Lscpu

```
csstudent299@csicluster:~$ lscpu
Architecture:            x86_64
CPU op-mode(s):          32-bit, 64-bit
Byte Order:              Little Endian
CPU(s):                  8
On-line CPU(s) list:     0-7
Thread(s) per core:      1
Core(s) per socket:      1
Socket(s):               8
NUMA node(s):            1
Vendor ID:               GenuineIntel
CPU family:              6
Model:                   63
Model name:              Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
Stepping:                2
CPU MHz:                 2397.223
BogoMIPS:                4794.44
Hypervisor vendor:       VMware
Virtualization type:     full
L1d cache:               32K
L1i cache:               32K
L2 cache:                256K
L3 cache:                15360K
NUMA node0 CPU(s):       0-7
```

L1 cache has a size of 32K for data storage.
Two store 3 matrices A,B & C in the L1 cache, let us consider the following block sizes:
block_size=b,
b=32 → total memory needed would be (32*32*8*3) ~ 24K
b=64 → total memory needed would be (64*64*8*3) ~ 96K

It would be wise to choose a cache block size around 32 so that all the submatrices can be loaded from cache minimising cache miss

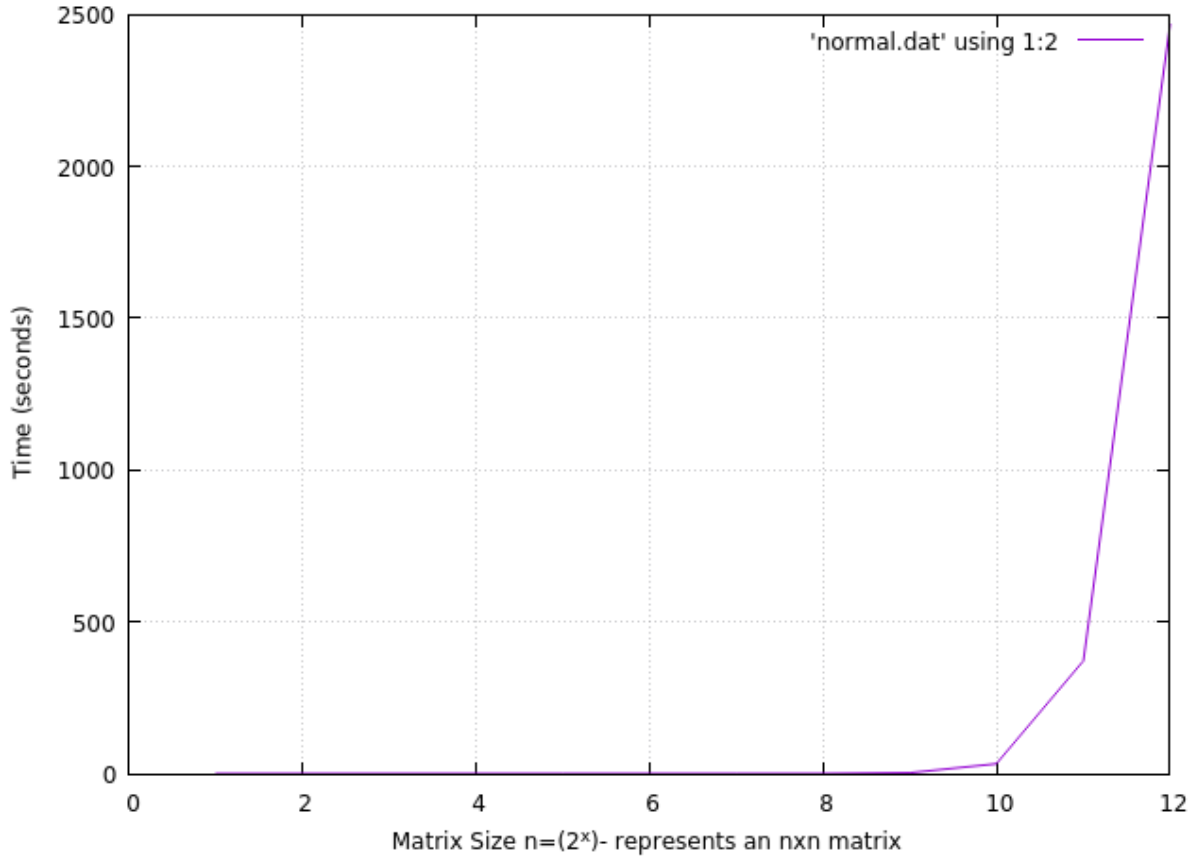
ALL READINGS ARE MENTIONED IN THE ATTACHED EXCEL FILE. THIS INCLUDES THE READINGS FOR

