МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Факультет «Информационные технологии и прикладная математика»
Кафедра «Вычислительная математика и программирование»

Лабораторная работа №1
по курсу «Параллельная обработка данных»

Изучение технологии CUDA

Выполнил: В.И. Лобов
Группа: 8О-406Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2019

Условие

Цель работы: Ознакомление и установка программного обеспечения для работы с программно-аппаратной архитектурой параллельных вычислений(CUDA). Реализация одной из примитивных операций над векторами.
Вариант 7: Поэлементное вычисление модуля вектора.

Программное и аппаратное обеспечение

GPU:

Название: GeForce GTX 1060
Compute apability: 6.1
Графическая память: 2075328512
Разделяемая память: 49152
Константная память: 65536
Количество регистров на блок: 65536
Максимальное количество блоков: (2147483647, 65535, 65535)
Максимальное количество нитей: (1024, 1024, 64)
Количество мультипроцессоров: 10

Сведения о системе:

Процессор: Intel Core i7-8700k 4.5GHz
Оперативная память: 16Gb
HDD: 1Tb
Операционная система: Ubuntu 18.04
IDE: Nsight
Компилятор: nvcc

Метод решения

Выделим память на видеокарте с помощью функции cudaMalloc и инициализируем её путём копирования данных о векторах из оперативной памяти с помощью cudaMemcpy. Запустим функцию kernel на заданной конфигурации, состоящей из некоторого количества блоков и потоков. Функция kernel будет запущена для каждого потока, при этом каждый поток обработает не несколько элементов вектора, расположенных подряд, а расположенные на равном расстоянии друг от друга, имеющие смещение, равное количеству потоков.

Описание программы

Элемент вектора заменяется абсолютным значением данного элемента.

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

#define CSC(call)                                       \
do {                                                    \
    cudaError_t res = call;                             \
    if (res != cudaSuccess) {                           \
        fprintf(stderr, "ERROR in %s:%d. Message: %s\n",        \
                __FILE__, __LINE__, cudaGetErrorString(res));   \
        exit(0);                                        \
    }                                                   \
} while(0)
```

```
__global__ void kernel(double *vector, int n) {
    int offset = blockDim.x * gridDim.x;

    for (int i = threadIdx.x + blockIdx.x * blockDim.x; i < n; i += offset) {
        vector[i] *= vector[i] < 0 ? -1 : 1;
    }
}

int main() {
    int n;
    scanf("%d", &n);

    int size = n * sizeof(double);
    double *vector = (double *) malloc(size);
    for (int i = 0; i < n; ++i) {
        scanf("%lf", &vector[i]);
    }
    double *device_vector;

    CSC(cudaMalloc(&device_vector, size));
    CSC(cudaMemcpy(device_vector, vector, size, cudaMemcpyHostToDevice));

    cudaEvent_t start, end;
    CSC(cudaEventCreate(&start));
    CSC(cudaEventCreate(&end));
    CSC(cudaEventRecord(start));

    int threads_per_block = 1024;
    int blocks_per_grid = (n + threads_per_block - 1) / threads_per_block;
    kernel<<<blocks_per_grid, threads_per_block>>>(device_vector, n);

    CSC(cudaGetLastError());

    CSC(cudaEventRecord(end));
    CSC(cudaEventSynchronize(end));

    float time;
    CSC(cudaEventElapsedTime(&time, start, end));
    CSC(cudaEventDestroy(start));
    CSC(cudaEventDestroy(end));

    //printf("Time = %f ms\n", time);

    CSC(cudaMemcpy(vector, device_vector, size, cudaMemcpyDeviceToHost));
    CSC(cudaFree(device_vector));
    for (int i = 0; i < n; ++i) {
        printf("%.10e ", vector[i]);
    }
    printf("\n");
    free(vector);
    return 0;
}
```

Результаты(ms):

N	CPU	1, 32	32, 64	128, 128	256, 256	1024,1024
100	0.007	0.012256	0.011264	0.011840	0.012192	0.023552
10000	0.319	0.085056	0.013696	0.013216	0.010656	0.024000
100000	0.805	0.735904	0.020160	0.014304	0.013600	0.020672
1000000	5.972	11.031072	0.219616	0.109440	0.113536	0.115136
33554431	208.033	354.921295	7.461632	3.570432	3.694560	3.552928

Выводы

Алгоритм, работы программы довольно прост. Основная задача данной лабораторной работы — научиться писать программы, выполняемые на видеокарте в различных конфигурациях и убедиться, в каких случаях выигрывает параллельный алгоритм(на GPU), а в каких последовательный(на CPU). Анализируя скорость работы на разных конфигурациях, можно сделать вывод, что при маленьком значении количества элементов вектора выигрывает CPU. Это происходит из-за того, что создание потоков на GPU в данном случае занимает больше времени, чем сам алгоритм. При росте N выигрыш GPU становится всё более значительным.