https://github.com/UCR-HPC/cs211-hw2-solving-large-linear-system-private-MoonlyTower

Rui Liu

862540175

Q1

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 13 & 18 \\ 7 & 54 & 78 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 40 & 57 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

M is the lower triangular,

$$M = \begin{pmatrix} 0 & 0 & 0 \\ 4 & 0 & 0 \\ 7 & 8 & 0 \end{pmatrix} \qquad L = M+I = \begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 7 & 8 & 1 \end{pmatrix}$$

U is the upper triangular,

$$U = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 5 & 6 \\ 0 & 0 & 9 \end{pmatrix}$$

so that, A= LU

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 7 & 8 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 5 & 6 \\ 0 & 0 & 9 \end{pmatrix}$$

Q2

step1:log into the hpc-001

scp -r /Users/ruiliu/Downloads/lapack-3.12.0 ssh -J rui001@bolt.cs.ucr.edu rui001@hpc-001
scp -r /Users/ruiliu/Downloads/lapack-3.12.0 rui001@hpc-001.cs.ucr.edu /home/rui001/CS211-hw/l_BaseKit_p_2024.2.1.100.sh
ssh -J rui001@bolt.cs.ucr.edu rui001@hpc-001

step2:download the MKL library form the Intel on hpc-001

```
The following tools have been installed successfully:

  Intel® oneAPI Math Kernel Library


Installation location: /home/rui001/intel/oneapi
```

```
 Intel® oneAPI Math Kernel Library | 2024.2.2
```

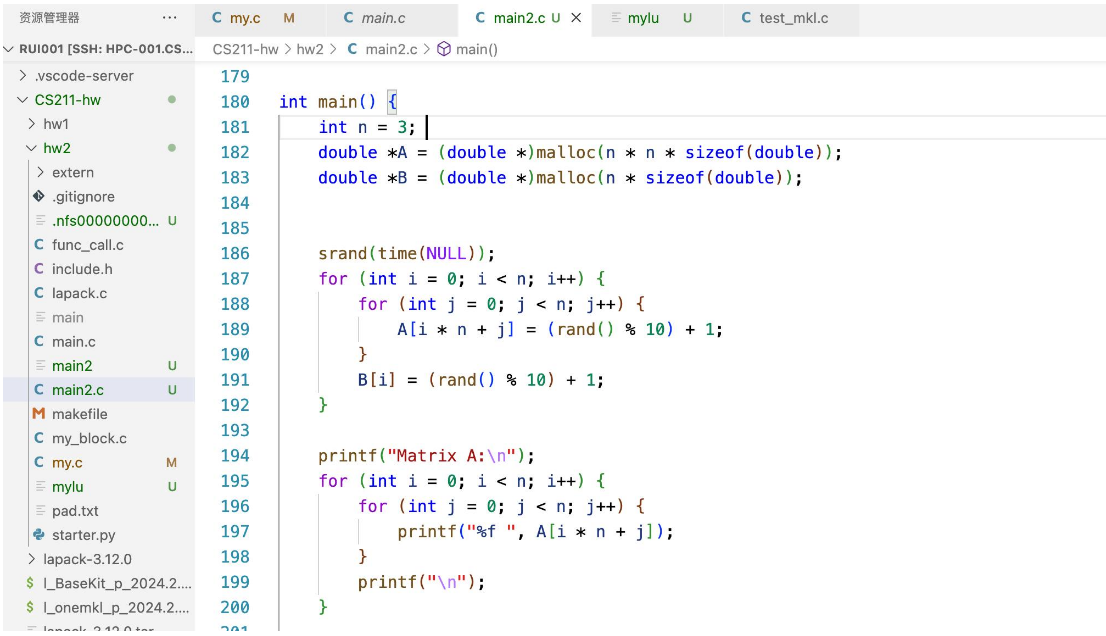
```
[rui001@cluster-001-login-node ~]$ source /home/rui001/intel/oneapi/setvars.sh

:: initializing oneAPI environment ...
   -bash: BASH_VERSION = 4.4.20(1)-release
   args: Using "$@" for setvars.sh arguments:
:: compiler -- latest
:: mkl -- latest
:: tbb -- latest
:: oneAPI environment initialized ::
```

step3:code /compiler and test on file main2.

