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

Faculty of Informational Technologies and Programming

**Report**

about laboratory work № 3

«The degree of objects similarity»

Student

Voronin Victor  J4132c c

(Surname, initials) Group

Saint-Petersburg, 2020

1. goal of laboratory work

Implement algorithm with using CUDA (with global and shared memory)

1. Task Definition

Given set of properties with matrix A[m][n].

It needs to calculate distance between each possible pair of rows in this matrix.

1. Brief Theory

CUDA – a General-Purpose Parallel Computing Platform and Programming Model based on following basic concepts.

* Host: The CPU and its memory (host memory). Serial code is executed here.
* Device: The GPU and its memory (device memory). Parallel code is executed here.
* Kernel: function that is executed N times in parallel. Kernels are executed by N different CUDA threads.
* Each thread has its own unique ID (treadIdx)
* Threads form a 1/2/3-dimensional block
* Blocks are organized into 1/2/3-dimensional grid

The contents of global memory are visible to all the threads of grid. Any thread can read and write to any location of the global memory. Global memory resides in device memory and device memory is accessed via 32-, 64-, or 128-byte memory transactions. These memory transactions must be naturally aligned: Only the 32-, 64-, or 128-byte segments of device memory that are aligned to their size (i.e., whose first address is a multiple of their size) can be read or written by memory transactions.

When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more of these memory transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads. In general, the more transactions are necessary, the more unused words are transferred in addition to the words accessed by the threads, reducing the instruction throughput accordingly. For example, if a 32-byte memory transaction is generated for each thread's 4-byte access, throughput is divided by 8.

CUDA shared memory is memory shared between the threads within a block, i.e. between blocks in a grid the contents of shared memory are undefined. Because it is on-chip, shared memory has much higher bandwidth and much lower latency than local or global memory.

To achieve high bandwidth, shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously. Any memory read or write request made of n addresses that fall in n distinct memory banks can therefore be serviced simultaneously, yielding an overall bandwidth that is n times as high as the bandwidth of a single module.

However, if two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized. The hardware splits a memory request with bank conflicts into as many separate conflict-free requests as necessary, decreasing throughput by a factor equal to the number of separate memory requests. If the number of separate memory requests is n, the initial memory request is said to cause n-way bank conflicts. Data inside of block is not visible to threads in other blocks. Use \_\_syncthreads() as a barrier to help to prevent occurs race condition between threads.

Global version of program is presented in appendix 1. The matrix is generated in main method with using function rand() %10 which generates random numbers in interval from 0 to 10. There is seqMatMul1Darray\_Test method calculates result of serial algorithm, measures time spent for this and examines correctness of obtained results from serial and parallel versions of program. matMult\_Global\_mem method services parallel computation of this task with using global memory.

Appendix 2 presents shared memory version of program. In different from global memory version description above, there is preliminaries needed to synchronize to make sure pieces of the matrices are distributed between blocks. After that only, it performs to calculate the task in finish of this procedure synchronously perform again to make sure that all operations have been accomplished.

1. Result And Experiments

Using CUDA Toolkit 11.0 and NVIDIA GTX1050, following results were obtained (see table 1). Time measurement was implemented with cudaEventElapsedTime command for parallel version and standard library <chrono> for serial version.

Table ‒ 1 Result obtained

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Order of matrix | Serial version, ms | GlobalGPU, ms | Speedup | SharedGPU, ms | Speedup |
| 50 | 0.9 | 0.5407 | 1.664595 | 0.5601 | 1.606776 |
| 100 | 4.4 | 3.0228 | 1.455596 | 1.0926 | 4.027062 |
| 250 | 33.3 | 22.173 | 1.501851 | 8.59 | 3.876601 |
| 500 | 524.3 | 347 | 1.50998 | 67.995 | 7.7109 |
| 750 | 1815.9 | 1004.4 | 1.807937 | 206.37 | 8.799385 |
| 1000 | 4277.6 | 2263.6 | 1.88975 | 473.09 | 9.041869 |
| 1500 | 14858 | 7611.1 | 1.952171 | 1392.2 | 10.67239 |
| 2000 | 35108 | 17557 | 1.999628 | 3269 | 10.73968 |

Figure 1 Time spent distribution

As seen from figure 1, serial version as expected loses parallel implementation of program. In turn, it is also expected that global memory version is cede in performance shared memory version. It happens thanks to fact that access to data in the last one is faster.

