实验名称 LU 分解

一. 实验内容:

问题描述:
给定矩阵A，求满足A=LU的下三角矩阵L和上三角矩阵U。
LU分解的串行算法请自行查询资料。
LU分解的并行算法请尽量使用负载均衡的矩阵划分方法。

输入:
第一行，整数n，表示方阵A的阶数
之后n行，每行n个浮点数，表示矩阵A
输出:
2n行，每行n个浮点数，表示矩阵L和U

要求：测试n=500，1000，5000时的实验性能

二. 实验环境:

操作系统:	ubuntu 16.04 LTS 64bit	
编译器:	gcc 5.3.1	(for openmp)
	nvcc 7.5.17	(for cuda)
CPU:	Intel i7-4720HQ CPU @ 2.60GHz * 8	
GPU:	NVIDIA GTX965M 2G GDDR5	
memory:	8GB DDR3L	

三. 实现步骤:

1. OpenMP

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>

#define MAXN 5001
#define NNN 5000
#define NUM_OF_THREADS 8

double A[MAXN][MAXN];
double L[MAXN][MAXN];
double U[MAXN][MAXN];
int nthreads,tid;

int factorize() {
	int i, j, k;
	#pragma omp parallel shared(A) private(tid, i, j, k)
	{
		for (k = 0; k < NNN; k++)
		{
			#pragma omp for
			for (i = k + 1; i < NNN; i++)
```

```
                        A[i][k] = A[i][k] / A[k][k];
                #pragma omp for
                for (i = k + 1; i < NNN; i++)
                        for (j = k + 1; j < NNN; j++)
                        {
                                A[i][j] = A[i][j] - A[i][k] * A[k][j];
                        }
                }
        }
        for (i = 0; i < NNN; i++)
                for (j = 0; j < NNN; j++)
                {
                        if (i <= j)
                                U[i][j] = A[i][j];
                        else
                                L[i][j] = A[i][j];
                }
        return 0;
}

int main()
{
        double wtime;
        int i, j;

        srand((unsigned)time(NULL));
        for (i = 0; i < NNN; i++)
                for (j = 0; j < NNN; j++)
                {
                        A[i][j] = rand() / 1000;
                        L[i][j] = 0;
                        U[i][j] = 0;
                }

        omp_set_num_threads(NUM_OF_THREADS);

        for (i = 0; i < NNN; i++)
                L[i][i] = 1;

        wtime = omp_get_wtime();
        factorize();
        wtime = omp_get_wtime() - wtime;

        printf("time = %lf s\n", wtime);

        return 0;
}
```

2. CUDA

```
#include <cuda.h>
#include <stdio.h>
#include <math.h>

__global__ void add( float *a, float *b, float *c) {
 int tid = blockIdx.x;
```

```
 c[tid] = a[tid] + b[tid];
}


__global__ void scale(float *a, int size, int index){
 int i;
 int start=(index*size+index);
 int end=(index*size+size);

 for(i=start+1;i<end;i++)
        a[i]=(a[i]/a[start]);
}

__global__ void reduce(float *a, int size, int index, int b_size){
 extern __shared__ float pivot[];
 int i;

 int tid=threadIdx.x;
 int bid=blockIdx.x;
 int block_size=b_size;

 int start;
 int end;
 int pivot_row;
 int my_row;

 if(tid==0)
        for(i=index;i<size;i++) pivot[i]=a[(index*size)+i];

 __syncthreads();

 pivot_row=(index*size);
 my_row=(((block_size*bid) + tid)*size);
 start=my_row+index;
 end=my_row+size;

 if(my_row >pivot_row)
        for(i=start+1;i<end;i++)
              a[i]=a[i]-(a[start]*pivot[(i-my_row)]);
}

float a[5001][5001];
float b[5001][5001];
float c[5001][5001];
float result[5001][5001];


int main(int argc, char *argv[]){
 int N;
 int blocks;
 float *dev_a;
 int i;
 int j;
 N = atoi(argv[1]);
 clock_t start, finish;
 double elapse;

