Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Дисциплина: Архитектура процессоров и технология CUDA

ОТЧЕТ
по лабораторной работе №3
на тему
ВВЕДЕНИЕ В ТЕХНОЛОГИЮ CUDA
Вариант 2.1

Студенты: Е.А. Петрович
М.А. Ходосевич

Преподаватель: Т.С. Жук

МИНСК 2024

1 ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

1.1 Цель работы

Изучить особенности написания программ на с использованием CUDA. Изучить и сравнить производительность алгоритмов перестановки элементов матрицы на центральном и графическом процессоре с использованием технологии CUDA.

1.2 Исходные данные к работе

В исходной матрице NxM, используя окно, переставить элементы с шагом N/2 и M/2. Размер выходной матрицы меняется пропорционально в зависимости от заданного окна. Входное окно согласно варианту: 2x2. Тип данных – int.

Требования:

1. Реализация CPU и GPU.
2. Сравнение времени работы в едином формате (микро-, миллисекунды и т.п.). Измерение времени на GPU через события в CUDA.
3. Сравнение результатов работы: полное поэлементное сравнения массива и вывод фрагмента на экран.
4. Допускается использование двух массивов: с входными и выходными данными. Дополнительные массивы запрещены.
5. Инициализация массива случайными числами.
6. Проверка ошибок выполнения cudaError_t.

Для сдачи лабораторной работы достаточно просто переместить элементы, но можно сделать дополнительную оптимизацию.

Каждая оптимизация – отдельное ядро, основанное на предыдущей реализации. Исключение – разный размер матриц. Без транзакций в глобальную память – самый простой вариант. Можно добавить: работа с матрицами, не кратными размеру блока и размерам обрабатываемого фрагмента, корректный доступ в глобальную память, разделяемая память и корректное обращение по банкам памяти, применение технологии CUDA Stream.

2 ВЫПОЛНЕНИЕ РАБОТЫ

В ходе экспериментов изучалась производительность графического и центрального процессора на разных размерах матриц. Основной задачей лабораторной является перестановка элементов матрицы и замер времени выполнения.

Предполагалось, что при небольших размерах матриц работа центрального процессора будет занимать меньше времени. Также, при повторном запуске с теми же матрицами центральный процессор должен отрабатывать быстрее предыдущего из-за занесения данных в кеш. При увеличении размера матриц время работы центрального процессора должно увеличиваться и в какой-то момент стать больше времени работы графического процессора. То есть, предполагается, что на небольших размерах матрицы центральный процессор будет производительнее, на больших – производительнее графический процессор.

В таблицу 2.1 сведены время перестановки элементов матрицы для центрального (CPU) и графического (GPU) процессоров на различных матрицах.

Таблица 2.1 – Результаты работы

Матрица	CPU, мс	GPU, мс	CPU/GPU
10x10	0.003	0.6707	0.0045
100x100	0.1928	0.7014	0.1352
250x250	1.2528	0.9851	1.2341
500x500	4.093	1.3655	4.2316
1000x1000	11.7319	3.6291	3.2328
5000x5000	303.084	74.2973	4.0793
10000x10000	1222.67	318.426	3.8397
20000x20000	19860.3	1810.7	10.9683

Теория о том, что на больших матрицах время работы графического процессора будет меньше, чем центрального, и наоборот, подтвердилась. Это связано с хороших уровнем параллелизма и эффективным распределением вычислений по ядрам графического процессора

3 ВЫВОДЫ

В ходе лабораторной работы были изучены особенности написание программы с использованием CUDA. Были написаны реализации как на CPU, так и на GPU. Были подтверждены предполагаемые исходы работы программы. Также были получены навыки по установке требуемого ПО и работе на CUDA. Удалось написать работоспособную программу, которая удовлетворяет всем поставленным требованиям. По окончании выполнения лабораторной работы было доказано, что на больших размерах матриц намного эффективнее и производительнее будет использовать графический процессов.

