Lab 4 - Matrix-Matrix Multiply and Third Party Libraries

NERS/ENGR 570 Fall 2020
September 25th, 2020
DUE BY: Oct. 5th, 2020

<u>Exercise 1 (100 pts) - BLAS</u>

1. Clone this Lab's git repository:
    a. From the terminal command line:

```
git clone https://github.com/bkochuna/Lab4-matmul.git
```

2. Read through the sample code we provided.

3. Read through the rest of this exercise and make a prediction about which implementation is fastest and how the run times will scale with the matrix size.
    - *It is predicted that the OpenBlas, being a more sophisticated library will be fastest.*
    - *It is predicted that the run times will scale quadratically (N^3) with matrix dimension (NxN), since a matrix multipliction involves N^2 sets of N inner multiplications.*

4. The Great Lakes cluster has a local installation of BLAS (/usr/lib/blas). Compile and link your program against this version of BLAS using one of the GNU compilers. Record the command used for compiling (don't forget optimizations!):
    - *To compile with the local library, the -<lib file name without extension> can be used to indicate the compiler should use that library, and the -L<lib path> can be used to indicate the compiler should look for libraries in that path. Optimization can be done through -O<N>, where N is either 0,1,2 for successively more optimization attempted. CC and EXT stand for the gnu compiler and file extension (either g++,cpp or gfortran,f90):*
    - *CC -lblas -O2 -o matmul.out matmul.EXT*

5. On Great Lakes there is also a module for a different BLAS implementation called OpenBLAS. Load this module and any other required modules. Then recompile the program against this version of BLAS. Record the command used for compiling:
    - *CC -lopenblas -O2 -o matmul.out matmul.EXT*

6. Run your executable for each BLAS implementation for the following matrix sizes and record the results. For consistency in the results, all results should be generated ***on the compute nodes.***

Matrix Size	BLAS dgemm time (s)	OpenBLAS dgemm time(s)
20	0.00	0.00
100	0.00	0.00
500	8e-2	1e-2
1000	6.8e-1	8e-2
2000	6.08	4.2e-1
4000	4.89e1	2.37

7. Provide your own `dgemm` implementation in either the Fortran or C++ program. Uncomment the code to compare the C matrix computed by your code to the C matrix computed by the BLAS library.

mydgemm

```
  for(unsigned int i=0; i<n; i++){
    for(unsigned int j=0; j<m; j++){
          c[i][j] *= beta;
          }
    };
  for(unsigned int i=0; i<n; i++){
    for(unsigned int j=0; j<m; j++){
          for(unsigned int l=0; l<k; l++){
              c[i][j] += alpha*a[i][l]*b[l][j];
                }
          }
    };
```

a. Again run your program for the various matrix sizes ***on the compute nodes***. Compare the run times.

Matrix Size	My MatMult time (s)
20	0.00
100	0.00
500	1.5e-1
1000	2.62
2000	3.63e1
4000	3.54e2

8. Write a similar matrix-matrix multiply program in Python using the numpy/scipy libraries and generate the same set of results as the other cases.

Matrix Size	Numpy dgemm time (s)
20	7.00e-5
100	2.80e-4
500	8.27e-3
1000	5.22e-2
2000	3.71e-1
4000	3.09

matmul.py

```
#!/usr/bin/env python3

import os,sys,time
import numpy as np


# Matrix Operations
def timer(quiet=False):
    def wrapper(func):
        def _wrapper(*args,**kwargs):
            time0 = time.time()
            output = func(*args,**kwargs)
            deltatime = time.time()-time0
            if not quiet:
                print('Time: %0.5f s'%(deltatime))
            return output
        return _wrapper
    return wrapper

@timer(quiet=True)
def generate_random_matrix(m,n,quiet=True,label=None):
    # return np.random.rand(m,n)
    arr = np.random.randint(1,10,(m,n)).astype(float)
    if not quiet:
        print_matrix(arr,label)
    return arr
```

```
def print_matrix(arr,label=None):
    if label is not None:
        print(label)
        print(arr)
        print()
    return


@timer(quiet=False)
def dgemm(A,B,C,alpha,beta):
    C *= beta
    C += alpha*np.dot(A,B)
    return


# Arguments
N = 10


Nargs = len(sys.argv)-1
if Nargs > 0:
    N = int(sys.argv[1])

print('N = %d'%(N))


# Initialize Matrices
arrs = {}
keys = ['A','B','C']
for k in keys:
    arrs[k] = generate_random_matrix(N,N,quiet=True,label=k)



# Multiply Matrices

params = {'alpha':1.0,'beta':0.0}

dgemm(**arrs,**params)

#print_matrix(arrs['C'],'New C')
```

