МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ
Кафедра дискретной математики и алгоритмики

Анализ эффективности нейросетевых вычислений с учетом аппаратных возможностей платформ

Курсовая работа

Бинцаровского Леонида Петровича
студента 3 курса
специальность «информатика»

Научный руководитель:
старший преподаватель
Д. И. Пиршток

Минск, 2024

Реферат

Курсовая работа, 36 стр., 2 иллюстр., 4 источника.

Ключевые слова: Visual Studio; C++; Onnxruntime; DirectML; QNN; oneDNN; openVINO; DefaultCPU; инференс.

Объекты исследования — анализ эффективности нейросетевых вычислений с учетом аппаратных возможностей платформ.

Цель исследования — реализация классов для инференсов в среде разработки Visual Studio с целью сравнения эффективности нейросетевых вычеслений с учетом аппаратных возможностей платформ.

Методы исследования — системный подход, изучение соответствующей литературы и электронных источников, постановка задачи и её решение.

В результате исследования были реализованы классы на языке программирования c++ для инференса нейросетей в среде разработки Visual Studio. Проведены сравнения эффективности нейросетевых вычеслений с учетом аппаратных возможностей платформ.

Области применения — инференс нейроных сетей.

Оглавление

Введение

Нейронные сети стали неотъемлемой частью современного мира, найдя применение в самых разнообразных областях, начиная от компьютерного зрения и обработки естественного языка, и заканчивая медицинской диагностикой и финансовым анализом. С развитием глубокого обучения и доступностью вычислительных ресурсов, нейронные сети становятся все более распространенными и мощными инструментами для решения сложных задач, которые ранее казались невозможными или непрактичными для автоматизации.

Всвязи с постоянным увеличением объема данных и сложности задач, стоящих перед нейронными сетями, вопрос эффективности и скорости их работы становится все более актуальным. Одним из ключевых аспектов, определяющих производительность нейросетевых моделей, является скорость инференса - процесса получения выводов от модели на основе входных данных.

ГЛАВА 1
ПОСТАНОВКА ЗАДАЧИ

На данный момент существует огромное количество моделей предназначенных под всевозможные цели: улучшение качества изображения, перевод изображения из черно-белого в цветное, распознование речи и перевод ее в текст, сегментация объектов, перевод текста на разные языки и т.д. В рамках работы будут протестированы модели для semantic segmentation (разделение изображения на два класса: 1 – объект, 0 – фон), такие как mediapipe, selfie_segmentation, SINet_Softmax_simple, rvm_mobilentv3_fp32 и pphumanseg_fp32. Все вышеперечисленные модели имеют разную архитектуру, что в свою очередь влияет на скорость инференса. Также скорость инференса очень сильно зависит от используемых инструментов. Например, можно разворачивать модели только на базовых фреймворках — PyTorch, TensorFlow, PaddlePaddle, TFLite, TorchScript — и получать не самые лучшие результаты. Такие инструменты больше подходят для обучения и тестового инференса моделей, когда нет потребности в высокой скорости ML-сервиса. Для эффективной работы нужно использовать более мощные фреймворки, такие как ONNX Runtime, OpenVINO или TVM.

Итак, сформулируем задачу анализа эффективности нейросетевых вычислений с учетом аппаратных возможностей платформ, которая будет изучаться в этой работе. Необходимо реализовать классы для инференса выбранной модели. На вход будет предоставлено изображение, на выходе - получена маска этого изображения и замерено время инференса. Затем, основываясь на полученных данных, будет проведена сравнительная характеристика провайдеров, которые были задействованы для инференса.

ГЛАВА 2
ОБЗОР ФРЕЙМВОРКА ONNX RUNTIME

2.1 Знакомство с ONNX Runtime

ONNX (Open Neural Network Exchange) - библиотека, реализующая хранение и обработку нейросетей, изначально называлась Toffe и разрабатывалась командой Pytorch (Meta). В 2017 проект был переименован в ONNX, и с тех пор поддерживается совместно Microsoft, Meta и другими большими компаниями.

Изначально ONNX задумывался как открытый формат представления нейросети, который свяжет представление моделей в разных фреймворках.

В первом релизе речь шла о Caffe2, PyTorch и CNTK. Сейчас многие крупные фреймворки стараются его поддерживать:

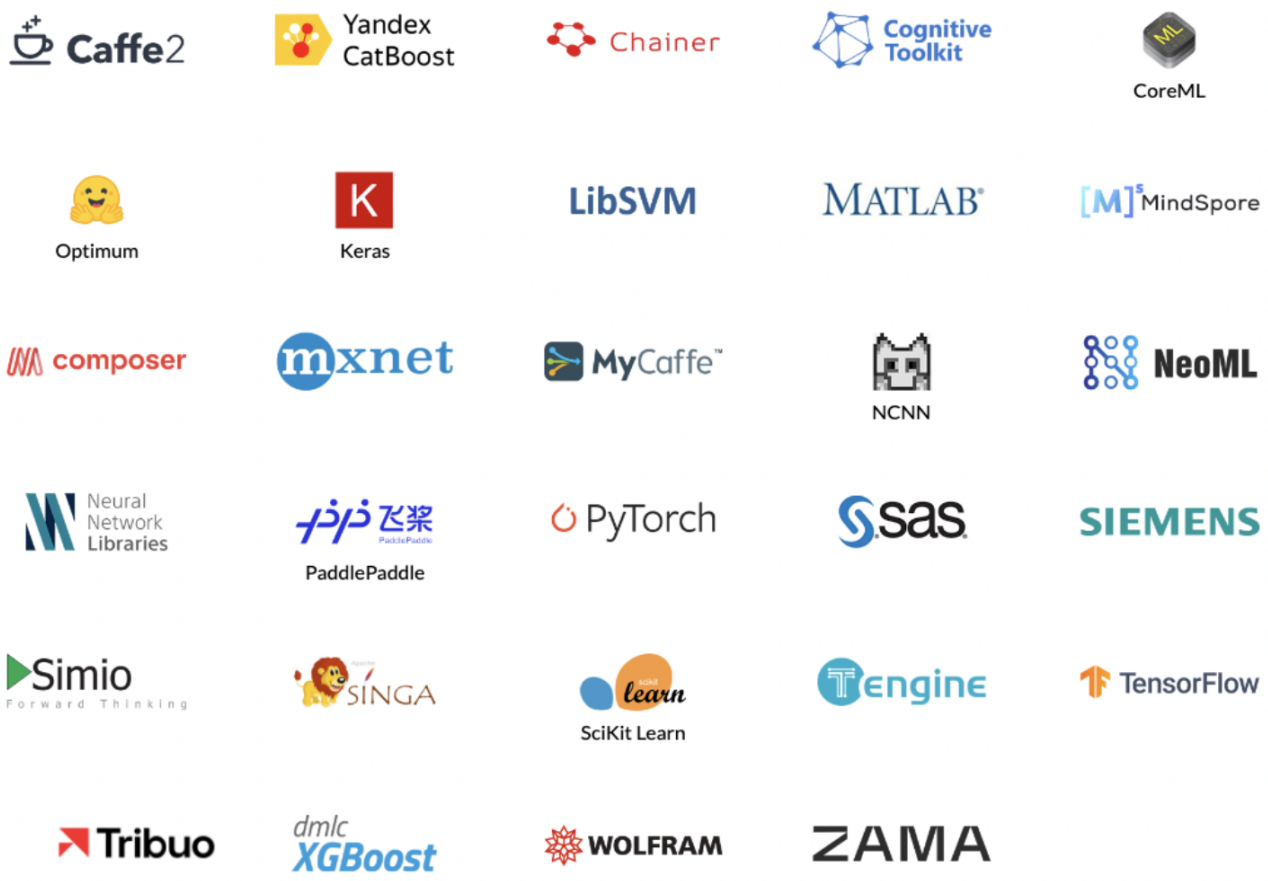

Рисунок 2.1 — Фреймворки, поддерживаемые ONNX Runtime

Внутри у ONNX есть свое промежуточное представление. Оно реализовано с помощью protocol buffers и служит промежуточным звеном для конвертации между фреймворками. Все компоненты ONNX, включая промежуточное представление, версионируются возрастающим числом или согласно SemVer.

Функции и операторы версионируются отдельно, а версия ONNX фиксируется в сериализуемой в protobuf модели.

Это дает гарантию, что представление любой модели, даже спустя время, будет конвертироваться между ONNX и другими форматами. Иначе было бы легко запутаться при обновлении зависимостей в инфраструктуре.

