Министерство образования Республики Беларусь

Учреждение образования «Полоцкий государственный университет имени Евфросинии Полоцкой»

Факультет информационных технологий

Кафедра технологий программирования

**Отчёт по лабораторной работе № 2 по курсу «Распределенные вычисления»**

«Типы памяти в CUDA. Оптимизация доступа»

ВЫПОЛНИЛ студент группы 21-ИТ-1

Шиковец Е.А.

ПРОВЕРИЛ преподаватель

Адамовский Е.Р.

Полоцк, 2024 г.

**ЦЕЛЬ РАБОТЫ**

Ознакомиться с рекомендациями по работе с памятью. Оптимизировать реализованный алгоритм.

**ОПИСАНИЕ ПРОДЕЛАННОЙ РАБОТЫ**

Задание:

Задание:

1. Модифицировать алгоритм для GPU из лабораторной работы №2 в соответствии с рекомендациями по работе с памятью, представленными в теории.
2. Реализовать предлагаемый алгоритм.

Листинг 1 – Program.cs

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <float.h>

#include <cuda.h>

#include <chrono>

#define BLOCK\_SIZE 256 // Размер блока потоков

#define MAX\_ITER 500 // Максимальное количество итераций

#define N 60000 // Количество точек данных

#define D 11 // Количество измерений

#define K 12 // Количество кластеров

// Функция CUDA для назначения кластеров

\_\_global\_\_ void assign\_clusters(float\* data, float\* centroids, int\* clusters) {

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

if (index < N) {

float min\_dist = FLT\_MAX;

int min\_index = -1;

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

float dist = 0.0;

for (int j = 0; j < D; j++) {

float data\_val = data[index \* D + j];

float centroid\_val = centroids[i \* D + j];

float diff = data\_val - centroid\_val;

dist += diff \* diff;

}

if (dist < min\_dist) {

min\_dist = dist;

min\_index = i;

}

}

clusters[index] = min\_index;

}

}

// Функция CUDA для вычисления центроидов

\_\_global\_\_ void compute\_centroids(float\* data, float\* centroids, int\* clusters) {

\_\_shared\_\_ float shared\_data[BLOCK\_SIZE][D];

\_\_shared\_\_ int shared\_clusters[BLOCK\_SIZE];

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

if (index < N) {

int cluster\_val = clusters[index];

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

float data\_val = data[index \* D + i];

shared\_data[threadIdx.x][i] = data\_val;

}

shared\_clusters[threadIdx.x] = cluster\_val;

}

\_\_syncthreads();

if (threadIdx.x < K) {

float sum[D] = { 0.0 };

int count = 0;

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

if (shared\_clusters[i] == threadIdx.x) {

for (int j = 0; j < D; j++) {

sum[j] += shared\_data[i][j];

}

count++;

}

}

if (count > 0) {

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

atomicAdd(&centroids[threadIdx.x \* D + i], sum[i]);

atomicAdd(&centroids[(K + threadIdx.x) \* D + i], (float)count);

}

}

}

}

int main() {

// Инициализация данных и центроидов

// Объявление указателей

float\* data;

float\* centroids;

int\* clusters;

// Выделение pinned-памяти

cudaMallocHost((void\*\*)&data, N \* D \* sizeof(float));

cudaMallocHost((void\*\*)&centroids, 2 \* K \* D \* sizeof(float));

cudaMallocHost((void\*\*)&clusters, N \* sizeof(int));

// Инициализация данных случайными значениями

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

for (int j = 0; j < D; j++) {

data[i \* D + j] = (float)rand() / RAND\_MAX;

}

}

// Инициализация центроидов случайными значениями

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

for (int j = 0; j < D; j++) {

centroids[i \* D + j] = (float)rand() / RAND\_MAX;

}

}

// Инициализация CUDA

float\* d\_data, \* d\_centroids;

int\* d\_clusters;

cudaMalloc((void\*\*)&d\_data, N \* D \* sizeof(float));

cudaMalloc((void\*\*)&d\_centroids, 2 \* K \* D \* sizeof(float));

cudaMalloc((void\*\*)&d\_clusters, N \* sizeof(int));

cudaMemcpyAsync(d\_data, data, N \* D \* sizeof(float), cudaMemcpyHostToDevice);

cudaMemcpyAsync(d\_centroids, centroids, K \* D \* sizeof(float), cudaMemcpyHostToDevice);

// Запуск таймера

auto start = std::chrono::high\_resolution\_clock::now();

// Запуск алгоритма K-средних

dim3 dimBlock(BLOCK\_SIZE, 1, 1);

dim3 dimGrid((N + BLOCK\_SIZE - 1) / BLOCK\_SIZE, 1, 1);

for (int iter = 0; iter < MAX\_ITER; iter++) {

assign\_clusters << <dimGrid, dimBlock >> > (d\_data, d\_centroids, d\_clusters);

cudaMemset(d\_centroids, 0, 2 \* K \* D \* sizeof(float));

compute\_centroids << <dimGrid, dimBlock >> > (d\_data, d\_centroids, d\_clusters);

cudaMemcpy(centroids, d\_centroids, 2 \* K \* D \* sizeof(float), cudaMemcpyDeviceToHost);

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

for (int j = 0; j < D; j++) {

if (centroids[(K + i) \* D + j] != 0) {

centroids[i \* D + j] /= centroids[(K + i) \* D + j];

}

}

}

cudaMemcpy(d\_centroids, centroids, K \* D \* sizeof(float), cudaMemcpyHostToDevice);

}

// Копирование кластеров обратно в хост-память

cudaMemcpy(clusters, d\_clusters, N \* sizeof(int), cudaMemcpyDeviceToHost);

// Остановка таймера

auto stop = std::chrono::high\_resolution\_clock::now();

// Вычисление продолжительности

auto duration = std::chrono::duration\_cast<std::chrono::milliseconds>(stop - start);

// Вывод принадлежности каждой точки данных к кластеру

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

printf("Data point %d is in cluster %d\n", i, clusters[i] + 1);

}

// Вывод центроидов

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

printf("Centroid %d: ", i + 1);

for (int j = 0; j < D; j++) {

printf("%f ", centroids[i \* D + j]);

}

printf("\n");

}

printf("Execution time: %lld milliseconds\n", duration.count());

// Очистка

cudaFree(d\_data);

cudaFree(d\_centroids);

cudaFree(d\_clusters);

cudaFreeHost(data);

cudaFreeHost(centroids);

cudaFreeHost(clusters);

return 0;

}

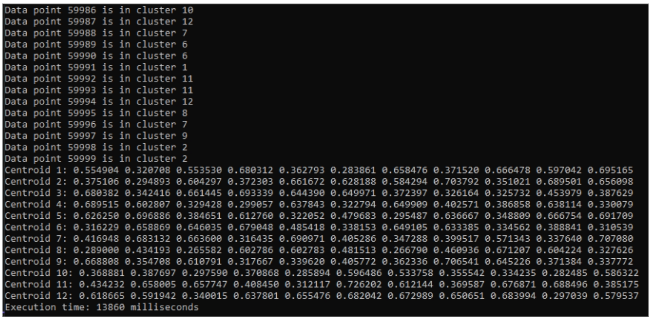


Рисунок 1 – Результат выполнения алгоритма на CPU

**ВЫВОД**

В результате выполнения лабораторной работы я ознакомился с рекомендациями по работе с памятью и оптимизировал реализованный алгоритм.