ПРИЛОЖЕНИЕ А
(обязательное)

Исходный текст программы

Содержимое файла kernel.cu:

```
#include <iostream>
#include <cuda_runtime.h>
#include <curand.h>
#include <curand_kernel.h>
#include <vector>
#include <random>
#include <chrono>

// Вспомогательная функция для проверки ошибок CUDA
void checkCudaError(cudaError_t err, const char* msg) {
    if (err != cudaSuccess) {
        std::cerr << "CUDA Error (" << msg << "): " << cudaGetErrorString(err) << std::endl;
        exit(EXIT_FAILURE);
    }
}

// Функция для создания матрицы с случайными значениями
std::vector<std::vector<int>> create_random_matrix(int rows, int cols) {
    std::vector<std::vector<int>> matrix(rows, std::vector<int>(cols));
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(0, 100); // Случайные значения от 0 до 100

    for (int i = 0; i < rows; ++i)
        for (int j = 0; j < cols; ++j)
            matrix[i][j] = dis(gen);

    return matrix;
}

// Ядро CUDA для перестановки элементов с шагом N/2 и M/2
__global__ void rearrange_gpu(const int* matrix, int* result, int rows, int cols) {
    int half_rows = rows / 2;
    int half_cols = cols / 2;

    int i = (blockIdx.y * blockDim.y + threadIdx.y) * 2;
    int j = (blockIdx.x * blockDim.x + threadIdx.x) * 2;

    if (i + 1 < rows && j + 1 < cols) {
        // Перестановка элементов из блока 2x2
        result[(i / 2) * cols + (j / 2)] = matrix[i * cols + j];
        result[(i / 2) * cols + (j / 2 + half_cols)] = matrix[i * cols + j + 1];
        result[((i / 2) + half_rows) * cols + (j / 2)] = matrix[(i + 1) * cols + j];
        result[((i / 2) + half_rows) * cols + (j / 2 + half_cols)] = matrix[(i + 1) * cols + j + 1];
    }
}
```

```
    // Функция для вывода части матрицы
    void print_partial_matrix(const std::vector<std::vector<int>>& matrix,
int N) {
        for (int i = 0; i < matrix.size() && i < N; ++i) {
            for (int j = 0; j < matrix[0].size() && j < N; ++j) {
                std::cout << matrix[i][j] << " ";
            }
            std::cout << std::endl;
        }
    }

    // Функция для вывода части одномерного массива как двумерной матрицы
    void print_partial_flat_matrix(const std::vector<int>& matrix, int rows,
int cols, int N) {
        for (int i = 0; i < rows && i < N; ++i) {
            for (int j = 0; j < cols && j < N; ++j) {
                std::cout << matrix[i * cols + j] << " ";
            }
            std::cout << std::endl;
        }
    }

    // Функция для замера времени выполнения на CPU
    void measure_cpu(const std::vector<std::vector<int>>& matrix,
std::vector<std::vector<int>>& result) {

        int half_rows = matrix.size() / 2;
        int half_cols = matrix[0].size() / 2;

        auto start = std::chrono::high_resolution_clock::now();

        // Перестановка элементов с шагом N/2 и M/2 на CPU
        for (int i = 0; i < half_rows * 2; i += 2) {
            for (int j = 0; j < half_cols * 2; j += 2) {
                result[i / 2][j / 2] = matrix[i][j];
                result[i / 2][j / 2 + half_cols] = matrix[i][j + 1];
                result[i / 2 + half_rows][j / 2] = matrix[i + 1][j];
                result[i / 2 + half_rows][j / 2 + half_cols] = matrix[i +
1][j + 1];
            }
        }

        auto end = std::chrono::high_resolution_clock::now();
        std::chrono::duration<float> duration = end - start;

        std::cout << "\nCPU Time: " << duration.count() * 1000 << "
milliseconds" << std::endl; // вывод времени в миллисекундах
    }

    // Функция для замера времени выполнения на GPU
    void measure_gpu(const int* d_matrix, int* d_result_matrix, int rows, int
cols) {
        int threads_per_block = 32;
        dim3 threads(threads_per_block, threads_per_block);