Через некоторое время появился ONNX Runtime - фреймворк для инференса в формате ONNX, который реализовал различные оптимизаторы поверх ONNX формата. Среди них ONNX Runtime DirectML, ONNX Runtime CoreML, ONNX Runtime TensorRT, ONNX Runtime CUDA, ONNX Runtime oneDNN, ONNX Runtime OpenVINO, ONNX Runtime QNN и т.д.

2.2 Конвертация моделей в формат ONNX

Для использования вышеперечисленных моделей (mediapipe.tflite, selfie_segmentation.tflite, pphumaseg_fp32.tflite, SINet_softmax_simple.tflite, rvm_mobilnetv3_fp32.tflite), их необходимо перевести из формата tflite в формат onnx. Для этого воспользуемся скриптом на python:

```python
import tf2onnx
import onnx

# Convert tflite model to ONNX
onnx_model, _ =
        tf2onnx.convert
            .from_tflite(model_path="pathToTfliteModel.tflite", opset=13)

# Save the ONNX model
onnx.save(onnx_model, "pathToSaveOnnxModel.onnx")
```

Листинг 2.1: Скрипт для конвертации tflite модели в onnx

В данном скрипте используется фреймворк tf2onnx, предоставленный на официальном сайте ONNX Runtime. При использовании рукописного преобразователя могут возникнуть проблемы, так как не все слои конвертируются в ONNX формат. Часть может преобразоваться совсем не так как хотелось бы, а часть после конвертации может начать некорректно вычислять значения.

Для удобства дальнейшего использования готового инференса, модели были загружены в h файлы. При использовании такого подхода, готовое приложение будет статически зависеть от модели и ее не нужно будет прикреплять к exe модулю. Для записи модели в массив был написан python скрипт:

```python
import binascii

def convert_to_c_array(bytes) -> str:
  hexstr = binascii.hexlify(bytes).decode("UTF-8")
  hexstr = hexstr.upper()
  array = ["0x" + hexstr[i:i + 2] for i in range(0, len(hexstr), 2)]
  array = [array[i:i+10] for i in range(0, len(array), 10)]
  return ",\n  ".join([", ".join(e) for e in array])
```

```
9
10 onnx_binary = open("modelName.onnx", 'rb').read()
11 ascii_bytes = convert_to_c_array(onnx_binary)
12 c_file = "const unsigned char modelName_model_onnx[] = {\n  "
13     + ascii_bytes
14     + "\n};\nunsigned int modelName_model_onnx_len = "
15     + str(len(tflite_binary)) + ";
16 open("modelName.h", "w").write(c_file)
```

Листинг 2.2: Скрипт для записи модели в массив char

ГЛАВА 3
РЕАЛИЗАЦИЯ КРОСС-ПЛАТФОРМЕННОЙ ЧАСТИ ПРИЛОЖЕНИЯ ДЛЯ ЗАМЕРА СКОРОСТИ ВЫЧИСЛЕНИЙ НЕЙРОСЕТЕЙ НА ЯЗЫКЕ C++

3.1 Структура ORTModelData

Структура ORTModelData представляет собой контейнер, содержащий данные, необходимые для работы с моделью, которая загружена с использованием библиотеки ONNX Runtime (ORT).

```
#ifndef ORTMODELDATA_H
#define ORTMODELDATA_H

#include <onnxruntime_cxx_api.h>

struct ORTModelData {
    std::unique_ptr<Ort::Session> session;
    std::unique_ptr<Ort::Env> env;
    std::vector<Ort::AllocatedStringPtr> inputNames;
    std::vector<Ort::AllocatedStringPtr> outputNames;
    std::vector<Ort::Value> inputTensor;
    std::vector<Ort::Value> outputTensor;
    std::vector<std::vector<int64_t>> inputDims;
    std::vector<std::vector<int64_t>> outputDims;
    td::vector<std::vector<float>> inputTensorValues;
    std::vector<std::vector<float>> outputTensorValues;
};

#endif /* ORTMODELDATA_H */
```

Листинг 3.1: Структура ORTModelData

В структуре ORTModelData определены поля:

- **session** — это уникальный указатель на объект сеанса Ort::Session, который является основным для выполнения модели. Сеанс содержит информацию о модели, включая ее граф вычислений, параметры и контекст выполнения. Он используется для загрузки и выполнения модели;

- **env** — уникальный указатель на объект окружения Ort::Env. Ort::Env представляет собой контекст выполнения ONNX Runtime, содержит ресурсы и параметры, необходимые для работы с библиотекой ONNX Runtime;

- **inputNames** — вектор указателей на строки Ort::AllocatedStringPtr, содержащих имена входных тензоров модели. Эти имена используются для связывания входных данных с соответствующими тензорами модели во время выполнения инференса;

- **outputNames** — подобно inputNames, это вектор указателей на строки Ort::AllocatedStringPtr, содержащих имена выходных тензоров модели. Используются для извлечения выходных данных из соответствующих тензоров модели после выполнения инференса;

- **inputTensor** — вектор объектов тензора Ort::Value, представляющих входные данные модели. Эти объекты тензора содержат входные данные, которые будут переданы в модель для инференса;

- **outputTensor** — вектор объектов тензора Ort::Value, представляющих выходные данные модели. После выполнения модели, результаты будут содержаться в этих объектах тензора;

- **inputDims** — вектор векторов целых чисел (int64_t), содержащих размерности входных тензоров. Эти размерности определяют форму (shape) входных тензоров модели;

- **outputDims** — аналогично inputDims, это вектор векторов целых чисел (int_64t), содержащих размерности выходных тензоров модели;

- **inputTensorValues** — вектор векторов чисел с плавающей запятой (float), содержащих значения входных тензоров модели. Эти значения представляют входные данные, которые будут переданы в модель для инференса;

- **outputTensorValues** — вектор векторов чисел с плавающей запятой (float), содержащих значения выходных тензоров модели. Эти значения представляют собой результаты инференса модели;

3.2 Классы Model и ModelBCHW

Для реализации класса Model была создана вспомогательная функция **vectorProduct**. Она принимает на вход вектор любого типа и возвращает произведение всех положительных элементов, так как 0 или -1 обычно обозначают "None".

```
template<typename T> T vectorProduct(const std::vector<T> &v)
{
  T product = 1;
  for (auto &i : v) {
    if (i > 0) { product *= i;}
  }
```

```
    return product;
}
```

Листинг 3.2: Вспомогательная функция vectorProduct

Далее был реализован базовый класс для всех моделей - Model. Учитывалось, что все модели имеют один вход и один выход. На вход модели поступает 4D тензор формы (1, H, W, C), где H и W - высота и ширина входного изображения, C - количество цветовых каналов. Обычно цветные изображения используют 3 канала: красного, зеленого и синего цветов. Входные данные представлены в формате BGR и их диапозон - [0, 255]. Цель данного класса состоит в том, чтобы предоставить общий интерфейс для работы с моделями машинного обучения, которые обрабатывают изображения.

```
#ifndef MODEL_H
#define MODEL_H

#include <onnxruntime_cxx_api.h>

#ifdef _WIN32
#include <wchar.h>
#endif

#include <opencv2/imgproc.hpp>
#include <algorithm>

#define UNUSED_PARAMETER(param) (void)param

class Model {
private:

public:
    Model() {};
    virtual ~Model() {};

    const char *name;

    virtual void populateInputOutputNames(
            const std::unique_ptr<Ort::Session> &session,
            std::vector<Ort::AllocatedStringPtr> &inputNames,
            std::vector<Ort::AllocatedStringPtr> &outputNames);

    virtual bool populateInputOutputShapes(
            const std::unique_ptr<Ort::Session> &session,
            std::vector<std::vector<int64_t>> &inputDims,
            std::vector<std::vector<int64_t>> &outputDims);

    virtual void allocateTensorBuffers(
            const std::vector<std::vector<int64_t>> &inputDims,
            const std::vector<std::vector<int64_t>> &outputDims,
            std::vector<std::vector<float>> &outputTensorValues,
            std::vector<std::vector<float>> &inputTensorValues,
            std::vector<Ort::Value> &inputTensor,
            std::vector<Ort::Value> &outputTensor);

    virtual void getNetworkInputSize(
            const std::vector<std::vector<int64_t>> &inputDims,
            uint32_t &inputWidth,
```

```
45          uint32_t &inputHeight)
46    {
47      // BHWC
48      inputWidth = (int)inputDims[0][2];
49      inputHeight = (int)inputDims[0][1];
50    }
51
52    virtual void prepareInputToNetwork(
53          cv::Mat &resizedImage,
54          cv::Mat &preprocessedImage)
55    {
56      preprocessedImage = resizedImage / 255.0;
57    }
58
59    virtual void postprocessOutput(cv::Mat &output) { UNUSED_PARAMETER(output); }
60
61    virtual void loadInputToTensor(
62          const cv::Mat &preprocessedImage,
63          uint32_t inputWidth,
64          uint32_t inputHeight,
65          std::vector<std::vector<float>> &inputTensorValues)
66    {
67      preprocessedImage.copyTo(
68          cv::Mat(inputHeight, inputWidth,
69                  CV_32FC3, &(inputTensorValues[0][0])));
70    }
71
72    virtual cv::Mat getNetworkOutput(
73          const std::vector<std::vector<int64_t>> &outputDims,
74          std::vector<std::vector<float>> &outputTensorValues)
75    {
76      // BHWC
77      uint32_t outputWidth = (int)outputDims[0].at(2);
78      uint32_t outputHeight = (int)outputDims[0].at(1);
79      int32_t outputChannels = CV_MAKE_TYPE(CV_32F, (int)outputDims[0].at(3));
80
81      return cv::Mat(outputHeight, outputWidth,
82                     outputChannels, outputTensorValues[0].data());
83    }
84
85    virtual void assignOutputToInput(std::vector<std::vector<float>> &,
86                                     std::vector<std::vector<float>> &) {}
87
88    virtual void runNetworkInference(
89          const std::unique_ptr<Ort::Session> &session,
90          const std::vector<Ort::AllocatedStringPtr> &inputNames,
91          const std::vector<Ort::AllocatedStringPtr> &outputNames,
92          const std::vector<Ort::Value> &inputTensor,
93          std::vector<Ort::Value> &outputTensor);
94  };
95
96  #endif
```