1. Conclusion

During the execution of the task, it used GPU to calculate degree of nearness of two vectors with CUDA Toolkit. Comparison of implementation with global memory and shared memory was enacted. The results obtained were analyzed.

Appendix 1

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include "cuda\_profiler\_api.h"

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <chrono>

#include <iostream>

using namespace std;

#define BLOCK\_SIZE 2

const int N = 100;

const bool debugInfoAllow = false;

\_\_global\_\_ void matMult\_Global\_mem(float\* a, int n, float\* c)

{

    int bx = blockIdx.x;

    int by = blockIdx.y;

    int tx = threadIdx.x;

    int ty = threadIdx.y;

    float sum = 0.0f;

    int ia = n \* BLOCK\_SIZE \* by + n \* ty;

    int ib = n \* BLOCK\_SIZE \* bx + n \* tx;

    int ic = n \* BLOCK\_SIZE \* by + n \* ty + BLOCK\_SIZE \* bx + tx;

    for (int k = 0; k < n; k++)

        sum += (a[ia + k] - a[ib + k])\* (a[ia + k] - a[ib + k]);

    c[ic] = sum;

}

bool seqMatMul1Darray\_Test(float\* hostInitialMatrix, float\* deviceResultMatrix) {

    int i, j, k;

    clock\_t start1;

    float\* C = (float\*)malloc(N \* N \* sizeof(float));

    auto beg = chrono::steady\_clock::now();

    for (i = 0; i < N; i++)

    {

        for (j = 0; j < N; j++)

        {

            C[N \* i + j] = 0;

            for (k = 0; k < N; k++)

            {

                C[N \* i + j] += (hostInitialMatrix[i \* N + k] - hostInitialMatrix[j \* N + k])\* (hostInitialMatrix[i \* N + k] - hostInitialMatrix[j \* N + k]);

            }

        }

    }

    if (debugInfoAllow == true)

        printf("\nResult matrix\n");

    bool testResult = true;

    for (i = 0; i < N; i++)

    {

        if (debugInfoAllow == true)

            std::cout << std::endl;

        for (j = 0; j < N; j++)

        {

            if (debugInfoAllow == true)

                std::cout << "\t" << deviceResultMatrix[i \* N + j] << "|" << C[i \* N + j];

            if (C[i \* N + j] != deviceResultMatrix[i \* N + j])

                testResult = false;

        }

    }

    auto end1 = chrono::steady\_clock::now();

    auto diff = end1 - beg;

    printf("\nTime spent for serial code: %.1f  ms\n", chrono::duration <double, milli>(diff).count());

    free(C);

    return testResult;

}

int main(int argc, char\* argv[])

{

    int i, j, k;

    int numBytes = N \* N \* sizeof(float);

    float\* hostInitialMatrix = (float\*)malloc(N \* N \* sizeof(float));;

    float\* hostResultMatrix = (float\*)malloc(N \* N \* sizeof(float));;

    srand(time(NULL));

    if (debugInfoAllow == true)

        printf("initial matrix");

    for (i = 0; i < N; i++) {

        if (debugInfoAllow == true)

            cout << endl;

        for (j = 0; j < N; j++)

        {

            k = N \* i + j;

            hostInitialMatrix[k] = rand() %10;

            if (debugInfoAllow == true)

                cout << "\t" << hostInitialMatrix[k];

        }

    }

    float\* deviceInitialMatrix;

    float\* deviceResultMatrix;

    cudaMalloc((void\*\*)&deviceInitialMatrix, numBytes);

    cudaMalloc((void\*\*)&deviceResultMatrix, numBytes);

    dim3 threads(BLOCK\_SIZE, BLOCK\_SIZE);

    dim3 blocks(N / BLOCK\_SIZE, N / BLOCK\_SIZE);

    cudaEvent\_t start, stop;

    float gpuTime = 0.0f;

    cudaEventCreate(&start);

    cudaEventCreate(&stop);

    cudaEventRecord(start, 0);

    cudaMemcpy(deviceInitialMatrix, hostInitialMatrix, numBytes, cudaMemcpyHostToDevice);

    cudaEventRecord(stop, 0);

    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&gpuTime, start, stop);

    printf("\nTime spent coping from HOST to DEVICE: %f millseconds", gpuTime);

    double totalTimeGPU = gpuTime;

    cudaEventRecord(start, 0);

    matMult\_Global\_mem <<<blocks, threads >> > (deviceInitialMatrix, N, deviceResultMatrix);

    cudaThreadSynchronize();

    cudaEventRecord(stop, 0);

