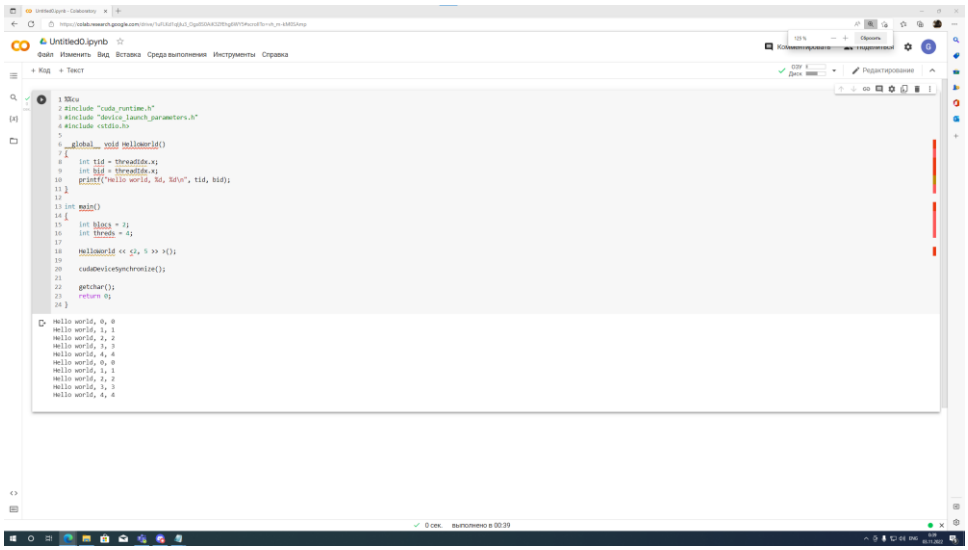
Отчет по лабораторной работе №5

Выполнил Герасимов АД, ИУСбд-01-20, 1132210569

Example 5.1

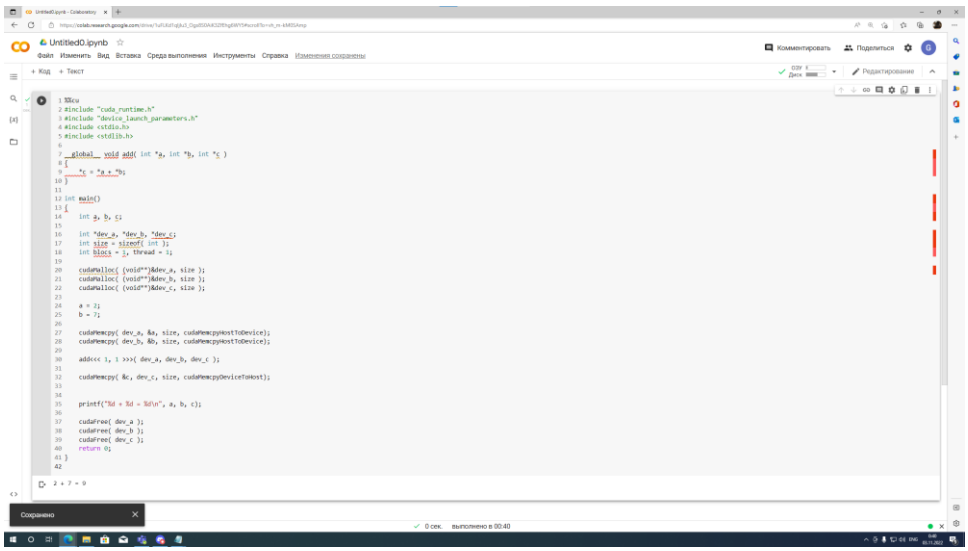
```
%%cu
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>

__global__ void HelloWorld()
{
    int tid = threadIdx.x;
    int bid = threadIdx.x;
    printf("Hello world, %d, %d\n", tid, bid);
}

int main()
{
    int blocks = 2;
    int threads = 4;

    HelloWorld << <blocks, threads >> >();

    cudaDeviceSynchronize();

    getchar();
    return 0;
}
```