Листинг 3.3: Класс Model

В данном классе реализованны базовые методы для инференса. В последствии некоторые методы будут изменены в классах конкретных моделей.

Метод **populateInputOutputNames** заполняет векторы inputNames и outputNames именами входных и выходных тензоров соответственно. В даль-

нейшем эти имена используются для запуска инференса модели.

```
virtual void Model::populateInputOutputNames(
        const std::unique_ptr<Ort::Session> &session,
        std::vector<Ort::AllocatedStringPtr> &inputNames,
        std::vector<Ort::AllocatedStringPtr> &outputNames)
  {
    Ort::AllocatorWithDefaultOptions allocator;

    inputNames.clear();
    outputNames.clear();
    inputNames.push_back(session->GetInputNameAllocated(0, allocator));
    outputNames.push_back(session->GetOutputNameAllocated(0, allocator));
  }
```

Листинг 3.4: Метод populateInputOutputNames

Метод **populateInputOutputShapes** заполняет векторы inputDims и outputDims размерностями входных и выходных тензоров соответственно. Он также выполняет проверку и корректировку размерностей, устанавливая любые отрицательные значения в размерности тензоров равными 1.

```
virtual bool Model::populateInputOutputShapes(
        const std::unique_ptr<Ort::Session> &session,
        std::vector<std::vector<int64_t>> &inputDims,
        std::vector<std::vector<int64_t>> &outputDims)
  {
    // Assuming model only has one input and one output image
    inputDims.clear();
    outputDims.clear();
    inputDims.push_back(std::vector<int64_t>());
    outputDims.push_back(std::vector<int64_t>());

    // Get output shape
    const Ort::TypeInfo outputTypeInfo = session->GetOutputTypeInfo(0);
    const auto outputTensorInfo = outputTypeInfo.GetTensorTypeAndShapeInfo();
    outputDims[0] = outputTensorInfo.GetShape();

    // fix any -1 values in outputDims to 1
    for (auto &i : outputDims[0]) {
      if (i == -1) {
        i = 1;
      }
    }

    // Get input shape
    const Ort::TypeInfo inputTypeInfo = session->GetInputTypeInfo(0);
    const auto inputTensorInfo = inputTypeInfo.GetTensorTypeAndShapeInfo();
    inputDims[0] = inputTensorInfo.GetShape();

    // fix any -1 values in inputDims to 1
    for (auto &i : inputDims[0]) {
      if (i == -1) {
        i = 1;
      }
    }

    if (inputDims[0].size() < 3 || outputDims[0].size() < 3) {
      std::cerr << "Input or output tensor dims are < 3. input = "
            << (int)inputDims.size() << ", output = "
```

```
39              << (int)outputDims.size() << "\n";
40        return false;
41      }
42
43      return true;
44    }
```

Листинг 3.5: Метод populateInputOutputShapes

Метод **allocateTensorBuffers** выделяет память для буферов входных и выходных тензоров и создает для них объекты Ort::Value.

```
1  virtual void Model::allocateTensorBuffers(
2          const std::vector<std::vector<int64_t>> &inputDims,
3          const std::vector<std::vector<int64_t>> &outputDims,
4          std::vector<std::vector<float>> &outputTensorValues,
5          std::vector<std::vector<float>> &inputTensorValues,
6          std::vector<Ort::Value> &inputTensor,
7          std::vector<Ort::Value> &outputTensor)
8    {
9      outputTensorValues.clear();
10     outputTensor.clear();
11     inputTensorValues.clear();
12     inputTensor.clear();
13
14     Ort::MemoryInfo memoryInfo =
15         Ort::MemoryInfo::CreateCpu(
16             OrtAllocatorType::OrtDeviceAllocator,
17             OrtMemType::OrtMemTypeDefault);
18
19     for (size_t i = 0; i < inputDims.size(); i++) {
20       inputTensorValues.push_back(
21             std::vector<float>(vectorProduct(inputDims[i]), 0.0f));
22       inputTensor.push_back(
23             Ort::Value::CreateTensor<float>(
24                 memoryInfo, inputTensorValues[i].data(),
25                 inputTensorValues[i].size(), inputDims[i].data(),
26                 inputDims[i].size()));
27     }
28
29     for (size_t i = 0; i < outputDims.size(); i++) {
30       outputTensorValues.push_back(
31         std::vector<float>(vectorProduct(outputDims[i]), 0.0f));
32       outputTensor.push_back(
33             Ort::Value::CreateTensor<float>(
34                 memoryInfo, outputTensorValues[i].data(),
35                 outputTensorValues[i].size(), outputDims[i].data(),
36                 outputDims[i].size()));
37     }
38   }
```

Листинг 3.6: Метод allocateTensorBuffers

Метод **getNetworkInputSize** получает размер входного изображения (ширину и высоту) из размерностей входного тензора.

Метод **prepareInputToNetwork** выполняет предварительную обработку входного изображения перед передачей его в сеть. В данном случае он нормализует значения пикселей изображения в диапазоне от 0 до 1.

Метод **postprocessOutput** выполняет постобработку выходных данных сети. Здесь он не выполняет никакой обработки, но будет переопределен в классах конкретных моделей для выполнения соотвествующей постобработки.

Метод **loadInputToTensor** загружает предобработанное изображение во входной тензор модели.

Метод **getNetworkOutput** получает выходные данные сети и возвращает их в виде объекта cv::Mat.

Метод **assignOutputToInput** присваивает выходные данные входному тензору для обратной передачи в сеть. В данном случае он не выполняет никакой обработки, но будет переопределен в классах конкретных моделей для присваивания соответсвующих данных.

Метод **runNetworkInference** является ключевым для запуска модели на входных данных и получения выходных данных. Он позволяет использовать один и тот же общий интерфейс для выполнения инференса различных моделей и управления входными и выходными данными.

```
virtual void Model::runNetworkInference(
        const std::unique_ptr<Ort::Session> &session,
        const std::vector<Ort::AllocatedStringPtr> &inputNames,
        const std::vector<Ort::AllocatedStringPtr> &outputNames,
        const std::vector<Ort::Value> &inputTensor,
        std::vector<Ort::Value> &outputTensor)
  {
    if (inputNames.size() == 0 || outputNames.size() == 0
        || inputTensor.size() == 0 || outputTensor.size() == 0) {
      std::cerr
          << "Error! Skip network inference. Inputs or outputs are null.\n";
      return;
    }

    std::vector<const char *> rawInputNames;
    for (auto &inputName : inputNames) {
      rawInputNames.push_back(inputName.get());
    }

    std::vector<const char *> rawOutputNames;
    for (auto &outputName : outputNames) {
      rawOutputNames.push_back(outputName.get());
    }

    session->Run(Ort::RunOptions{nullptr}, rawInputNames.data(),
                 inputTensor.data(), inputNames.size(),
                 rawOutputNames.data(), outputTensor.data(),
                 outputNames.size());
  }
```

