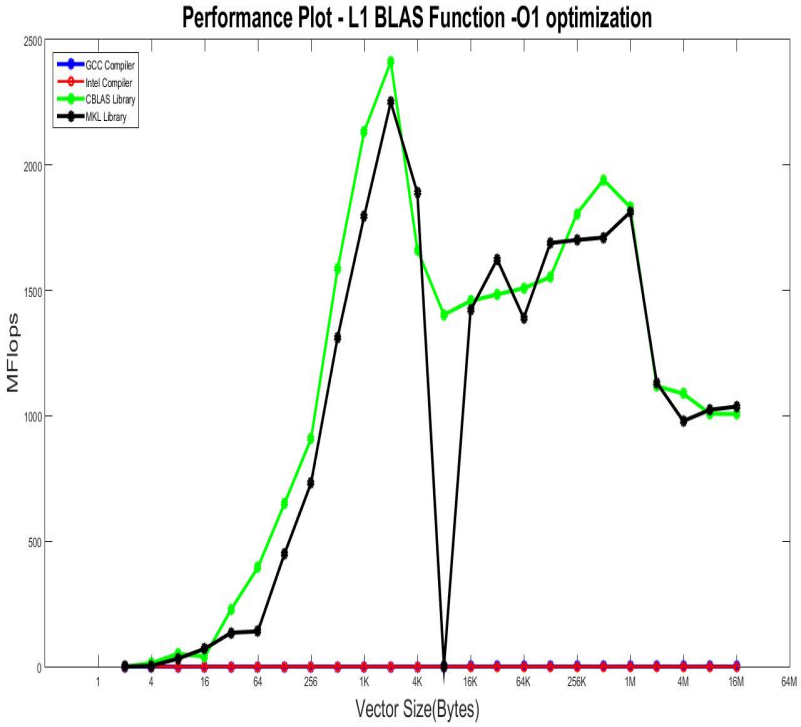

Figure 2: L1 BLAS Function - O1 optimization

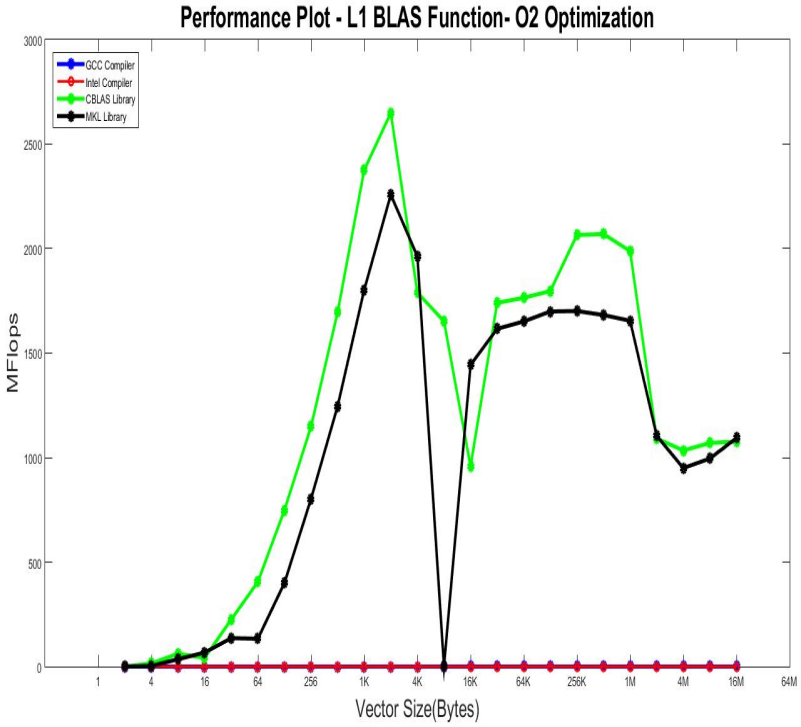

Figure 3: L1 BLAS Function - O2 optimization

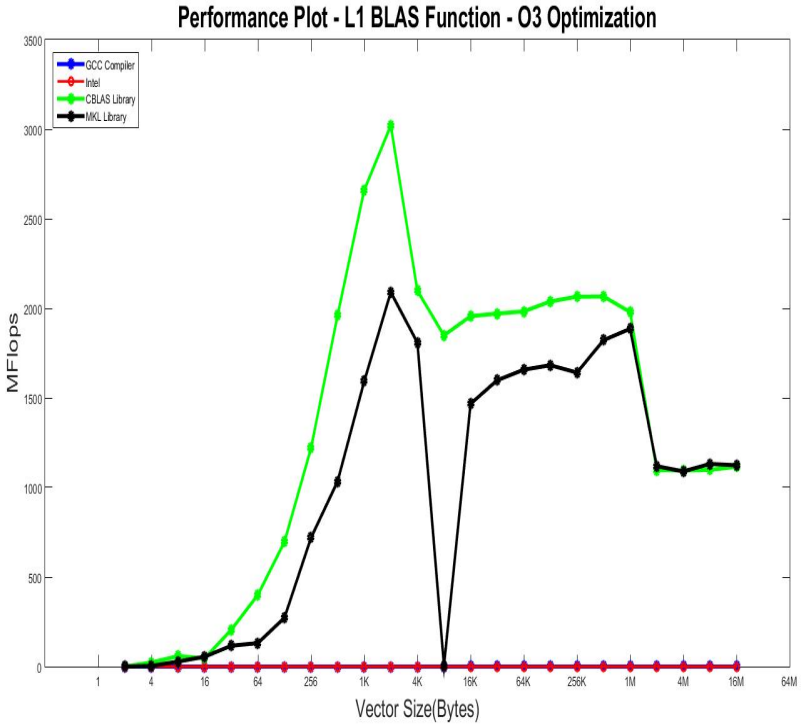

Figure 4: L1 BLAS Function - O3 optimization

From Fig. 1 it can be observed that for L1 BLAS function code developed, and when compiled using GCC compiler the highest performance achieved is 315 MFlops. When the same code was compiled using the Netlib BLAS library function ddot, the peak performance obtained was 2809.3 MFlops which is 2.8 GFlops. The `ddot` function of Netlib library have very good results over L1 BLAS Function program that was written. Furthermore, when the MKL library was used the peak performance achieved was 3280 MFlops or 3.28 GFlops. This is the maximum achieved peak performance among all the experiments. When compared with the peak performance computed above, we can achieve just 4.52% of the peak. Various optimization techniques were performed to experiment the changes. It is shown in Fig. 2, 3, 4 above. It can be observed from the plots that the performance achieved by all the three optimization techniques is reduced as compared to without optimization. Also we can observe that the performance for the Netlib `ddot` function has improved over MKL library. In case of O1 optimization, the peak performance is obtained for Netlib CBLAS library and is 2.4 GFlops. When we use O2 optimization technique, this performance is improved to 2.65 GFlops. In case of O3 optimization, we obtain the best result for CBLAS library with 3.1 GFlops. However, for optimized MKL library, the best performance achieved was without any optimization. The performances for L1 BLAS function program developed is negligible when optimization techniques are used. When Intel compiler was used for L1 BLAS function, the results obtained looked weird with the performance in the range of 2×10^6 MFlops, which obviously is incorrect and probably some glitch in the program. The performance plot with Intel compiler is shown in Fig. 5