Example 5.2

```
%%cu
#include "cuda_runtime.h"
```

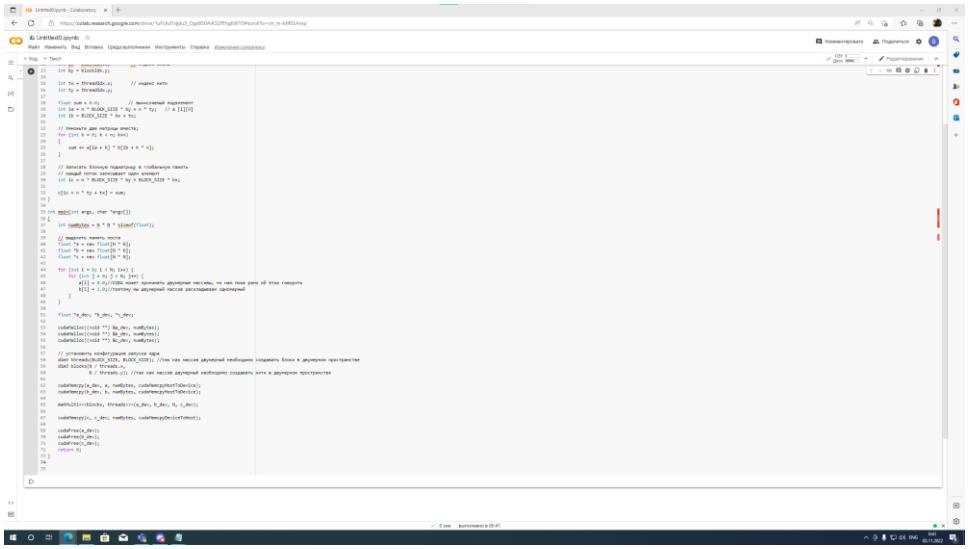
```
#include "device_launch_parameters.h"
#include <stdio.h>
#include <stdlib.h>

__global__ void add( int *a, int *b, int *c )
{
    *c = *a + *b;
}

int main()
{
    int a, b, c;

    int *dev_a, *dev_b, *dev_c;
    int size = sizeof( int );
    int blocs = 1, thread = 1;

    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = 2;
    b = 7;

    cudaMemcpy( dev_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy( dev_b, &b, size, cudaMemcpyHostToDevice);

    add<<< 1, 1 >>>( dev_a, dev_b, dev_c );

    cudaMemcpy( &c, dev_c, size, cudaMemcpyDeviceToHost);


    printf("%d + %d = %d\n", a, b, c);

    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```



Example 5.3

```
%%cu
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <stdlib.h>

#define BLOCK_SIZE  10           // размер подматрицы
#define N           10         // размер матрицы N * N

__global__ void matMulti(float *a, float *b, int n, float *c)
{
    int bx = blockIdx.x;         // индекс блока
    int by = blockIdx.y;

    int tx = threadIdx.x;        // индекс нити
    int ty = threadIdx.y;

    float sum = 0.0;             // вычисляемый подэлемент
    int ia = n * BLOCK_SIZE * by + n * ty;    // a [i][0]
    int ib = BLOCK_SIZE * bx + tx;

    // Умножьте две матрицы вместе;
    for (int k = 0; k < n; k++)
    {
        sum += a[ia + k] * b[ib + k * n];
    }


    // Записать блочную подматрицу в глобальную память
    // каждый поток записывает один элемент
    int ic = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;

    c[ic + n * ty + tx] = sum;
```

```
}

int main(int argc, char *argv[])
{
    int numBytes = N * N * sizeof(float);

    // выделить память хоста
    float *a = new float[N * N];
    float *b = new float[N * N];
    float *c = new float[N * N];

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            a[i] = 0.0;//CUDA может принимать двумерные массивы, но нам пока
 рано об этом говорить
            b[i] = 1.0;//поэтому мы двумерный массив раскладываем одномерный
        }
    }

    float *a_dev, *b_dev, *c_dev;

    cudaMalloc((void **) &a_dev, numBytes);
    cudaMalloc((void **) &b_dev, numBytes);
    cudaMalloc((void **) &c_dev, numBytes);

    // установить конфигурацию запуска ядра
    dim3 threads(BLOCK_SIZE, BLOCK_SIZE); //так как массив двумерный необход
имо создавать блоки в двумерном пространстве
    dim3 blocks(N / threads.x,
                N / threads.y); //так как массив двумерный необходимо создав
ать нити в двумерном пространстве

    cudaMemcpy(a_dev, a, numBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_dev, b, numBytes, cudaMemcpyHostToDevice);

    matMulti<<<blocks, threads>>>(a_dev, b_dev, N, c_dev);

    cudaMemcpy(c, c_dev, numBytes, cudaMemcpyDeviceToHost);