Листинг 3.7: Метод runNetworkInference

Для реализации класса ModelBCHW были реализованы вспомогательные функции. Функция **hwc_to_chw** преобразует входное изображение из формата HWC (высота x ширина x количество каналов) в формат CHW (количество каналов x высота x ширина).

```
static void hwc_to_chw(cv::InputArray src, cv::OutputArray dst)
{
  std::vector<cv::Mat> channels;
  cv::split(src, channels);
  for (auto &img : channels) { img = img.reshape(1, 1);}
  cv::hconcat(channels, dst);
}
```

Листинг 3.8: Вспомогательная функция hwc_to_chw

Функция **chw_to_hwc_32f** преобразует изображение из формата CHW (количество каналов x высота x ширина) в формат HWC (высота x ширина x количество каналов) для изображений с плавающей точкой (float32).

```
static void chw_to_hwc_32f(cv::InputArray src, cv::OutputArray dst)
{
  const cv::Mat srcMat = src.getMat();
  const int channels = srcMat.channels();
  const int height = srcMat.rows;
  const int width = srcMat.cols;
  const int dtype = srcMat.type();
  assert(dtype == CV_32F);
  const int channelStride = height * width;

  cv::Mat flatMat = srcMat.reshape(1, 1);

  std::vector<cv::Mat> channelsVec(channels);
  for (int i = 0; i < channels; i++) {
    channelsVec[i] =
      cv::Mat(height, width, CV_MAKE_TYPE(dtype, 1),
            flatMat.ptr<float>(0) + i * channelStride);
  }

  cv::merge(channelsVec, dst);
}
```

Листинг 3.9: Вспомогательная функция chw_to_hwc_32f

Далее был реализован сам класс ModelBCHW. Он является наследником базового класса Model и представляет собой специализированную версию модели, которая работает с данными в формате BCHW (количество каналов x высота x ширина).

```
class ModelBCHW : public Model {
public:
  ModelBCHW() {}
  ~ModelBCHW() {}
  virtual void prepareInputToNetwork(
        cv::Mat &resizedImage,
        cv::Mat &preprocessedImage)
  {
    resizedImage = resizedImage / 255.0;
    hwc_to_chw(resizedImage, preprocessedImage);
  }

  virtual void postprocessOutput(cv::Mat &output)
  {
    cv::Mat outputTransposed;
    chw_to_hwc_32f(output, outputTransposed);
```

```
17      outputTransposed.copyTo(output);
18    }
19
20    virtual void getNetworkInputSize(
21          const std::vector<std::vector<int64_t>> &inputDims,
22          uint32_t &inputWidth,
23          uint32_t &inputHeight)
24    {
25      // BCHW
26      inputWidth = (int)inputDims[0][3];
27      inputHeight = (int)inputDims[0][2];
28    }
29
30    virtual cv::Mat getNetworkOutput(
31          const std::vector<std::vector<int64_t>> &outputDims,
32          std::vector<std::vector<float>> &outputTensorValues)
33    {
34      // BCHW
35      uint32_t outputWidth = (int)outputDims[0].at(3);
36      uint32_t outputHeight = (int)outputDims[0].at(2);
37      int32_t outputChannels = CV_MAKE_TYPE(CV_32F, (int)outputDims[0].at(1));
38
39      return cv::Mat(outputHeight, outputWidth,
40              outputChannels, outputTensorValues[0].data());
41    }
42
43    virtual void loadInputToTensor(
44          const cv::Mat &preprocessedImage,
45          uint32_t, uint32_t,
46          std::vector<std::vector<float>> &inputTensorValues)
47    {
48      inputTensorValues[0].assign(
49              preprocessedImage.begin<float>(),
50              preprocessedImage.end<float>());
51    }
52  };
```

Листинг 3.10: Метод runNetworkInference

Метод **prepareInputToNetwork** выполняет предварительную обработку входного изображения перед передачей его в сеть. Он нормализует значения пикселей изображения в диапазоне от 0 до 1 и затем преобразует изображение из формата HWC в формат BCHW с помощью функции hwc_to_chw.

Метод **postprocessOutput** выполняет постобработку выходных данных сети. Он транспонирует изображение из формата BCHW обратно в формат HWC с помощью функции chw_to_hwc_32f, чтобы его можно было корректно отобразить или использовать дальше.

Метод **getNetworkInputSize** получает размер входного изображения (ширину и высоту) из размерностей входного тензора в формате BCHW.

Метод **getNetworkOutput** получает выходные данные сети и возвращает их в виде объекта cv::Mat в формате HWC.

Метод **loadInputToTensor** загружает предобработанное изображение во входной тензор модели. Он копирует значения пикселей изображения в одномерный вектор входного тензора.

3.3 Структуа FilterData

Эта структура предназначена для хранения основных данных, необходимых для фильтров ORT (ONNX Runtime). FilterData наследует все поля, определенные в структуре ORTModelData, так как она должна иметь доступ к базовым данным модели ORT.

```
#ifndef FILTERDATA_H
#define FILTERDATA_H

#include "models/Model.h"
#include "OrtUtils/ORTModelData.h"

struct FilterData : public ORTModelData {
  std::string useGPU;
  uint32_t numThreads;
  std::string modelSelection;
  std::unique_ptr<Model> model;

  cv::Mat inputRGB;

  std::mutex inputRGBLock;

  const unsigned char* modelInfo = nullptr;
    unsigned int modelSize = 0;
};

#endif /* FILTERDATA_H */
```

Листинг 3.11: Структура FilterData

Непосредственно в структуре FilterData определены поля:

- **useGPU** — строка, содержащая информацию о том, какой провайдер использовать. Может быть установлен флаг CPU, DirectML, QNN, oneDNN или OpenVINO;
- **numThreads** — переменная, определяющая количество потоков, используемых для выполнения инференса;
- **modelSelection** — строка, содержащая название модели, которая будет использоваться для обработки данных;
- **model** — Уникальный указатель на объект модели, который будет использоваться для выполнения инференса;
- **inputBGRA** — матрица изображения (cv::Mat) в формате BGRA (синий, зеленый, красный, альфа), которая будет подаваться на вход модели;
- **inputBGRALock** — объект мьютекса для безопасного доступа к входным данным;

- **modelInfo** и **modelSize** — указатель на информацию о модели и ее размер, которые будут использованы для хранения информации о модели;

3.4 Класс ModelMediapipe

Класс **ModelMediapipe** является производным от базового класса Model. Он предназначен для работы непосредственно с моделью mediapipe. Так как модель mediapipe возвращает тензор размерности (B, H, W, C) и изображение с двумя каналами (0 - маска фона, 1 - маска человека), необходимо переопределить методы getNetworkOutput и postprocessOutput.

```
#ifndef MODELMEDIAPIPE_H
#define MODELMEDIAPIPE_H

#include "Model.h"

class ModelMediaPipe : public Model {
public:
  ModelMediaPipe() {}
  ~ModelMediaPipe() {}

  virtual cv::Mat getNetworkOutput(
        const std::vector<std::vector<int64_t>> &outputDims,
        std::vector<std::vector<float>> &outputTensorValues)
  {
    uint32_t outputWidth = (int)outputDims[0].at(2);
    uint32_t outputHeight = (int)outputDims[0].at(1);
    int32_t outputChannels = CV_32FC2;

    return cv::Mat(outputHeight, outputWidth,
            outputChannels, outputTensorValues[0].data());
  }

  virtual void postprocessOutput(cv::Mat &outputImage)
  {
    // take 2nd channel
    std::vector<cv::Mat> outputImageSplit;
    cv::split(outputImage, outputImageSplit);
    outputImage = outputImageSplit[1];
  }
};

#endif // MODELMEDIAPIPE_H
```

Листинг 3.12: Класс ModelMediapipe

3.5 Класс ModelPPHumanseg

Класс **ModelPPHumanseg** является производным от класса ModelBCHW. Он предназначен для работы с моделью pphumanseg_fp32.

