МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Информационные технологии и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

**Лабораторная работа №2**

**по курсу «Программирование графических процессоров»**

**Обработка изображений на GPU. Фильтры.**

Выполнил: А. О. Тояков

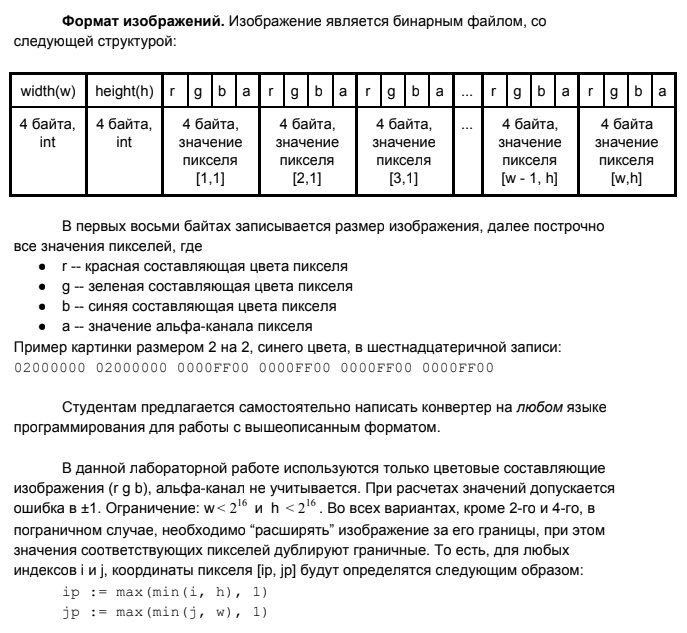
Группа: М8О-407Б-18

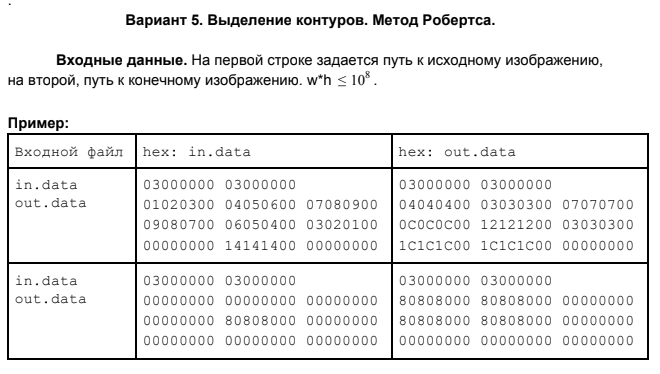
Преподаватели: К. Г. Крашенинников,

А. Ю. Морозов

# Условие

**Цель работы:** научится использовать GPU для обработки изображений. Использование текстурной памяти.





# Программное и аппаратное обеспечение

Device: GeForce MX250

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 Ubuntu 18.04

Редактор: VSCode

Компилятор: nvcc версии 11.4 (g++ версии 7.5.0)

# Метод решения

Для реализации метода Робертса необходимо вычислить яркость пикселя (i, j, где i и j – координаты равные номерам потока сетки по idx и idy), а также трёх соседних с координатами (i + 1, j), (i, j + 1) и (i + 1, j + 1). Учитывая, что интерфейс текстурной ссылки предусматривает выход за границы, нам об этом не нужно беспокоиться. Затем посчитать разности 2 пар яркостей (4 и 1, 2 и 3). Результирующая яркость res будет вычислять как корень квадратный из суммы квадратов этих разностей и если она будет > 255, то необходимо приравнять её к 255. После чего в результирующий массив out мы запишем точку (res, res, res, 255) по координатам: номер потока в сетке по y \* ширину картинки + номер потока в сетке по x. Потоки будут обрабатывать по несколько элементов сразу, используя переменные offsetx и offsety, как шаг. Затем нужно передать готовый массив обратно хосту и вывести результат в бинарный файл.

# Описание программы

Макрос **CSC**отвечает за отслеживание ошибок в функциях cuda, поэтому все cuda-вызовы оборачиваются в него и при cudaError\_t != cudaSuccess выводится сообщение об ошибке.

Texture<uchar4, 2, cudaReadModeElementType> tex – текстурная ссылка, которая будет использоваться для работы с данными которая имеет три параметра <тип данных, размерность, режим нормализации>

\_\_global\_\_ void kernel(int\* arr, n) – функция на GPU, в которой происходит обработка массива по потокам.

\_\_device\_\_ uchar4 get\_pixel(int i, int j) – функция на GPU, которая позволяет получить пиксель из данных, хранящихся в текстурной ссылке.

