CS 133 Lab1 Report
Maolei Tang
604501038

1.Methods to parallelize and optimize:

In mmul1.c:
I use two methods to speed up it. The code is executed in parallel by OpenMP so that there are 32 threads executed at same time. The other method is changing the order of inner two loops. Because A[i][k]xB[k][j] in the original loop have good spacial locality for A but not for B, there exists many memory misses for B[k][j] if we read B[k][j] first and b[k+1][j] then. The solution is exchange the loop for k and j, which calculates A[i][k]xB[k][j] first and A[i][k]xB[k][j+1] then. It gives us good spacial locality for B[k][j]. After testing, it has more than 45 Gflops and 90X speed up.

In mmul2.c:
I use three methods to speed up it. First, we try to find the good algorithm to get better temporal and spacial locality than naive way. The method is that we could use blocked matrix multiplication to get smaller block at a time, and calculate the product for each block. A block is small enough so that all the data in the block can be cached in the fast memory, which is cache of CPU. I set block size to be 64. Second, we could calculate the result for each block in parallel so that there are 32 threads to calculate the matrices production. The third method is what we did in mmul1.c, we exchange the inner loops in case that there is still memory missed after separating the matrix into blocks.

2.

	1024x1024	**2048x2048**	**4096x4096**
mmul1.c	`26.00 GFlop/s (42.8X)`	`45.79 GFlop/s (115.7X)`	`42.37 GFlop/s (550.4X)`
mmul2.c	`9.24 GFlop/s (33.1X)`	`19.24 GFlop/s (39.8X)`	`12.76 GFlop/s (24.5X)`

For mmul1, there is no obvious difference between various matrix size. The little difference comes from subtle random task such us cache missed.

For mmul2, the block size is equal to 2048/32=64, which means that each thread calculate one block in matrix B of 2048 by 2048. If we use the block size in matrix of 1024 by 1024, there will be 16 threads access the same memory at the same time. This will have influence on the performance result.

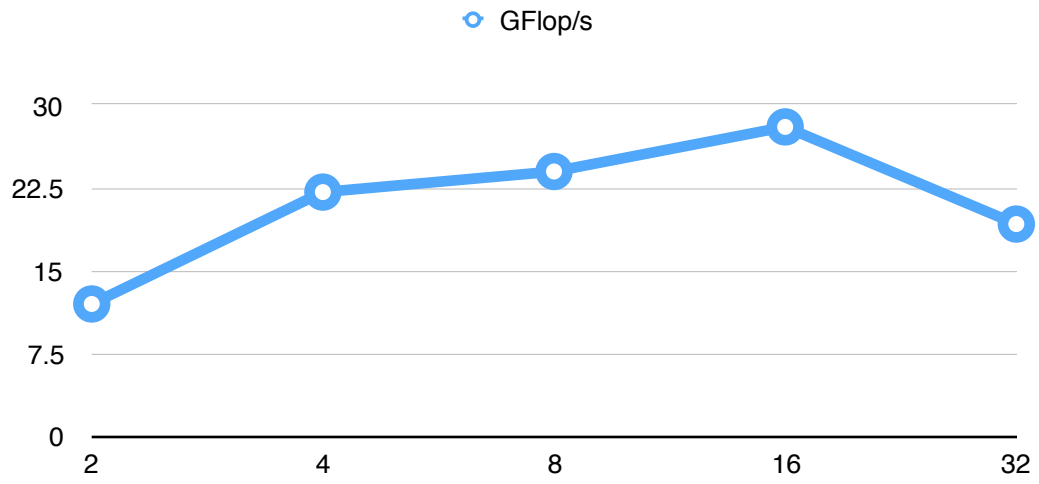
3.
My OpenMp performance in GFlop/s for mmul2.c is 19.74 GFlop/s (37.3X) compared to the serial version. Although the throughput does not reach more than 45 GFlop/s, it reaches the maximum of blocked matrix multiplication with parallel. The speedup for blocked matrix multiplication is around 10 GFlop/s (17.4X) and exchanging inner loop with blocked matrix multiplicati is 19.24 GFlop/s (39.8X).

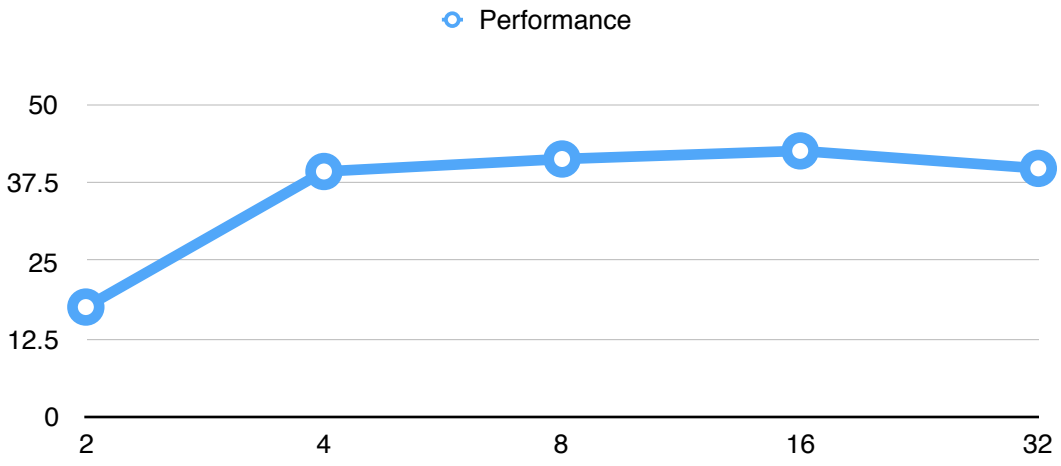
4.

For the input size 2048x2048, the speedup for blocked matrix multiplication is around 10 GFlop/s (17.4X) and exchanging inner loop with blocked matrix multiplicati is 19.24 GFlop/s (39.8X).

5.

threads	performance
2	12.02 GFlop/s (17.6X)
4	22.13 GFlop/s (39.3X)
8	23.99 GFlop/s (41.3X)
16	28.02 GFlop/s (42.6X)
32	19.24 GFlop/s (39.8X)





6.
My result is kind of acceptable. However, the blocked matrix multiplication is slower than what I expect. As a method to get a perfect temporal and spacial locality, it is expected to be dramatically better than exchanging inner loop order. However, there must be something wrong or some components which I did not notice which has a huge influence on the blocked version algorithm. Furthermore, the good memory hit is not occurred well in my solution after taking several trials of test. I use exchanging loop order strategy to catch the missed part in block. Therefore, the performance increase a little bit, but it does not arrive the level where we use exchanging loop orders individually.