9. Questions
   a. Which BLAS implementation was fastest?
      - *The openBLAS implemention as fastest, by almost an order of magnitude from BLAS and the Naive, but was barely faster than the Numpy python.*
   b. How did each BLAS implementation scale with matrix size?
      - *As per the plot below in Fig 1., the BLAS multiplications scaled as T ~ N^3, as predicted.*
   c. Did the speed of each implementation agree with your prediction?
      - *The speed of each implementation was roughly T ~ C*N^p, with p ~ 3 and small constants of C, as per in Fig 1. However the openBLAS scaled as around p ~ 2.5, however had a much larger C than the python, which had p ~ 3, and so open BLAS is expected to be much faster at larger N. The mkl scaled almost identically to openBLAS.*
   d. If you observed differences from your prediction why?
      - *The small differences from predictions, for the p_Naive > 3, can be attributed to overhead in the naive implementation and no parallalization of any sort. For the p_openBLAS < 3, it can be attributed to the highly optimized code that eliminates extraneous operations.*
   e. For your implementation, change the order of the loops that are executed and rerun. Describe what you observe? Explain what might be going on.
      - *The mydgemm was reran with the inner loop over the inner A and B matrices first, and then the nested loops for the C components. As per the table below, the time increases not insignificantly compared to the inner multiplication loop being the inner for loop. It is assumed that the compiler sees the outer loops as being identical for all C components in the original implementation, and can optimize performing the inner loop for each C component. Whereas when the inner multiplication is first, and each C element is then iterated over in the inner loops, the optimization is less obvious and the compiler can't quite rearrange the loops.*

Matrix Size	My MatMult time (s)
20	0.00
100	0.00
500	1.5e-1
1000	2.62
2000	3.63e1