        dim3 blocks((cols + threads_per_block * 2 - 1) / (threads_per_block
* 2),
            (rows + threads_per_block * 2 - 1) / (threads_per_block * 2));

        cudaEvent_t start, stop;
        checkCudaError(cudaEventCreate(&start), "cudaEventCreate (start)");
        checkCudaError(cudaEventCreate(&stop), "cudaEventCreate (stop)");
```

```
        checkCudaError(cudaEventRecord(start), "cudaEventRecord (start)");

        // Запуск CUDA ядра
        rearrange_gpu << <blocks, threads >> > (d_matrix, d_result_matrix,
rows, cols);
        checkCudaError(cudaGetLastError(), "Kernel execution"); // Проверка
на ошибку ядра
        checkCudaError(cudaDeviceSynchronize(), "cudaDeviceSynchronize");

        checkCudaError(cudaEventRecord(stop), "cudaEventRecord (stop)");
        checkCudaError(cudaEventSynchronize(stop), "cudaEventSynchronize");

        float milliseconds = 0;
        checkCudaError(cudaEventElapsedTime(&milliseconds,   start,   stop),
"cudaEventElapsedTime");
        std::cout << "\nGPU Time: " << milliseconds << " milliseconds" <<
std::endl; // вывод времени в миллисекундах

        checkCudaError(cudaEventDestroy(start), "cudaEventDestroy (start)");
        checkCudaError(cudaEventDestroy(stop), "cudaEventDestroy (stop)");
    }

    // Полное поэлементное сравнение матриц
    bool  compare_results(const  std::vector<std::vector<int>>&  cpu_matrix,
const std::vector<int>& gpu_matrix, int rows, int cols) {
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                if (abs(cpu_matrix[i][j] - gpu_matrix[i * cols + j]) > 1) {
// Учитываем, что тип int
                    return false;
                }
            }
        }
        return true;
    }

    int main() {
        int rows = 10000, cols = 10000;
        size_t matrix_size = rows * cols * sizeof(int);

        auto cpu_matrix = create_random_matrix(rows, cols);

        std::cout << "Original Matrix (first 10x10):" << std::endl;
        print_partial_matrix(cpu_matrix, 10);

        // Копирование данных в линейный массив для GPU
        std::vector<int> flat_cpu_matrix(rows * cols);
        for (int i = 0; i < rows; ++i)
            std::copy(cpu_matrix[i].begin(),           cpu_matrix[i].end(),
flat_cpu_matrix.begin() + i * cols);

        int* d_matrix;
        int* d_matrix_res;
        checkCudaError(cudaMalloc(&d_matrix,    matrix_size),    "cudaMalloc
(d_matrix)");
        checkCudaError(cudaMalloc(&d_matrix_res,  matrix_size),  "cudaMalloc
(d_matrix_res)");

        checkCudaError(cudaMemcpy(d_matrix,          flat_cpu_matrix.data(),
matrix_size, cudaMemcpyHostToDevice), "cudaMemcpy HostToDevice");
```

```
        std::vector<std::vector<int>>                        cpu_result(rows,
std::vector<int>(cols));
        measure_cpu(cpu_matrix, cpu_result);

        measure_gpu(d_matrix, d_matrix_res, rows, cols);

        std::vector<int> gpu_result(rows * cols);
        checkCudaError(cudaMemcpy(gpu_result.data(),          d_matrix_res,
matrix_size, cudaMemcpyDeviceToHost), "cudaMemcpy DeviceToHost");

        if (compare_results(cpu_result, gpu_result, rows, cols)) {
            std::cout << "CPU and GPU results match!" << std::endl;
        }
        else {
            std::cout << "Results differ!" << std::endl;
        }

        std::cout << "\nPartial Matrix CPU (first 10x10):" << std::endl;
        print_partial_matrix(cpu_result, 10);

        std::cout << "\n\nPartial Matrix GPU (first 10x10):" << std::endl;
        print_partial_flat_matrix(gpu_result, rows, cols, 10);

        checkCudaError(cudaFree(d_matrix), "cudaFree (d_matrix)");
        checkCudaError(cudaFree(d_matrix_res), "cudaFree (d_matrix_res)");

        return 0;
    }
```