Так как модель pphumanseg_fp32 принимает на вход тензор BCHW, возвращает тензор размерности (B, H, W, C) и изображение с двумя каналами (0 - маска фона, 1 - маска человека), необходимо переопределить методы prepareInputToNetwork, getNetworkOutput и postprocessOutput.

```
#ifndef MODELPPHUMANSEG_H
#define MODELPPHUMANSEG_H

#include "Model.h"

class ModelPPHumanSeg : public ModelBCHW {
public:
  ModelPPHumanSeg() {}
  ~ModelPPHumanSeg() {}

  virtual void prepareInputToNetwork(
        cv::Mat &resizedImage, cv::Mat &preprocessedImage)
  {
    resizedImage =
        (resizedImage / 256.0 - cv::Scalar(0.5, 0.5, 0.5))
        / cv::Scalar(0.5, 0.5, 0.5);

    hwc_to_chw(resizedImage, preprocessedImage);
  }

  virtual cv::Mat getNetworkOutput(
        const std::vector<std::vector<int64_t>> &outputDims,
        std::vector<std::vector<float>> &outputTensorValues)
  {
    uint32_t outputWidth = (int)outputDims[0].at(2);
    uint32_t outputHeight = (int)outputDims[0].at(1);
    int32_t outputChannels = CV_32FC2;

    return cv::Mat(outputHeight, outputWidth,
            outputChannels, outputTensorValues[0].data());
  }

  virtual void postprocessOutput(cv::Mat &outputImage)
  {
    std::vector<cv::Mat> outputImageSplit;
    cv::split(outputImage, outputImageSplit);
    cv::normalize(outputImageSplit[1], outputImage, 1.0, 0.0, cv::NORM_MINMAX);
  }
};

#endif // MODELPPHUMANSEG_H
```

Листинг 3.13: Класс ModelPPHumanseg

3.6 Класс ModelSelfie

Класс **ModelSelfie** является производным от класса Model. Он предназначен для работы с моделью selfie_segmentation. Для этого необходимо переопределить лишь метод postprocessOutput.

```
#ifndef MODELSELFIE_H
```

```
#define MODELSELFIE_H

#include "Model.h"

class ModelSelfie : public Model {
public:
  ModelSelfie() {}
  ~ModelSelfie() {}

  virtual void postprocessOutput(cv::Mat &outputImage)
  {
    cv::normalize(outputImage, outputImage, 1.0, 0.0, cv::NORM_MINMAX);
  }
};

#endif // MODELSELFIE_H
```

Листинг 3.14: Класс ModelSelfie

3.7 Класс ModelSINET

Класс **ModelSINET** является производным от класса ModelBCHW. Он предназначен для работы с моделью SINet_softmax_simple. Для этого необходимо переопределить лишь метод prepareInputToNetwork, так как модель SINet_softmax_simple принимает на вход тензор BCHW.

```
#ifndef MODELSINET_H
#define MODELSINET_H

#include "Model.h"

class ModelSINET : public ModelBCHW {
  public:
  ModelSINET(/* args */) {}
  ~ModelSINET() {}

  virtual void prepareInputToNetwork(
        cv::Mat &resizedImage, cv::Mat &preprocessedImage)
  {
    cv::subtract(resizedImage,
            cv::Scalar(102.890434, 111.25247, 126.91212), resizedImage);
    cv::multiply(resizedImage,
            cv::Scalar(1.0 / 62.93292, 1.0 / 62.82138, 1.0 / 66.355705) / 255.0,
            resizedImage);
    hwc_to_chw(resizedImage, preprocessedImage);
  }
};

#endif // MODELSINET_H
```

Листинг 3.15: Класс ModelSINET

3.8 Функции createOrtSession и runFilterModelInference

Так как модели были преобразованы в ONNX, а затем записаны в массив char, необходимо подключить нужный хедер. Для этого были установлены макросы, определяющие какая именно модель необходима для текущего инференса:

```
#include <BackgroundFilter.h>
#include <onnxruntime_cxx_api.h>
#include <cpu_provider_factory.h>

#ifdef _WIN32
#include <dml_provider_factory.h>
#include <dnnl_provider_facrtory.h>
#include <wchar.h>
#include <iostream>
#endif // _WIN32

#include "ort-session-utils.h"
#include "consts.h"
#if _MODEL_MEDIAPIPE
#include "models/models_h/mediapipe.h"
#endif
#ifdef _MODEL_SELFIE
#include "models/models_h/selfie_segmentation.h"
#endif
#ifdef _MODEL_SINET
#include "models/models_h/SINet_Softmax_simple.h"
#endif
#ifdef _MODEL_RVM
#include "models/models_h/rvm_mobilenetv3_fp32.h"
#endif
#ifdef _MODEL_PPHUMANSEG
#include "models/models_h/pphumanseg_fp32.h"
#endif
```

Листинг 3.16: Подключение необходимых хедеров для функции createOrtSession

Функция **createOrtSession** реализована, чтобы создать сеанс для выполнения инференса конкретной модели.

Изначально проверяется задана ли модель. Если нет – возвращается ошибка, и инференс останавливается. Затем задаются настройки параметров сеанса. В зависимости от текущей модели, задаются modelInfo и modelSize. Далее устанавливается текущий провайдер. Это может быть DirectML, OpenVINO, QNN или oneDNN. Если ни один из них не установлен, инференс будет происходить на CPU. Если для инференса используется CPU – устанавливается заданное число используемых ядер CPU. Далее создается новая сессия для модели с использованием указанных параметров, информации о ней и опции сеанса. Если при создании сеанса возникает ошибка – выводится сообщение об ней. Далее для созданной модели вызываются методы populateInputOutputNames, populateInputOutputShapes

и allocateTensorBuffers.

```
void createOrtSession(FilterData *tf)
{
  if (tf->model.get() == nullptr) {
    std::cerr << "Error! Model object is not initialized!\n";
    return;
  }

  Ort::SessionOptions sessionOptions;

  sessionOptions.SetGraphOptimizationLevel(
        GraphOptimizationLevel::ORT_ENABLE_ALL);
  if (tf->useGPU != USEGPU_CPU) {
    sessionOptions.DisableMemPattern();
    sessionOptions.SetExecutionMode(ExecutionMode::ORT_SEQUENTIAL);
  } else {
    sessionOptions.SetInterOpNumThreads(tf->numThreads);
    sessionOptions.SetIntraOpNumThreads(tf->numThreads);
  }

  char *modelSelection_rawPtr = (char *)tf->modelSelection.c_str();

  if (modelSelection_rawPtr == nullptr) {
    std::cerr << "Error! Unable to get model filename "
        << tf->modelSelection.c_str() << " from plugin!\n";
    return;
  }

#ifdef _MODEL_SINET
  if (tf->modelSelection == MODEL_SINET) {
      tf->modelInfo = SINet_Softmax_simple_onnx;
      tf->modelSize = SINet_Softmax_simple_onnx_len;
  }
#endif
#ifdef _MODEL_SELFIE
  if (tf->modelSelection == MODEL_SELFIE) {
      tf->modelInfo = selfie_segmentation_onnx;
      tf->modelSize = selfie_segmentation_onnx_len;
  }
#endif
#if _MODEL_MEDIAPIPE
  if (tf->modelSelection == MODEL_MEDIAPIPE) {
      tf->modelInfo = mediapipe_onnx;
      tf->modelSize = mediapipe_onnx_len;
  }
#endif
#ifdef _MODEL_RVM
  if (tf->modelSelection == MODEL_RVM) {
      tf->modelInfo = rvm_mobilenetv3_fp32_onnx;
      tf->modelSize = rvm_mobilenetv3_fp32_onnx_len;
  }
#endif
#ifdef _MODEL_PPHUMANSEG
  if (tf->modelSelection == MODEL_PPHUMANSEG) {
      tf->modelInfo = pphumanseg_fp32_onnx;
      tf->modelSize = pphumanseg_fp32_onnx_len;
  }
#endif

  try {
```

```
#ifdef _WIN32
    if (tf->useGPU == USEGPU_DML) {
      auto &api = Ort::GetApi();
      OrtDmlApi *dmlApi = nullptr;
      Ort::ThrowOnError(
        api.GetExecutionProviderApi("DML", ORT_API_VERSION,
                (const void **)&dmlApi));
      Ort::ThrowOnError(
            dmlApi->SessionOptionsAppendExecutionProvider_DML(
                    sessionOptions, 0));
    } else if (tf->useGPU == USEGPU_VINO) {
      OrtOpenVINOProviderOptions openvino;
      openvino.num_of_threads = 1;
      openvino.device_type = "CPU_FP32";
      sessionOptions.AppendExecutionProvider_OpenVINO(openvino);
      Ort::ThrowOnError(
            OrtSessionOptionsAppendExecutionProvider_CPU(
                    sessionOptions, 0));
    } else if (tf->useGPU == USEGPU_QNN) {
      Ort::Env env = Ort::Env{ ORT_LOGGING_LEVEL_ERROR, "Default" };
      std::unordered_map<std::string, std::string> qnn_options;
      qnn_options["backend_path"] = "QnnHtp.dll";
      sessionOptions.AppendExecutionProvider("QNN", qnn_options);
    } else if (tf->useGPU == USEGPU_DNN) {
      Ort::Env env = Ort::Env{ ORT_LOGGING_LEVEL_ERROR, "Default" };
      Ort::SessionOptions sf;
      bool enable_cpu_mem_arena = true;
      Ort::ThrowOnError(
        OrtSessionOptionsAppendExecutionProvider_Dnnl(
                sf, enable_cpu_mem_arena));

    }
#endif
    tf->session.reset(
            new Ort::Session(*tf->env, tf->modelInfo,
                    tf->modelSize, sessionOptions));
  } catch (const std::exception &e) {
    std::cerr << "Error! " << e.what() << "\n";
    return;
  }

  Ort::AllocatorWithDefaultOptions allocator;
  tf->model->populateInputOutputNames(
        tf->session, tf->inputNames, tf->outputNames);

  if (!tf->model->populateInputOutputShapes(
        tf->session, tf->inputDims, tf->outputDims)) {
    std::cerr << "Error! Unable to get model input and output shapes\n";
    return;
  }

  tf->model->allocateTensorBuffers(
        tf->inputDims, tf->outputDims,
        tf->outputTensorValues, tf->inputTensorValues,
        tf->inputTensor, tf->outputTensor);
}
```

