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»**

**Освоение программного обеспечения для работы с технологией CUDA.**

**Примитивные операции над векторами.**

Выполнил: М.А. Бронников

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.

**Вариант 4.** Поэлементное нахождение минимума векторов.

**Программное и аппаратное обеспечение**

Device: GeForce GT 545

Размер глобальной памяти: 3150381056

Размер константной памяти : 65536

Размер разделяемой памяти: 49152

Регистров на блок: 32768

Максимум потоков на блок: 1024

Количество мультипроцессоров : 3

OS: Linux Mint 20 Cinnamon

Редактор: VSCode

**Метод решения**

Для нахождения поэлементного минимума двух векторов достаточно вызвать количество нитей равное размеру массивов и записать в качестве результата минимум 2-ух соответствующих элементов массива по идентификатору в третий.

**Описание программы**

Для выполнения программы я реализовал собственный вектор в методе которого и вызывался kernel. Для того, чтобы выполнить поэлементную операцию минимума необходимо выделить 3 дополнительных блока памяти на device: в первых двух будут храниться 2 входных вектора, а в третий записываться результат. Для более оптимального обращения с памятью можно было обойтись 2 блоками, однако для лучшей читаемости кода и в целях лучше разобраться со структурой программирования на CUDA я не стал этого делать. После аллокации я скопировал данные из векторов в выделенные массивы с помощью функции cudaMemcpy. После работы kernel я скопировал результат в выходной вектор с помощью аналогичной функции.

Для запуска kernel на device необходимо задать количество блоков и потоков в каждом из блоков. Для одномерного массива нам достаточно вызывать блоки и нити в одном измерении. Вызов kernel с количеством нитей на блок — 512 и количеством блоков, достаточным для того, чтобы каждому отдельному потоку достался один элемент из каждого массива:

elem\_min<<<blocks + plus\_one, 512>>>(d\_left, d\_right, d\_ans, ans.\_size);

В самом kernel мы вычисляем общий индекс исполняемой нити который и будет индексом в массиве при условии idx < размер массива. Далее выполняем операцию нахождения минимума двух чисел из массивов с записью результа в третий:

template<typename T>

\_\_global\_\_ void elem\_min(T\* d\_left, T\* d\_right, T\* d\_ans, int size){

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

if(idx >= size){

return;

}

T l\_v = d\_left[idx];

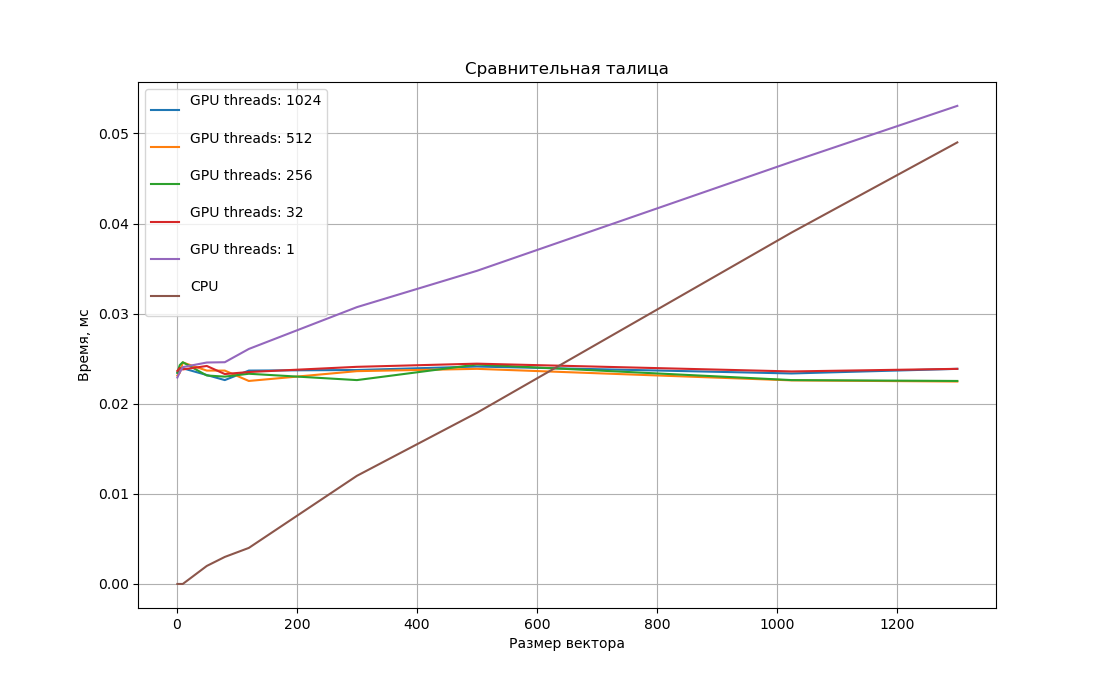
T r\_v = d\_right[idx];

d\_ans[idx] = l\_v < r\_v ? l\_v : r\_v;

}

**Результаты**

В силу идеи описанного мной решения, количество вызываемых блоков функционально заависит от количества нитей на блок, поэтому привожу результат замеров в зависимости от различного количества нитей на один блок:



Как видно для небольших векторов целесообразнее использовать CPU. Также не рекомендуется использовать блоки с одним потоком, поскольку они сильно уступают GPU с количеством потоков больше 32. Отметим, что размер вектора практически не влияет на время работы GPU в силу параллелизма.

Разница GPU по сравнению с CPU очевидна при предельном размере вектора:

**GPU (threads: 1024)**

**size: 33500000**

**time:** **26.35**ms

**CPU**

**size: 33500000**

**time:** **419.563**ms

**Выводы**

Реализованный мной алгоритм является простым в программировании, поскольку это вводное задание в курс программирования графических процессоров. Но даже не смотря на это он может найти свое применение как небольшая часть крупных систем, в которых требуются быстрые операции над массивами и матрицами.

В ходе выполнения столкнулся с неудобством от того, что я не являюсь счастливым обладателем видеокарты Nvidia, что заставило искать обходные пути. Первая лабораторная работа была отлажена с помощью сервиса Google Collaboratory, котррый бесплатно предостовляет бесплатный доступ к видеокарте Nvidia Tesla 80K.

Однако несмотря на все трудности я выполнил работу, которая наглядно показала мне, насколько большое преимущество может дать использование графического процессора вместо центрального.