实验名称 矩阵乘法

一. 实验内容:

问题描述:
给定两个方阵A,B,计算其乘积C

输入:
第一行，一个整数n:输入方阵的阶数
后面2n行，每行n个单精度浮点数，前n行表示A，后n行表示B

输出:
n行，每行n个双精度浮点数，表示矩阵C=AB

要求：测试n=500, 1000, 5000时的实验性能
提示：MPI实现请使用cannon乘法或者fox乘法。CUDA实现需要分块处理，而且最好多利用特殊存储器。

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

#define NUM_THREAD 1
#define DEBUG 0

int main (int argc, char *argv[])
{
    int i, j, k;
    double wtime;
    int chunk = 10;
    int scale = atoi(argv[1]);

    float** a = (float**)malloc(sizeof(float*) * scale);
    for (i = 0; i < scale; i++)
        a[i] = (float*)malloc(sizeof(float) * scale);
    float** b = (float**)malloc(sizeof(float*) * scale);
    for (i = 0; i < scale; i++)
```

```
        b[i] = (float*)malloc(sizeof(float) * scale);
    float** c = (float**)malloc(sizeof(float*) * scale);
    for (i = 0; i < scale; i++)
        c[i] = (float*)malloc(sizeof(float) * scale);

    wtime = omp_get_wtime();
    #pragma omp parallel shared(a,b,c,scale,chunk) private(i,j,k) num_threads(NUM_THREAD)
    {
        srand(time(0));
        #pragma omp for schedule (static, chunk)
        for (i=0; i<scale; i++)
            for (j=0; j<scale; j++)
                a[i][j]= rand() / 1000;
        #pragma omp for schedule (static, chunk)
        for (i=0; i<scale; i++)
            for (j=0; j<scale; j++)
                b[i][j]= rand() / 1000;
        #pragma omp for schedule (static, chunk)
        for (i=0; i<scale; i++)
            for (j=0; j<scale; j++)
                c[i][j]= 0;

        #pragma omp for schedule (static, chunk)
        for (i=0; i<scale; i++)
        {
            for(j=0; j<scale; j++)
                for (k=0; k<scale; k++)
                    c[i][j] += a[i][k] * b[k][j];
        }
    }
    wtime = omp_get_wtime() - wtime;
    printf("time: %lf s\n", wtime);

    if (DEBUG)
    {
        for (i = 0; i < scale; i++)
        {
            for (j = 0; j < scale; j++)
                printf("%f", c[i][j]);
            printf("\n");
        }
    }

    return 0;
}
```

2. CUDA

```
// matrix multiplication
// CA_LAB4

#include<stdio.h>
#include<iostream>
#include<cstdlib>
#include<time.h>
#include<cuda.h>
```

```
#define TILE_WIDTH 32
#define DEBUG 0
using namespace std;

void print(float *A, int n, int m)
{
 for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
        cout << A[n*i+j] << " ";
        cout<<endl;
}

void init_matrix (float *mat, float value, int n, int m)
{
 int size = n * m;
 for (int i = 0; i < size; i++)
        mat[i] = value;
}

void multMatrixSeq (float *mA, float *mB, float *mC, int n, int m, int o)
{
 for (int i = 0; i < n; i++)
 {
        for (int j = 0; j < o; j++)
        {
                float sum = 0;
                for (int k = 0; k < m; k++)
                {
                        sum += mA[m*i+k] * mB[o*k+j];
                }
                mC[o*i+j] = sum;
        }
 }
}

__global__ void CU_multMatrixThread (float *mA, float *mB, float *mC, int n, int m, int o)
{
 int row = blockIdx.y * blockDim.y + threadIdx.y;
 int col = blockIdx.x * blockDim.x + threadIdx.x;

 if ((row<n) && (col<o))
 {
        float temp = 0;
        for (int i = 0; i < m; i++)
                temp += mA[row*m+i] * mB[i*o+col];
        mC[row*o+col] = temp;
 }
}