\_\_device\_\_ double brightness(uchar4 pixel) – функция на GPU, которая расчитывает яркость пикселя, поданного на вход по формуле 0.299 \* x + 0.587 \* y + 0.114 \* z

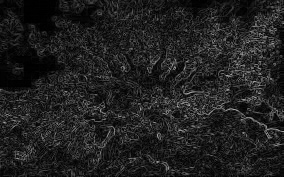
int main() ­– отвечает за ввод, передачу данных в kernel и вывод.

# Примеры работы программы

До обработки:

После обработки:

# тесты производительности

Для обеих программ данные генерировались рандомно. Изменялись только размеры картинки и конфигурации ядер. Работа на GPU:

|  |  |
| --- | --- |
| Тест: | Результат: |
| 10 \* 10 | kernel = «<1, 32»>, time = 0.028128 kernel = «<1, 64»>, time = 0.015296 kernel = «<1, 128»>, time = 0.015296 kernel = «<1, 256»>, time = 0.014336 kernel = «<1, 512»>, time = 0.013248 kernel = «<1, 1024»>, time = 0.015360 kernel = «<2, 32»>, time = 0.014336 kernel = «<2, 64»>, time = 0.013312 kernel = «<2, 128»>, time = 0.012288 kernel = «<2, 256»>, time = 0.012288 kernel = «<2, 512»>, time = 0.013312 kernel = «<2, 1024»>, time = 0.013312 kernel = «<4, 32»>, time = 0.013312 kernel = «<4, 64»>, time = 0.014176 kernel = «<4, 128»>, time = 0.014336 kernel = «<4, 256»>, time = 0.015360 kernel = «<4, 512»>, time = 0.017408 kernel = «<4, 1024»>, time = 0.012288 kernel = «<8, 32»>, time = 0.012288 kernel = «<8, 64»>, time = 0.012288 kernel = «<8, 128»>, time = 0.012288 kernel = «<8, 256»>, time = 0.017408 kernel = «<8, 512»>, time = 0.015360 kernel = «<8, 1024»>, time = 0.015360 kernel = «<16, 32»>, time = 0.014336 kernel = «<16, 64»>, time = 0.014336 kernel = «<16, 128»>, time = 0.014336 kernel = «<16, 256»>, time = 0.012288 kernel = «<16, 512»>, time = 0.014336 kernel = «<16, 1024»>, time = 0.015296 kernel = «<32, 32»>, time = 0.014336 kernel = «<32, 64»>, time = 0.015200 kernel = «<32, 128»>, time = 0.013312 kernel = «<32, 256»>, time = 0.017408 kernel = «<32, 512»>, time = 0.015360 kernel = «<32, 1024»>, time = 0.019360 kernel = «<64, 32»>, time = 0.015360 kernel = «<64, 64»>, time = 0.017408 kernel = «<64, 128»>, time = 0.015264 kernel = «<64, 256»>, time = 0.017280 kernel = «<64, 512»>, time = 0.017408 kernel = «<64, 1024»>, time = 0.028672 kernel = «<128, 32»>, time = 0.014336 kernel = «<128, 64»>, time = 0.014336 kernel = «<128, 128»>, time = 0.016384 kernel = «<128, 256»>, time = 0.015360 kernel = «<128, 512»>, time = 0.028672 kernel = «<128, 1024»>, time = 0.049152 kernel = «<256, 32»>, time = 0.017408 kernel = «<256, 64»>, time = 0.015360 kernel = «<256, 128»>, time = 0.015360 kernel = «<256, 256»>, time = 0.028576 kernel = «<256, 512»>, time = 0.048128 kernel = «<256, 1024»>, time = 0.097280 kernel = «<512, 32»>, time = 0.017408 kernel = «<512, 64»>, time = 0.015360 kernel = «<512, 128»>, time = 0.026624 kernel = «<512, 256»>, time = 0.048128 kernel = «<512, 512»>, time = 0.092160 kernel = «<512, 1024»>, time = 0.187360 kernel = «<1024, 32»>, time = 0.021504 kernel = «<1024, 64»>, time = 0.025600 kernel = «<1024, 128»>, time = 0.050176 kernel = «<1024, 256»>, time = 0.096256 kernel = «<1024, 512»>, time = 0.183296 kernel = «<1024, 1024»>, time = 0.364544 |
| 100 \* 100 | kernel = «<1, 32»>, time = 0.415200 kernel = «<1, 64»>, time = 0.249856 kernel = «<1, 128»>, time = 0.116672 kernel = «<1, 256»>, time = 0.116704 kernel = «<1, 512»>, time = 0.116736 kernel = «<1, 1024»>, time = 0.116736 kernel = «<2, 32»>, time = 0.249856 kernel = «<2, 64»>, time = 0.115712 kernel = «<2, 128»>, time = 0.115712 kernel = «<2, 256»>, time = 0.115712 kernel = «<2, 512»>, time = 0.115744 kernel = «<2, 1024»>, time = 0.116736 kernel = «<4, 32»>, time = 0.116736 kernel = «<4, 64»>, time = 0.115712 kernel = «<4, 128»>, time = 0.116736 kernel = «<4, 256»>, time = 0.115712 kernel = «<4, 512»>, time = 0.116736 kernel = «<4, 1024»>, time = 0.117760 kernel = «<8, 32»>, time = 0.115712 kernel = «<8, 64»>, time = 0.091136 kernel = «<8, 128»>, time = 0.115712 kernel = «<8, 256»>, time = 0.113664 kernel = «<8, 512»>, time = 0.115712 kernel = «<8, 1024»>, time = 0.119776 kernel = «<16, 32»>, time = 0.122752 kernel = «<16, 64»>, time = 0.118784 kernel = «<16, 128»>, time = 0.119808 kernel = «<16, 256»>, time = 0.120832 kernel = «<16, 512»>, time = 0.128000 kernel = «<16, 1024»>, time = 0.126976 kernel = «<32, 32»>, time = 0.097280 kernel = «<32, 64»>, time = 0.120832 kernel = «<32, 128»>, time = 0.125952 kernel = «<32, 256»>, time = 0.124928 kernel = «<32, 512»>, time = 0.130048 kernel = «<32, 1024»>, time = 0.130048 kernel = «<64, 32»>, time = 0.101376 kernel = «<64, 64»>, time = 0.113664 kernel = «<64, 128»>, time = 0.120832 kernel = «<64, 256»>, time = 0.133120 kernel = «<64, 512»>, time = 0.147456 kernel = «<64, 1024»>, time = 0.184320 kernel = «<128, 32»>, time = 0.108544 kernel = «<128, 64»>, time = 0.118784 kernel = «<128, 128»>, time = 0.133120 kernel = «<128, 256»>, time = 0.139296 kernel = «<128, 512»>, time = 0.179200 kernel = «<128, 1024»>, time = 0.356352 kernel = «<256, 32»>, time = 0.119648 kernel = «<256, 64»>, time = 0.125952 kernel = «<256, 128»>, time = 0.130080 kernel = «<256, 256»>, time = 0.180224 kernel = «<256, 512»>, time = 0.348160 kernel = «<256, 1024»>, time = 0.687104 kernel = «<512, 32»>, time = 0.104448 kernel = «<512, 64»>, time = 0.138240 kernel = «<512, 128»>, time = 0.179200 kernel = «<512, 256»>, time = 0.345088 kernel = «<512, 512»>, time = 0.686080 kernel = «<512, 1024»>, time = 1.375232 kernel = «<1024, 32»>, time = 0.126976 kernel = «<1024, 64»>, time = 0.182272 kernel = «<1024, 128»>, time = 0.344064 kernel = «<1024, 256»>, time = 0.678912 kernel = «<1024, 512»>, time = 1.349632 kernel = «<1024, 1024»>, time = 3.923968 |
| 1000 \* 1000 | kernel = «<1, 32»>, time = 27.872671 kernel = «<1, 64»>, time = 13.858816 kernel = «<1, 128»>, time = 10.720256 kernel = «<1, 256»>, time = 8.166400 kernel = «<1, 512»>, time = 6.575104 kernel = «<1, 1024»>, time = 6.896640 kernel = «<2, 32»>, time = 15.527968 kernel = «<2, 64»>, time = 11.299840 kernel = «<2, 128»>, time = 4.594688 kernel = «<2, 256»>, time = 5.416960 kernel = «<2, 512»>, time = 3.768320 kernel = «<2, 1024»>, time = 8.294432 kernel = «<4, 32»>, time = 12.028928 kernel = «<4, 64»>, time = 4.701184 kernel = «<4, 128»>, time = 3.727264 kernel = «<4, 256»>, time = 3.980288 kernel = «<4, 512»>, time = 3.243008 kernel = «<4, 1024»>, time = 6.144000 kernel = «<8, 32»>, time = 3.879936 kernel = «<8, 64»>, time = 3.751936 kernel = «<8, 128»>, time = 2.112512 kernel = «<8, 256»>, time = 3.913728 kernel = «<8, 512»>, time = 2.985984 kernel = «<8, 1024»>, time = 5.609472 kernel = «<16, 32»>, time = 4.930560 kernel = «<16, 64»>, time = 2.072576 kernel = «<16, 128»>, time = 2.066432 kernel = «<16, 256»>, time = 4.213760 kernel = «<16, 512»>, time = 3.980288 kernel = «<16, 1024»>, time = 5.898240 kernel = «<32, 32»>, time = 2.071552 kernel = «<32, 64»>, time = 2.066432 kernel = «<32, 128»>, time = 2.073600 kernel = «<32, 256»>, time = 3.745792 kernel = «<32, 512»>, time = 2.837504 kernel = «<32, 1024»>, time = 5.965760 kernel = «<64, 32»>, time = 2.066432 kernel = «<64, 64»>, time = 2.072576 kernel = «<64, 128»>, time = 2.091008 kernel = «<64, 256»>, time = 3.447808 kernel = «<64, 512»>, time = 3.103744 kernel = «<64, 1024»>, time = 5.478400 kernel = «<128, 32»>, time = 2.065408 kernel = «<128, 64»>, time = 2.091008 kernel = «<128, 128»>, time = 2.115552 kernel = «<128, 256»>, time = 2.802688 kernel = «<128, 512»>, time = 4.364288 kernel = «<128, 1024»>, time = 6.580224 kernel = «<256, 32»>, time = 2.075648 kernel = «<256, 64»>, time = 2.116608 kernel = «<256, 128»>, time = 2.153472 kernel = «<256, 256»>, time = 3.445760 kernel = «<256, 512»>, time = 3.566592 kernel = «<256, 1024»>, time = 10.571776 kernel = «<512, 32»>, time = 2.104320 kernel = «<512, 64»>, time = 2.189280 kernel = «<512, 128»>, time = 3.591168 kernel = «<512, 256»>, time = 3.526656 kernel = «<512, 512»>, time = 7.270400 kernel = «<512, 1024»>, time = 14.372864 kernel = «<1024, 32»>, time = 2.094080 kernel = «<1024, 64»>, time = 2.504704 kernel = «<1024, 128»>, time = 3.403776 kernel = «<1024, 256»>, time = 6.112256 kernel = «<1024, 512»>, time = 16.571392 kernel = «<1024, 1024»>, time = 34.676735 |