    cudaFree(a_dev);
    cudaFree(b_dev);
    cudaFree(c_dev);
    return 0;
}
```



Упражнение.

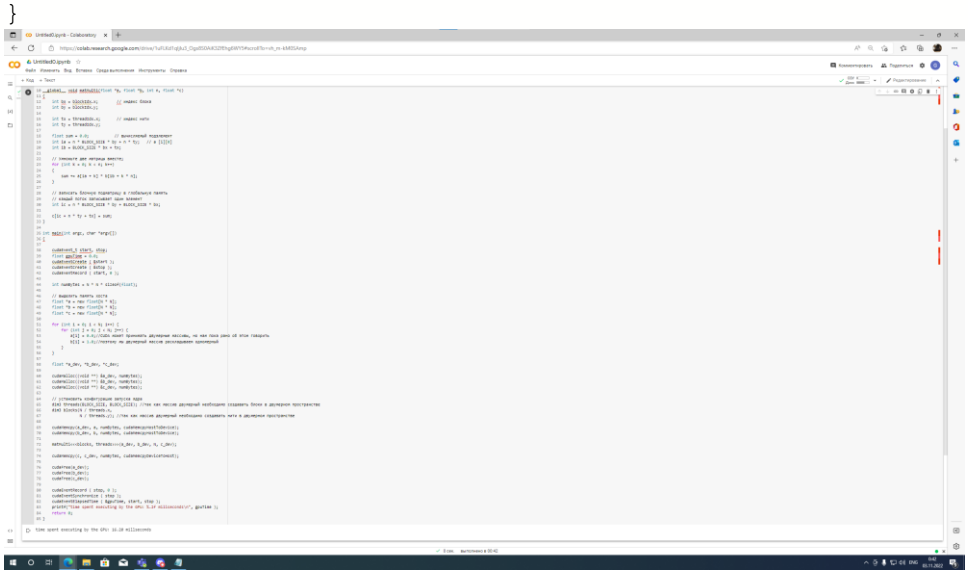
Замерить время выполнения функции с помощью спецификаторов CUDA

* cudaEvent_t start, stop;
* float gpuTime = 0.0;
* cudaEventCreate (&start);
* cudaEventCreate (&stop);
* cudaEventRecord (start, 0);
* .....
* cudaEventRecord (stop, 0);
* cudaEventSynchronize (stop);
* cudaEventElapsedTime (&gpuTime, start, stop);
* printf("time spent executing by the GPU: %.2f millseconds\n", gpuTime);

```
%%cu
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <stdlib.h>

#define BLOCK_SIZE  16          // размер подматрицы
#define N           1024        // размер матрицы N * N

__global__ void matMulti(float *a, float *b, int n, float *c)
{
    int bx = blockIdx.x;        // индекс блока
    int by = blockIdx.y;
```

```
    int tx = threadIdx.x;       // индекс нити
    int ty = threadIdx.y;

    float sum = 0.0;            // вычисляемый подэлемент
    int ia = n * BLOCK_SIZE * by + n * ty;   // a [i][0]
    int ib = BLOCK_SIZE * bx + tx;

    // Умножьте две матрицы вместе;
    for (int k = 0; k < n; k++)
    {
        sum += a[ia + k] * b[ib + k * n];
    }

    // Записать блочную подматрицу в глобальную память
    // каждый поток записывает один элемент
    int ic = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;

    c[ic + n * ty + tx] = sum;
}

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

    cudaEvent_t start, stop;
    float gpuTime = 0.0;
    cudaEventCreate ( &start );
    cudaEventCreate ( &stop );
    cudaEventRecord ( start, 0 );

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

    // выделить память хоста
    float *a = new float[N * N];
    float *b = new float[N * N];
    float *c = new float[N * N];

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            a[i] = 0.0;//CUDA может принимать двумерные массивы, но нам пока
 рано об этом говорить
            b[i] = 1.0;//поэтому мы двумерный массив раскладываем одномерный
        }
    }

    float *a_dev, *b_dev, *c_dev;

    cudaMalloc((void **) &a_dev, numBytes);
    cudaMalloc((void **) &b_dev, numBytes);
    cudaMalloc((void **) &c_dev, numBytes);