I give the detailed comments process when using file main2 to debugging and verify the correctness of the code.


```
-0.865367 0.324562 -0.571642 -0.402492 0.546081 -0.143504 0.536445 0.567530 0.988226 -0.115777
-0.522515 1.071072 0.034204 -1.042182 -0.692322 -0.592811 -0.407617 0.442498 1.298387 -0.600616
-0.777394 -0.478120 -1.070278 -1.644396 0.366007 -0.703901 -0.255174 0.936692 1.094184 -0.275433
-0.040643 -0.030346 0.793164 2.070068 -0.181983 0.376359 -0.212329 -1.629902 -1.104095 0.368456
-0.853179 0.875634 0.371728 0.133576 1.788122 0.000349 0.724988 -0.683310 0.123207 0.972633
-0.168867 0.637587 0.068753 -0.042380 -0.012966 1.218015 0.072017 -0.919788 -1.632044 -0.048442
-0.156482 0.620882 -0.733388 -0.151293 0.662255 -0.091264 -1.205219 -0.373143 -0.167320 -0.605230
0.129645 -0.798361 0.377964 -0.058655 -0.804866 -0.665317 0.554355 -2.110945 -0.884717 0.688650
0.895295 0.513499 -0.196381 -0.148092 0.034006 -0.415157 0.185580 0.370581 -2.675141 -0.580604
-0.204266 -0.661267 0.589822 -0.280163 -0.662701 0.031582 0.472380 0.339359 -0.361098 -0.495711

Matrix A after LU factorization:
-0.865367 0.324562 -0.571642 -0.402492 0.546081 -0.143504 0.536445 0.567530 0.988226 -0.115777
-0.522515 1.071072 0.034204 -1.042182 -0.692322 -0.592811 -0.407617 0.442498 1.298387 -0.600616
-0.777394 -0.478120 -1.070278 -1.644396 0.366007 -0.703901 -0.255174 0.936692 1.094184 -0.275433
-0.040643 -0.030346 0.793164 2.070068 -0.181983 0.376359 -0.212329 -1.629902 -1.104095 0.368456
-0.853179 0.875634 0.371728 0.133576 1.788122 0.000349 0.724988 -0.683310 0.123207 0.972633
-0.168867 0.637587 0.068753 -0.042380 -0.012966 1.218015 0.072017 -0.919788 -1.632044 -0.048442
-0.156482 0.620882 -0.733388 -0.151293 0.662255 -0.091264 -1.205219 -0.373143 -0.167320 -0.605230
0.129645 -0.798361 0.377964 -0.058655 -0.804866 -0.665317 0.554355 -2.110945 -0.884717 0.688650
0.895295 0.513499 -0.196381 -0.148092 0.034006 -0.415157 0.185580 0.370581 -2.675141 -0.580604
-0.204266 -0.661267 0.589822 -0.280163 -0.662701 0.031582 0.472380 0.339359 -0.361098 -0.495711
Pivot indices:
7 9 5 0 8 1 3 2 6 4

tempB[0] after pivoting: -0.949390
tempB[1] after pivoting: -0.378045
tempB[2] after pivoting: 0.330522
tempB[3] after pivoting: 0.348874
tempB[4] after pivoting: -0.101949
tempB[5] after pivoting: 0.916656
tempB[6] after pivoting: 0.777232
tempB[7] after pivoting: 0.784441
tempB[8] after pivoting: -0.366271
tempB[9] after pivoting: -0.485835
B[0] before forward substitution (after pivoting): -0.949390
B[1] before forward substitution (after pivoting): -0.378045
B[2] before forward substitution (after pivoting): 0.330522
B[3] before forward substitution (after pivoting): 0.348874
B[4] before forward substitution (after pivoting): -0.101949
B[5] before forward substitution (after pivoting): 0.916656
B[6] before forward substitution (after pivoting): 0.777232
B[7] before forward substitution (after pivoting): 0.784441
B[8] before forward substitution (after pivoting): -0.366271
B[9] before forward substitution (after pivoting): -0.485835
B[0] after forward substitution: -0.949390
```


```
0.129645 -0.798361 0.377964 -0.058655 -1.439199 -0.810646 -1.299551 -1.155876 0.009189 -0.397472
0.895295 0.513499 -0.196381 -0.148092 0.060807 -0.505655 -0.228909 -0.492904 -2.352308 -0.384535
-0.156482 0.620882 -0.733388 -0.151293 1.184193 -0.110929 -0.731664 -0.741725 0.063221 0.043322
-0.853179 0.875634 0.371728 0.133576 1.788122 0.000349 0.724988 -0.683310 0.123207 0.972633
-0.168867 0.637587 0.068753 -0.042380 -0.023184 1.218011 0.062617 -0.910928 -1.633641 -0.061053

Step 4: Current max = 1.184990 at row 4
Swapping rows 4 and 8
