NMDL+ Руководство пользователя

НАУЧНО-ТЕХНИЧЕСКИЙ ЦЕНТР

Оглавление

1. Общие сведения

1.1. Назначение

Программный модуль *NMDL+* позволяет запускать предварительно обученную глубокую сверточную нейронную сеть на вычислительных модулях *MC127.05*, *NMCard*, *NMMezzo*, *NMQuad* и на симуляторе модуля *MC127.05*. Программный модуль состоит из 2 частей. Одна часть работает на персональном компьютере (хост) под управлением 64 разрядных ОС Microsoft® Windows 7/10 или Linux, другая часть запускается и работает на процессоре вычислительного модуля. Для связи устройств с хостом используется интерфейс PCIe.

Для работы с *NMDL+* необходимо предварительно установить в системе ПО поддержки используемых вычислительных модулей. Для работы с симулятором установка ПО поддержки не требуется.

NMDL+ выполняет обработку пользовательских исходных изображений в соответствии с заданной моделью нейросети. Перед обработкой необходимо подготовить данные модели и изображений.

Модель предварительно подготавливается специальным компилятором из состава *NMDLP*. Исходные модели могут быть представлены в формате *ONNX* или *DarkNet*. Компилятором *NMDLP* поддерживаются не все операции, определённые в *ONNX*. Список поддерживаемых операций и другие ограничения приведены в разделе Поддерживаемые операции.

Изображения также должны быть предварительно обработаны специальным конвертером изображений. Только подготовленные модели и изображения могут быть загружены и обработаны на вычислительных модулях.

Библиотека предоставляет программный интерфейс C/C++, Python.

1.2. Режимы обработки

Обработка входных данных (инференс) производится в соответствии с графом обработки, определённым в нейросетевой модели. Каждая из моделей обрабатывается на вычислительном устройстве - юните. Модули *MC127.05*, *NMCard*, *NMMezzo* и прочие устройства, выполненные на базе *СБИС К1879ВМ8Я* имеют четыре юнита.

На устройствах с процессором *К1879ВМ8Я* возможна одновременная и независимая обработка различных моделей. Такая обработка иллюстрируется на Рисунок 1. Каждый юнит независимо обрабатывает "свою" модель. Такой режим обработки обозначается "single unit" - инференс выполняется на одном юните.