__global__ void CU_multMatrixTiled(float *mA, float *mB, float *mC, int n, int m, int o){
 __shared__ float tmpM1[TILE_WIDTH][TILE_WIDTH];
 __shared__ float tmpM2[TILE_WIDTH][TILE_WIDTH];

 int bx = blockIdx.x;
 int by = blockIdx.y;
 int tx = threadIdx.x;
 int ty = threadIdx.y;

 int row = by * TILE_WIDTH + ty;
 int col = bx * TILE_WIDTH + tx;

 float Pvalue = 0;

 for (int k = 0; k < (m + TILE_WIDTH - 1) / TILE_WIDTH; ++k)
 {
```

```
        if (k*TILE_WIDTH + tx < m && row < n)
            tmpM1[ty][tx] = mA[row * m + k*TILE_WIDTH + tx];
        else
            tmpM1[ty][tx] = 0;

        if (k*TILE_WIDTH + ty < m && col < o)
            tmpM2[ty][tx] = mB[(k*TILE_WIDTH + ty) * o + col];
        else
            tmpM2[ty][tx] =0;

        __syncthreads();

        for(int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += tmpM1[ty][k] * tmpM2[k][tx];

        __syncthreads();
 }

 if (row < n && col < o)
        mC[row * o + col] = Pvalue;
}

void multMatrixTiled(float *A, float *B, float *C, int n, int m, int o)
{
 float blockSize = TILE_WIDTH;
 float *mA, *mB, *mC;

 cudaMalloc(&mA, n * m * sizeof(float));
 cudaMalloc(&mB, m * o * sizeof(float));
 cudaMalloc(&mC, n * o * sizeof(float));

 cudaMemcpy(mA, A, n * m * sizeof(float), cudaMemcpyHostToDevice);
 cudaMemcpy(mB, B, m * o * sizeof(float), cudaMemcpyHostToDevice);

 dim3 threads(blockSize,blockSize,1);
 dim3 blocks(ceil(o/blockSize),ceil(n/blockSize),1);
 CU_multMatrixThread<<<blocks,threads>>>(mA,mB,mC,n,m,o);

 cudaMemcpy (C, mC, n * o * sizeof(float), cudaMemcpyDeviceToHost);

 cudaFree(mA);
 cudaFree(mB);
 cudaFree(mC);

}

void multMatrixThread(float *A, float *B, float *C, int n, int m, int o)
{
 float blockSize = TILE_WIDTH;
 float *mA, *mB, *mC;

 cudaMalloc(&mA, n * m * sizeof(float));
 cudaMalloc(&mB, m * o * sizeof(float));
 cudaMalloc(&mC, n * o * sizeof(float));

 cudaMemcpy(mA, A, n * m * sizeof(float), cudaMemcpyHostToDevice);
 cudaMemcpy(mB, B, m * o * sizeof(float), cudaMemcpyHostToDevice);

 dim3 threads(blockSize,blockSize,1);
 dim3 blocks(ceil(o/blockSize),ceil(n/blockSize),1);
 CU_multMatrixThread<<<blocks,threads>>>(mA,mB,mC,n,m,o);

 cudaMemcpy (C, mC, n * o * sizeof(float), cudaMemcpyDeviceToHost);

 cudaFree(mA);
 cudaFree(mB);
```

```
 cudaFree(mC);
}

int compareMatrix (float *A, float *B,int n, int m)
{
 int size = n * m;
 for (int i = 0; i < size; i++ )
 {
 if (A[i] != B[i])
 {
        cout<<"the sequential result and parallel result are not equal"<<endl;
        return 0;
 }
 }
 cout<<"the sequential result and parallel result are equal"<<endl;
 return 0;
}

int main(int argc, char* argv[])
{
 clock_t start, finish;
 double elapsedsequential, elapsedParallel, elapsedParallelTiles, optimizationP, optimizationT;
 int kkkkk = atoi(argv[1]);
 int n = kkkkk;
 int m = kkkkk;
 int o = kkkkk;

 float *matA = (float *) malloc(n * m * sizeof(float));
 float *matB = (float *) malloc(m * o * sizeof(float));
 float *matCS = (float *) malloc(n * o * sizeof(float));
 float *matCP = (float *) malloc(n * o * sizeof(float));
 float *matCPT = (float *) malloc(n * o * sizeof(float));