Row 5 after division: 0.129645 -0.798361 0.377964 -0.058655 -0.804866 -0.810646 -1.299551 -1.155876 0.009189 -0.397472
Row 6 after division: 0.895295 0.513499 -0.196381 -0.148092 0.034006 -0.505655 -0.228909 -0.492904 -2.352308 -0.384535
Row 7 after division: -0.156482 0.620882 -0.733388 -0.151293 0.662255 -0.110929 -0.731664 -0.741725 0.063221 0.043322
Row 8 after division: -0.204266 -0.661267 0.589822 -0.280163 -0.662701 0.038236 -1.047497 -0.468853 0.453519 -0.984350
Row 9 after division: -0.168867 0.637587 0.068753 -0.042380 -0.012966 1.218011 0.062617 -0.910928 -1.633641 -0.061053
Matrix A after step 4:
-0.865367 0.324562 -0.571642 -0.402492 0.546081 -0.143504 0.536445 0.567530 0.988226 -0.115777
-0.522515 1.071072 0.034204 -1.042182 -0.692322 -0.592811 -0.407617 0.442498 1.298387 -0.600616
-0.777394 -0.478120 -1.070278 -1.644396 0.366007 -0.703901 -0.255174 0.936692 1.094184 -0.275433
-0.040643 -0.030346 0.793164 2.070068 -0.181983 0.376359 -0.212329 -1.629902 -1.104095 0.368456
-0.853179 0.875634 0.371728 0.133576 1.788122 0.000349 0.724988 -0.683310 0.123207 0.972633
0.129645 -0.798361 0.377964 -0.058655 -0.804866 -0.810366 -0.716033 -1.705848 0.108354 0.385368
0.895295 0.513499 -0.196381 -0.148092 0.034006 -0.505667 -0.253562 -0.469668 -2.356498 -0.417611
-0.156482 0.620882 -0.733388 -0.151293 0.662255 -0.111160 -1.211791 -0.289200 -0.018374 -0.600809
-0.204266 -0.661267 0.589822 -0.280163 -0.662701 0.038468 -0.567047 -0.921683 0.535168 -0.339785
-0.168867 0.637587 0.068753 -0.042380 -0.012966 1.218015 0.072017 -0.919788 -1.632044 -0.048442

Step 5: Current max = 0.810366 at row 5
Swapping rows 5 and 9
Row 6 after division: 0.895295 0.513499 -0.196381 -0.148092 0.034006 -0.415157 -0.253562 -0.469668 -2.356498 -0.417611
Row 7 after division: -0.156482 0.620882 -0.733388 -0.151293 0.662255 -0.091264 -1.211791 -0.289200 -0.018374 -0.600809
Row 8 after division: -0.204266 -0.661267 0.589822 -0.280163 -0.662701 0.031582 -0.567047 -0.921683 0.535168 -0.339785
Row 9 after division: 0.129645 -0.798361 0.377964 -0.058655 -0.804866 -0.665317 -0.716033 -1.705848 0.108354 0.385368
Matrix A after step 5:
-0.865367 0.324562 -0.571642 -0.402492 0.546081 -0.143504 0.536445 0.567530 0.988226 -0.115777
-0.522515 1.071072 0.034204 -1.042182 -0.692322 -0.592811 -0.407617 0.442498 1.298387 -0.600616
-0.777394 -0.478120 -1.070278 -1.644396 0.366007 -0.703901 -0.255174 0.936692 1.094184 -0.275433
-0.040643 -0.030346 0.793164 2.070068 -0.181983 0.376359 -0.212329 -1.629902 -1.104095 0.368456
-0.853179 0.875634 0.371728 0.133576 1.788122 0.000349 0.724988 -0.683310 0.123207 0.972633
-0.168867 0.637587 0.068753 -0.042380 -0.012966 1.218015 0.072017 -0.919788 -1.632044 -0.048442
0.895295 0.513499 -0.196381 -0.148092 0.034006 -0.415157 -0.223664 -0.851524 -3.034051 -0.437722
-0.156482 0.620882 -0.733388 -0.151293 0.662255 -0.091264 -1.205219 -0.373143 -0.167320 -0.605230
-0.204266 -0.661267 0.589822 -0.280163 -0.662701 0.031582 -0.569322 -0.892634 0.586711 -0.338255
0.129645 -0.798361 0.377964 -0.058655 -0.804866 -0.665317 -0.668119 -2.317798 -0.977472 0.353138

Step 6: Current max = 0.223664 at row 6
Swapping rows 6 and 7
Row 7 after division: 0.895295 0.513499 -0.196381 -0.148092 0.034006 -0.415157 0.185580 -0.851524 -3.034051 -0.437722
```




```c

int main() {
    int n = 3;
    double *A = (double *)malloc(n * n * sizeof(double));
    double *B = (double *)malloc(n * sizeof(double));


    srand(time(NULL));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            A[i * n + j] = (rand() % 10) + 1;
        }
        B[i] = (rand() % 10) + 1;
    }

    printf("Matrix A:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%f ", A[i * n + j]);
        }
        printf("\n");
    }
```