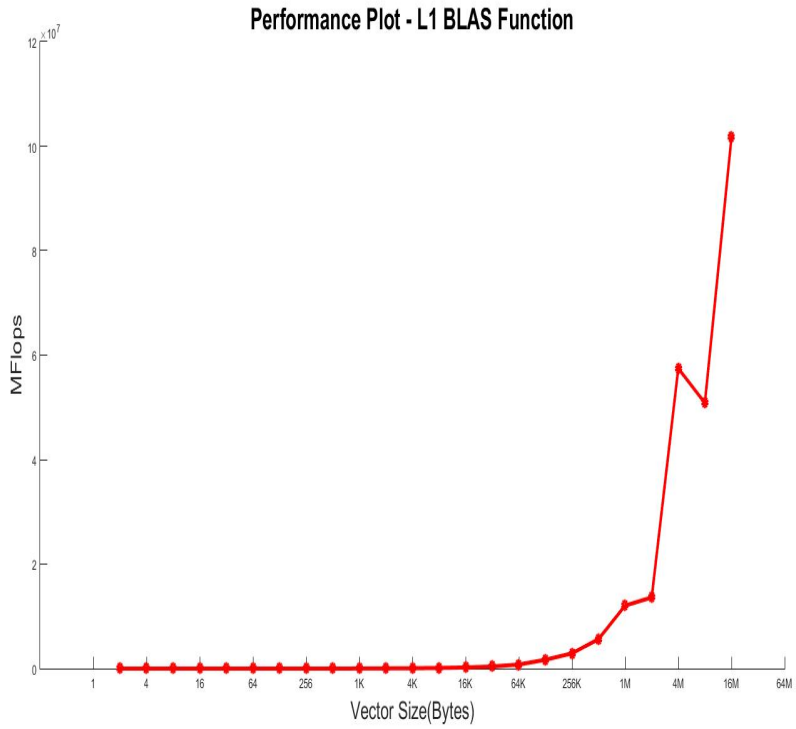

Figure 5: L1 BLAS Function - O3 optimization

Problem 5

A program was written that computes matrix-matrix multiplication which is also defined as L3 BLAS function under Basic Linear Algebra Subprogram. This is a level 3 BLAS function which computes matrix-matrix multiplication with $2n^3$ operations occurring. This program was compiled using both GCC and Intel compilers. The performance for both the compilers was plotted. Furthermore, -O1, -O2 and -O3 optimization techniques were used to experiment and observe the variations. This was further compared with the optimized `dgemm` function in MKL Library. The performance plots are shown in figures 6 - 10 below.