Листинг 3.17: Вспомогательная функция createOrtSession

Функция **runFilterModelInference** используется для выполнения инфе-

ренса модели с применением сеанса, который был предварительно создан в createOrtSession, и подготовленных в структуре FilterData данных. Сначала идет проверка того, что сеанс ONNX Runtime и объект модели инициализированы и готовы к использованию. Если нет – функция возвращает false и выводит сообщение об ошибке. Если они инициализированы – изображение изменяется до размера, ожидаемого входными данными модели, с использованием cv::resize. Размер определяется методом getNetworkInputSize. Затем подготовленное изображение преобразуется в формат, который ожидает модель и загружается в объект тензора для передачи в сеанс выполнения. Выполняется инференс модели с помощью метода runNetworkInference. После выполнения инференса данные преобразуются в формат cv::Mat с использованием метода getNetworkOutput. При необходимоти результаты могут быть перенаправлены на входные данные для последующих итераций модели с помощью метода assignOutputToInput. Вывод модели обрабатывается с помощью метода postprocessOutput и конвертируется в формат CV_8U (8-битное изображение) для дальнейшего использования.

```
bool runFilterModelInference(
        FilterData *tf,
        const cv::Mat &imageRGB,
        cv::Mat &output)
{
  if (tf->session.get() == nullptr) {
    // Onnx runtime session is not initialized. Problem in initialization
    std::cerr << "Error! Session isn't initialized!\n";
    return false;
  }
  if (tf->model.get() == nullptr) {
    // Model object is not initialized
    std::cerr << "Error! Model isn't initialized!\n";
    return false;
  }

  // Resize to network input size
  uint32_t inputWidth, inputHeight;
  tf->model->getNetworkInputSize(
        tf->inputDims, inputWidth, inputHeight);

  cv::Mat resizedImageRGB;
  cv::resize(imageRGB, resizedImageRGB, cv::Size(inputWidth, inputHeight));

  // Prepare input to nework
  cv::Mat resizedImage, preprocessedImage;
  resizedImageRGB.convertTo(resizedImage, CV_32F);

  tf->model->prepareInputToNetwork(resizedImage, preprocessedImage);

  tf->model->loadInputToTensor(
            preprocessedImage, inputWidth,
            inputHeight, tf->inputTensorValues);

  // Run network inference
  tf->model->runNetworkInference(
        tf->session, tf->inputNames,
        tf->outputNames, tf->inputTensor, tf->outputTensor);
```

```
39
40    // Get output
41    // Map network output to cv::Mat
42    cv::Mat outputImage =
43            tf->model->getNetworkOutput(
44                        tf->outputDims,
45                        tf->outputTensorValues);
46
47    // Assign output to input in some models that have temporal information
48    tf->model->assignOutputToInput(
49            tf->outputTensorValues, tf->inputTensorValues);
50
51    // Post-process output. The image will now be in [0,1] float, BHWC format
52    tf->model->postprocessOutput(outputImage);
53
54    // Convert [0,1] float to CV_8U [0,255]
55    outputImage.convertTo(output, CV_8U, 255.0);
56    return true;
57 }
```

Листинг 3.18: Вспомогательная функция runFilterModelInference

3.9 Класс BackgroundFilter

Класс **BackgroundFilter** является связующим звеном. Именно в нем реализованы методы, в которые пользователь передает входные данные, может установить параметры для модели и сессии инференса.

```
1  #pragma once
2  #include <onnxruntime_cxx_api.h>
3
4  #include <opencv2/imgproc.hpp>
5
6  #include <numeric>
7  #include <memory>
8  #include <exception>
9  #include <new>
10 #include <mutex>
11 #include <iostream>
12
13 #include "models/ModelSINET.h"
14 #include "models/ModelMediapipe.h"
15 #include "models/ModelSelfie.h"
16 #include "models/ModelRVM.h"
17 #include "models/ModelPPHumanSeg.h"
18 #include "FilterData.h"
19 #include "consts.h"
20
21 class BackgroundFilter
22 {
23 public:
24    BackgroundFilter() {
25       tf = new FilterData();
26       std::string instanceName{"background-removal-inference"};
27       tf->env.reset(
28               new Ort::Env(OrtLoggingLevel::ORT_LOGGING_LEVEL_ERROR,
29                       instanceName.c_str()));
```

```
30      tf->modelSelection = MODEL_MEDIAPIPE;
31    }
32
33      void filterUpdateThreads(const uint32_t numThreads_ = 1);
34
35    void filterUpdateModel(const std::string moadel_ = MODEL_MEDIAPIPE);
36
37    void filterUpdateProvider(const std::string useGPU_ = USEGPU_CPU);
38
39      void filterActivateChanges() {
40          setDefines();
41          if (!initialized) {
42              createOrtSession(tf);
43              initialized = true;
44          }
45      }
46
47      void filterVideoTick();
48
49      bool setInputImage(const int height, const int width,
50      const int type, unsigned char* data);
51
52    cv::Mat getMask() {
53      return tf->backgroundMask;
54    }
55
56  private:
57      FilterData *tf;
58      bool initialized = false;
59
60      void setDefines() {
61      if (tf->modelSelection == MODEL_SINET) {
62        #define _MODEL_SINET 1
63      }
64      if (tf->modelSelection == MODEL_SELFIE) {
65        #define _MODEL_SELFIE 1
66      }
67      if (tf->modelSelection == MODEL_MEDIAPIPE) {
68        #define _MODEL_MEDIAPIPE 1
69      }
70      if (tf->modelSelection == MODEL_RVM) {
71        #define _MODEL_RVM 1
72      }
73      if (tf->modelSelection == MODEL_PPHUMANSEG) {
74        #define _MODEL_PPHUMANSEG 1
75      }
76    }
77
78    void processImageForBackground(
79              const cv::Mat& imageBGRA, cv::Mat& backgroundMask);
80  };
```

Листинг 3.19: Класс BackgroundFilter

В конструкторе данного класса инициализируется объект tf, который содержит базовую информацию для работы с ORT. Создается новая среда для работы с ORT, и ее уровень логирования устанавливается на уровень ошибок. Выбирается модель по умолчанию – modiapipe.

Метод **filterUpdateThreads** обновляет количество задействованных

ядер CPU и устанавливает их в указанное значение.

```
void BackgroundFilter::filterUpdateThreads(const int32_t numThreads_ = 1) {
    const uint32_t newNumThreads = numThreads_;

    if (tf->modelSelection.empty() || tf->numThreads != newNumThreads) {
        // Re-initialize model if it's not already the selected
        // one or switching inference device
        tf->numThreads = newNumThreads;
        initialized = false;
    }
}
```

Листинг 3.20: Метод filterUpdateThreads

Метод **filterUpdateModel** устанавливает текущую модель для инференса. По умолчанию модель инициализируется как модель mediapipe. На вход принимает строку с названием модели для инференса.

```
void BackgroundFilter::filterUpdateModel(const std::string model_) {
    const std::string newModel = model_ /*MODEL_***/;
    if (tf->modelSelection.empty() || tf->modelSelection != newModel) {
        // Re-initialize model if it's not already the selected
        // one or switching inference device
        tf->modelSelection = newModel;

        if (tf->modelSelection == MODEL_SINET) {
            tf->model.reset(new ModelSINET);
        }
        if (tf->modelSelection == MODEL_SELFIE) {
            tf->model.reset(new ModelSelfie);
        }
        if (tf->modelSelection == MODEL_MEDIAPIPE) {
            tf->model.reset(new ModelMediaPipe);
        }
        if (tf->modelSelection == MODEL_RVM) {
            tf->model.reset(new ModelRVM);
        }
        if (tf->modelSelection == MODEL_PPHUMANSEG) {
            tf->model.reset(new ModelPPHumanSeg);
        }
        initialized = false;
    }
}
```

Листинг 3.21: Метод filterUpdateModel

Метод **filterUpdateProvider** устанавливает текущий провайдер. По умолчанию установлен провайдер CPU от ONNXRuntime.

```
void BackgroundFilter::filterUpdateGpu(const std::string useGPU_) {
    const std::string newUseGpu = useGPU_ /*USEGPU_**/;
    if (tf->modelSelection.empty() || tf->useGPU != newUseGpu) {
        // Re-initialize model if it's not already the selected
        // one or switching inference device
        tf->useGPU = newUseGpu;
        initialized = false;
    }
}
```

Листинг 3.22: Метод filterUpdateProvider

Метод **setDefines** устанавливает макрос текущей модели. В дальнейшем данный макрос будет использован в функции createOrtSession для загрузки необходимой модели в память.

