Shenliang Wang

05/14/2018

CSC 656

Project # 3

Part a:

// File: MaxCol.cu

// Compile: nvcc MaxCol.cu -o mc

// Run: ./mc [width of matrix] [threads per block]

// Description: finds the max of each column of a randomly generated matrix

// in kernel findMax(), each thread finds the max of one column

#include <stdio.h>

#include <stdlib.h>

#include <cuda.h>

#define THREADSPERBLOCK 4

int checkArray(int [], int [], int);

\_\_global\_\_ void findMax(int \*m, int \*rs, int n);

int main(int argc, char \*\*argv)

{

/\* variables for timing \*/

cudaEvent\_t start, stop;

float time;

if (argc != 3) {

printf("Usage: ./SR [width of matrix] [threads per block]\n");

exit(0);

}

int n = atoi(argv[1]); // number of matrix rows/cols

int \*hm, // host matrix

\*dm, // device matrix

\*hcs, // host column sums

\*dcs; // device column sums

int \*checkCs;

int msize = n \* n \* sizeof(int); // size of matrix in bytes

int rssize = n \* sizeof(int);

int threadsPerBlock = atoi(argv[2]); // get threads per block

if (n % threadsPerBlock != 0) {

printf("Warning: width of matrix not divisible by # threads per block\n");

}

// allocate space for host matrix

hm = (int \*) malloc(msize);

// create timer events

cudaEventCreate(&start);

cudaEventCreate(&stop);

// as a test, fill matrix with random integers

int i, j;

for (i = 0; i < n; i++) {

for (j = 0; j < n; j++) {

hm[i\*n+j] = random() % RAND\_MAX;

}

}

// compute max of columns on CPU for checking

checkCs = (int \*) malloc(rssize);

for (i=0; i<n; i++) {

checkCs[i] = hm[i];

for (j=0; j<n; j++) {

if (checkCs[i] < hm[i + j\*n])

checkCs[i] = hm[i + j\*n];

}

}

// allocate space for device matrix

cudaMalloc((void \*\*)&dm,msize);

// copy host matrix to device matrix

cudaMemcpy(dm,hm,msize,cudaMemcpyHostToDevice);

// allocate host, device rowsum arrays

hcs = (int \*) malloc(rssize);

cudaMalloc((void \*\*)&dcs,rssize);

// record start timestamp

cudaEventRecord(start, 0);

// invoke the kernel

findMax<<<n/threadsPerBlock,threadsPerBlock>>>(dm,dcs,n);

// wait for kernel to finish

cudaThreadSynchronize();

// copy row vector from device to host

cudaMemcpy(hcs,dcs,rssize,cudaMemcpyDeviceToHost);

// get elapsed time

cudaEventRecord(stop, 0);

cudaEventSynchronize(stop);

cudaEventElapsedTime(&time, start, stop);

printf("Elapsed time = %f\n", time);

// check results

int diff = checkArray(hcs, checkCs, n);

if (diff == 0) {

printf("Arrays match\n");

}

else {

printf("Arrays do not match\n");

}

// clean up

free(hm);

cudaFree(dm);

free(hcs);

cudaFree(dcs);

}

int checkArray(int x[], int y[], int size) {

int i;

int numDiff = 0;

for (i=0; i<size; i++) {

if (x[i] != y[i]) {

numDiff++;

}

}

return numDiff;

}

// findMax(int \*m, int \*cs, int n)

// m: n x n matrix (input)

// cs: cs[i] contains max of columnn i of m (output)

// n: number of elements in each row/column of m

\_\_global\_\_ void findMax(int \*m, int \*cs, int n)

{

// your code goes here

int column = blockDim.x \*blockIdx.x + threadIdx.x;

int maxnum =0;

for (int i=0; i < n ;i++){

if (maxnum <=m[i\*n +column])

maxnum = m[i\*n +column];

}

cs[column]=maxnum;

}

Result table for Part a:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 8 threads | 16 threads | 32 threads | 64 threads | 128 threads |
| Size 1024 | 0.217376 | 0.2239168 | 0.2313152 | 0.212544 | 0.2137088 |
| Size 2048 | 0.6244928 | 0.364992 | 0.3675968 | 0.3648512 | 0.3684608 |
| Size 4096 | 1.672224 | 1.138528 | 0.644608 | 0.6486592 | 0.6529728 |

Part b:

/\*\*\*\*

File: findRedsDriver.cu

Date: 5/14/2018

By: Shenliang Wang

Compile: nvcc findRedsDriver.cu -o findreadsdriver

Run: ./findreadsdriver

\*\*\*\*/

#include <stdio.h>

#include <math.h>

#include <stdlib.h>

#include <cuda.h>

#define NUMPARTICLES 32768

#define NEIGHBORHOOD .05

#define THREADSPERBLOCK 128

void initPos(float \*);

float findDistance(float \*, int, int);

\_\_device\_\_ float findDistanceGPU(float \*, int, int);

void dumpResults(int index[]);

\_\_global\_\_ void findRedsGPU(float \*p, int \*numI);

int main() {

cudaEvent\_t start, stop;

float time;

float \*pos, \*dpos;

int \*numReds, \*dnumReds;

pos = (float \*) malloc(NUMPARTICLES \* 4 \* sizeof(float));

numReds = (int \*) malloc(NUMPARTICLES \* sizeof(int));

initPos(pos);

// your code to allocate device arrays for pos and numReds go here

cudaMalloc((void \*\*)&dpos,NUMPARTICLES \* 4 \* sizeof(float));

cudaMalloc((void \*\*)&dnumReds,NUMPARTICLES \* sizeof(int));

cudaMemcpy(dpos,pos,NUMPARTICLES \* 4 \* sizeof(float),cudaMemcpyHostToDevice);

