Реализация эффективного использования нейронных сетей на графических процессорах

Дипломная работа

Бинцаровский Леонид Петрович

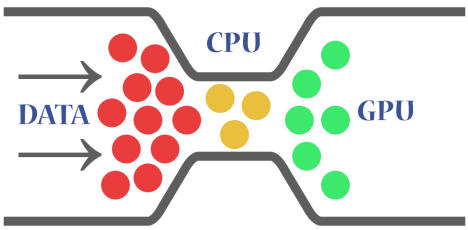
Белорусский государственный университет
ФПМИ, ДМА, 4 курс
руководитель: старший преподаватель Пирштук Д. И.

Минск, 2025

Постановка задачи

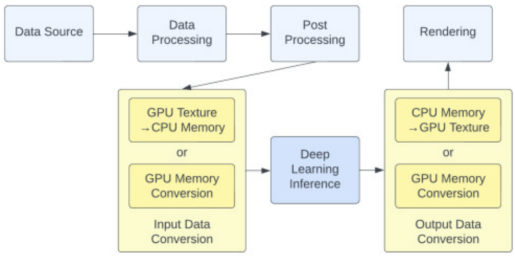
Реализовать инференс нейросетевой модели (или её подграфа) в виде последовательности шейдеров, каждый из которых реализует один или несколько операторов модели. При этом необходимо обеспечить:

- Минимальное число проходов по данным и пересчетов;
- Хранение промежуточных результатов в GPU-текстурах;
- Минимизацию сложных операций, плохо распараллеливаемых на GPU;
- Оптимальное размещение весов и параметров модели.



Актуальность поставленной задачи

TensorFlow Lite (TF-Lite), PyTorch Mobile и CoreML — три наиболее популярных фреймворка для мобильного инференса — предоставляют абстрактные интерфейсы для работы с нейросетевыми моделями, но не обеспечивают тесной связки с текстурной памятью GPU и графическим пайплайном, основанным на OpenGL или Vulkan.



- Модели, содержащие редкие операторы, могут не поддерживаться GPU-бэкендом.
- Форматы Tensor или Mat, используемые в большинстве фреймворков, не совместимыми с текстурами OpenGL.
- Невозможность непосредственной передачи данных из шейдера в шейдер без выхода за пределы графического пайплайна затрудняет реализацию end-to-end решений.

Описание среды разработки

Для реализации GPU-only пайплайна в рамках задачи эффективного использования нейронных сетей на графических процессорах будут использоваться:

API доступа к GPU

OpenGL ES версии 3.20

Модель

ESPCN_2X_16_16_4.h5

Языки программирования, среда разработки и операционные системы

C++, Visual Studio и Windows, Android Studio и Android

Основные этапы конвейера

Разработанный пайплайн можно разделить на четыре основных этапа:

1. Преобразование модели в удобный для работы формат — json файл;
2. Загрузка модели и манипуляции с ней;
3. Генерация вычислительного графа и необходимая предобработка;
4. Выполнение соответствующих операторов для обработки модели.

Архитектура фрагментных шейдеров для нейросетевых слоев

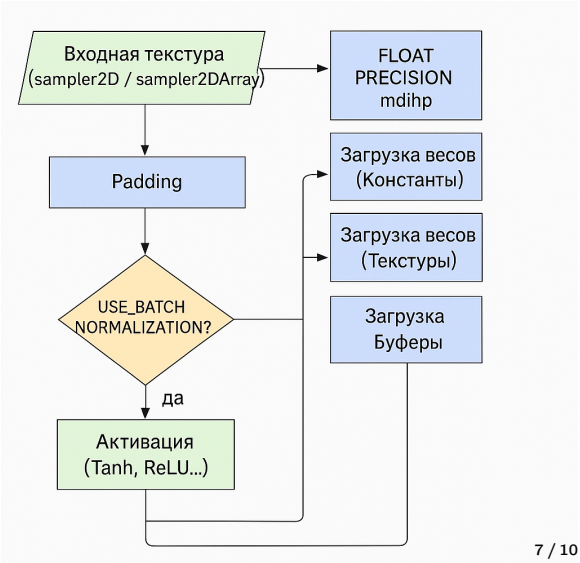
Для универсальности фрагментные шейдеры будут создаваться непосредственно после извлечения данных из файла модели, при этом будут использоваться шаблоны (для каждого слоя они уникальны), написанные заранее.
Ниже приведена таблица, в которой перечислены слои модели ESPCN и соответствующие им количества render pass:

