МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Информационные технологии и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

**Лабораторная работа №1**
**по курсу «Программирование графических процессоров»**

# Изучение технологии CUDA

Выполнила: Алексюнина Ю.В.
Группа: 8О-407Б
Преподаватели:
К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2020

**Условие**

Цель работы. Ознакомление и установка программного обеспечения для работы с программно-аппаратной архитектурой параллельных вычислений(CUDA). Реализация одной из примитивных операций над векторами. В качестве вещественного типа данных необходимо использовать тип данных double. Все результаты выводить с относительной точностью $10^{-10}$ . Ограничение: $n < 2^{25}$ .

***Вариант 1. Сложение векторов***.

**Программное и аппаратное обеспечение**

**GPU:**

- Name: GeForce GTX 750 Ti
- Compute capability: 5.0
- Графическая память: 4294967295
- 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: 5

**Сведения о системе:**

- Процессор: Intel Core i5-4460 3.20GHz
- ОЗУ: 16 ГБ
- HDD: 930 ГБ

**Программное обеспечение:**

- OS: Windows 8.1
- IDE: Visual Studio 2019
- Компилятор: nvcc

**Метод решения**

Введем данные 2х векторов(массивов) в оперативную память. Затем перенесем их на видеопамять. Также выделим память на графическом процессоре под результат нахождения сложения двух векторов. Запустим ядро, которое будет пробегать по всем элементам каждого вектора, складывать их поэлементно и записывать в результирующий вектор. После вычисления, для обработки

результатов, перенесем данные обратно. Выведем полученный вектор.

**Описание программы**

1) Программа принимает на вход число n - размер каждого вектора.
2) Выделяем память на CPU для трех векторов размера n и заполняем два из них через потоковый ввод.
3) Выделяем память под них на GPU с помощью cudaMalloc().
4) Копируем данные из исходных векторов на GPU через cudaMemcpy().
5) Запускаем ядро - функция kernelAdd () со спецификатором вызова функции __global__, который определяет, что функция вызывается из CPU, а выполняется на GPU.
6) В функции kernelAdd () рассчитываем id текущей нити (index) и смещение (offset). В цикле запускаем расчет сложения элементов векторов.
7) Копируем обратно данные результирующего вектора из GPU в CPU с помощью функции cudaMemcpy()
8) Выводим получившийся вектор на печать.
9) Освобождаем используемую память функцией MemFree .

**Файл kernel.cu:**

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <iostream>

using namespace std;

__global__ void kernelAdd(double* v1, double* v2, double* v3, unsigned
long long n) {
   unsigned long long i = blockDim.x * blockIdx.x + threadIdx.x;
   unsigned long long offset = gridDim.x * blockDim.x;
   for (; i < n; i += offset) {
      v3[i] = v1[i] + v2[i];
   }
}
```

```
void MemFree(double* va, double* vb, double* vc, double* v1, double* v2,
double* v3)
{
   cudaFree(va);
   cudaFree(vb);
   cudaFree(vc);
   delete[] v1;
   delete[] v2;
   delete[] v3;
}

int main() {

   unsigned long long n;

   cin >> n;

   double* v1;
   v1 = new double[n] {};
   double* v2;
   v2 = new double[n] {};
   double* v3;
   v3 = new double[n] {};

   for (int i = 0; i < n; i++)
      cin >> v1[i];
   for (int i = 0; i < n; i++)
      cin >> v2[i];

   double* va = 0, * vb = 0, * vc = 0;

   cudaError_t cudaStatus = cudaSetDevice(0);
   if (cudaStatus != cudaSuccess) {
      cout << "cudaSetDevice failed!  Do you have a CUDA-capable GPU
installed?:)";
      MemFree(va, vb, vc, v1, v2, v3);
      return 0;
```

