Saint Petersburg National Research University of Information Technologies, Mechanics and Optics (ITMO University)
Faculty of Informational Technologies and Programming

REPORT

about laboratory work № 1

« Parallel Clustering Methods»

Student
Nerinovsky Arseny **_M41331_**
(Surname, initials) Group

Saint-Petersburg, 2018

Report

June 28, 2019

1 Goal of laboratory work

The purpose of this work is to implement an algorithm for calculating the degree of closeness of objects described by a set of numerically expressed properties using CUDA technology.

2 Task Definition

We have a set of objects that can be described by sets of properties expressed numerically. In this task, it is necessary to calculate the degree of similarity of objects. As input parameters we have a matrix:

$$A(m \times n) = \begin{pmatrix} a_{0,0} & \cdots & a_{0,n} \\ \vdots & \ddots & \vdots \\ a_{m,0} & \cdots & a_{m,n} \end{pmatrix}, m \neq n.$$

It is necessary to calculate the distance between each possible pair of rows of the matrix.

$$d_{i,j} = \sum_{k=0}^{n} (a_{i,k} - a_{j,k})^2.$$

The paper assumes the implementation of this algorithm using the software-hardware architecture of parallel computing CUDA with global and shared memory. It is necessary to determine the kernel configuration and perform acceleration tests with various examples of input parameters.

3 Brief Theory

CUDA technology introduces a number of additional extensions for the C language, which are necessary for writing code for the GPU: 1. Function specifiers that show how and where functions

will be performed from; 2. The variable specifiers that serve to indicate the type of memory used by the GPU; 3. GPU kernel launch qualifiers; 4. Built-in variables for identifying threads, blocks and other parameters when executing code in the GPU core. 5. Additional types of variables.

Function specifiers determine how and from where the functions will be called. There are 3 such specifiers in CUDA: - **host** - **global** - **device**

Kernel launch qualifiers are used to describe the number of blocks, threads, and memory that need to be allocated when calculating on a GPU. The CUDA host API is the link between the CPU and the GPU.

4 ALGORITHM (METHOD) of IMPLEMENTATION

[4]:
```
%cat mm.cu
```

```
#include <cuda_runtime.h>

#include <iostream>
#include <memory>
#include <string>

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




#ifndef BLOCK_SIZE
# define BLOCK_SIZE 16
#endif

#ifndef _M
# define _M 10000
#endif

#ifndef _N
# define _N 10000
#endif

#if !defined(CUDA) && !defined(CPU) && !defined(CHECK)
# define CUDA
#endif

#define gpuErrchk(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, const char *file, int line, bool
abort=true)
{
   if (code != cudaSuccess)
   {
```

```
      fprintf(stderr,"gpuAssert: %s %s %d\n", cudaGetErrorString(code), file,
line);
      if (abort) exit(code);
   }
}

__global__ void mx_dist(float *m_in, float *m_out, int m, int n)
{
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;
        float s = 0, sum = 0;

    if( i < m && j < m) {

        for(int k = 0; k < n; ++k) {
                s = m_in[i*m + k] - m_in[j*m + k];
                sum += s*s;
        }

        // printf("--> %d %d %f %f\n", j, i, m_in[j*n], sum);
        m_out[i*m + j] = sum;
    }
}

void mx_dist_cpu(float *m_in, float *m_out, int m, int n)
{
        float s, sum;

        for(int i = 0; i < m; ++i)
                for(int j = 0; j < m; ++j) {
                        sum = 0;
                        for(int k = 0; k < n; ++k) {
                                s = m_in[i*m + k] - m_in[j*m + k];
                                sum += s*s;
                        }
                        m_out[i*m + j] = sum;
                }
}

void init_mx(float *A, size_t m, size_t n)
{
        for(int i = 0; i < m; ++i) {
                for(int j = 0; j < n; ++j) {
                        float t = sin(i*m + j) * 10 + 1;
                        A[i*m + j] = t;
                }
        }
}
```

```
void print_mx(float *A, size_t m, size_t n)
{
        for(int i = 0; i < m; ++i) {
                for(int j = 0; j < n; ++j) {
                        printf("%d %d %f\n", i, j, A[i*m + j]);
                }
        }
}

void cmp_mx(float *A, float *B, size_t m, size_t n)
{
        for(int i = 0; i < m; ++i) {
                for(int j = 0; j < n; ++j) {
                        if( abs(A[i*m + j] - B[i*m + j]) > 0.01) {
                                printf("not equal %f %f\n", A[i*m + j], B[i*m +
j]);
                                return;
                        } else {
                                printf("Equal\n");
                        }
                }
        }
}



float *run_cuda(float *A, size_t m, size_t n)
{
        cudaError_t e;

        float *A_d;
        float *B, *B_d;

        B = (float*) malloc(m*m*sizeof(float));


        e = cudaMalloc(&A_d, m*n*sizeof(float));
        gpuErrchk(e);
        e = cudaMalloc(&B_d, m*m*sizeof(float));
        gpuErrchk(e);


        e = cudaMemcpy(A_d, A, m*n*sizeof(float),
                                cudaMemcpyHostToDevice);
        gpuErrchk(e);


    unsigned int grid_rows = (m + BLOCK_SIZE - 1) / BLOCK_SIZE;
```

```
    unsigned int grid_cols = (n + BLOCK_SIZE - 1) / BLOCK_SIZE;

    dim3 dimGrid(grid_cols, grid_rows);
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);

        mx_dist<<<dimGrid, dimBlock>>>(A_d, B_d, m, n);


        e = cudaMemcpy(B, B_d, m*m*sizeof(float),
                                cudaMemcpyDeviceToHost);
        gpuErrchk(e);


        cudaFree(A_d);
        cudaFree(B_d);


        return B;
}


float *run_cpu(float *A, size_t m, size_t n)
{

        float *B;
        B = (float*) malloc(m*m*sizeof(float));

        mx_dist_cpu(A, B, m, n);

        return B;
}

int main()
{

        int m = _M, n = _N;
        float *A;
        A = (float*) malloc(m*n*sizeof(float));
        init_mx(A, m, n);

#if defined(CUDA) | defined(CHECK)
        float *gpu = run_cuda(A, m, n);
#endif

#if defined(CPU) | defined(CHECK)
        float *cpu = run_cpu(A, m, n);
#endif
```

```
#if defined(CHECK)
        cmp_mx(gpu, cpu, m, m);
#endif
        //for(int _j = 0; _j < size; ++_j) printf("%f ", h_vec[2][_j]);
        // printf("\n");


    return 0;
```

