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

Москва, 2021

## 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 2: Вычитание векторов.*

Входные данные: На первой строке задано число $n$ – размер векторов. В следующих 2-х строках, записано по $n$ вещественных чисел -- элементы векторов.

Выходные данные: Необходимо вывести $n$ чисел – результат вычитания исходных векторов.

## Программное и аппаратное обеспечение

```
##### CUDA Info

/opt/cuda/extras/demo_suite/deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "NVIDIA GeForce MX150"
  CUDA Driver Version / Runtime Version          11.4 / 11.4
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 2003 MBytes (2099904512
bytes)
  ( 3) Multiprocessors, (128) CUDA Cores/MP:     384 CUDA Cores
  GPU Max Clock rate:                            1532 MHz (1.53 GHz)
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              64-bit
  L2 Cache Size:                                 524288 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072,
65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048
layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
```

```
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device
simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.4, CUDA Runtime
Version = 11.4, NumDevs = 1, Device0 = NVIDIA GeForce MX150
Result = PASS

##### CPU Info

Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Address sizes:                   39 bits physical, 48 bits virtual
Byte Order:                      Little Endian
CPU(s):                          8
On-line CPU(s) list:             0-7
Vendor ID:                       GenuineIntel
Model name:                      Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
CPU family:                      6
Model:                           142
Thread(s) per core:              2
Core(s) per socket:              4
Socket(s):                       1
Stepping:                        10
CPU max MHz:                     3400.0000
CPU min MHz:                     400.0000
BogoMIPS:                        3601.00
Flags:                           fpu vme de pse tsc msr pae mce cx8 apic
sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm
pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts
rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64
monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2
x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm
abm 3dnowprefetch cpuid_fault epb invpcid_single pti ibrs ibpb stibp
tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2
smep bmi2 erms invpcid mpx rdseed adx smap clflushopt intel_pt xsaveopt
xsavec xgetbv1 xsaves dtherm ida arat pln pts hwp hwp_notify hwp_act_window
hwp_epp
Virtualization:                  VT-x
L1d cache:                       128 KiB (4 instances)
L1i cache:                       128 KiB (4 instances)
L2 cache:                        1 MiB (4 instances)
L3 cache:                        6 MiB (1 instance)
NUMA node(s):                    1
NUMA node0 CPU(s):               0-7
Vulnerability Itlb multihit:     KVM: Mitigation: VMX disabled
Vulnerability L1tf:              Mitigation; PTE Inversion; VMX conditional
cache flushes, SMT vulnerable
Vulnerability Mds:               Vulnerable: Clear CPU buffers attempted,
no microcode; SMT vulnerable
Vulnerability Meltdown:          Mitigation; PTI
```

```
Vulnerability Spec store bypass: Vulnerable
Vulnerability Spectre v1:        Mitigation; usercopy/swapgs barriers and
__user pointer sanitization
Vulnerability Spectre v2:        Mitigation; Full generic retpoline, IBPB
conditional, IBRS_FW, STIBP conditional, RSB filling
Vulnerability Srbds:             Vulnerable: No microcode
Vulnerability Tsx async abort:   Not affected

##### RAM Info

               total        used        free      shared  buff/cache
available
Mem:           7.6Gi       1.7Gi       4.6Gi       633Mi       1.4Gi
5.1Gi
Swap:             0B          0B          0B
```

## Метод решения

Выделить одну область в куче – буффер - куда будут записаны поступающие значения и результирующие после выполнения, а также две области на GPU, куда будут записаны два вектора для подсчета разницы. Считать размер вектором и два раза считать эти вектора в буффер, попутно копируя их на GPU.

Ядро запускается на единственной сетке, размер который можно посчитать двумя методами:

1. Можно размер задать вручную, например <<< 1024, 1024 >>>. Тогда нужно в ядро добавить for-цикл, который начинает с текущего потока `blockDim.x * blockIdx.x + threadIdx.x` и имеет шаг размером с количеством потоков в сетке `blockDim.x * gridDim.x`. For-цикл содержит вычитание.
2. Количество блоков можно высчитывать вручную через `num_elements / block_size + 1`. Тогда, учитываю количество блоков, каждый поток будет высчитывать единственную пару значений под индексами, равными текущему потоку.

Во обоих случаях, вычитание деструктивно: теряется весь первый вектор и в него записывается результат.

Результат копируется в буффер и выводится.