 init_matrix(matA,1.5,n,m);
 init_matrix(matB,1.5,m,o);
 init_matrix(matCS,0,n,o);
 init_matrix(matCP,0,n,o);
 init_matrix(matCPT,0,n,o);


 start = clock();
 multMatrixSeq(matA,matB,matCS,n,m,o);
 finish = clock();
 elapsedsequential = (((double) (finish - start)) / CLOCKS_PER_SEC );
 cout<< "sequential matrix multiplication: " << elapsedsequential << "sec"<< endl<< endl;

 start = clock();
 multMatrixThread(matA,matB,matCP,n,m,o);
 finish = clock();
 elapsedParallel = (((double) (finish - start)) / CLOCKS_PER_SEC );
 cout<< "parallel matrix multiplication without using Tiles: " << elapsedParallel << "sec"<< endl<<
endl;

 start = clock();
 multMatrixTiled(matA,matB,matCPT,n,m,o);
 finish = clock();
 elapsedParallelTiles = (((double) (finish - start)) / CLOCKS_PER_SEC );
 cout<< "parallel matrix multiplication using Tiles: " << elapsedParallelTiles << "sec"<< endl<< endl;

 optimizationP = elapsedsequential/elapsedParallel;
 cout<< "speedup without using Tiles: " << optimizationP <<endl;

 optimizationT = elapsedsequential/elapsedParallelTiles;
 cout<< "speedup using Tiles: " << optimizationT <<endl;

 cout<< "check parallel result without using Tiles: " <<endl;
```

```
    compareMatrix(matCS,matCP,n,o);
    cout<< "check parallel result using Tiles: " <<endl;
    compareMatrix(matCS,matCPT,n,o);

    if (DEBUG)
    {
        print(matCS,n,o);
        cout<<endl;
        print(matCP,n,o);
        cout<<endl;
        print(matCPT,n,o);
    }

    free (matA);
    free (matB);
    free (matCS);
    free (matCP);
    free (matCPT);
    return 0;
}
```

四. 实验结果报表

1. OpenMP

本次实验代码总行数	65 lines
累计耗费时间	？？？

运行时间(sec)

规模 \ 核数	1	2	4	8
500	0.7597	0.4875	0.2960	0.2887
1000	7.3753	3.8828	2.4509	2.1799
5000	1817.4004	892.2873	508.1902	400.8447

加速比

规模 \ 核数	1	2	4	8
500	1	1.5584	2.5666	2.6315
1000	1	1.8995	3.0092	3.3833
5000	1	2.0368	3.5762	4.5339

3.CUDA

本次实验代码总行数	225 lines

累计耗费时间	？？？

运行时间(sec)

规模 线程数 串行	without using tiles blocks_num = 32 * 32 threads_num = (1000/32) * (1000/32)	using tiles(分块) tiles_num = 32 * 32 threads_num = (1000/32) * (1000/32)
500 0.3854	0.06684	0.0052
1000 3.0481	0.0937	0.0383
5000 904.4637	3.3436	3.2870

加速比

规模 线程数 串行	without using tiles blocks_num = 32 * 32 threads_num = (1000/32) * (1000/32)	using tiles(分块) tiles_num = 32 * 32 threads_num = (1000/32) * (1000/32)
500 1	5.7660	74.115
1000 1	32.530	79.585
5000 1	270.51	275.16

五. 对本次实验的收获和总结

矩阵乘法是一个很经典的可以并行化的算法。通过**CUDA**来进行加速，可以得到很好的加速比。而用**openmp**来进行加速，效果也挺好的，但是**cpu**在这种问题上显然比不过显卡。一般**pc**上的**cpu**能有**8**个核，而显卡核的数量则要多得多。

注意到在规模为**500**和**1000**时，分块的并行算法比不分块的并行算法加速比要高得多，然而在规模为**5000**时加速比就差别不大了。我也不知道为什么（也许是因为分块的大小不合适？）

另外，在数据规模大的时候，串行算法要跑很久很久。这说明，一个码农，一定要会**cuda**编程，要不然训练个神经网络都要**1**个多月才能训练完。

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)