test on a main2 as shown below:

gcc main2.c -o main2 -lm

./main2

```
● [rui001@cluster-001-login-node hw2]$ gcc main2.c -o main2 -lm
● [rui001@cluster-001-login-node hw2]$ ./main2
Matrix A:
6.000000 3.000000 1.000000
1.000000 3.000000 4.000000
3.000000 7.000000 5.000000
Vector B:
2.000000 10.000000 5.000000
Initial Matrix A:
6.000000 3.000000 1.000000
1.000000 3.000000 4.000000
3.000000 7.000000 5.000000
Initial Vector B:
2.000000 10.000000 5.000000

Step 0: Current max = 6.000000 at row 0
Row 1 after division: 0.166667 3.000000 4.000000
Row 2 after division: 0.500000 7.000000 5.000000
Matrix A after step 0:
6.000000 3.000000 1.000000
0.166667 2.500000 3.833333
0.500000 5.500000 4.500000

Step 1: Current max = 2.500000 at row 1
Swapping rows 1 and 2
Row 2 after division: 0.166667 0.454545 3.833333
Matrix A after step 1:
6.000000 3.000000 1.000000
0.500000 5.500000 4.500000
0.166667 0.454545 1.787879

Matrix A after LU factorization:
6.000000 3.000000 1.000000
0.500000 5.500000 4.500000
0.166667 0.454545 1.787879
Pivot indices:
0 2 1
```

```
tempB[0] after pivoting: 2.000000
tempB[1] after pivoting: 5.000000
tempB[2] after pivoting: 10.000000
B[0] before forward substitution (after pivoting): 2.000000
B[1] before forward substitution (after pivoting): 5.000000
B[2] before forward substitution (after pivoting): 10.000000
B[0] after forward substitution: 2.000000
B[i]: 4.000000
A[i * n + j]]: 0.500000
B[1] after forward substitution: 4.000000
B[i]: 9.666667
A[i * n + j]]: 0.166667
B[i]: 7.848485
A[i * n + j]]: 0.454545
B[2] after forward substitution: 7.848485
Vector B after forward substitution:
2.000000 4.000000 7.848485

B[i]: 7.848485
A[i * n + i]]: 1.787879
#########B[2] after backward substitution: 4.389831
B[i]: 4.000000
A[i * n + i]]: 5.500000
#########B[1] after backward substitution: -2.864407
B[i]: 2.000000
A[i * n + i]]: 6.000000
#########B[0] after backward substitution: 1.033898
Vector B after backward substitution (Solution X):
1.033898 -2.864407 4.389831

Solution X:
1.033898 -2.864407 4.389831
```