    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&gpuTime, start, stop);

    printf("\nTime spent executing by the GPU: %f millseconds", gpuTime);

    totalTimeGPU += gpuTime;

    cudaEventRecord(start, 0);

    cudaMemcpy(hostResultMatrix, deviceResultMatrix, numBytes, cudaMemcpyDeviceToHost);

    cudaEventRecord(stop, 0);

    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&gpuTime, start, stop);

    printf("\nTime spent coping from DEVICE to HOST: %f millseconds", gpuTime);

    totalTimeGPU += gpuTime;

    printf("\nTime spent with using GPU: %f millseconds", totalTimeGPU);

    bool tested = seqMatMul1Darray\_Test(hostInitialMatrix, hostResultMatrix);

    printf("\nTestResult = %s", tested ? "true" : "false");

    // release resources

    cudaEventDestroy(start);

    cudaEventDestroy(stop);

    cudaFree(deviceInitialMatrix);

    cudaFree(deviceResultMatrix);

    delete hostInitialMatrix;

    delete hostResultMatrix;

    return 0;

}

Appendix 2

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include "cuda\_profiler\_api.h"

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <chrono>

#include <iostream>

using namespace std;

#define BLOCK\_SIZE 16

const int N = 2000;

const bool debugInfoAllow = false;

\_\_global\_\_ void matMult\_Shared\_mem1(float\* a, float\* b, int n, float\* c)

{

int bx = blockIdx.x;

int by = blockIdx.y;

int tx = threadIdx.x;

int ty = threadIdx.y;

int aBegin = n \* BLOCK\_SIZE \* by;

int aEnd = aBegin + n - 1;

int bBegin = BLOCK\_SIZE \* bx;

int aStep = BLOCK\_SIZE;

int bStep = BLOCK\_SIZE \* n;

float sum = 0.0f;

\_\_shared\_\_ float as[BLOCK\_SIZE][BLOCK\_SIZE];

\_\_shared\_\_ float bs[BLOCK\_SIZE][BLOCK\_SIZE];

for (int ia = aBegin, ib = bBegin; ia <= aEnd; ia += aStep, ib += bStep) {

as[tx][ty] = a[ia + n \* ty + tx];

bs[tx][ty] = a[ib + n \* ty + tx];

//printf("\nas = %f , bs = %f, ty = %f, tx = %f", as[tx][ty], bs[tx][ty],ty,tx);

\_\_syncthreads(); // Synchronize to make sure the matrices are loaded

for (int k = 0; k < BLOCK\_SIZE; k++) {

sum += (as[k][ty] - bs[tx][k]) \* (as[k][ty] - bs[tx][k]);

//printf("\nsum[%f,%f] = %f", ty, tx, sum);

}

\_\_syncthreads();

}

c[aBegin + bBegin + n \* ty + tx] = sum;

}

\_\_global\_\_ void matMult\_Shared\_mem5(float\* a, float\* b, int n, float\* c)

{

int bx = blockIdx.x;

int by = blockIdx.y;

int tx = threadIdx.x;

int ty = threadIdx.y;

int aBegin = n \* BLOCK\_SIZE \* by;

int aEnd = aBegin + n - 1;

int bBegin = BLOCK\_SIZE \* bx;

int aStep = BLOCK\_SIZE;

int bStep = BLOCK\_SIZE \* n;

float sum1 = 0.0f, sum2 = 0.0f, sum3 = 0.0f, sum4 = 0.0f;

\_\_shared\_\_ float as[BLOCK\_SIZE][BLOCK\_SIZE];

\_\_shared\_\_ float bs[BLOCK\_SIZE][BLOCK\_SIZE];

for (int ia = aBegin, ib = bBegin; ia <= aEnd; ia += aStep, ib += bStep) {

as[ty][tx] = a[ia + n \* ty + tx];

bs[ty][tx] = b[ib + n \* ty + tx];

as[ty + 8][tx] = a[ia + n \* (ty + 8) + tx];

bs[ty + 8][tx] = b[ib + n \* (ty + 8) + tx];

as[ty + 16][tx] = a[ia + n \* (ty + 16) + tx];

bs[ty + 16][tx] = b[ib + n \* (ty + 16) + tx];

as[ty + 24][tx] = a[ia + n \* (ty + 24) + tx];

bs[ty + 24][tx] = b[ib + n \* (ty + 24) + tx];