1. BASIC IJK (WITHOUT CACHE BLOCKING)
2. BASIC BLAS (WITHOUT CACHE BLOCKING)
3. CACHE BLOCKED BLAS

PREFACE
—----------

Observing behaviour of naive ijk matrix multiplication without any blocking.

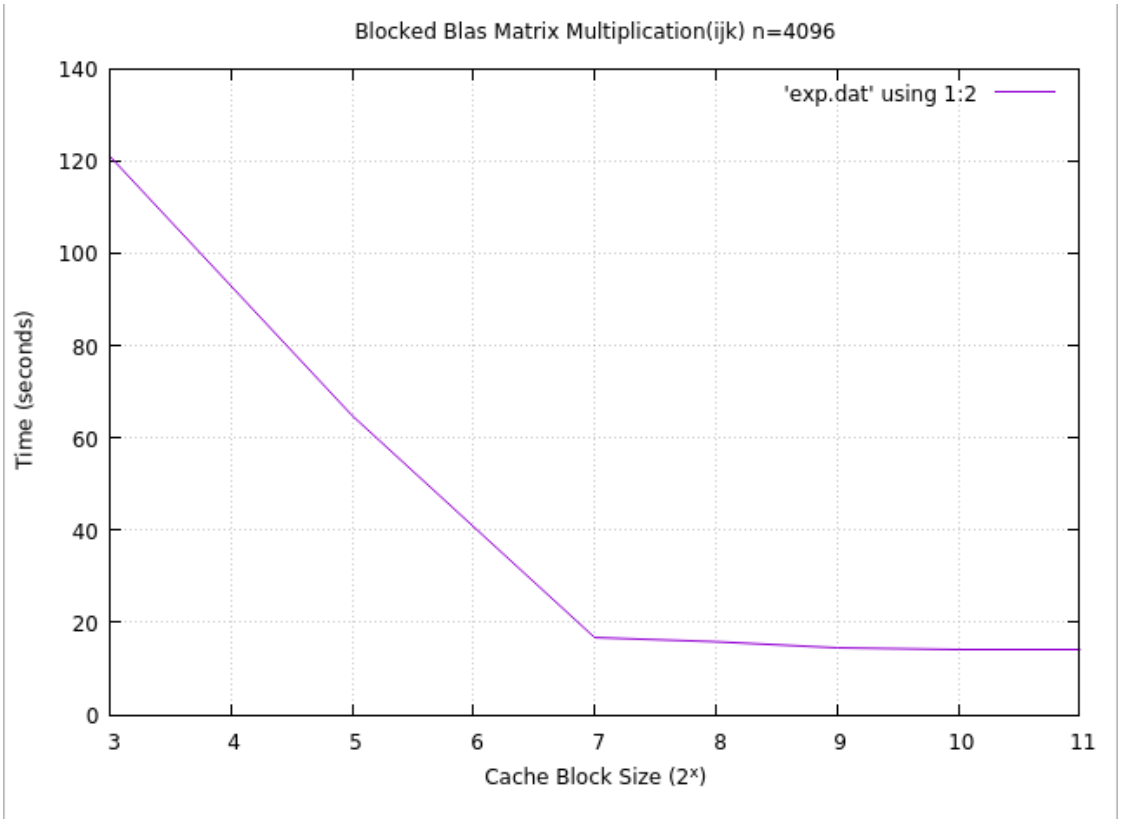
The following graph illustrates that time taken for matrix multiplication increases exponentially with the increase in size. Only for small block sizes where the entire matrices can be used from processor cache does it show fast and efficient performance.