step4:code /compiler and test on file main.

complier：

cd /path/to/your/project
CURDIR=$(pwd)
export CURDIR
LD_LIBRARY_PATH=$CURDIR/extern:/act/opt/intel/composer_xe_2013.3.1/mkl/lib/intel64:$LD_LIBRARY_PATHexport LD_LIBRARY_PATH
make

I have shown the final experiment outcome as above.

./main function num

```
[rui001@cluster-001-login-node hw2]$ ./main my 1000
n=1000, pad=1
time=0.146002s
[rui001@cluster-001-login-node hw2]$ ./main my 2000
n=2000, pad=1
time=1.642781s
[rui001@cluster-001-login-node hw2]$ ./main my 3000
n=3000, pad=1
time=6.559644s
[rui001@cluster-001-login-node hw2]$ ./main my 4000
n=4000, pad=1
time=15.823135s
[rui001@cluster-001-login-node hw2]$ ./main my 5000
n=5000, pad=1
time=30.574100s

[rui001@cluster-001-login-node hw2]$ ./main lapack 100
n=100, pad=1
time=0.040017s
[rui001@cluster-001-login-node hw2]$ ./main lapack 1000
n=1000, pad=1
time=0.073281s
[rui001@cluster-001-login-node hw2]$ ./main lapack 1000
n=1000, pad=1
time=0.074497s
[rui001@cluster-001-login-node hw2]$ ./main lapack 2000
n=2000, pad=1
time=0.239922s
[rui001@cluster-001-login-node hw2]$ ./main lapack 3000
n=3000, pad=1
time=0.707063s
[rui001@cluster-001-login-node hw2]$ ./main lapack 4000
n=4000, pad=1
time=1.424908s
[rui001@cluster-001-login-node hw2]$ ./main lapack 5000
n=5000, pad=1
time=2.850201s
[rui001@cluster-001-login-node hw2]$
```

srun main function num

```
[rui001@cluster-001-login-node hw2]$ export LD_LIBRARY_PATH=/home/act-software/opt/intel/cor
b/intel64:/home/rui001/CS211-hw/hw2/extern:$LD_LIBRARY_PATH
[rui001@cluster-001-login-node hw2]$ srun main my 1000
n=1000, pad=1
time=0.133002s
[rui001@cluster-001-login-node hw2]$ srun main my 2000
n=2000, pad=1
time=1.292453s
[rui001@cluster-001-login-node hw2]$ srun main my 3000
n=3000, pad=1
time=5.619809s
[rui001@cluster-001-login-node hw2]$ srun main my 4000
n=4000, pad=1
time=14.073875s
[rui001@cluster-001-login-node hw2]$ srun main my 5000
n=5000, pad=1
time=28.704089s
```

```
[rui001@cluster-001-login-node hw2]$ srun main lapack 1000
n=1000, pad=1
time=0.039766s
[rui001@cluster-001-login-node hw2]$ srun main lapack 2000
n=2000, pad=1
time=0.203340s
[rui001@cluster-001-login-node hw2]$ srun main lapack 3000
n=3000, pad=1
time=0.640516s
[rui001@cluster-001-login-node hw2]$ srun main lapack 4000
n=4000, pad=1
time=1.304958s
[rui001@cluster-001-login-node hw2]$ srun main lapack 5000
n=5000, pad=1
time=2.614344s
```

Compare the outcome of my function and the lapack:

We firstly calculate the Number of Floating-Point Operations (FLOPs).

