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: Гамов Павел Антонович
Группа: 8О-407Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2021

Условие

Цель работы:

Научиться использовать GPU для обработки изображений.

Использование текстурной памяти.

Вариант 6:

Выделение контуров. Метод Превитта.

Программное и аппаратное обеспечение

nvcc 7.0

Ubuntu 14.04 LTS

Compute capability	6.1
Name	GeForce GTX 1050
Total Global Memory	2096103424
Shared Mem per block	49152
Registers per block	65534
Max thread per block	(1024,1024,64)
Max block	(2147483647, 65535, 65535)
Total constant memory	65536
Multiprocessor’s count	5

Метод решения

Создание текстуры на основании данных. Далее обработка двумерным ядром по обоим плоскостям текстуры, так как мы работаем с картинкой. Нахождение градиентов по оси X и оси Y. Взятие нормы вектора, приведение к чёрно-белому формату картинки используя YUV (яркость, цветоразностные компоненты сигналов) преобразование. Запись в результирующий массив. Освобождение текстуры, памяти, запись в файл.

Использованная литература:

https://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf

http://harmanani.github.io/classes/csc447/Notes/Lecture15.pdf

Описание программы

Разделение по файлам, описание основных типов данных и функций. Обязательно описать реализованные ядра.

Реализовано одно ядро prewitt, в котором вычисляется два градиента, после он записывается в результирующий массив.
Пример вычисления градиента по оси X:

```
int sx1[2] = {max(min(x+1, w-1),0),max(min(x-1, w-1),0)};
int sy1[3] = {max(min(y, h-1),0),max(min(y+1, h-1),0),max(min(y-1, h-1),0)};
```

Находим координаты пикселей из которых будем брать значения для градиента.

```
for (int j=0; j<3; j++) {
   piv = tex2D(tex, sx1[0], sy1[j]);
   gx += (float)piv.x * 0.299 + (float)piv.y * 0.587 + (float)piv.z * 0.114;
   piv = tex2D(tex, sx1[1], sy1[j]);
   gx -= (float)piv.x * 0.299 + (float)piv.y * 0.587 + (float)piv.z * 0.114;
}
```

Далее в цикле для всех найденных точек обращаемся к текстуре и получаем нужные пиксели, далее переводим их в черно-белый формат на основании яркости, прибавляем к градиенту, или вычитаем.

```
float g = sqrt(gx*gx + gy*gy);
unsigned char mean = (unsigned char) min(255,(int)g);
res[y * w + x] = make_uchar4(mean, mean, mean, p.w);
```

Получив два градиента, находим корень из суммы квадратов, обязательно проверяем на переполнение функцией min. Записываем значения в результирующий массив.

```
// текстурная ссылка <тип элементов, размерность, режим нормализации>
texture<uchar4, 2, cudaReadModeElementType> tex;
```

Выделяем глобальную текстуру. Данный объект предоставляет специфический вид памяти – текстурный. Особенности использования: только чтение, кеширование данных, доступ из всех блоков. Данный тип данных надо использовать когда кол-во чтений велико, а мы для каждого блока читаем 6*2 + 1 пикселя, так как данные только читаются и нет записи, данный вид памяти быстрее обычного выделенного массива.

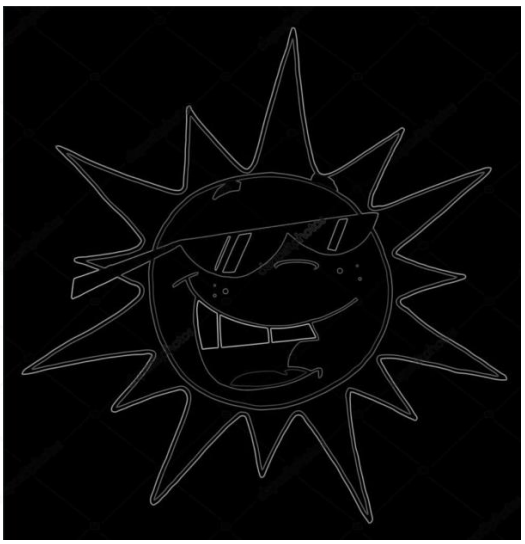
```
// Подготовка данных для текстуры
cudaArray *arr;
cudaChannelFormatDesc ch = cudaCreateChannelDesc<uchar4>();
cudaMallocArray(&arr, &ch, w, h);
cudaMemcpyToArray(arr,0,0,data,sizeof(uchar4)*w*h, cudaMemcpyHostToDevice);

// Подготовка текстурной ссылки, настройка интерфейса работы с данными
// Политика обработки выхода за границы по каждому измерению
tex.addressMode[0] = cudaAddressModeClamp;
tex.addressMode[1] = cudaAddressModeClamp;
tex.channelDesc = ch;
tex.filterMode = cudaFilterModePoint; // Без интерполяции при обращении по дробным координатам
tex.normalized = false; // Режим нормализации координат: без нормализации
```

Формально, используя параметр border или clamp в обработках выхода за границу текстуры, мне **не придется** обрабатывать выход за границы в ядре.

```
// Связываем интерфейс с данными
cudaBindTextureToArray(tex, arr, ch);

uchar4 * dev_out;
cudaMalloc(&dev_out, sizeof(uchar4) * w * h);

prewitt<<<dim3(16, 16), dim3(16, 32)>>>(dev_out, w, h);

cudaMemcpy(data, dev_out, sizeof(uchar4) * w * h, cudaMemcpyDeviceToHost);

// Отвязываем данные от текстурной ссылки
cudaUnbindTexture(tex);
```

Результаты

	1500x1500	2500x2500	5000x5000
(32,32), (32,32)	2.2816e-05	2.6272e-05	2.4768e-05
(64,64), (64,64)	1.1072e-05	1.3984e-05	1.3792e-05
(128,128), (128,128)	1.136e-05	1.4208e-05	1.3632e-05
(256,256), (256,256)	1.2512e-05	1.2928e-05	1.3312e-05
(512,512), (512,512)	1.2096e-05	1.3952e-05	1.3344e-05
(1024,1024), (1024,1024)	1.216e-05	1.3216e-05	1.2704e-05
C++	0.549501	1.50992	6.06611



Выводы

Использование технологии Cuda позволяет существенно сократить время обработки изображений. Встроенные структуры, как например текстуры, позволяют писать ядра обработки изображений, которые выполняются быстрее и умнее, нежели на ЦП. Главным образом сложности были с понимаем работы текстуры, как к ней обращаться и выделять.