University of California, Riverside
Department of Computer Science & Engineering

Title: **High-Performance Computing – Project 2 Solving Large Linear System**

Student Name:
Mahbod Afarin

Student ID:
862186340

Fall 2020

Part 1: In the table 1 and table 2 we can see the execution time (seconds) and performance (Gflops) for LAPAK and naïve LU. As we can see in the tables, the execution time for LAPAK is much better compared to the execution time for naïve LU. It requires ${2n^3}/{3}$ operations for basic LU factorization and in total it requires $2n^2$ operations.

Table 1: The execution time and performance for MKL LAPAK

n	1024	2048	3072	4096	5120
Execution time (Seconds)	0.120017 sec	0.655016 sec	1.830898 sec	4.530101 sec	8.285191 sec
Performance (Gflops)	5.96438740067	8.74272240882	10.5562149459	10.1130161316	10.7998096041

Table 2: The execution time and performance for naïve LU

n	1024	2048	3072	4096	5120
Execution time (Seconds)	4.178531 sec	34.906586 sec	115.258259 sec	288.866282 sec	542.736801 sec
Performance (Gflops)	0.17131089434	0.16405566162	0.16768735706	0.1585958187	0.16486533651

In figure 1, I show the execution time for LAPAK for different matrix size. As we can see in the figure, the execution time will rise as the matrix dimension is growing.

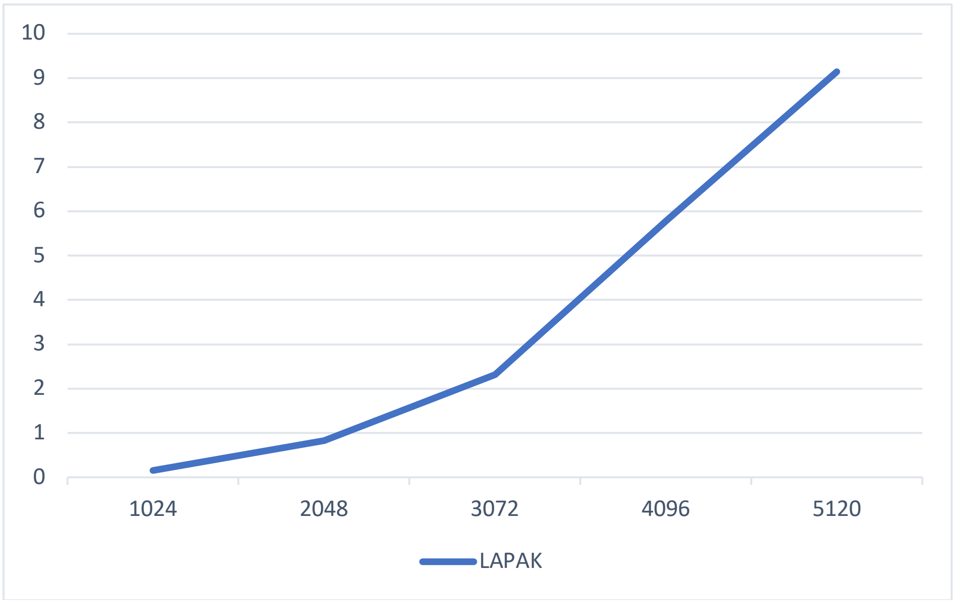


Figure 1: Comparing the execution time for different matrix size for LAPAK

In figure 2, I show the execution time for naïve LU for different matrix size. As we can see in the figure, the execution time will rise as the matrix dimension is growing.