For an LU decomposition of an n×n matrix, the number of operations can be approximated by $\frac{n^3+6n^2-4n}{3}$.

So Gflops= $\frac{n^3+6n^2-4n}{3}$ / time(seconds)* 10^9.

Matrix Size 1000

FLOPs=3.35332*10^8

my function:Time: 0.133002s

Gflops=3.35332*10^8 / 0.133002×10^9≈2.5212 Gflops

LAPACK:Time: 0.039766s

Gflops=3.35332*10^8 / 0.039766×109≈8.43051 Gflops

Matrix Size 2000

FLOPs=2.674664*10^9

my function:Time: 1.242953s

Gflops=2.674664*10^9 / 1.242953×10^9≈2.151826 Gflops

LAPACK:Time: 0.203340s

Gflops=2.674664*10^9 / 0.203340×10^9=13.153654 Gflops

Matrix Size 3000

FLOPs=9.017996*10^9

my function:Time: 5.619809s
Gflops=9.017996*10^9 / 5.619809×10^9≈1.60468Gflops
LAPACK:Time: 0.640516s
Gflops=9.017996*10^9 /   0.640516×10^9≈14.0793 Gflops

Matrix Size 4000
FLOPs=2.1365328*10^10
my function:Time: 17.044483s
Gflops=2.1365328*10^10 / 14.073875×10^9≈ 1.518Gflops
LAPACK:Time: 1.424908s
Gflops=2.1365328*10^10 / 1.304958×10^9≈16.3724Gflops

Matrix Size 5000
FLOPs=4.171666*10^10
my function:Time: 33.782017s
Gflops=4.171666*10^10 / 28.704089×10^9≈ 1.4534Gflops
LAPACK:Time: 2.850201s
Gflops=4.171666*10^10 / 2.614344×10^9≈18.0412 Gflops

Q3

A= $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 9 & 12 & 15 \\ 3 & 26 & 41 & 49 \\ 5 & 40 & 107 & 135 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 9 & 12 & 15 \\ 3 & 26 & 41 & 49 \\ 5 & 40 & 107 & 135 \end{pmatrix}$

$= \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 5 & 12 & 15 \\ 3 & 20 & 41 & 49 \\ 5 & 30 & 107 & 135 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 5 & 12 & 15 \\ 3 & 4 & 41 & 49 \\ 5 & 6 & 107 & 135 \end{pmatrix}$

$= \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 5 & 6 & 7 \\ 3 & 4 & 41 & 49 \\ 5 & 6 & 107 & 135 \end{pmatrix}$

we calculate $\begin{pmatrix} 3 & 4 \\ 5 & 6 \end{pmatrix} \begin{pmatrix} 3 & 4 \\ 6 & 7 \end{pmatrix} = \begin{pmatrix} 33 & 44 \\ 51 & 62 \end{pmatrix}$

so A= $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 5 & 6 & 7 \\ 3 & 4 & 8 & 9 \\ 5 & 6 & 56 & 73 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 5 & 6 & 7 \\ 3 & 4 & 8 & 9 \\ 5 & 6 & 7 & 10 \end{pmatrix}$

L is the lower triangular, which is $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 3 & 4 & 1 & 0 \\ 5 & 6 & 7 & 1 \end{pmatrix}$,

U is the upper triangular, which is $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 0 & 5 & 6 & 7 \\ 0 & 0 & 8 & 9 \\ 0 & 0 & 0 & 10 \end{pmatrix}$.