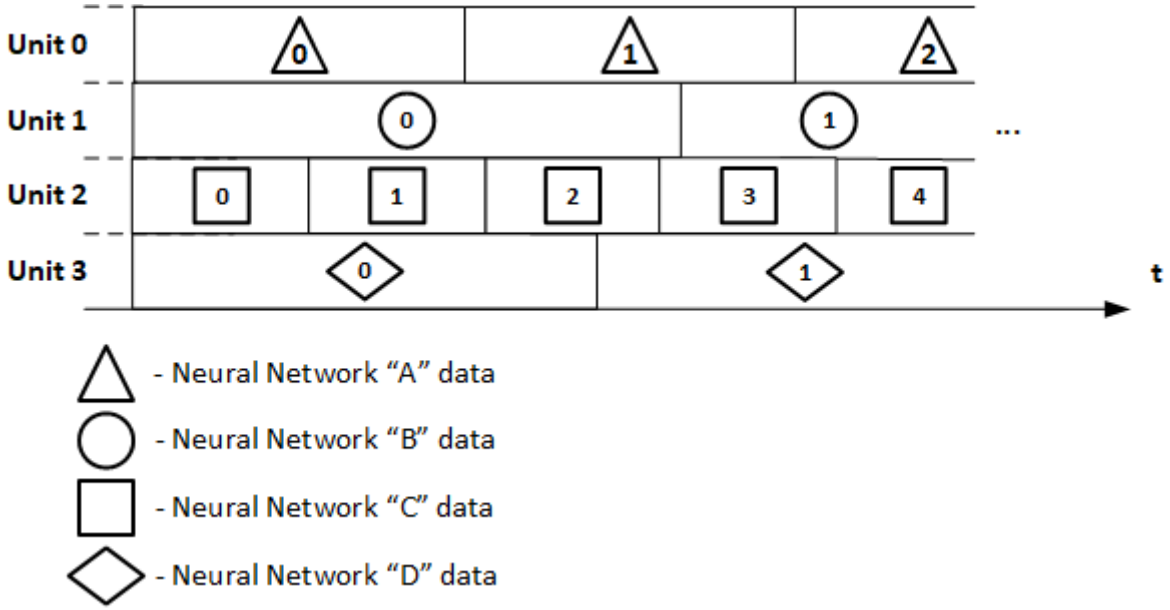

Рисунок 1. Одновременная обработка нескольких нейронных сетей в режиме "single unit"

Четыре юнита можно настроить на обработку одинаковых моделей, когда каждый юнит выполняет один и тот же граф обработки, при этом можно организовать пакетную обработку, достигая максимальной производительности обработки потока данных, например, потока кадров от видеокамеры (см. Рисунок 2).

Пакетная обработка характеризуется высоким значением задержки (Latency) - времени от запуска обработки до получения результата.

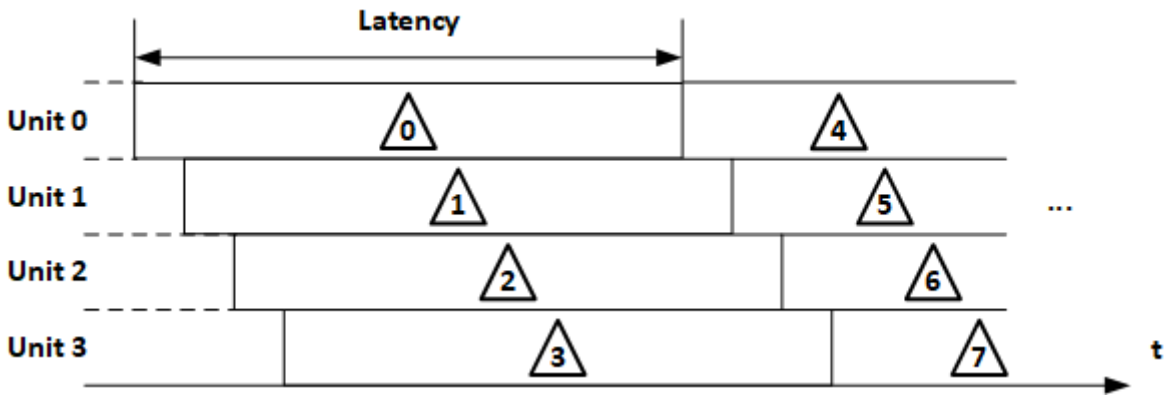

Рисунок 2. Пакетная обработка в режиме "single unit"

Для устройств на базе *СБИС К1879ВМ8Я* в *NMDL+* реализован режим обработки одной модели на четырёх юнитах с равномерным разделением данных (*"multi unit"*) (см. Рисунок 3). При этом достигается минимальная задержка. Производительность здесь, как правило, несколько ниже из-за накладных расходов по организации параллелизма. Например, при выполнении свёрток над разделёнными тензорами необходимо компенсировать обработку границ, выполнять переупаковки, транзит данных между кластерами и т.п.

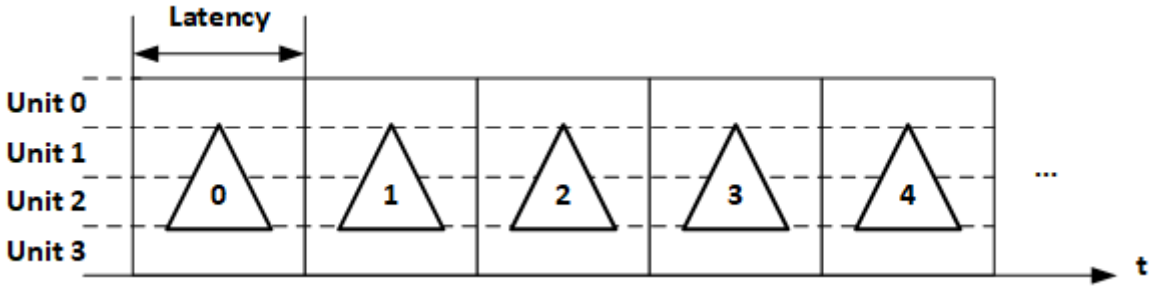

Рисунок 3. Режим обработки "multi unit"