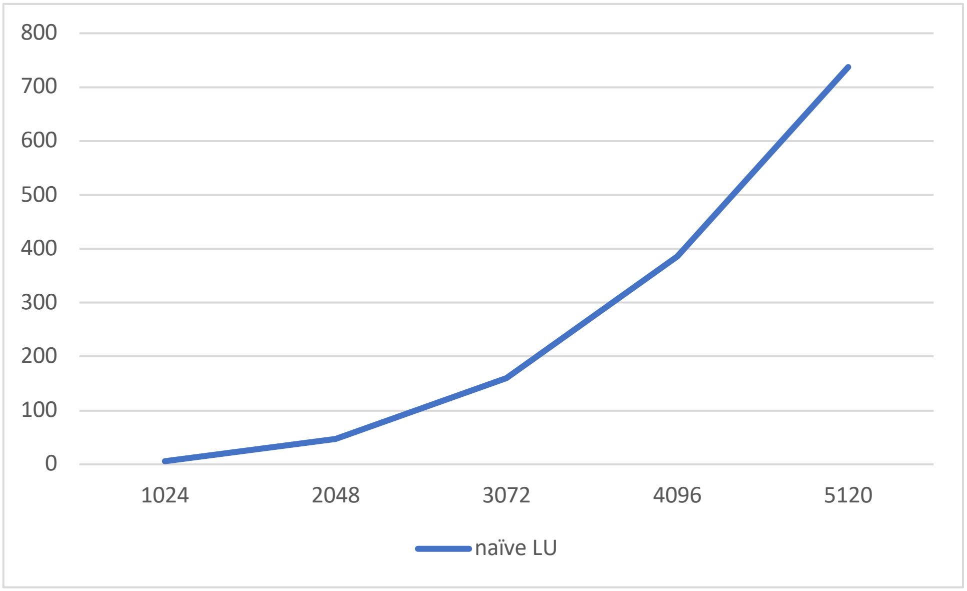


Figure 2: Comparing the execution time for different matrix size for naïve LU

Part 2: In this part first, I implemented blocked GEPP. There are two main problem with basic GE algorithm:

- The first problem is when $A(i,i)$ becomes zero or very small. As we can see in the figure 3, it can lead to the problem if $A(i,i)$ becomes very small or zero because $A(i,i)$ is located denominator of the equation.
- The second problem is the low performance of the algorithm. As we can see in figure 3, in the algorithm we use BLAS level 1 and BLAS level 2 in the algorithm and they have a very low performance compared to the BLAS level 3. In the figure 4, we can see the performance of BLAS 1, BLAS 2, and BLAS 3. As we can see in the figure, the performance of BLAS 3 is much better than the performance of BLAS 1 and BLAS 2.

```
$for\ i = 1\ to\ n-1$
    $A(i+1{:}n,i) = A(i+1{:}n,i)\ /\ A(i,i)$ ... BLAS 1 (scale a vector)
    $A(i+1{:}n,i+1{:}n) = A(i+1{:}n,i+1{:}n)$ ... BLAS 2 (rank-1 update)
        $- A(i+1{:}n,i) * A(i,i+1{:}n)$
```

