МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №*2*
по курсу «Программирование графических процессоров»

Обработка изображений на GPU. Фильтры.

Выполнил: *А. Ю. Голов*
Группа: *М8О-401*
Преподаватель: А.Ю. Морозов

Москва, 2025

Условие

Цель работы. Научиться использовать GPU для обработки изображений.
Использование текстурной памяти и двухмерной сетки потоков.
Вариант 6. Выделение контуров. Метод Превитта.

Программное и аппаратное обеспечение

Видеокарта: NVIDIA GeForce RTX 2060 Mobile

- Compute Capability: 7.5
- Графическая память: 6 ГБ GDDR6, с 192-битной шиной и пропускной способностью 336 ГБ/с
- Разделяемая память: до 64 КБ на мультипроцессор
- Константная память: 64 КБ
- Количество регистров на блок: 65 536
- Максимальное количество блоков на мультипроцессор: 16
- Максимальное количество нитей на мультипроцессор: 1 024
- Количество мультипроцессоров (SM): 30

Процессор: Intel Core i7-9750H имеет следующие характеристики:

- Количество ядер: 6
- Количество потоков: 12
- Техпроцесс: 14 нм

Оперативная память:

- Объём: 16 ГБ
- Тактовая частота: 3500 MHz
- Поколение: DDR4

Жёсткий диск:

- Объём: 512 ГБ
- Формат: SSD M2

Программное обеспечение:

- Операционная система: Ubuntu 24.04 LTS
- IDE: Lunar Vim

Метод решения

Программа реализует вычисление градиента изображения с использованием CUDA. Она считывает изображение в формате `uchar4` (RGBA), загружает его в текстурную память GPU и применяет оператор Собеля для выделения границ. Основные этапы решения:

1. Чтение входного изображения – данные загружаются из бинарного файла, извлекаются размеры изображения и массив пикселей.
2. Копирование данных на GPU – создаётся массив `cudaArray`, куда копируются данные изображения. Затем создаётся текстурный объект для удобного доступа к пикселям в ядре CUDA.
3. Запуск CUDA-ядра – используется сетка потоков `dim3(16, 16)`, каждый поток вычисляет градиент яркости пикселя, используя оператор Собеля:
    - Для каждого пикселя берутся соседние пиксели в горизонтальном и вертикальном направлениях.
    - Вычисляются градиенты `accX` и `accY` (производные по осям X и Y).
    - Итоговый градиент определяется как accX2+accY2\sqrt{accX^2 + accY^2}, ограничивается в пределах [0,255][0, 255] и записывается в выходной массив.
4. Копирование результата с GPU на CPU – обработанное изображение передаётся обратно в память хоста.
5. Сохранение результата – записывается в бинарный файл с тем же форматом, что и входные данные.

Таким образом, программа выполняет фильтрацию изображения, выделяя границы с использованием операторов градиента, эффективно реализованных на GPU.

Описание программы

Программа для конвертирования jpg в заданный формат данных и для обратного конвертирования.

```
def convert_jpg_to_custom_format(input_path, output_path):
    image = Image.open(input_path).convert("RGBA")
    width, height = image.size
    pixels = list(image.getdata())

    with open(output_path, "wb") as f:
        f.write(struct.pack("<II", width, height))

        for pixel in pixels:
            r, g, b, a = pixel
            f.write(struct.pack("<BBBB", r, g, b, a))

def convert_custom_to_jpg(input_path, output_path):
    with open(input_path, "rb") as f:
        width, height = struct.unpack("<II", f.read(8))
```

```
pixels = []
for _ in range(width * height):
    r, g, b, a = struct.unpack("<BBBB", f.read(4))
    pixels.append((r, g, b))

image = Image.new("RGB", (width, height))
image.putdata(pixels)
image.save(output_path, "JPEG")
```

Программа вычислений на графическом процессоре:

1. CUDA-ядро `kernel`
    - Загружает пиксели из текстуры:

```
tmp = tex2D<uchar4>(texture, x + curX, y + curY);
```

    - Преобразует цвет в яркость:

```
float Y = 0.299 * tmp.x + 0.587 * tmp.y + 0.114 * tmp.z;
```

    - Вычисляет градиент по Собелю:

```
float grad = min(max(sqrt(accX * accX + accY * accY), 0.0f),
255.0f);
```

    - Записывает результат:

```
res[y * width + x] = make_uchar4(grad, grad, grad, tmp.w);
```

2. Создание текстурного объекта
    - Выделяется `cudaArray`:

```
cudaMallocArray(&arr, &channel, width, height);
```

    - Описывается ресурс:

```
cudaResourceDesc resDesc = {};
resDesc.resType = cudaResourceTypeArray;
resDesc.res.array.array = arr;
```

    - Настраивается текстурный объект:

```
cudaTextureDesc texDesc = {};
texDesc.addressMode[0] = cudaAddressModeClamp;
texDesc.addressMode[1] = cudaAddressModeClamp;
texDesc.filterMode = cudaFilterModePoint;
texDesc.readMode = cudaReadModeElementType;
texDesc.normalizedCoords = false;

cudaTextureObject_t tex = 0;
```

```
cudaCreateTextureObject(&tex, &resDesc, &texDesc, NULL);
```