Метод **filterActivateChanges** устанавливает макрос для текущей модели при помощи функции setDefines и, если сессия не создана (флаг initialized == false), создает ее с помощью функции createOrtSession.

Метод **getMask** возвращает текущую маску для входного изображения.

Метод **setInputImage** устанавливает поле tf->inputRGB. Для этого в данный метод передается размерность изображения (высота x ширина), тип входного изображения и его данные. В методе создается временный cv::Mat для преобразования типа из BGR в RGB. Затем блокируется мьютекс inputRGBLock с помощью std::lock_guard, и при помощи std::move перемещаются данные из tempRGB в tf->inputRGB. Таким образом удается минимизировать накладные расходы на копирование данных.

```
bool BackgroundFilter::setInputImage(const int height, const int width,
    const int type, unsigned char* data);
    cv::Mat temp_mat(height, width, type, data), tempRGB;
    cv::cvtColor(tempMat, tempRGB, cv::COLOR_BGR2RGB);
    {
        std::lock_guard<std::mutex> lock(tf->inputRGBLock);
        tf->inputRGB = std::move(tempRGB);
    }
}
```

Листинг 3.23: Метод setInputImage

Метод **filterVideoTick** обрабатывает входное изображение и возвращает маску в поле tf->backgroundMask. Метод начинается с проверки входного изображения. Если оно пустое – метод заканчивает работу и выдает ошибку. Далее изображение извлекается из tf->inputRGBLock, чтобы предотвратить возможный одновременный доступ к нему. Если все выполнилось успешно – изображение передается на инференс модели. Для этого используется вспомогательная функция processImageForBackground. Если на каком-то из этапов инференса происходит ошибка, она выводится, и работа метода завершается.

```
void BackgroundFilter::filterVideoTick() {
    if (tf->inputBGRA.empty()) {
        std::cerr << "Error! Input image is empty!\n";
        return;
    }

    cv::Mat imageRGB;
    {
        std::unique_lock<std::mutex> lock(
                tf->inputRGBLock, std::try_to_lock);
        if (!lock.owns_lock()) {
            return;
        }
        imageRGB = tf->inputRGB.clone();
    }

    if (tf->backgroundMask.empty()) {
```

```
18            // First frame. Initialize the background mask.
19            tf->backgroundMask =
20                cv::Mat(imageRGB.size(), CV_8UC1, cv::Scalar(255));
21        }
22
23        try {
24            cv::Mat backgroundMask;
25
26            // Process the image to find the mask.
27            processImageForBackground(imageRGB, backgroundMask);
28            cv::resize(backgroundMask, backgroundMask, imageRGB.size());
29
30            // Save the mask for the next frame
31            backgroundMask.copyTo(tf->backgroundMask);
32        }
33        catch (const Ort::Exception& e) {
34            std::cerr << "Error! " << e.what() << "\n";
35            // TODO: Fall back to CPU if it makes sense
36        }
37        catch (const std::exception& e) {
38            std::cerr << "Error! " << e.what() << "\n";
39        }
40    }
```

Листинг 3.24: Метод filterVideoTick

Метод **processImageForBackground** запускает инференс модели фильтра. Для этого вызывается функция runFilterModelInference, в которой выполняется инференс модели фильтра на входном изображении и возвращает результат в переменной outputImage. Если инференс не удался – метод завершается.

```
1  void BackgroundFilter::processImageForBackground(
2          const cv::Mat& imageRGB,
3          cv::Mat& backgroundMask)
4  {
5      cv::Mat outputImage;
6      if (!runFilterModelInference(tf, imageRGB, outputImage)) {
7          return;
8      }
9      // Assume outputImage is now a single channel, uint8 image with values
           between 0 and 255
10
11     const uint8_t threshold_value = (uint8_t)(0.05f * 255.0f);
12     backgroundMask = outputImage < threshold_value;
13 }
```

Листинг 3.25: Метод processImageForBackground

ГЛАВА 4
ТЕСТИРОВАНИЕ ФРЕЙМВОРКОВ

4.1 Реализация тестовой программы

Для тестирования работы инференса была реализована программа, в которой реализовано два основных цикла. Первый цикл предназначен для "разогрева"модели. Для большей точности были замерены результаты "разогрева"на 100, 200 и 300 итерациях. Второй цикл работал в основном режиме. Замеры были проведены на 3000 итерациях инференса. В обоих случаях на вход модели предоставлялось изображение:

Рисунок 4.1 — Входное изображение

Ниже представлен код для тестирования инференса модели pphumnseg_fp32 на провайдере OpenVINO с использованием 1 ядра CPU:

```
#include <iostream>
#include <filesystem>
#include <BackgroundFilter.h>
#include <opencv2/opencv.hpp>

int main() {
    BackgroundFilter filter;
    filter.filterUpdateThreads(1);
    filter.filterUpdateModel(MODEL_PPHUMANSEG);
    filter.filterUpdateProvider(USECPU_VINO);
    filter.filterActivateChanges();

    std::string imagePath = std::filesystem::current_path()
                                .append("input.png").string();
    cv::Mat image_mat = cv::imread(imagePath);

    int max_inference_time_prepairing = 0;
    int min_inference_time_prepairing = INT32_MAX;
    int sum_inference_time_prepairing = 0;

```

```
21      int numOfPreporation = 100;
22
23      for (int i = 0; i < numOfPreporation; i++) {
24          auto start_time_prepairing = std::chrono::high_resolution_clock::now();
25          filter.filterVideoTick(
26                  image_mat.rows, image_mat.cols,
27                  image_mat.type(), image_mat.data);
28          auto end_time_prepairing = std::chrono::high_resolution_clock::now();
29          int inference_time_ms =
30              std::chrono::duration_cast<std::chrono::milliseconds>(
31                      end_time_prepairing - start_time_prepairing).count();
32          sum_inference_time_prepairing += inference_time_ms;
33          max_inference_time_prepairing =
34              std::max(inference_time_ms, max_inference_time_prepairing);
35          min_inference_time_prepairing =
36              std::min(inference_time_ms, min_inference_time_prepairing);
37      }
38
39      std::cout << "Minimum infirence time of preporation: "
40          << min_inference_time_prepairing << " ms\n";
41      std::cout << "Maximun infirence time of preporation: "
42          << max_inference_time_prepairing << " ms\n";
43      std::cout << "Average infirence time of preporation: "
44          << sum_inference_time_prepairing / numOfPreporation << " ms\n";
45
46      int max_inference_time = 0;
47      int min_inference_time = INT32_MAX;
48      int sum_inference_time = 0;
49
50      int numOfIteration = 3000;
51
52      for (int i = 0; i < numOfIteration; i++) {
53          auto start_time = std::chrono::high_resolution_clock::now();
54          filter.filterVideoTick(
55              image_mat.rows, image_mat.cols,
56              image_mat.type(), image_mat.data);
57          auto end_time = std::chrono::high_resolution_clock::now();
58          int inference_time_ms =
59              std::chrono::duration_cast<std::chrono::milliseconds>(
60                      end_time - start_time).count();
61          sum_inference_time += inference_time_ms;
62          max_inference_time =
63              std::max(inference_time_ms, max_inference_time);
64          min_inference_time =
65              std::min(inference_time_ms, min_inference_time);
66      }
67
68      std::cout
69          << "\nMinimum infirence time: " << min_inference_time << " ms\n";
70      std::cout
71          << "Maximun infirence time: " << max_inference_time << " ms\n";
72      std::cout << "Average infirence time: "
73          << sum_inference_time / numOfIteration << " ms\n";
74
75      return 0;
76  }
```

Листинг 4.1: Код тестирующей программы

4.2 Анализ результатов

При тестировании инференса модели замерялось максимальное, минимальное и среднее время выполнения. Результаты измерялись в миллисекундах. Замеры производились на разных провайдерах: DefaultCPU, OpenVINO, QNN, DirectML и oneDNN. При использовании провайдера DefaultCPU были измерены зависимости скорости инференса от разного количества задействованных ядер CPU: 1 и 3.