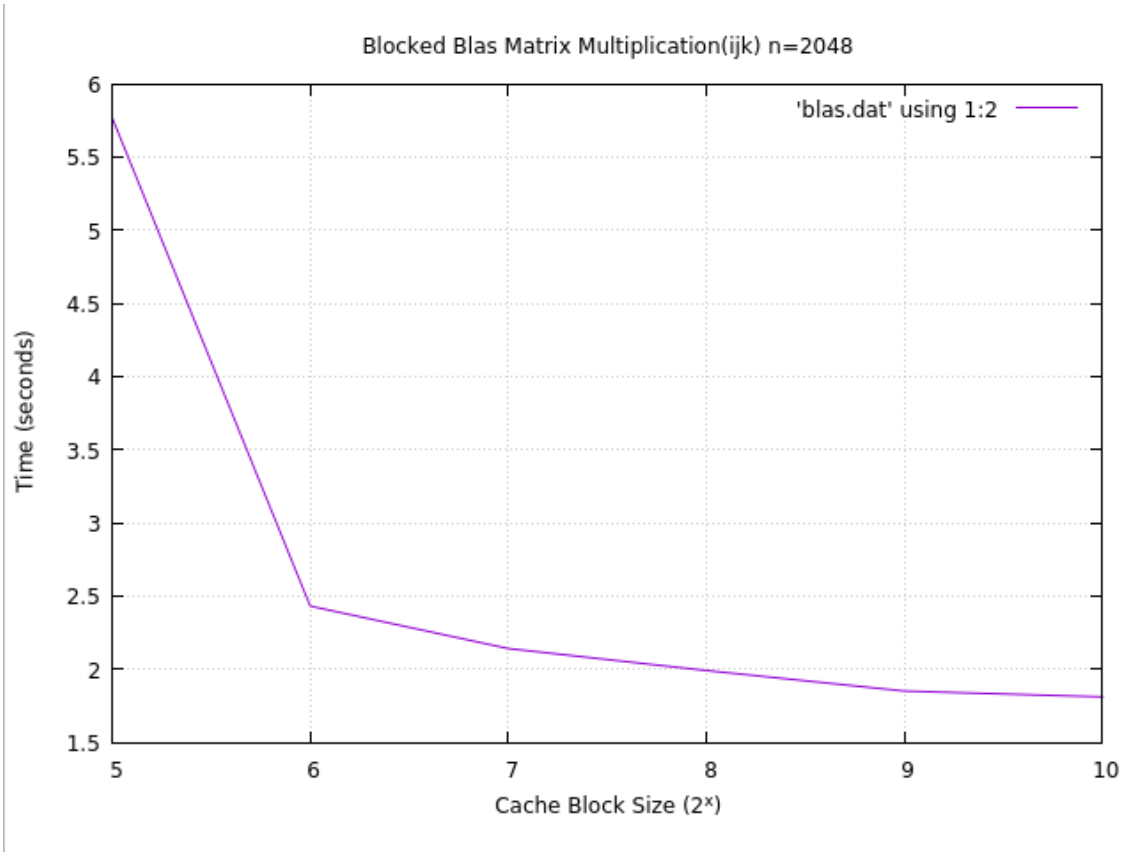


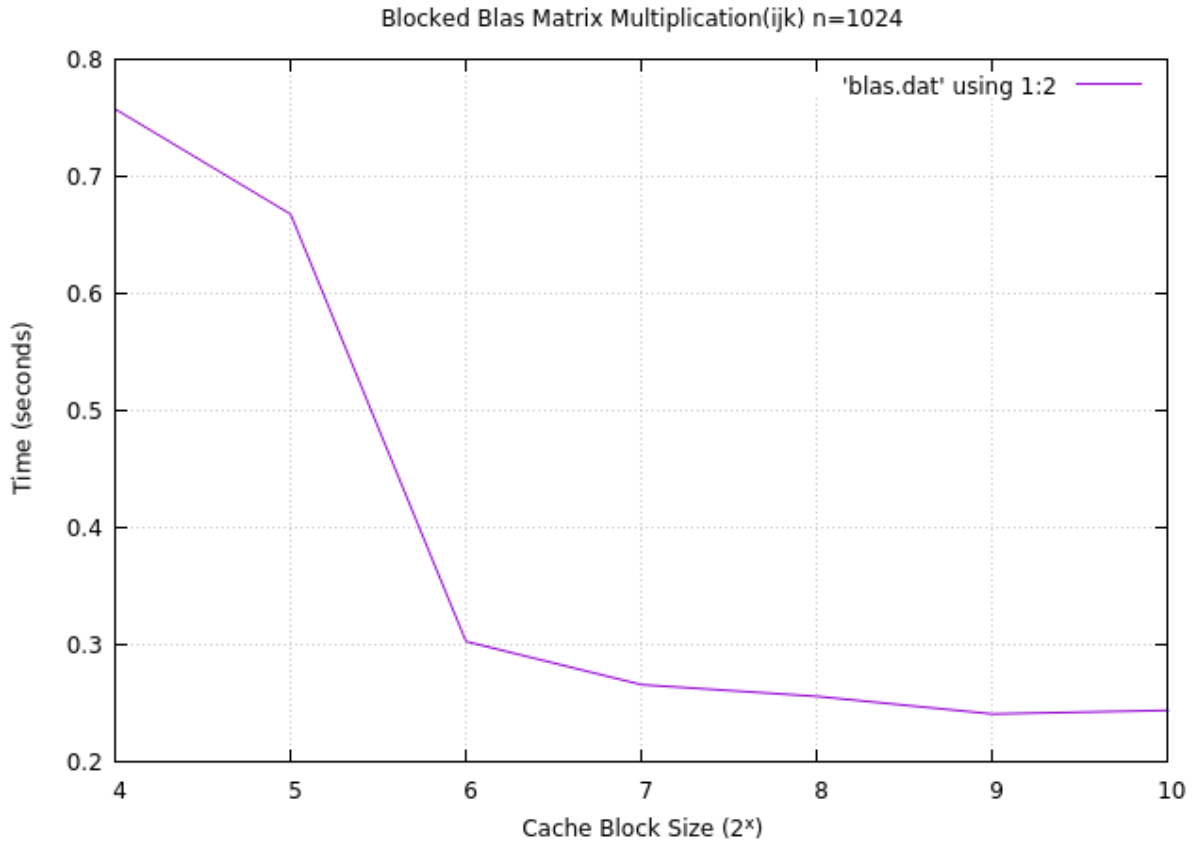
Variants received **1-b 2-a**

1-b Blocked matrix using ijk

Blocked Blas for matrix multiplication was done for matrices with size 4096,2048,1024.
The performance of blas seems to dip with a blocked implementation suggesting the blas is already optimised for matrix multiplication and blocking seems to be implicitly taken care of.