Figure 3: Basic GE algorithm

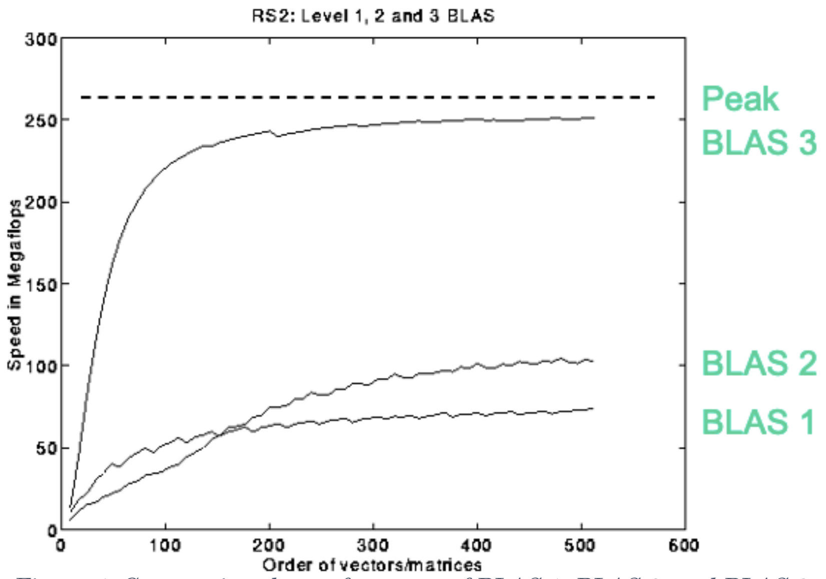


Figure 4: Comparting the performance of BLAS 1, BLAS 2, and BLAS 3

For solving the first problem we need to find the maximum element of that column and we name it $A(k,i)$ and then swap it with $A(i,i)$. If $A(k,i)$ is zero it means and the A matrix is singular and if $A(i,i)$ is not zero, we will do swapping and the we will do the computation. For the second problem we should use blocking technique. In this technique we will do the computation block by block and we will delay the update of the tanning matrix. In the basic GE algorithm, we processed the matrix row by row, but in blocked GEPP we will process the matrix column by column. We can see the main idea of the blocked GEPP algorithm in the figure 5. We have step b which determines the number of the columns which we are going to process at a time. Then we will apply BLAS 2 to the blue part of the figure 6 to factorize it and we will get $A(ib{:}\,n\,, ib{:}\,end)$. Then we will update the pink park of the figure 6 in the same way as before.

```
for   ib = 1 to n-1 step b      … Process matrix b columns at a time
    end = ib + b-1               … Point to end of block of b columns
    apply BLAS2 version of GEPP to  get A(ib:n , ib:end) = P' * L' * U'
    … let LL denote the strict lower triangular part of A(ib:end , ib:end) + I
    A(ib:end , end+1:n) = LL^-1 * A(ib:end , end+1:n)        … update next b rows of U
    A(end+1:n , end+1:n ) = A(end+1:n , end+1:n )
                - A(end+1:n , ib:end) * A(ib:end , end+1:n)
                                   … apply delayed updates with single matrix-multiply
                                   … with inner dimension b
```

Figure 5: Blocked GEPP algorithm

We named it delay update because we will first update the blue part of the figure 6 and then update the pink part. It means that we are delaying the update of the oink park. Finally, we should update the green part. The green part is BLAS 3 which is the matrix multiplication. The performance of BLAS 3 is much higher than the BLAS 1 and BLAS 2 and that is why the block GEPP has better performance compared to the basic GE. So, in total we will do blocked GEPP in three steps. The computing of the pink part is n^2b and the computation of the green part is n^3 which is the major computation of this algorithm. Therefore, the computation of this algorithm will be n^3. The total complexity of this algorithm is (2n^3)/3.

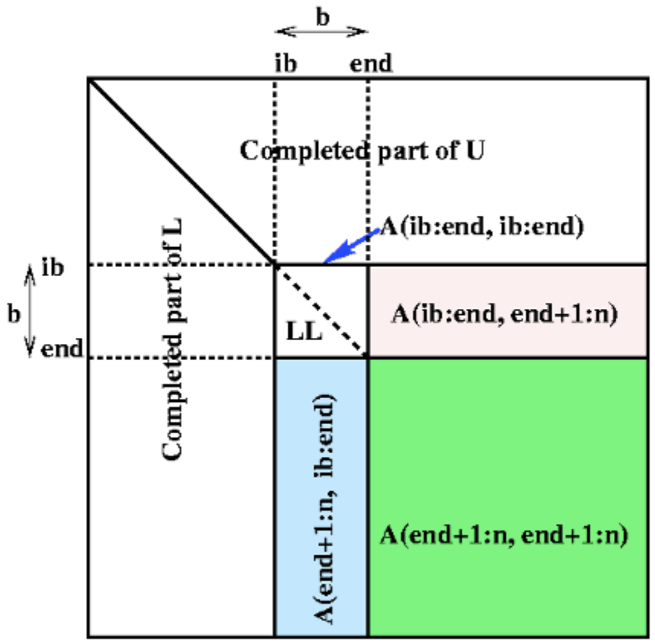


Figure 6: blocked GEPP algorithm using BLAS level 3

For the implementation part first, I implemented the basic blocked GEPP without any optimization and then I tried to optimized it. In the table 3, I shewed the execution time in seconds and the performance in Gflops for blocked GEPP. As we can see in the table, the execution time improves spectacularly compared to the previous versions, because we used BLAS 3 in the implementation of blocked GEPP and BLAS 3 has a better performance compared to the BLAS 1 and BLAS 2.