\_\_syncthreads();

for (int k = 0; k < BLOCK\_SIZE; k++) {

sum1 += as[ty][k] \* bs[k][tx];

sum2 += as[ty + 8][k] \* bs[k][tx];

sum3 += as[ty + 16][k] \* bs[k][tx];

sum4 += as[ty + 24][k] \* bs[k][tx];

}

\_\_syncthreads();

}

c[aBegin + bBegin + n \* ty + tx] = sum1;

c[aBegin + bBegin + n \* (ty + 8) + tx] = sum2;

c[aBegin + bBegin + n \* (ty + 16) + tx] = sum3;

c[aBegin + bBegin + n \* (ty + 24) + tx] = sum4;

}

bool seqMatMul1Darray\_Test(float\* hostInitialMatrix, float\* h\_b, float\* deviceResultMatrix) {

int i, j, k;

clock\_t start1;

float\* C = (float\*)malloc(N \* N \* sizeof(float));

auto beg = chrono::steady\_clock::now();

for (i = 0; i < N; i++)

{

for (j = 0; j < N; j++)

{

C[N \* i + j] = 0;

for (k = 0; k < N; k++)

{

C[N \* i + j] += (hostInitialMatrix[i \* N + k] - hostInitialMatrix[j \* N + k]) \* (hostInitialMatrix[i \* N + k] - hostInitialMatrix[j \* N + k]);

}

}

}

if (debugInfoAllow == true)

printf("\nResult matrix\n");

bool testResult = true;

for (i = 0; i < N; i++)

{

if (debugInfoAllow == true)

std::cout << std::endl;

for (j = 0; j < N; j++)

{

if (debugInfoAllow == true)

std::cout << "\t" << deviceResultMatrix[i \* N + j] << "|" << C[i \* N + j];

if (C[i \* N + j] != deviceResultMatrix[i \* N + j])

testResult = false;

}

}

auto end1 = chrono::steady\_clock::now();

auto diff = end1 - beg;

printf("\nTime spent for serial code: %.1f ms\n", chrono::duration <double, milli>(diff).count());

free(C);

return testResult;

}

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

{

int i, j, k;

int numBytes = N \* N \* sizeof(float); //define memory size

// allocate host memory

float\* h\_A = (float\*)malloc(N \* N \* sizeof(float));;

float\* h\_B = (float\*)malloc(N \* N \* sizeof(float));;

float\* h\_C = (float\*)malloc(N \* N \* sizeof(float));;

//init matrix

srand(time(NULL));

if (debugInfoAllow == true)

printf("initial matrix");

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

if (debugInfoAllow == true)

cout << endl;

for (j = 0; j < N; j++)

{

k = N \* i + j;

h\_A[k] = rand() % 3;//\*1.0f;

h\_B[k] = rand() % 3;//\*1.0f;

if (debugInfoAllow == true)

cout << "\t" << h\_A[k];

}

}

float\* d\_A;

float\* d\_B;

float\* d\_C;

cudaMalloc((void\*\*)&d\_A, numBytes);

cudaMalloc((void\*\*)&d\_B, numBytes);

cudaMalloc((void\*\*)&d\_C, numBytes);

dim3 threads(BLOCK\_SIZE, BLOCK\_SIZE);

dim3 blocks(N / BLOCK\_SIZE, N / BLOCK\_SIZE);

cudaEvent\_t start, stop;

float gpuTime = 0.0f;

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventRecord(start, 0);

cudaMemcpy(d\_A, h\_A, numBytes, cudaMemcpyHostToDevice);

cudaMemcpy(d\_B, h\_B, numBytes, cudaMemcpyHostToDevice);

matMult\_Shared\_mem1 << <blocks, threads >> > (d\_A, d\_B, N, d\_C); //kernel launch

cudaMemcpy(h\_C, d\_C, numBytes, cudaMemcpyDeviceToHost);

cudaEventRecord(stop, 0);

cudaEventSynchronize(stop);

cudaEventElapsedTime(&gpuTime, start, stop);

printf("\ntime spent executing by the GPU: %f millseconds\n", gpuTime);

// release resources

cudaEventDestroy(start);

cudaEventDestroy(stop);

cudaFree(d\_A);

cudaFree(d\_B);

cudaFree(d\_C);

delete h\_A;

delete h\_B;

delete h\_C;

return 0;

}