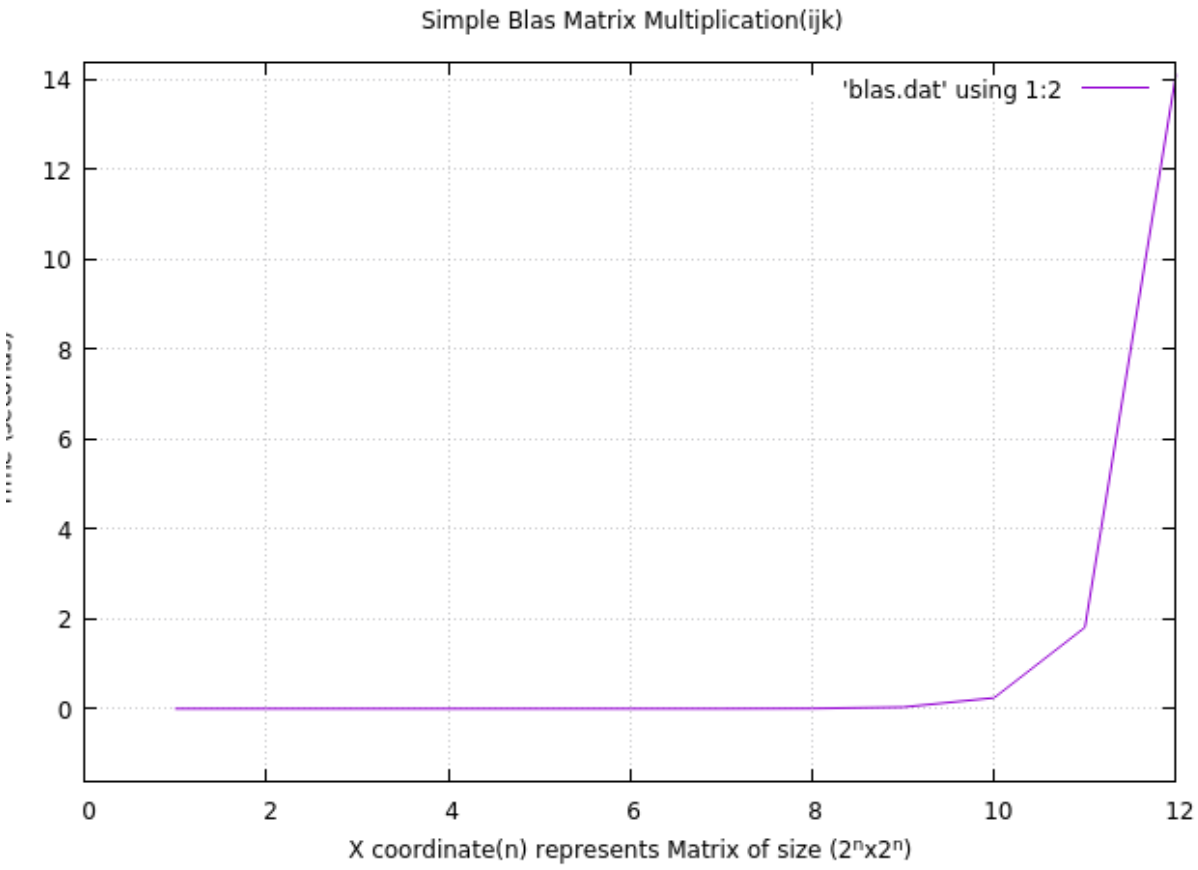

Blocked Blas Matrix Multiplication(ijk) n=4096
'exp.dat' using 1:2
Time (seconds)
Cache Block Size (2^x)
0
20
40
60
80
100
120
140
3
4
5
6
7
8
9
10
11


Blocked Blas Matrix Multiplication(ijk) n=2048
'blas.dat' using 1:2
Time (seconds)
Cache Block Size (2^x)
1.5
2
2.5
3
3.5
4
4.5
5
5.5
6
5
6
7
8
9
10



2-a Straightforward matrix multiplication using blas

Following graph shows how blas performed for all matrices of dimensions (2^n x 2^n) where 1<=n<=12. From the figure it can be deduced that identical matrices of similar dimensions perform much better with direct non blocking blas implementation.



OBSERVATION

—----------.

The performance of blocked BLAS for matrix multiplication is severely impacted by small block sizes for large data. For eg. when n=4096 and b=8 we get a time of around 121 seconds.
This is significantly worse compared to a normal blas multiplication of the same matrix. (Normal unblocked would be around 14 sec.) We can see that the execution time progressively reduces with increase in block size. When the block size ~ n the execution time also becomes ~ 14, the normal unblocked blas execution time. In this scenario the blocked blas in effect acts like a normal basic blas implementation with the entire matrix being sent to the blas routine.

Following are the possible reasons why blocked cache behaves in this way.

Possible reasons.

- Explicit blocking of the matrix into submatrices and then invoking the blas routines may cause additional overheads within blas. This can but may not only involve calculating the starting addresses, dimensions of the submatrices.

Calculating the correct offsets for each submatrix adds another layer of complexity.These need to be done as well so that the submatrices align with their corresponding larger ones.

- Blas is extremely optimised to handle the multiplication of matrices in their entirety with strategies in places to load and access matrix entries from memory for processing. The blocking of matrices and the subsequent submatrix multiplication may potentially be hindering this.

  It seems that the blocking mode is in turn impeding BLAS dgemm routines' optimal memory access algorithms.

CONCLUSION

—----------

While blocking of submatrices can yield better performance of matrix multiplication under some scenarios, with BLAS that does not seem to be the case. As expected for a large matrix, a small block size gave a very poor performance.
The performance gradually improved with increasing block size finally culminating with the best performance when block_size=matrix dimension.
However there was no optimal block size that could be found which was more efficient compared to non blocking BLAS.

BLAS whether blocked or non blocked performs better than naive ijk matrix multiplication. This is especially true for larger matrices. The optimisation of BLAS libraries are more visible as you keep increasing the size. For extremely small matrices (2x2,4x4) the straightforward matrix multiplication may be faster.