В данной таблице представлены результаты инференса модели mediapipe:

"Разогрев"	**100**	**300**
QNN	Min:6ms Max:18ms Average:6ms	Min:5ms Max:11ms Average:6ms
DirectML	Min:7ms Max:17ms Average:7ms	Min:6ms Max:14ms Average:6ms
OpenVINO	Min:8ms Max:38ms Average:9ms	Min:8ms Max:43ms Average:8ms
oneDNN	Min:10ms Max:58ms Average:12ms	Min:9ms Max:45ms Average:11ms
DefaultCPU 1 ядро	Min:11ms Max:40ms Average:11ms	Min:11ms Max:47ms Average:11ms
DefaultCPU 3 ядра	Min:7ms Max:54ms Average:8ms	Min:7ms Max:33ms Average:9ms

Таблица 4.1 — Результаты инференса модели mediapipe

Тесты всех провайдеров, кроме DirectML, проводились на Macbook с чипом M1. Для этого была установлена виртуальная машина windows 11, при помощи программы Parallels Desktop. Тестирование провайдера DirectML пришлось проводить на стороннем устройстве, так как виртуальная машина windows 11 не могла получить доступ к GPU.

Из приведенных таблиц видно, что при обработке на DefaultCPU, от количества используемых ядер меньше всего зависят модели mediapipe и selfie_segmentation. При разработке данных моделей учитывалось, что они будут использоваться на мобильных устройствах, поэтому они и демонстрируют такие результаты, независимо от количества задействованных ядер CPU.

"Разогрев"	100	300
QNN	Min:7ms Max:18ms Average:8ms	Min:6ms Max:36ms Average:7ms
DirectML	Min:8ms Max:24ms Average:9ms	Min:8ms Max:30ms Average:8ms
OpenVINO	Min:10ms Max:30ms Average:10ms	Min:9ms Max:45ms Average:11ms
oneDNN	Min:12ms Max:54ms Average:14ms	Min:11ms Max:32ms Average:12ms
DefaultCPU 1 ядро	Min:16ms Max:67ms Average:17ms	Min:15ms Max:42ms Average:16ms
DefaultCPU 3 ядра	Min:9ms Max:24ms Average:9ms	Min:9ms Max:22ms Average:9ms

Рисунок 4.2 — Инференс selfie_segmentation

"Разогрев"	100	300
QNN	Min:28ms Max:58ms Average:29ms	Min:24ms Max:64ms Average:24ms
DirectML	Min:30ms Max:62ms Average:31ms	Min:28ms Max:57ms Average:28ms
OpenVINO	Min:45ms Max:137ms Average:45ms	Min:43ms Max:126ms Average:44ms
oneDNN	Min:49ms Max:145ms Average:49ms	Min:44ms Max:137ms Average:44ms
DefaultCPU 1 ядро	Min:126ms Max:298ms Average:130ms	Min:120ms Max:243ms Average:126ms
DefaultCPU 3 ядра	Min:51ms Max:170ms Average:54ms	Min:50ms Max:156ms Average:52ms

Рисунок 4.3 — Инференс rvm_mobilenetv3_fp32

"Разогрев"	100	300
QNN	Min:12ms Max:27ms Average:13ms	Min:10ms Max:23ms Average:12ms
DirectML	Min:12ms Max:24ms Average:12ms	Min:11ms Max:35ms Average:12ms
OpenVINO	Min:19ms Max:70ms Average:20ms	Min:20ms Max:78ms Average:21ms
oneDNN	Min:21ms Max:54ms Average:22ms	Min:19ms Max:65ms Average:19ms
DefaultCPU 1 ядро	Min:25ms Max:53ms Average:25ms	Min:24ms Max:78ms Average:25ms
DefaultCPU 3 ядра	Min:16ms Max:38ms Average:16ms	Min:16ms Max:57ms Average:17ms

Рисунок 4.4 — Инференс SINet_softmax_simple

"Разогрев"	100	300
QNN	Min:13ms Max:20ms Average:13ms	Min:11ms Max:28ms Average:12ms
DirectML	Min:13ms Max:19ms Average:14ms	Min:10ms Max:26ms Average:11ms
OpenVINO	Min:18ms Max:84ms Average:20ms	Min:19ms Max:120ms Average:19ms
oneDNN	Min:20ms Max:98ms Average:21ms	Min:19ms Max:87ms Average:21ms
DefaultCPU 1 ядро	Min:35ms Max:74ms Average:36ms	Min:35ms Max:95ms Average:37ms
DefaultCPU 3 ядра	Min:15ms Max:117ms Average:17ms	Min:15ms Max:123ms Average:17ms

Рисунок 4.5 — Инференс pphumanseg_fp32

По итогу тестирования, лучшими провайдерами оказались QNN и DirectML. Они на всех тестируемых моделях показали лучший результат, в некоторых случаях с большим отрывом от других провайдеров (рис. 4.3). Это связано с архитектурой процессора, под который проектировался провайдер QNN: ARM. Что же касается DirectML – это единственный провайдер, ко-

торый был реализован на GPU, из-за чего он и тестировался на стороннем компьютере.

Следующими по скорости инференса для моделей mediapipe (таблица 4.1) и rvm_mobilenetv3_fp32 (рис. 4.3) стали результаты провайдера OpenVINO. Для остальных моделей – Default CPU с использованием 3 ядер. Хотелось бы отметить, что при такой конфигурации (DefaultCPU, 3 ядра) было задействовано приблизительно 80% CPU. Провайдер OpenVINO разрабатывался Intel и существенный прирост производительности отмечается имеено на их процессорах. В связи с чем OpenVINO и продемонстрировал результат сопоставимый с DefaultCPU, 3 ядра.

Последним по скорости инференса всех моделей стал провайдер DefaultCPU с использованием 1 ядра. В некоторых случаях результат работы данного провайдера был сравним с работой oneDNN (таблица 4.1, рис. 4.4), но в большинстве своем сильно уступал. При использовании модели rvm_mobilenetv3_fp32 скорость инференса на провайдере DefaultCPU (1 ядро) была почти в 4 раза медленнее чем на провайдере QNN.

Заключение

В ходе курсовой работы:

1. Сделан краткий обзор фреймворка ONNX Runtime и провайдеров OpenVINO, DirectML, oneDNN, QNN и DefaultCPU;

2. Реализованы классы для кросс-платформенного инференса моделей mediapipe, Selfie_segmentation, pphumanseg_fp32, SINet_softmax_simple и rvm_mobilenetv3_fp32;

3. Реализованы платформозависимые компоненты: DirectML, OpenVINO, oneDNN, QNN;

4. Создана тестовая программа для замера скорости инференса выбранной модели;

5. Проведено сравнение скорости инференса моделей в зависимости от используемого провайдера или количеста ядер CPU.

Список использованных источников

1. Qualcomm Technologies, Inc. (2024). Qualcomm Neural Processing SDK for Windows on Snapdragon [Электронный ресурс]: Qualcomm Developer Network. Режим доступа: https://developer.qualcomm.com/software/qualcomm-neural-processing-sdk/wi ndows-on-snapdragon. Дата доступа: 15.05.2024

2. ONNX Runtime Execution Providers [Электронный ресурс]: ONNX Runtime. – Redmond, WA: Microsoft Corporation, 2021. – Режим доступа: https://onnxruntime.ai/docs/execution-providers/. Дата доступа: 15.05.2024

3. Ashfaq, S., AskariHemmat, M., Sah, S., Saboori, E., Mastropietro, O., & Hoffman, A. (2022). Accelerating Deep Learning Model Inference on Arm CPUs with Ultra-Low Bit Quantization and Runtime [Электронный ресурс]: Deeplite Inc. – Montreal, Canada: Deeplite Inc., 2022. – Режим доступа: https://arxiv.org/pdf/2207.08820.pdf. Дата доступа: 15.05.2024

4. Lee, J., Chirkov, N., Ignasheva, E., Pisarchyk, Y., Shieh, M., Riccardi, F., Sarokin, R., Kulik, A., & Grundmann, M. (2019). On-Device Neural Net Inference with Mobile GPUs [Электронный ресурс]: Google Research. – Режим доступа: https://ar5iv.labs.arxiv.org/html/1907.01989. Дата доступа: 15.05.2024