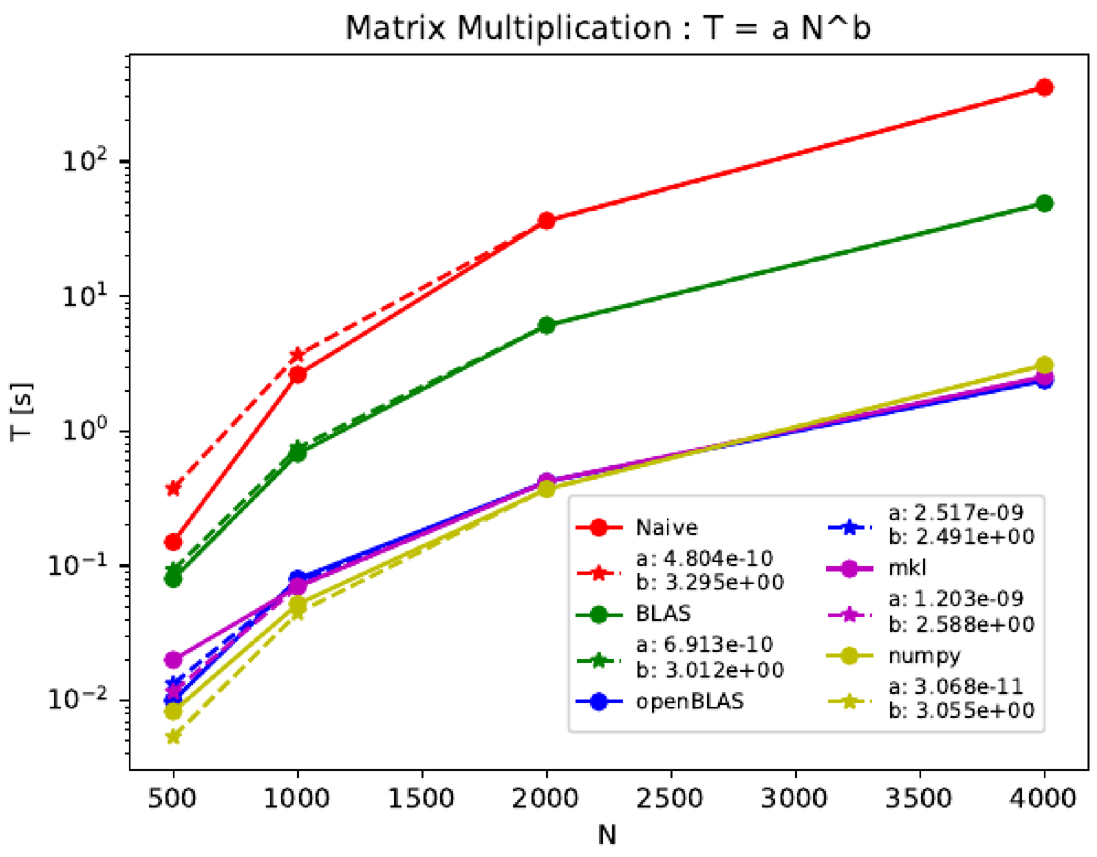

Fig 1: Run times for matrix-matrix multiplication as a function of matrix, for different implementations.

<u>submit.slrm</u>

```
#!/bin/bash


#SBATCH --job-name=NERS570_Lab4
#SBATCH --mail-user=mduschen@umich.edu
#SBATCH --mail-type=END
#SBATCH --cpus-per-task=1
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --mem-per-cpu=4000m
#SBATCH --time=10:00:00
#SBATCH --output=./logs/%x-%j.log

#make run $n

submit=1
file=matmul
ext=out
#lib=mkl_intel_ilp64  #openblas
lib=openblas
#LFLAGS="-L${MKLROOT}/lib/intel64 -Wl,--no-as-needed -lmkl_intel_ilp64"
LFLAGS=""
CCFLAGS="-O2"
if [[ "${ext}" == "out" ]]
then
	echo $lib
	echo g++ -o ${file}.out ${file}.cpp -l${lib} ${LFLAGS} ${CCFLAGS}
	g++ -o ${file}.out ${file}.cpp -l${lib} ${LFLAGS} ${CCFLAGS}
else
	echo python
fi


for n in 20 100 500 1000 2000; # 4000;
do
	echo Running $n
	if [ "${submit}" == 1 ]
	then
		./${file}.${ext} $n
	else
		echo No Submit
	fi
done
```

~~*Extra Credit (**5 pts**)*~~
Install the ATLAS Library and compare this with your previously generated results.
Automatically Tuned Linear Algebra Software (ATLAS)

*Extra Credit (**5 pts**)*
Take the Fortran or C++ program and link it to the MKL library (using the GNU compilers). Generate results for this BLAS implementation, and compare to the other results.

- *Used https://software.intel.com/content/www/us/en/develop/articles/intel-mkl-link-line-advisor.html to determine compiling and linking options. Followed suggested commands for dynamic library with mkl.2018.0.4, and compiled without error.*

- *g++ -o matmul.out matmul.cpp -m64 -I${MKLROOT}/include -L${MKLROOT}/lib/intel64 -Wl,--no-as-needed -lmkl_intel_lp64 -lmkl_sequential -lmkl_core -lpthread -lm -ldl*

- *As can be seen in the plot in Fig 1., mkl performs almost identically to openBLAS.*

Matrix Size	MKL dgemm time (s)
20	0.0
100	0.0
500	2e-2
1000	7e-2
2000	4.2e-1
4000	2.53

Deliverables and how to submit:

For the overall deliverable you may submit a single word doc or pdf based on these instructions. Name the file as follows:

<username>_Lab4.doc (or docx or pdf).

For exercise 1, include the commands you executed for each part of the exercise for compiling. Also include one version of your Slurm script used to generate the results. You will run several cases and this may necessitate several Slurm files. Just provide one. For your naive implementation of matrix-matrix multiply include the code snippet for this routine. For your python code, copy the source into this word document as well (or use the listings environment in LaTeX if you're so inclined).

Include answers for all questions.

Please upload your doc/docx/pdf file to canvas.

For the Extra-Credit include this in your submission with this document.