```
    }

    cudaStatus = cudaMalloc((void**)&va, n * sizeof(double));
    if (cudaStatus != cudaSuccess) {
        cout<< "cudaMalloc failed!";
        MemFree(va, vb, vc, v1, v2, v3);
        return 0;
    }

    cudaStatus = cudaMalloc((void**)&vb, n * sizeof(double));
    if (cudaStatus != cudaSuccess) {
        cout << "cudaMalloc failed!";
        MemFree(va, vb, vc, v1, v2, v3);
        return 0;
    }

    cudaStatus = cudaMalloc((void**)&vc, n * sizeof(double));
    if (cudaStatus != cudaSuccess) {
        cout << "cudaMalloc failed!";
        MemFree(va, vb, vc, v1, v2, v3);
        return 0;
    }
    cudaStatus = cudaMemcpy(va, v1, n * sizeof(double),
cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        cout << "cudaMemcpy failed!";
        MemFree(va, vb, vc, v1, v2, v3);
        return 0;
    }

    cudaStatus = cudaMemcpy(vb, v2, n * sizeof(double),
cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        cout << "cudaMemcpy failed!";
        MemFree(va, vb, vc, v1, v2, v3);
        return 0;
    }
```

```
    kernelAdd << <1024, 1024>> > (va, vb, vc, n);
    cudaStatus = cudaMemcpy(v3, vc, n * sizeof(double),
cudaMemcpyDeviceToHost);
    if (cudaStatus != cudaSuccess) {
        cout << "cudaMemcpy failed!";
        MemFree(va, vb, vc, v1, v2, v3);
        return 0;
    }

    for (int i = 0; i < n; i++)
        printf("%.10e ", v3[i]);

    cudaStatus = cudaDeviceReset();
    if (cudaStatus != cudaSuccess) {
        cout << "cudaDeviceReset failed!";
        MemFree(va, vb, vc, v1, v2, v3);
        return 1;
    }

    MemFree(va, vb, vc, v1, v2, v3);
    return 0;
}
```

**Результаты**

**Конфигурация ядра (1, 32)**

| Количество элементов | Время (мс) |
|---|---|
| 10 | 0.006880 |
| 100 | 0.006404 |
| 1000 | 0.007999 |
| 10000 | 0.008942 |
| 100000 | 0.038160 |
| 1000000 | 0.468160 |

**Конфигурация ядра (32, 32)**

| Количество элементов | Время (мс) |
|---|---|
| 10 | 0.006720 |
| 100 | 0.005504 |
| 1000 | 0.007232 |
| 10000 | 0.008352 |
| 100000 | 0.032960 |
| 1000000 | 0.408800 |

**Конфигурация ядра (64, 64)**

| Количество элементов | Время (мс) |
|---|---|
| 10 | 0.006528 |
| 100 | 0.005632 |
| 1000 | 0.006176 |
| 10000 | 0.008064 |
| 100000 | 0.014240 |
| 1000000 | 0.147264 |

**Конфигурация ядра (128, 128)**

| Количество элементов | Время (мс) |
|---|---|
| 10 | 0.006720 |
| 100 | 0.006752 |
| 1000 | 0.005952 |
| 10000 | 0.007616 |
| 100000 | 0.010528 |
| 1000000 | 0.116704 |

**Конфигурация ядра (256, 256)**

| Количество элементов | Время (мс) |
|---|---|
| 10 | 0.006496 |
| 100 | 0.005888 |
| 1000 | 0.005984 |
| 10000 | 0.005952 |
| 100000 | 0.011008 |
| 1000000 | 0.120512 |

**Конфигурация ядра (512, 512)**

| Количество элементов | Время (мс) |
|---|---|
| 10 | 0.007263 |
| 100 | 0.004201 |
| 1000 | 0.005039 |
| 10000 | 0.006352 |
| 100000 | 0.019919 |
| 1000000 | 0.080124 |

**Конфигурация ядра (1024, 1024)**

| Количество элементов | Время (мс) |
|---|---|
| 10 | 0.004140 |
| 100 | 0.005136 |
| 1000 | 0.006358 |
| 10000 | 0.006974 |
| 100000 | 0.017845 |
| 1000000 | 0.078980 |

**CPU**

| Количество элементов | Время (мс) |
|---|---|
| 10 | 0.006123 |
| 100 | 0.061804 |
| 1000 | 0.161795 |
| 10000 | 0.163084 |
| 100000 | 0.830208 |
| 1000000 | 1.243007 |

Выигрыш на графическом процессоре можно получить только при действительно больших данных. Иначе целесообразнее и быстрее произвести расчеты на процессоре

**Выводы**

Данная лабораторная работа знакомит с основами параллельного программирования на видеокартах NVIDIA — с технологией CUDA. Необходимо было разобраться с базовыми функциями и операциями. Так же, нужно было посмотреть, при каких условиях мы будем получать наибольшую производительность. Сам алгоритм предельно простой и сложностей при его написании не возникло.