3. Запуск CUDA-ядра

- Выделяется память:

```
cudaMalloc(&res, sizeof(uchar4) * width * height);
```

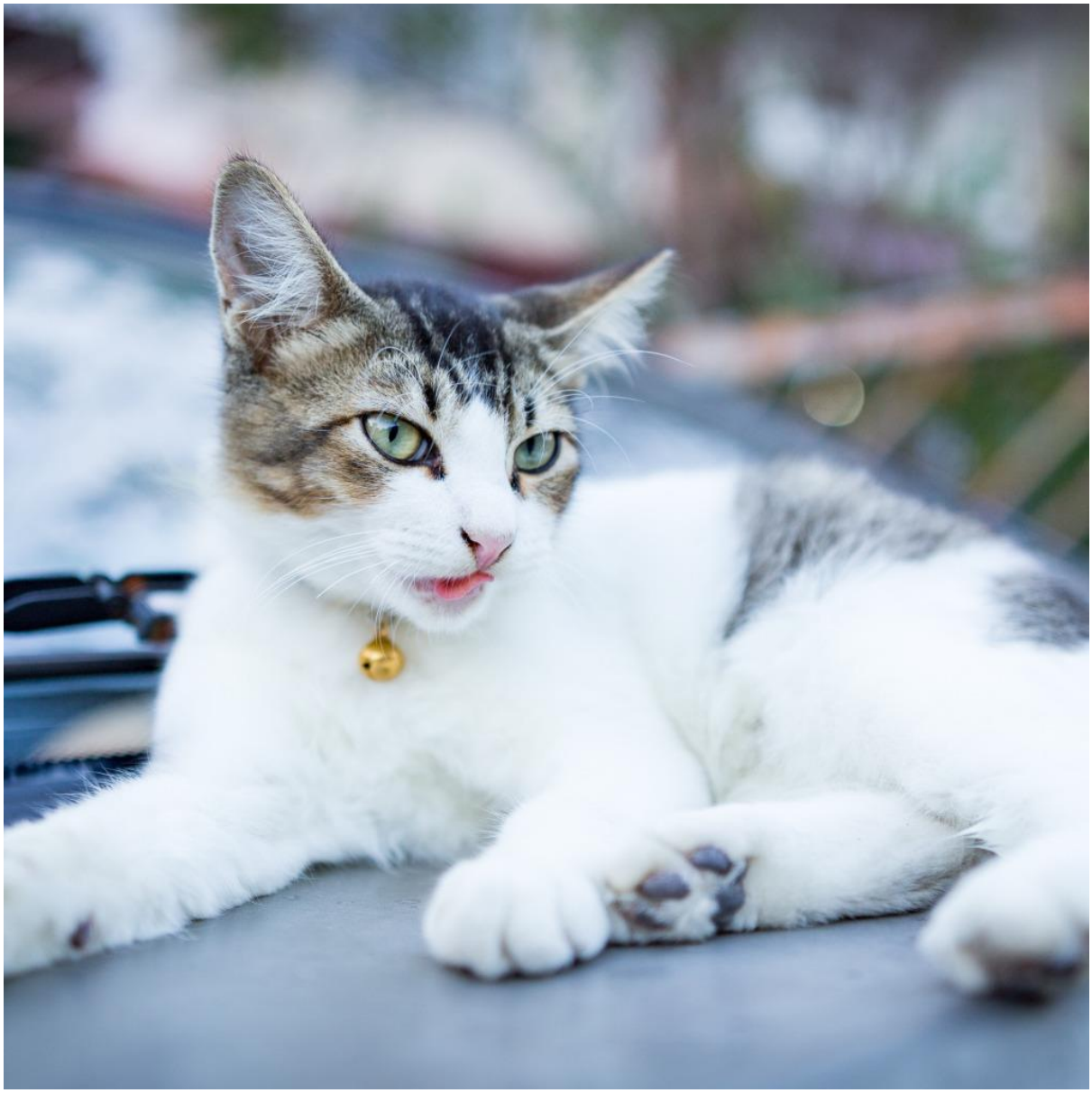
- Запускается обработка:

```
kernel<<< dim3(16, 16), dim3(32, 32) >>>(tex, width, height, res);
```

Результаты

В качестве показательных входных данных было использовано два изобораження: размером 1024x1024 пикселя и размером 4096x4096 пикселей.

1. (1024x1024)



2. (4096x4096)

Замеры производительности:
Измерения указаны в миллисекундах

Конфигурация ядра	Картинка №1 (1024x1024)	Картинка №2 (4096x4096)
<<<64, 64>>>	1.45416	37.5297
<<<128, 128>>>	1.45701	41.2902
<<<256, 256>>>	3.7171	42.5209
<<<512, 512>>>	7.15654	45.5664
<<<1024, 1024>>>	19.7872	36.8498

Выводы

Реализованный алгоритм решает задачи предобработки изображений перед сегментацией, обнаружения контуров в медицинских снимках и спутниковых данных, а также улучшения качества изображения в системах машинного зрения.

CUDA-реализация требует оптимизации работы с памятью и грамотного распределения потоков. Основная сложность заключалась в работе с границами изображения и балансировке сетки потоков для разных размеров входных данных. Возникли проблемы с подбором оптимальной конфигурации сетки потоков, особенно при обработке больших изображений, а также с неравномерной загрузкой GPU при малых входных данных.

При сравнении результатов стало очевидно, что для маленьких изображений ускорение незначительно из-за накладных расходов запуска ядер. Для больших изображений производительность падает при увеличении количества потоков из-за конкуренции за ресурсы. Оптимальной оказалась конфигурация `<<<64, 64>>>`, обеспечивающая минимальное время работы при больших входных данных.