 cudaMalloc((void**)&dev_a, N * N * sizeof(float));
 srand((unsigned)2);
 for (i = 0; i <= N; i++)
        for (j = 0; j <= N; j++)
        a[i][j] = ((rand() % 10) + 1);
 cudaMemcpy(dev_a, a, N * N * sizeof(float), cudaMemcpyHostToDevice);

 start = clock();
 for(i = 0; i < N; i++){
```

```
        scale<<<1,1>>>(dev_a, N, i);
        blocks = N / 50;
        reduce<<<blocks,50,N*sizeof(float)>>>(dev_a, N, i, 50);
    }
    finish = clock();
    elapse = (((double) (finish - start)) / CLOCKS_PER_SEC);
    printf("lu decompositon: %lf sec\n", elapse);

    cudaMemcpy(c, dev_a, N * N * sizeof(float),cudaMemcpyDeviceToHost);
    cudaFree( dev_a );

    return 0;
}
```

四. 实验结果报表

1. OpenMP

本次实验代码总行数	69 lines
累计耗费时间	？？？

运行时间(sec)

规模 \ 核数	1	2	4	8
500	0.1211	0.0632	0.0390	0.04567
1000	0.9929	0.5148	0.2889	0.3095
5000	125.5536	69.4698	68.4940	70.4312

加速比

规模 \ 核数	1	2	4	8
500	1	1.9161	3.1051	2.6516
1000	1	1.9287	3.4368	3.2081
5000	1	1.8073	1.8331	1.7826

3.CUDA

本次实验代码总行数	85 lines
累计耗费时间	？？？

运行时间(sec)

规模 \ 线程数	blocks_num = size / 50 threads_num = 50
500	0.0021
1000	0.4508
5000	35.2180

加速比

规模 \ 线程数	blocks_num = size / 50 threads_num = 50
500	57.6667
1000	2.2025
5000	3.5650

五. 对本次实验的收获和总结

规模为1000的时候和规模为5000的时候，耗时相差不大。
这估计是因为我所用的算法不太适合并行，而且并行的参数也没有调好。

六. 其他问题（请在符合自己情况的地方填入*）

1.你使用的编程模型是 (多选)

OpenMP	MPI	CUDA	MapReduce
*		*	

注：以下只需填实验中选择的实现方式对应的行，每行单选。

2.在进行并行编程时，我愿意经常使用此编程模型 (单选)

	非常符合	比较符合	不确定	比较不符合	非常不符合
OpenMP	*				
MPI					
CUDA		*			
MapReduce					

3.我认为此编程模型有许多非必要而复杂的部分(单选)

	非常符合	比较符合	不确定	比较不符合	非常不符合

OpenMP				*	
MPI					
CUDA				*	
MapReduce					

4.我认为在此编程模型下编程是容易的(单选)

	非常符合	比较符合	不确定	比较不符合	非常不符合
OpenMP	*				
MPI					
CUDA				*	
MapReduce					

5.我认为我需要有技术人员的支持才能使用此编程模型 (单选)

	非常符合	比较符合	不确定	比较不符合	非常不符合
OpenMP				*	
MPI					
CUDA				*	
MapReduce					

6.我认为在此编程模型中，有许多功能都很好地整合在一起 (单选)

	非常符合	比较符合	不确定	比较不符合	非常不符合
OpenMP	*				
MPI					
CUDA		*			
MapReduce					

7.我认为在此编程模型中有许多不一致的地方 (单选)

	非常符合	比较符合	不确定	比较不符合	非常不符合
OpenMP				*	
MPI					
CUDA				*	
MapReduce					

8.我认为大多数人都能很快地掌握此编程模型 (单选)

	非常符合	比较符合	不确定	比较不符合	非常不符合
OpenMP		*			
MPI					
CUDA				*	
MapReduce					

9.我认为此编程模型不灵活 (单选)

	非常符合	比较符合	不确定	比较不符合	非常不符合
OpenMP		*			
MPI					
CUDA				*	
MapReduce					

10.对于在此编程模型下编程，我对问题能够被解决感到很自信 (单选)

	非常符合	比较符合	不确定	比较不符合	非常不符合
OpenMP			*		
MPI					
CUDA			*		
MapReduce					

11.在使用此编程模型前，我需要了解大量的知识(单选)