Модель ESPCN	Количество render pass
Layer 1: Conv2D output: $n \times n \times 16$	Layer 1 output: 4 render passes
Layer 2: Conv2D output: $n \times n \times 16$	Layer 2 output: 4 render passes
Layer 3: Conv2D output: $n \times n \times 4$	Layer 3 output: 1 render pass
Layer 4: Subpixel output: $2n \times 2n \times 1$	Layer 4 output: 1 render pass

Реализация слоя Conv2D

Главные аспекты шейдера:

- Поддерживаемые режимы pading: clamped, const, replicate, checkboard и remove_zero.
- Три режима работы с весами: constants (загружаются на этапе создания шейдера), textures (передаются как текстуры) и buffers (передаются как буфферы).
- Реализована встроенная batch нормализация (если слой ее поддерживает).
- Присутствует режим MRT.



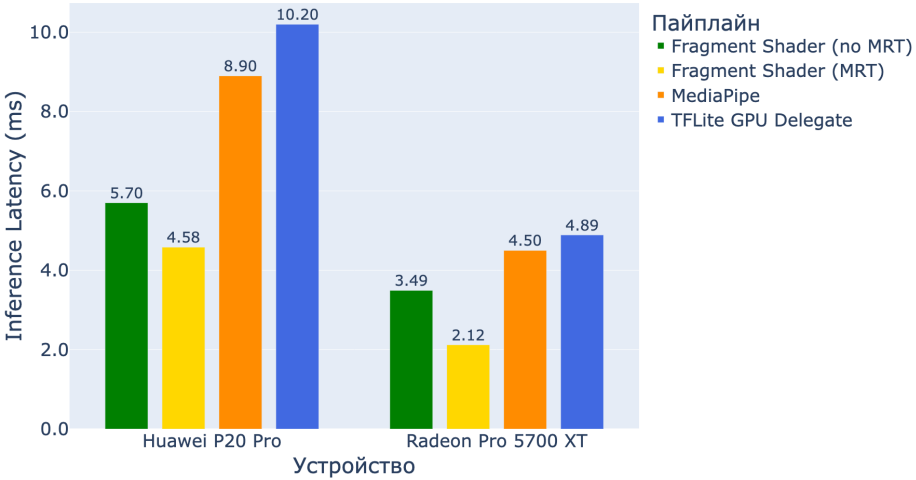
Реализация слоя Activation

```
56  #if PLANE_COUNT > 2
57  #ifdef INPUT_TEXTURE_2D
58  |   s = texelFetch(inputTextures0, ivec2(outLoc), 0);
59  #else
60  |   s = texelFetch(inputTextures0, ivec3(outLoc, layer3), 0);
61  #endif
62  #endif
63  |
64  |   _PLACEHOLDER_ACTIVATION_
65  |   o_pixel = s;
66  #if PLANE_COUNT > 1
67  |   o_pixel1 = s;
68  #endif
69  #if PLANE_COUNT > 2
70  |   o_pixel2 = s;
71  #endif
72  #if PLANE_COUNT > 3
73  |   o_pixel3 = s;
74  #endif
75  }
```

Реализованные функции активации:

- **ReLU:** s = max(s, vec4(0.0));
- **ReLU6:** s = clamp(s, vec4(0.0), vec4(6.0));
- **Tanh:** s = tanh(s);
- **Sigmoid:** s = vec4(1.0f) / (vec4(1.0f) + exp(-s));
- **LeakyReLU (с параметром alpha):** s = max(s, (s * vec4(alpha)));
- **SiLU (Swish):** s = s * vec4(1.0f) / (vec4(1.0f) + exp(-s));

Сравнительное тестирование реализованного пайплайна



Выводы

1. Произведен анализ существующих реализаций GPU пайплайнов и их особенностей. Показано, что главном проблемой является отсутствие фреймворков с поддержкой аппаратных структур данных GPU для входа.
2. Был реализован пайплайн на языке программирования C++, в основе которого лежит движок OpenGL.
3. Разработана архитектура обработки данных на шейдерах (структура вершинного и фрагментного шейдера). Также реализованы шаблоны фрагментных шейдеров для слоев Conv2D, Activation и Subpixel.
4. Проанализированы результаты сравнения построенного пайплайна с фреймворками MediaPipe и TFLite с GPU delegate. Среднее суммарное время обработки кадра нейросетью в реализованном пайплайне более, чем в 2 меньше, чем с помощью универсальных библиотек, что доказывает эффективность выбранного подхода.