1.3. Быстрый старт

В разделе приводится пошаговая инструкция для быстрого начала работы с демонстрационной программой. Более полную информацию о *NMDL+* можно получить в соответствующих разделах руководства. Здесь описывается демонстрация обработки изображения с применением нейронной сети для классификации *Squeezenet* на симуляторе устройства на базе *СБИС К1879ВМ8Я*.

- Установите дистрибутив *NMDL+* так, как описано в разделе Установка
- Перейдите в каталог *bin* в директории установки *NMDL+* (по-умолчанию в Windows "C:\Program Files\Module\NMDLPlus", по-умолчанию в Linux "/opt/nmdlplus") и запустите демонстрационную программу *nmdlp_gui*
- Выберите устройство с помощью меню

  File – Open Board...

  Убедитесь, что в диалоговом окне выбран симулятор.

- Выберите описание модели с помощью меню

  File – Open Description...

  В диалоговом окне выберите файл

  $PATH_TO_NMDLPLUS/ref_data/squeezenet_imagenet/description08.xml

- Выберите обрабатываемое изображение с помощью меню

  File – Open Picture...

  В диалоговом окне выберите файл

  $PATH_TO_NMDLPLUS/ref_data/squeezenet_imagenet/frame.bmp

- Запустите обработку с помощью меню

  File – Run

  В результате обработки всплывёт окно классификации с вычисленными вероятностями в порядке уменьшения. Верный класс для выбранного изображения - "lakeside".

1.4. Производительность NMDL+

В таблицах приводятся значения производительности (кадры в секунду FPS) и значения задержки от начала обработки кадра до получения результата (Latency). Рядом с названием сети в скобках указан размер обрабатываемого изображения.

SU - Single unit mode

MU - Multi unit mode

Таблица 1. FPS

	СБИС К1879ВМ8Я **(MU)**	*СБИС К1879ВМ8Я* **(SU)**
alexnet_imagenet (227x227x3)	12.7376	13.1061
inception_v3_imagenet (299x299x3)	13.7975	21.9992
iva (112x112x3)	9.32752	13.8304
resnet_18_imagenet (224x224x3)	24.4118	46.1858
resnet_50_imagenet (224x224x3)	11.8142	20.1852
mobilenet_v2_imagenet (224x224x3)	nan	30.1471
squeezenet_imagenet (224x224x3)	102.294	156.448
unet_covid (572x572x1)	0.10381	0.34009
unet_tiny_covid (48x48x1)	229.784	347.795
yolo_v2_tiny_pascal_voc (416x416x3)	23.1283	31.263
yolo_v2_tiny_wega (416x416x3)	27.5956	35.7081
yolo_v3_coco (416x416x3)	3.7449	4.58138
yolo_v3_tiny_coco (416x416x3)	29.0247	36.7978
yolo_v3_wega (704x576x3)	1.91577	1.98855
yolo_v5s_coco (640x640x3)	4.4736	4.8231
yolo_v7 (640x640x3)	1.1806	1.2065
yolo_v7_tiny (640x640x3)	11.07	12.2886

Таблица 2. Latency (ms)