Q4

```
void mydgemm(int n, double *A, double *B, double *C) {
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < n; i += BLOCK_SIZE) {
        for (int j = 0; j < n; j += BLOCK_SIZE) {
            for (int k = 0; k < n; k += BLOCK_SIZE) {
                for (int ii = i; ii < i + BLOCK_SIZE && ii < n; ++ii) {
                    for (int jj = j; jj < j + BLOCK_SIZE && jj < n; ++jj) {
                        double sum = 0.0;
                        for (int kk = k; kk < k + BLOCK_SIZE && kk < n; ++kk) {
                            sum += A[ii * n + kk] * B[kk * n + jj];
                        }
                        C[ii * n + jj] += sum;
                    }
                }
            }
        }
    }
}
```

```
void mydgetrf_block(int n, double *A) {
    for (int k = 0; k < n; k += BLOCK_SIZE) {
        int end = k + BLOCK_SIZE < n ? k + BLOCK_SIZE : n;

        // Factorize the diagonal block
        for (int i = k; i < end; ++i) {
            for (int j = k; j < i; ++j) {
                A[i * n + j] /= A[j * n + j];
                for (int l = j + 1; l < end; ++l) {
                    A[i * n + l] -= A[i * n + j] * A[j * n + l];
                }
            }
        }

        // Update the trailing submatrix
        #pragma omp parallel for collapse(2)
        for (int i = end; i < n; ++i) {
            for (int j = k; j < end; ++j) {
                A[i * n + j] /= A[j * n + j];
                for (int l = end; l < n; ++l) {
                    A[i * n + l] -= A[i * n + j] * A[j * n + l];
                }
            }
        // Perform matrix multiplication for the trailing submatrix
        #pragma omp parallel for collapse(2)
        for (int i = end; i < n; i += BLOCK_SIZE) {
            for (int j = end; j < n; j += BLOCK_SIZE) {
                for (int ii = i; ii < i + BLOCK_SIZE && ii < n; ++ii) {
                    for (int jj = j; jj < j + BLOCK_SIZE && jj < n; ++jj) {
                        double sum = 0.0;
                        for (int kk = k; kk < end; ++kk) {
                            sum += A[ii * n + kk] * A[kk * n + jj];
                        }
                        A[ii * n + jj] -= sum;
                    }
                }
            }
        }
    }
}
```

test result:

on block size=32

```
n=1000, pad=1
Time taken: 0.068316 seconds

n=2000, pad=1
Time taken: 0.537873 seconds

n=3000, pad=1
Time taken: 1.862734 seconds

n=4000, pad=1
Time taken: 4.064807 seconds

n=5000, pad=1
Time taken: 8.126369 seconds
```

on block size=64:

```
n=1000, pad=1
Time taken: 0.074267 seconds

n=2000, pad=1
Time taken: 0.535702 seconds

n=3000, pad=1
Time taken: 1.738807 seconds

n=4000, pad=1
Time taken: 4.140256 seconds

n=5000, pad=1
Time taken: 8.497961 seconds
```

on block size=128

```
n=1000, pad=1
Time taken: 0.099635 seconds

n=2000, pad=1
Time taken: 0.577570 seconds

n=3000, pad=1
Time taken: 1.811601 seconds

n=4000, pad=1
Time taken: 4.347364 seconds

n=5000, pad=1
Time taken: 8.669026 seconds
```

We can see that compare to non-block algorithm is much quicker, especially as the n size increase, the block pattern get a better time cost than non-block one.

```
[rui001@cluster-001-login-node hw2]$ export LD_LIBRARY_PATH=/home/act-software/opt/intel/cor
b/intel64:/home/rui001/CS211-hw/hw2/extern:$LD_LIBRARY_PATH
[rui001@cluster-001-login-node hw2]$ srun main my 1000
n=1000, pad=1
time=0.133002s
[rui001@cluster-001-login-node hw2]$ srun main my 2000
n=2000, pad=1
time=1.292453s
[rui001@cluster-001-login-node hw2]$ srun main my 3000
n=3000, pad=1
time=5.619809s
[rui001@cluster-001-login-node hw2]$ srun main my 4000
n=4000, pad=1
time=14.073875s
[rui001@cluster-001-login-node hw2]$ srun main my 5000
n=5000, pad=1
time=28.704089s
```

We get the conclusion that maybe as the block size is too large, the performance is began to lose, so the appreciate size like 32 or 64 would be much better.