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

Выполнил: А. О. Дубинин
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

GeForce 940MX

Compute capability:	5.0
Dedicated video memory:	4096 MB
shared memory per block:	49152 bytes
constant memory:	65536 bytes
Total number of registers available per block:	65536
Maximum number of threads per multiprocessor:	2048
Maximum number of threads per block:	1024
(3) Multiprocessors, (128) CUDA Cores/MP:	384 CUDA Cores

Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz

Architecture:	x86_64
Byte Order:	Little Endian
CPU(s):	4
Thread(s) per core:	2
Core(s) per socket:	2
CPU MHz: CPU max MHz: CPU min MHz:	713.848 3100,0000 400,0000
L1d cache: L1i cache: L2 cache: L3 cache:	64 KiB 64 KiB 512 KiB 3 MiB

RAM	8GiB SODIMM DDR4 Synchronous Unbuffered (Unregistered) 2400 MHz (0,4 ns)

SSD(SPCC_M.2_SSD)	223,6G
HDD(ST1000LM035-1RK172)	931,5G

OS: Ubuntu 20.04 focal
IDE: subl
compiler: nvcc

Метод решения

Пришлось совсем чуть чуть переписать код лекции, чтобы сложить два вектора на device'e.

Описание программы

Для решения задачи, было выделено 3 double массива: результирующий массив и два вектора. Тип double использовался для удовлетворения точности задачи. Исходя из результатов тестирование, использования разного кол-во блоков, потоков, я решил использовать <<<256, 256>>>.

```
__global__ void kernel(double *res, double *vec1, double *vec2, int n) {
        int i, idx = blockDim.x * blockIdx.x + threadIdx.x;
        int offset = blockDim.x * gridDim.x;
        for(i = idx; i < n; i += offset)
                res[i] = vec1[i] + vec2[i];
}
```

Результаты

1.

	Небольшой тест(2^6)	Средний тест(2^16)	Предельный тест(2^24)
<<<1, 32>>>	0.02	0.05	15.07
<<<32, 32>>>	0.01	0.09	17.27
<<<1, 128>>>	0.02	0.25	59.95
<<<128, 128>>>	0.02	0.06	16.28
<<<1, 256>>>	0.01	0.15	33.32
<<<256, 256>>>	0.02	0.05	16.82
<<<1, 512>>>	0.02	0.1	21.69
<<<512, 512>>>	0.02	1.51	15.12
<<<1, 1024>>>	0.02	0.09	16.89
<<<1024, 1024>>>	0.06	0.16	17.35

2.

	Небольшой тест(2^6)	Средний тест(2^16)	Предельный тест(2^24)
GPU<<<256, 256>>>	0.02	0.05	16.82
CPU	0.002	0.46	125.79

Выводы

Применять данный алгоритм можно в математических библиотеках, для которых сильно нужна скорость вычисления. Сложность возникла, что сначала я не увидел ограничения по точности задачи и решил, что использовать float будет разумнее, так как на лекции рассказывалось, что его применение быстрее, но как чекер меня огорчил, то я сразу понял в чем ошибка. Как показало сравнение с cpu. На cpu быстрее выполняются маленькие тесты, но на больших тестах значительно выигрывает gpu. Это естественно и понятно, что мы на маленьких тестах тратим больше времени на поддержание правильной работы gpu, чем на сами вычисления.