    // установить конфигурацию запуска ядра
```

```
    dim3 threads(BLOCK_SIZE, BLOCK_SIZE); //так как массив двумерный необход
имо создавать блоки в двумерном пространстве
    dim3 blocks(N / threads.x,
                N / threads.y); //так как массив двумерный необходимо создав
ать нити в двумерном пространстве

    cudaMemcpy(a_dev, a, numBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_dev, b, numBytes, cudaMemcpyHostToDevice);

    matMulti<<<blocks, threads>>>(a_dev, b_dev, N, c_dev);

    cudaMemcpy(c, c_dev, numBytes, cudaMemcpyDeviceToHost);

    cudaFree(a_dev);
    cudaFree(b_dev);
    cudaFree(c_dev);

    cudaEventRecord ( stop, 0 );
    cudaEventSynchronize ( stop );
    cudaEventElapsedTime ( &gpuTime, start, stop );
    printf("time spent executing by the GPU: %.2f millseconds\n", gpuTime );
    return 0;
}
```



Домашнее задание(базовое)

1. Вычисление числа ПИ.

Напишите программу для параллельного вычисления числа пи.

Для вычислений воспользуйтесь одной из формул:

- Формула Мадхавы-Лейбница (15 век)

  1 - 1/3 + 1/5 - 1/7 + 1/9 - ... = pi/4

- Формула Валлиса (17 век)

  2 * 2/3 * 4/3 * 4/5 * 6/5 * 6/7 * 8/7 * 8/9 * ... = pi/2

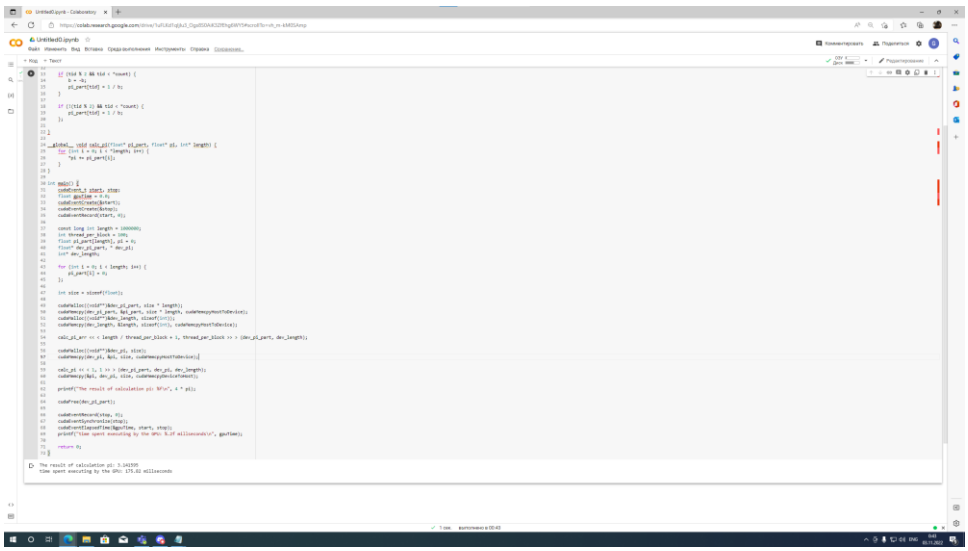
Это хорошо распаралеливаемая задача, так как вычисление членов ряда можно разбить на части для потоков и потом суммировать результаты.

Вычислить число пи используя CUDA. Замерить время выполнения 1млн итерраций на CPU и на GPU.

Сделать то же самое на 100 итеррациях, сделать выводы о времени выполнения.

```
%%cu
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <cstdlib>

__global__ void calc_pi_arr(float* pi_part, int* count) {

    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    float b = float(1 + (tid * 2));
    pi_part[tid] = 0;

    if (tid % 2 && tid < *count) {
        b = -b;
        pi_part[tid] = 1 / b;
    }

    if (!(tid % 2) && tid < *count) {
        pi_part[tid] = 1 / b;
    };

}

__global__ void calc_pi(float* pi_part, float* pi, int* length) {
    for (int i = 0; i < *length; i++) {
        *pi += pi_part[i];
    }
}

int main() {
    cudaEvent_t start, stop;
    float gpuTime = 0.0;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);
```

```
    const long int length = 1000000;
    int thread_per_block = 100;
    float pi_part[length], pi = 0;
    float* dev_pi_part, * dev_pi;
    int* dev_length;

    for (int i = 0; i < length; i++) {
        pi_part[i] = 0;
    };

    int size = sizeof(float);

    cudaMalloc((void**)&dev_pi_part, size * length);
    cudaMemcpy(dev_pi_part, &pi_part, size * length, cudaMemcpyHostToDevice)
;
    cudaMalloc((void**)&dev_length, sizeof(int));
    cudaMemcpy(dev_length, &length, sizeof(int), cudaMemcpyHostToDevice);

    calc_pi_arr << < length / thread_per_block + 1, thread_per_block >> > (d
ev_pi_part, dev_length);

    cudaMalloc((void**)&dev_pi, size);
    cudaMemcpy(dev_pi, &pi, size, cudaMemcpyHostToDevice);

    calc_pi << < 1, 1 >> > (dev_pi_part, dev_pi, dev_length);
    cudaMemcpy(&pi, dev_pi, size, cudaMemcpyDeviceToHost);

    printf("The result of calculation pi: %f\n", 4 * pi);

    cudaFree(dev_pi_part);