// create timer events

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventRecord(start, 0);

/\* invoke kernel findRedsGPU here \*/

findRedsGPU<<<NUMPARTICLES/THREADSPERBLOCK,THREADSPERBLOCK>>>(dpos,dnumReds);

cudaThreadSynchronize();

// your code to copy results to numReds[] go here

cudaMemcpy(numReds,dnumReds,NUMPARTICLES \* sizeof(int),cudaMemcpyDeviceToHost);

cudaEventRecord(stop, 0);

cudaEventSynchronize(stop);

cudaEventElapsedTime(&time, start, stop);

printf("Elapsed time = %f\n", time);

dumpResults(numReds);

}

void initPos(float \*p) {

// your code for initializing pos goes here

int i;

int j;

for (i=0; i<NUMPARTICLES; i++) {

p[i\*4] = rand() / (float) RAND\_MAX;

p[i\*4+1] = rand() / (float) RAND\_MAX;

p[i\*4+2] = rand() / (float) RAND\_MAX;

j = rand() % 3;

if (j == 0)

p[i\*4+3] = 0xff0000;

else if (j == 1)

p[i\*4+3] = 0x00ff00;

else

p[i\*4+3] = 0x0000ff;

}

}

\_\_device\_\_ float findDistanceGPU(float \*p, int i, int j) {

// your code for calculating distance for particle i and j

float x, y, z;

x = p[i\*4] - p[j\*4];

y = p[i\*4+1] - p[j\*4+1];

z = p[i\*4+2] - p[j\*4+2];

return(sqrt(x\*x + y\*y + z\*z));

}

\_\_global\_\_ void findRedsGPU(float \*p, int \*numI) {

int index = blockDim.x \* blockIdx.x + threadIdx.x;

int i;

float d;

numI[index] = 0;

for (i=0; i<NUMPARTICLES; i++) {

if (index!=i) {

d = findDistanceGPU(p, index, i);

if (d < NEIGHBORHOOD && p[i\*4+3] == 0xff0000) {

numI[index]++;

}

}

}

}

void dumpResults(int index[]) {

int i;

FILE \*fp;

fp = fopen("./dump.out", "w");

for (i=0; i<NUMPARTICLES; i++) {

fprintf(fp, "%d %d\n", i, index[i]);

}

fclose(fp);

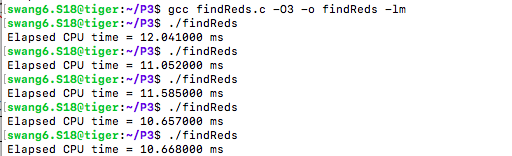
}

Result table for b:

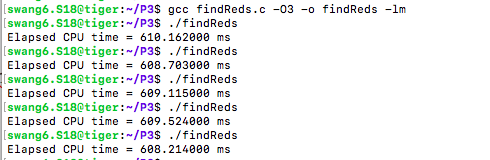
CPU:

|  |  |  |  |
| --- | --- | --- | --- |
|  | NUMPARTICLES = 1024 | NUMPARTICLES = 8192 | NUMPARTICLES = 32768 |
| time | 9711.069800ms | 609.143600ms | 11.200600ms |

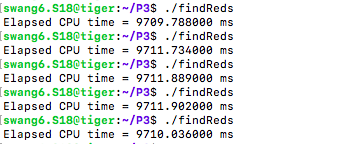
Data

1024:

8192



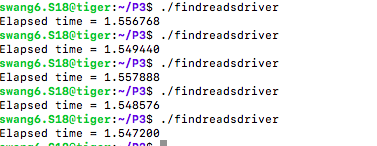
32768:



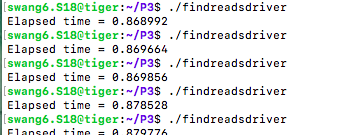
GPU：

|  |  |  |  |
| --- | --- | --- | --- |
|  | 4 threads | 16 threads | 64 threads |
| NUMPARTICLES = 1024 | 1.5519744 | 0.8733632 | 0.8882816 |
| NUMPARTICLES = 8192 | 54.241005 | 19.8361792 | 8.979104 |
| NUMPARTICLES = 32768 | 711.3262816 | 209.1328644 | 133.829152 |

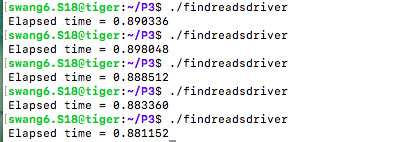
NUMPARTICLES = 1024. 4 threads



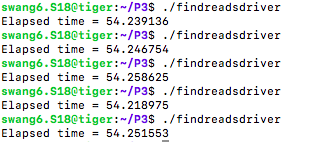
NUMPARTICLES = 1024. 16 threads



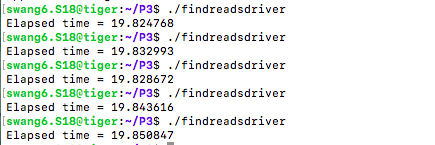
NUMPARTICLES = 1024. 64 threads



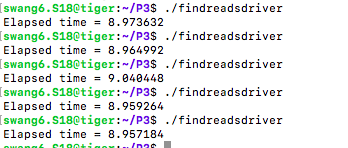
NUMPARTICLES = 8192. 4 threads



NUMPARTICLES = 8192. 16 threads



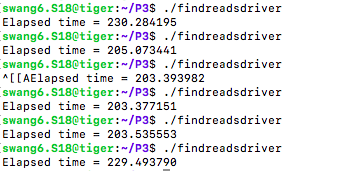
NUMPARTICLES = 8192. 64 threads



NUMPARTICLES = 32768. 4 threads



NUMPARTICLES = 32768. 16threads



NUMPARTICLES = 32768. 64 threads