5 Result And Experiments

[1]:
```
import subprocess
import os

def compile(*defs, **defskw):
    args = [f"-D{k}" for k in defs] + [f"-D{k}={v}" for k, v in defskw.items()]
    _cmd = 'nvcc mm.cu -o mm'.split() + args
    # print(' '.join(_cmd))
    cmd = subprocess.run(_cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE)

    if(cmd.stdout): print('cmd.stdout', cmd.stdout)
    if(cmd.stderr): print('cmd.stderr', cmd.stderr)

def run(env=None):
    cmd = subprocess.run('./mm', stdout=subprocess.PIPE, stderr=subprocess.
 ↪PIPE, env=env)
    return cmd.stdout.decode('utf8')
```

Lets see how the performance on CPU compares with the performace on GPU

[7]:
```
compile('CPU', _N=10_00, _M=50_0)
print("Execution time on CPU", end="\n\t")
%timeit run()
print()

compile('CUDA', _N=10_00, _M=50_0)
print("Execution time on GPU", end="\n\t")
%timeit run()
print()
```

```
Execution time on CPU
        1.45 s ś 113 ţs per loop (mean ś std. dev. of 7 runs, 1 loop each)

Execution time on GPU
        532 ms ś 21.6 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)
```

We see a clear gain of onGPU execution.

Lets analyze how the block size affects performance.

[9]:
```
for bs in [8, 16, 32, 64, 128, 256, 512, 1024, 2048]:
    compile('CUDA', _N=10_00, _M=50_0, BLOCK_SIZE=bs)
    print(f"Execution time with block size {bs}", end="\n\t")
    %timeit run()
    print()
```

```
Execution time with block size 8
        578 ms ś 42.6 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)

Execution time with block size 16
        554 ms ś 27.2 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)

Execution time with block size 31
        533 ms ś 35.1 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)

Execution time with block size 64
        520 ms ś 38.2 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)

Execution time with block size 128
        519 ms ś 37.2 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)

Execution time with block size 256
        531 ms ś 30.4 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)

Execution time with block size 512
        521 ms ś 35.6 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)

Execution time with block size 1024
        516 ms ś 41.6 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)

Execution time with block size 2048
        510 ms ś 31 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)
```

6 Conclusion

In this paper, we implemented an algorithm for calculating the degree of proximity of objects described by a set of numerically expressed properties using CUDA technology. Tests were performed for the kernel with global and shared memory. As a result of this work, the skill of working with CUDA technology of writing code for the GPU was obtained.