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&gpuTime, start, stop);
    printf("time spent executing by the GPU: %.2f millseconds\n", gpuTime);

    return 0;
}
```



2. Вычисление интеграла.

Аналогичным образом можно разбить на ряд подзадач и вычисление определённого интеграла. Напишите паралельную реализацию

методов интегрирования по одному из следующих алгоритмов:

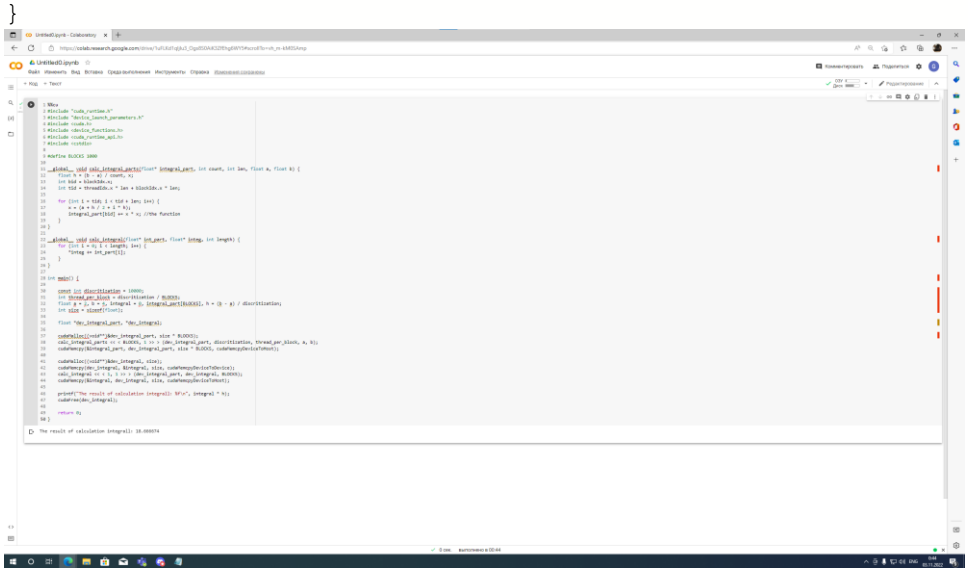
Метод прямоугольников (https://ru.wikipedia.org/wiki/Метод_прямоугольников)

Метод трапеций (https://ru.wikipedia.org/wiki/Метод_трапеций)

Метод Симпсона (https://ru.wikipedia.org/wiki/Формула_Симпсона)

```
%%cu
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <cuda.h>
#include <device_functions.h>
#include <cuda_runtime_api.h>
#include <cstdio>

#define BLOCKS 1000

__global__ void calc_integral_parts(float* integral_part, int count, int len
, float a, float b) {
    float h = (b - a) / count, x;
    int bid = blockIdx.x;
    int tid = threadIdx.x * len + blockIdx.x * len;

    for (int i = tid; i < tid + len; i++) {
        x = (a + h / 2 + i * h);
        integral_part[bid] += x * x; //the function
    }
}
```

```
__global__ void calc_integral(float* int_part, float* integ, int length) {
    for (int i = 0; i < length; i++) {
        *integ += int_part[i];
    }
}

int main() {

    const int discritization = 10000;
    int thread_per_block = discritization / BLOCKS;
    float a = 2, b = 4, integral = 0, integral_part[BLOCKS], h = (b - a) / d
iscritization;
    int size = sizeof(float);

    float *dev_integral_part, *dev_integral;

    cudaMalloc((void**)&dev_integral_part, size * BLOCKS);
    calc_integral_parts << < BLOCKS, 1 >> > (dev_integral_part, discritizati
on, thread_per_block, a, b);
    cudaMemcpy(&integral_part, dev_integral_part, size * BLOCKS, cudaMemcpyD
eviceToHost);