Table 3: The execution time and performance for blocked GEPP

n	1024	2048	3072	4096	5120
Execution time (Seconds)	1.732467 sec	11.834723 sec	38.872354 sec	90.764567 sec	165.568436 sec
Performance (Gflops)	0.41318413722	0.48388315141	0.49720047394	0.50474525472	0.54043202614

In order to optimize the blocked GEPP, I did these changes in the basic blocked GEPP algorithm.

1. The first optimization is to maximize the cache hit in the pink part of the figure 6. We will do the computation of $A(ib\colon end\ , end+1\colon n)$ for each row and doing the computation row by row may lead to cache miss and it will happen when the cache size is bigger than each row. Therefore, loading the elements each

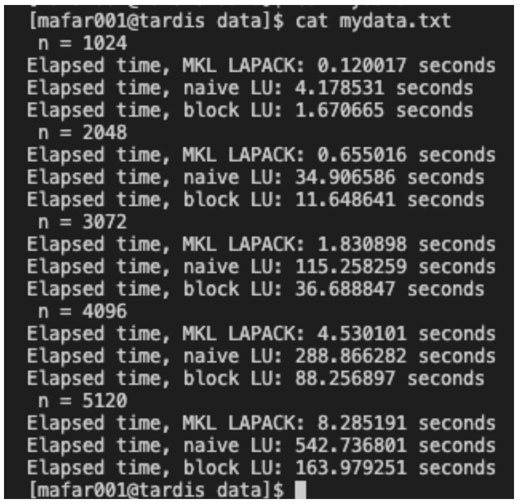
time from the memory will lead to huge additional execution time. In order to avoid that we can tile the computation and with this technique we will maximize the cache reuse for that part.

2. We can also maximize the cache hit in the blue part of the figure 6. In the figure 6, the factorization part (blue part) will completed column by column. Therefore, if the size of the column becomes too large, the cache misses will increase spectacularly because the size of the column will be bigger than the size of the cache. In order to avoid that we can tile it to maximize the cache hit for that part.
3. The third optimization is optimizing the matrix multiplication for this algorithm. I removed all the boundary checks and maximize the register reuse for that to maximize the implementation of the matrix multiplication.
4. Using BLAS 3 make the program faster compared to use BLAS 1 and BLAS 2.
5. We can use register reuse in the computations to maximize register reuse and increase the execution time of the program.

We applied the optimization scenarios to the program and in the table 4, we can see the results. As we can see in the table 4, the optimized version of blocked GEPP has a better performance compare to the naïve version.

Table 4: The execution time and performance for blocked GEPP

n	1024	2048	3072	4096	5120
Execution time (Seconds)	1.6706056 sec	11.648641 sec	36.688847 sec	88.256897 sec	163.979251 sec
Performance (Gflops)	0.42848406749	0.49161297539	0.52679095726	0.51908673483	0.5456695575



```
[mafar001@tardis data]$ cat mydata.txt
 n = 1024
Elapsed time, MKL LAPACK: 0.120017 seconds
Elapsed time, naive LU: 4.178531 seconds
Elapsed time, block LU: 1.670665 seconds
 n = 2048
Elapsed time, MKL LAPACK: 0.655016 seconds
Elapsed time, naive LU: 34.906586 seconds
Elapsed time, block LU: 11.648641 seconds
 n = 3072
Elapsed time, MKL LAPACK: 1.830898 seconds
Elapsed time, naive LU: 115.258259 seconds
Elapsed time, block LU: 36.688847 seconds
 n = 4096
Elapsed time, MKL LAPACK: 4.530101 seconds
Elapsed time, naive LU: 288.866282 seconds
Elapsed time, block LU: 88.256897 seconds
 n = 5120
Elapsed time, MKL LAPACK: 8.285191 seconds
Elapsed time, naive LU: 542.736801 seconds
Elapsed time, block LU: 163.979251 seconds
[mafar001@tardis data]$ 
```

Figure 7: The output of the program