CS 211: High Performance Computing Project 2

Zhenyu Yang (862187998)

Fall 2019

Solving Large Linear Systems

1.1 naive GE

The basic idea is using convenience of triangular matrix to solve large linear equations. For basic LU factorization, it requires $\frac{2n^3}{3}$ operations. For two substitutions(forward and backward), it totally requires $2n^2$ operations. The performance (i.e., Gflops) of my naive LU factorization and LAPACK version are shown as follows.

Table 1: **LAPACK** Performance

n	runtime(s)	performance(Gflops)
1002	0.131762	5.10529
2001	0.541297	9.88246
3000	1.712678	10.5204
4002	3.496786	12.2292
5001	6.609571	12.6231

Table 2: **naive LU** Performance

n	runtime(s)	performance(Gflops)
1002	4.064462	0.165503
2001	32.843274	0.162875
3000	111.804068	0.161157
4002	264.094468	0.161922
5001	516.365470	0.161578

1.2 Blocked GEPP

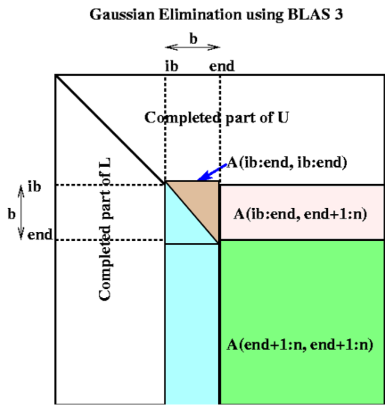

Figure 1: Blocked GEPP

Blocked GEPP using BLAS3 to maximize performance. First, I finished a basic version, and then, I tried some basic techniques to optimize the code. Here is what I did.

the Original version I named **'mydgetrf_non_squrare_naive(...)'**, which has a lot of extra operations to achieve , such as inverse LL^-1, transpose LL^-1, use BLAS-3 matrix multiplication to solve the part of U and large amount of memory allocation, copy and release.

1. Skip computations of LL^-1, directly calculate the part of U **A(ib:end , end+1:n)**.
   This step saved **n*b** operations from inverse LL^-1 and transpose LL^-1, and also saved $\boldsymbol{2 * b^2 * sizeof(double)}$ bytes space and according memory operations.

2. Tile the **A(ib:n, ib:end)** factorization and us 3x3 registers block to maximize cache reuse and register reuse.
   This step optimized the basic LU part, because the LU factorization have to update **a large square** by large amount of 1*1 double sub-multiplication, which column size is **too big to cause great cache misses**. So tile it to increase cache hit and use register to accelerate computations.

3. Tile the computations for **A(ib:end , end+1:n)** to maximize cache reuse.
   Again. This part optimized cache reuse by tiling for reusing completed part. Computation of **A(ib:end , end+1:n)** needs to be ordered row by row. However, completing one entire row may exceeds the cache size, which means the begin of this row may not available in cache and they need to be used many times to generate the following elements at same columns. Thus, tiling it to make sure **all computations of a column finished by caching one times**.

4. Use BLAS3 matrices multiplication to compute **A(end+1:n , end+1:n)**

5. reduce the parameters of mydgemm, and try to make it inline(maybe not work at -O0 which without compiler optimization)
   This step optimize the function invoking from compiling side by saving stack using.

The optimized funtion I named **'mydgetrf_non_squrare(...)'** All of this bring me almost **0.1s** performance improvement in **n = 2001**(almost **5%** improvement).
To find out which part has potential to continue to be optimized deeply. I tested different parts of this code. The result shows that the dgemm part is the **bottleneck** of performance, **it costs almost 1.8s when n = 2001(total is 1.84, which means it consumes almost 99% time)**.

1.3 Optimized Matrices Multiplication

Considered the **bottleneck** is the Matrices Multiplication, so optimizing the **mygemm()** is necessary.

From last lab1, I achieved **11.2 seconds** for **2048*2048 Matrices Multiplication** from **mygemm()**. which is extremely slow for solving linear equations. Although I have **removed almost all boundary check** to achieve **4.82 seconds** for **2048*2048 Matrices Multiplication** in previous part, it is far from my expectation.

To improve deeply the performance of Matrices Multiplication, I used the **local variable** to store the common index, and I also tried put those index in **integer registers**. This time it achieved **4.78 seconds** for **2048*2048 Matrices Multiplication**.

I also made a specialized optimization. I **removed one outer loop k** because the matrices is not square and one dimension is exactly same as our block size, which means there is only one block used from each input matrix to compute single block of result matrix.

After all above optimizations done, the performance has been **highly improved**, which achieved almost **double** than naive version, and **comparing with non-block naive LU version, Optimized Blocked GEPP improves almost 500% performance.**. The result of optimization Blocked GEPP are shown as follows, I named it **'mydgemm_sub_best(...)'**.

Table 3: **Optimized Blocked GEPP**

n	runtime(s)	performance(Gflops)
1002	0.852416	**0.789148**
2001	6.838943	**0.782189**
3000	22.855887	**0.788331**
4002	55.102992	**0.776051**
5001	106.132280	**0.786126**

Table 4: **naive LU**

n	runtime(s)	performance(Gflops)
1002	4.382704	0.153486
2001	34.882417	0.153354
3000	114.872045	0.156853
4002	272.453497	0.156954
5001	521.850689	0.159880

Table 5: **unoptimized Blocked GEPP**

n	runtime(s)	performance(Gflops)
1002	1.817282	0.370159
2001	14.514987	0.368539
3000	49.464173	0.364264
4002	117.492669	0.363961
5001	228.624235	0.364937

Table 6: **LAPACK**

n	runtime(s)	performance(Gflops)
1002	0.136527	4.92710
2001	0.541638	9.87624
3000	1.619845	11.1233
4002	3.502706	12.2085
5001	6.612467	12.6176

1.3 try SSE instructions

To improve the dgemm performance, I also tried to use 3x3 SSE instructions computations Unfortunately, the results get worse. The SSE instruction has more latency than c code, **dgemm cost almost 5s when n=2001(worse 3 times)**.