    cudaMalloc((void**)&dev_integral, size);
    cudaMemcpy(dev_integral, &integral, size, cudaMemcpyDeviceToDevice);
    calc_integral << < 1, 1 >> > (dev_integral_part, dev_integral, BLOCKS);
    cudaMemcpy(&integral, dev_integral, size, cudaMemcpyDeviceToHost);

    printf("The result of calculation integrall: %f\n", integral * h);
    cudaFree(dev_integral);

    return 0;
}
```



3. Инверсия

Создайте массив P[x][y][3] из x*y*3 элементо, где x и y соответствуют размерам вашего экрана в пикселях. Заполните её случайными значениями

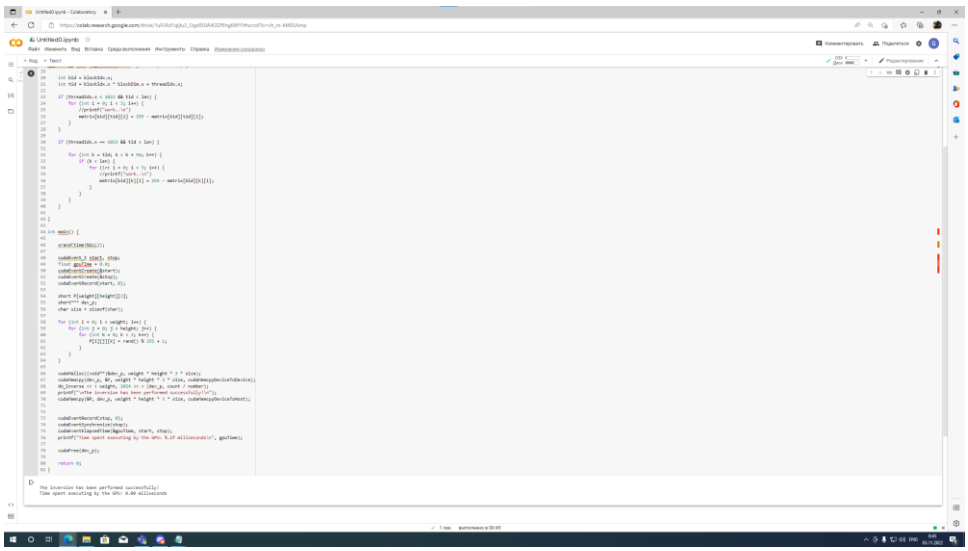
от 0 до 255.

Напишите паралельные программы с помощью OpenMP и CUDA проводящие инверсию цветов (замену каждого значения на P[i][j][k] = 255 - P[i][j][k]).

Оцените врямя работы программ и время на пересылку данных.

```
%%cu

#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <cuda.h>
#include <cuda_runtime_api.h>
#include <cstdio>
#include <ctime>
#include <iostream>

using namespace std;

#define weight 1280
#define height 720
#define number 3

#define count weight*height*number

__global__ void do_inverse(short*** matrix, int len) {

    int bid = blockIdx.x;
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (threadIdx.x < 1023 && tid < len) {
        for (int i = 0; i < 3; i++) {
            //printf("work..\n");
            matrix[bid][tid][i] = 255 - matrix[bid][tid][i];
        }
    }

    if (threadIdx.x == 1023 && tid < len) {

        for (int k = tid; k < k + 56; k++) {
            if (k < len) {
                for (int i = 0; i < 3; i++) {
                    //printf("work..\n");
                    matrix[bid][k][i] = 255 - matrix[bid][k][i];
                }
            }
        }
    }
```

```
}

int main() {

    srand(time(NULL));

    cudaEvent_t start, stop;
    float gpuTime = 0.0;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);

    short P[weight][height][3];
    short*** dev_p;
    char size = sizeof(char);

    for (int i = 0; i < weight; i++) {
        for (int j = 0; j < height; j++) {
            for (int k = 0; k < 3; k++) {
                P[i][j][k] = rand() % 255 + 1;
            }
        }
    }

    cudaMalloc((void**)&dev_p, weight * height * 3 * size);
    cudaMemcpy(dev_p, &P, weight * height * 3 * size, cudaMemcpyDeviceToDevi
ce);
    do_inverse << < weight, 1024 >> > (dev_p, count / number);
    printf("\nThe inversion has been performed successfully!\n");
    cudaMemcpy(&P, dev_p, weight * height * 3 * size, cudaMemcpyDeviceToHost
);


    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&gpuTime, start, stop);
    printf("Time spent executing by the GPU: %.2f millseconds\n", gpuTime);

    cudaFree(dev_p);

    return 0;
}
```



```
*/
```