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: М.А. Бронников

Группа: 8О-407Б

Преподаватели: К.Г. Крашенинников,

А.Ю. Морозов

Москва, 2020

**Условие**

**Цель работы:** Научиться использовать GPU для обработки изображений.

Использование текстурной памяти.

**Вариант 4.** Метод Собеля.

**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 каждого пикселя входного изображения следует выделить по отдельному потоку, каждый из которых будет обрабатывать окрестность этого потока. Поскольку при будет произведено много обращений к памяти, следует воспользоваться текстурной памятью, которая работает быстрее благодаря кэшированию. Результат обработки каждого пикселя будет записываться в выходной массив. Обработка производиться при помощи соответствующего ядра свертки.

**Описание программы**

Для выполнения программы я реализовал собственный класс изображения в методе которого и вызывался kernel. Чтобы влезть в ограничения по размеру с учетом размера текстурной памяти я считываю матрицу в транспонированном виде, если высота превышает ширину.

Для выполнения операции я создал текстурную ссылку в качестве глобального объекта, после чего выделил память для массива на девайсе, скопировал в нее массив-изображение и сделал бинд ссылки с массивом:

throw\_on\_cuda\_error(

cudaMalloc3DArray(&a\_data, &cfDesc, {\_widht, \_height, 0})

);

throw\_on\_cuda\_error(

cudaMemcpy2DToArray(

a\_data, 0, 0, \_data, \_widht \* sizeof(uint32\_t),

\_widht \* sizeof(uint32\_t), \_height, cudaMemcpyHostToDevice

)

);

throw\_on\_cuda\_error(

cudaBindTextureToArray(g\_text, a\_data, cfDesc)

);

Для аппаратной обработки граничных условий я использую Clamp аддресацию.

После расчета количества блоков по заданному количеству потоков на каждое из измерений я вызываю kernel:

dim3 threads = dim3(MAX\_X, MAX\_Y);

dim3 blocks = dim3(bloks\_y, bloks\_x);

// run filter

sobel<<<blocks, threads>>>(d\_data, \_height, \_widht);

throw\_on\_cuda\_error(cudaGetLastError());

В самом kernel мы вычисляем общий индекс исполняемой нити который и будет индексом в массиве при условии idx < размер массива. Далее производим расчет оператора Собеля для соответствующего пикселя:

\_\_global\_\_ void sobel(uint32\_t\* d\_data, uint32\_t h, uint32\_t w){

int32\_t idx = blockIdx.x \* blockDim.x + threadIdx.x;

int32\_t idy = blockIdx.y \* blockDim.y + threadIdx.y;

if(idx >= w || idy >= h){

return;

}

// ans pixel

uint32\_t ans = 0;

// locate area in mem(32 bite)

// compute grey scale for all pixels in area

float w11 = GREY(tex2D(g\_text, idx - 1, idy - 1));

float w12 = GREY(tex2D(g\_text, idx, idy - 1));

float w13 = GREY(tex2D(g\_text, idx + 1, idy - 1));

float w21 = GREY(tex2D(g\_text, idx - 1, idy));

float w23 = GREY(tex2D(g\_text, idx + 1, idy));

float w31 = GREY(tex2D(g\_text, idx - 1, idy + 1));

float w32 = GREY(tex2D(g\_text, idx, idy + 1));

float w33 = GREY(tex2D(g\_text, idx + 1, idy + 1));

// compute Gx Gy

float Gx = w13 + w23 + w23 + w33 - w11 - w21 - w21 - w31;

float Gy = w31 + w32 + w32 + w33 - w11 - w12 - w12 - w13;

// full gradient

int32\_t gradf = (int32\_t)sqrt(Gx\*Gx + Gy\*Gy);

// max(grad, 255)

gradf = gradf > 255 ? 255 : gradf;

// store values in variable for minimize work with global mem

ans ^= (gradf << 16);

ans ^= (gradf << 8);

ans ^= (gradf);

// locate in global mem

d\_data[idy\*w + idx] = ans;

}

После вызова kernel я копирую данные в массив и освобождаю выделенную память.

**Результаты**

Для иллюстрации результатов работы алгоритма я выбрал 3 изображения:





****

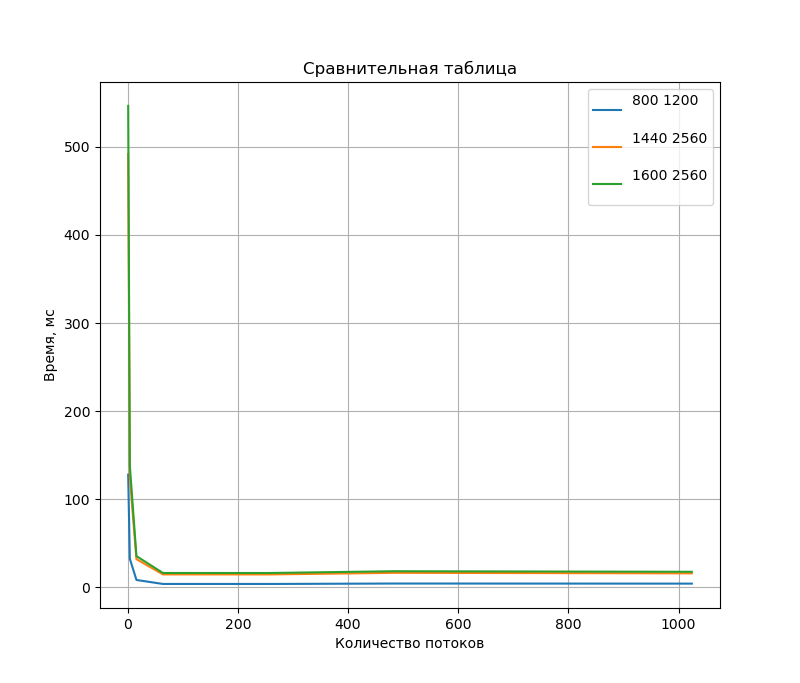
После чего я применил конвертер в заданный в задании формат, который я взял из материалов от преподавателя. После чего я применил к ним свою программу, и конвертировал обратно в jpg. Полученные результаты:







Я посмотрел как зависит время работы на этих изображениях от колличества потоков на один блок и получил следующий график:



Отсюда видно, что при количество потоков на блок не сильно влияет на производительность, начиная с 32 потоков, составляющих один варп. Впрочем это ожидаемый результат, поскольку каждый пиксель обрабатывается отдельной нитью вне зависимости от количества потоков на одну нить в силу специфики программы.

Однако разница значительна, если сравнивать с алгоритмом, написанным на CPU:

GPU threads: 1024

time: 4.21114

GPU threads: 1024

time: 15.8575

GPU threads: 1024

time: 17.6306

CPU

time: 1436.52

CPU

time: 5990.75

CPU

time: 5631.32

**Выводы**

Реализованный мной алгоритм широко применяется при обработке изображений, поскольку позволяет четко выделить контуры, что бывает полезно в задачах машинного обучения. При этом этот алгоритм является достаточно шумным(на изображении можно заметить светлые размывы), поэтому перед его применением рекомендуется применять сглаживающие фильтры.

Алгоритмы свертки хорошо распараллеливаются, что делает эффективным их использование на графических процессорах. Недаром, что современные свёрточные нейронные сети обучаются гораздо быстрее на GPU.

В ходе выполнения работы возникла трудность в том, как расположить данные в текстурной памяти так, чтобы влезть в ограничения, а также было совсем не очевидно, что следует решать задач в оттенках серого, а не в 3-ех каналах RGB изображения.