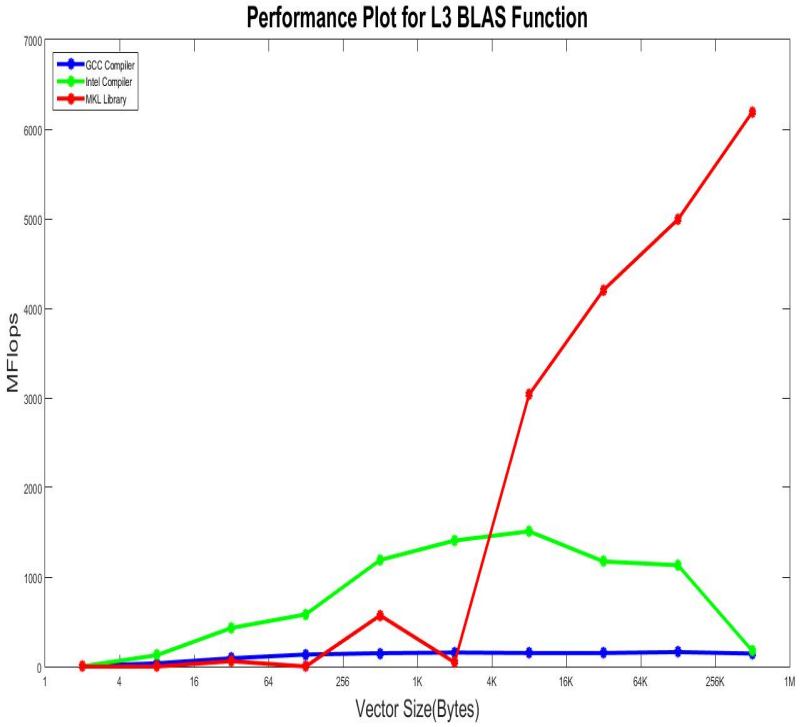

Figure 6: L3 BLAS Function - GCC, Intel and MKL Comparison

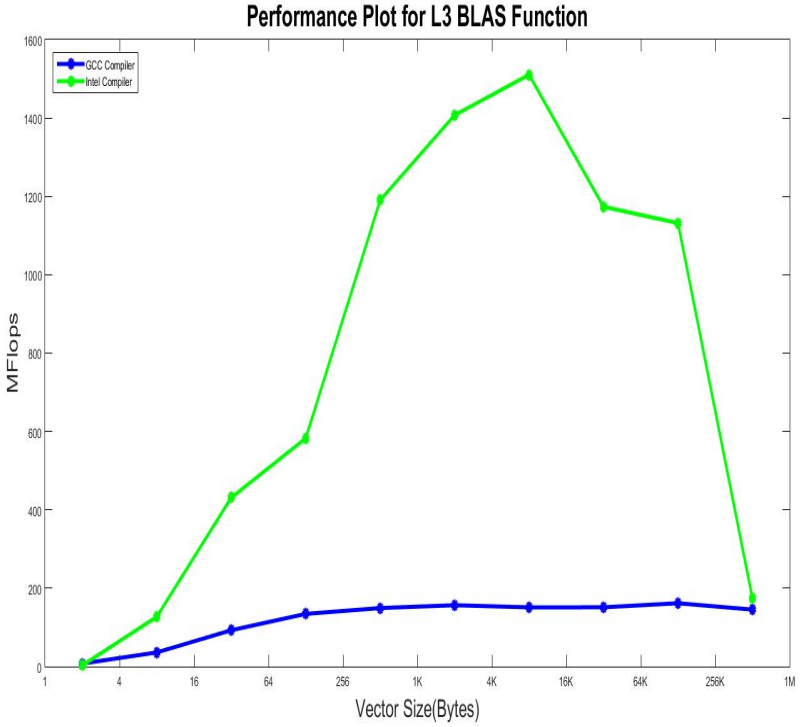

Figure 7: L3 BLAS Function - GCC and Intel Comparison

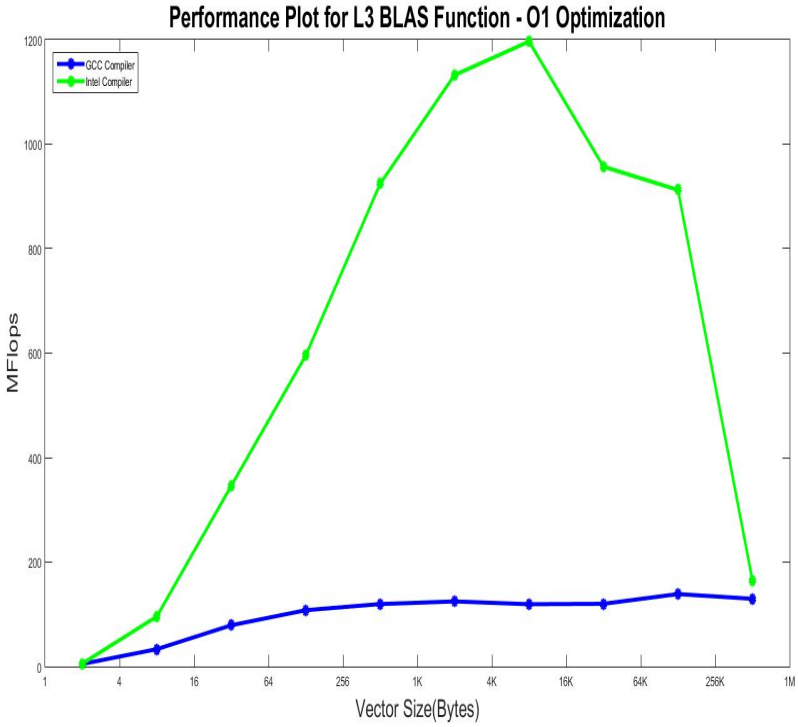

Figure 8: L3 BLAS Function - GCC and Intel - O1 optimization

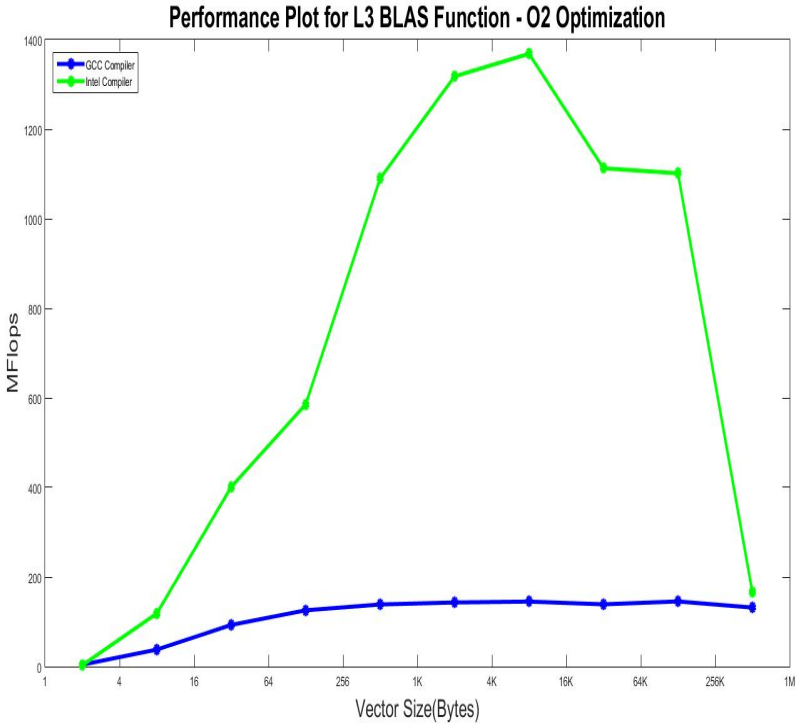

Figure 9: L3 BLAS Function - GCC and Intel - O2 optimization

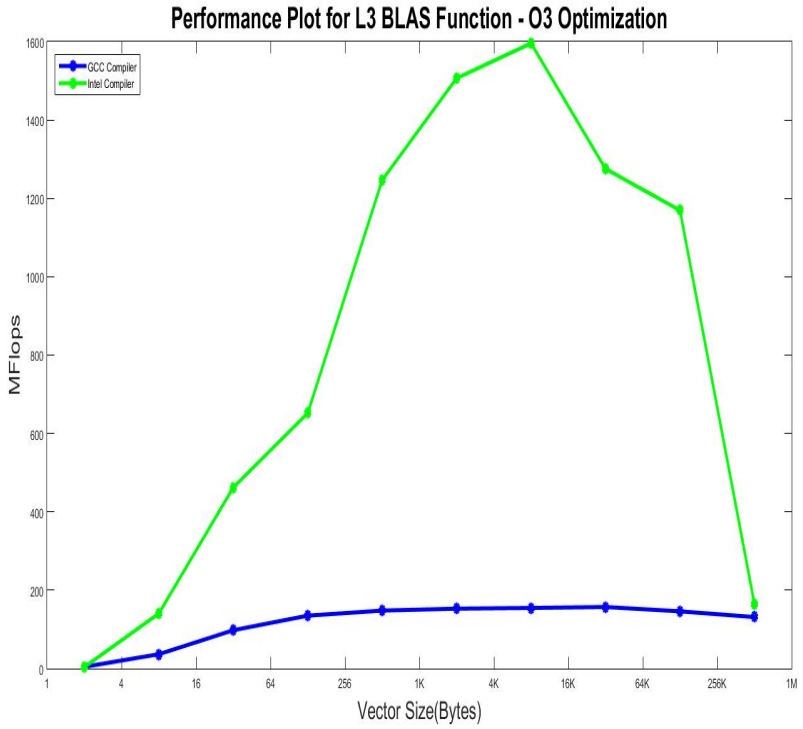

Figure 10: L3 BLAS Function - GCC and Intel - O3 optimization

Figure 6 depicts the performance plots for non-optimized L3 BLAS Function and its comparison with `dgemm`. To generate this plot, matrices of size 1024×1024 elements was generated. This can help in generating 16 MBytes of data. The GCC compiler provided the performance of 156 MFlops, Intel compiler provided 1.7 GFlops and the `dgemm` function provided the performance of 6.2 GFlops and the plot shows that MKL goes on increasing. Thus with the amount of data generated in matrix-matrix multiplication, the MKL library provided 8.55% of the peak performance that was calculated t 72.64 GFlops. Figure 7 shows the same plot without MKL comparison. Furthermore, as earlier three optimization techniques, O1, O2 and O3 were performed on the L3 BLAS Function and the performance plots are shown in figures 8 - 10. When the O1 optimization was performed, the GCC compiler gave highest performance of 138 MFlops and the Intel compiler provided 1.2 GFlops. With the O2 optimization, GCC compiler achieved highes performance of 146 MFlops whereas the Intel compiler obtained 1.37 GFlops. O2 optimization technique provided an improved performance over O1 but less compared to non-optimized. Next, O3 optimization technique was performed to see if the results are further improves, and GCC provided its highest performance overall of 158 MFlops. The Intel compiler achieved 1.6 GFlops.