Работа на CPU:

|  |  |
| --- | --- |
| Тест: | Результат: |
| 10 \* 10 | 0.072 |
| 100 \* 100 | 4.623 |
| 1000 \* 1000 | 100.242 |

Как можно заметить на самом маленьком тесте программа на GPU даёт совсем не существенный выигрыш в скорости, а при выборе количества блоков и потоков > 1024 вообще проигрывает по времени. На среднем и большом тестах распараллеливание даёт большой выигрыш в скорости работы, однако указывать параметры ядра очень большими снова нецелесообразно.

# Выводы

Сделав вторую лабораторную работу, я узнал основы использования текстурной памяти и её особенности. Для начала скажу, что мне достался вариант № 5 – метод Робертса, который совсем прост в реализации. Используя пример с лекции, мне было легко разобраться с тем, как получать данные по текстурной ссылке. Для работы с текстурами нужно сначала записать данные в CUDA структуру cuda\_Array. Затем настроить интерфейс текстурной ссылки, где будут учтены и обработаны следующие параметры:

1. Обработка выхода за границы по каждому измерению, что сильно облегчает реализацию алгоритма. CUDA сама видит, попадает ли точка в заданный отрезок и если нет, то проводит преобразование.
2. Фильтрация данных, а именно, когда адрес и данные разных типов и непонятно какое значение возвращать из текстуры, CUDA выполняет один из способов: Point – берёт ближайшее значение из массива или Linear – значение берётся после выполнения линейной интерполяции.
3. Нормализация координат – переводит заданный отрезок в отрезок [0, 1].

После того, как интерфейс будет настроен, нам нужно привязать его к данным с помощью команды cudaBindTextureToArray. Теперь можно работать с текстурной ссылкой на девайсе.

Я думаю, что использовать технологию CUDA для обработки изображений фильтрами не очень эффективно, особенно на больших данных, т. к. частое обращение к памяти сильно замедляет программу. Гораздо лучше параллелить на видеокарте какие-нибудь нейросети. Однако, исходя из тестов производительности, на маленьких картинках программа на GPU имеет преимущество. В целом было интересно реализовывать эту программу, т. к. можно было наблюдать результат сразу же после выполнения, всего лишь конвертировав бинарный файл обратно в pic\_name.png.