## Описание программы

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

// Считывает из ввода значения в массив размером num_elements
void read_vec(double* vec, unsigned int num_elements) {
    for (unsigned int i = 0; i < num_elements; ++i) {
        scanf("%lf", &vec[i]);
    }
}
```

```
// Ядро, вычитающий из vec1 vec2; каждая пара элементов в своем потоке

__global__
void vec_diff(double* vec1, double* vec2, unsigned int num_elements) {
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < num_elements) {
        vec1[i] -= vec2[i];
   }
}

int main() {
    unsigned int num_elements;
    scanf("%u", &num_elements);
    size_t mem_size = num_elements * sizeof(double);

    double *h_a = (double*) malloc(mem_size);
    double *h_b = (double*) malloc(mem_size);
    read_vec(h_a, num_elements);
    read_vec(h_b, num_elements);

    double *d_a;
    double *d_b;
    cudaMalloc(&d_a, mem_size);
    cudaMalloc(&d_b, mem_size);
    cudaMemcpy(d_a, h_a, mem_size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, mem_size, cudaMemcpyHostToDevice);

    unsigned int blockSize = 512;
    unsigned int gridSize  = num_elements / blockSize + 1;
    vec_diff<<<gridSize, blockSize>>>(d_a, d_b, num_elements);

    cudaMemcpy(h_a, d_a, mem_size, cudaMemcpyDeviceToHost);

    for (unsigned int i = 0; i < num_elements; ++i) {
        printf("%.10lf ", h_a[i]);
    }
    printf("\n");

    free(h_a);
    free(h_b);
    cudaFree(d_a);
    cudaFree(d_b);
}
```

## Результаты

| **CPU** | |
|---|---|
| 100 | 0.1655us |
| 1000 | 0.3824us |
| 10000 | 5.6783us |
| 100000 | 35.1492us |
| 1000000 | 274.9287us |

| <<<1, 32>>> | |
|---|---|
| 100 | 3.5200us |
| 1000 | 9.4080us |
| 10000 | 68.160us |
| 100000 | 1080.2us |
| 1000000 | 10735us |

| <<<32, 32>>> | |
|---|---|
| 100 | 2.3680us |
| 1000 | 3.0080us |
| 10000 | 4.8640us |
| 100000 | 58.592us |
| 1000000 | 604.20us |

| <<<64, 64>>> | |
|---|---|
| 100 | 2.8480us |
| 1000 | 2.8800us |
| 10000 | 3.7120us |
| 100000 | 53.120us |
| 1000000 | 562.63us |

| <<<128, 128>>> | |
|---|---|
| 100 | 3.5840us |
| 1000 | 3.6160us |
| 10000 | 4.3510us |
| 100000 | 53.028us |
| 1000000 | 564.20us |

| <<<256, 256>>> | |
|---|---|
| 100 | 7.6800us |
| 1000 | 7.6480us |
| 10000 | 8.8400us |
| 100000 | 50.944us |
| 1000000 | 564.29us |

| <<<512, 512>>> | |
|---|---|
| 100 | 24.448us |
| 1000 | 24.608us |
| 10000 | 25.408us |
| 100000 | 65.440us |
| 1000000 | 561.67us |

| <<<1024, 1024>>> | |
|---|---|
| 100 | 103.68us |
| 1000 | 103.84us |
| 10000 | 105.63us |
| 100000 | 144.48us |
| 1000000 | 562.69us |

| <<<gridSize, blockSize>>> | |
|---|---|
| 100 | 3.2320us |
| 1000 | 3.1680us |
| 10000 | 4.3840us |
| 100000 | 53.376us |
| 1000000 | 558.88us |

## Выводы

Распараллеливание в CUDA имеет место там, где задача может быть разбита на какие-либо маленькие по размеру под задачи, например, подсчет положения и цвета треугольника в полигональной сетке.

Единственную сложность, которая появилась при написании алгоритма разности двух векторов, была понять, как происходит индексация в случае ограниченного числа блоков – для этого достаточно каждый блок считать за элемент массива “сетка”, каждый поток считать за элемент массива “блок”.

Измерение скорости на процессоре было выполнено с помощью библиотеки `<chrono>`, а на GPU – с помощью утилиты nvprof:

- На процессоре скорость выполнения была выше скорости на GPU, возможно, из-за того, что задача была больно простая и на запуск потоков уходило больше времени;
- Sweet spot при ручном подборе размеров блоков были размеры `<<<64, 64>>>`, после чего оптимизация была незначительная.