	СБИС К1879ВМ8Я **(MU)**	*СБИС К1879ВМ8Я* **(SU)**
alexnet_imagenet (227x227x3)	78	305
inception_v3_imagenet (299x299x3)	72	181
iva (112x112x3)	107	289
resnet_18_imagenet (224x224x3)	40	86
resnet_50_imagenet (224x224x3)	84	198
mobilenet_v2_imagenet (224x224x3)	nan	132
squeezenet_imagenet (224x224x3)	9	25
unet_covid (572x572x1)	9633	11761
unet_tiny_covid (48x48x1)	4	11
yolo_v2_tiny_pascal_voc (416x416x3)	43	127
yolo_v2_tiny_wega (416x416x3)	36	112
yolo_v3_coco (416x416x3)	267	873
yolo_v3_tiny_coco (416x416x3)	34	108
yolo_v3_wega (704x576x3)	522	2010
yolo_v5s_coco (640x640x3)	223	832
yolo_v7 (640x640x3)	845	3305
yolo_v7_tiny (640x640x3)	90	325

2. Состав ПО

2.1. C/C++

ПО реализации нейронных сетей состоит из программных модулей (API), утилит и руководства.

Файлы API для разработки программ с использованием *NMDL+*:

- *nmdlp[.dll, .so]* - программный модуль для инференса обученной нейронной сети

  См. раздел Описание идентификаторов, функций и структур nmdl+

- *nmdlp.lib* - библиотека для раннего связывания программ с *NMDL+* в среде MSVC++
- *nmdlp.h* - заголовочный файл с описанием структур и функций API
- *nmdlp_compiler[.dll, .so]* - программный модуль - компилятор моделей ONNX/DarkNet во внутреннее представление.

  См. раздел Описание идентификаторов и функций nmdlp_compiler

- *nmdlp_compiler.lib* - библиотека для раннего связывания модуля компилятора моделей в среде MSVC++
- *nmdlp_compiler.h* - заголовочный файл с описанием структур и функций компилятора моделей.
- *nmdlp_image_converter[.dll, .so]* - программный модуль для подготовки обрабатываемых изображений.

  См. Описание идентификаторов и функций nmdlp_image_converter

- *nmdlp_image_converter.lib* - модуль для раннего связывания модуля подготовки изображений в среде MSVC++
- *nmdlp_image_converter.h* - заголовочный файл с описанием структур и функций для подготовки изображений

Заголовочные файлы и библиотеки раннего связывания размещаются в каталогах *include* и *lib* директории *NMDL+*.

Утилиты:

- *nmdlp_compiler_console* - утилита командной строки для компиляции моделей из форматов ONNX/DarkNet во внутренний формат для загрузки на вычислительные модули.

  Файл модели ONNX обычно имеет расширение *.onnx*. Модель в формате DarkNet сохраняется в двух файлах - с расширением *.cfg* и расширением *.weights*. Подготовленная модель имеет расширение *.nm8*.

  См. раздел Компиляция модели

- *nmdlp_image_converter_console* - утилита командной строки для подготовки обрабатываемых изображений.

  См. раздел Подготовка изображений

- *nmdlp_gui* - оконная утилита для демонстрации функциональных возможностей *NMDL+*.

  См. раздел Демонстрационная программа

2.2. Python

NMDL+ Python API предоставляет возможность пользователю работать с нейронными сетями и пользоваться всеми функциями основного набора библиотек *NMDL+* с использованием языка Python.

Данное *API* включает в себя следующий набор пакетов для импорта:

- *compiler* - программная обертка над компилятором моделей
- *image_converter* - программная обертка над конвертером изображений
- *nmdlp* - программная обертка над базовым набором команд для работы с вычислительным модулем и инференса
- *session* - интерфейс, обеспечивающий доступ к набору высокоуровневых функций для работы с библиотекой *NMDL+* из *Python* используя "сессионную модель" по аналогии с *TensorFlow Session, OnnxRuntime InferenceSession*

Помимо этого, в состав API также входит набор вспомогательных пакетов для импорта, включающих в себя необходимые для корректной работы типы данных и постоянные:

- *constants*
- *datatypes*

3. Установка

3.1. Установка в Windows

NMDL+ распространяется в виде установочного дистрибутива. Поддерживаются только 64-х битовые системы.

Для установки дистрибутива запустите исполняемый файл инсталлятора с правами администратора. Следуйте инструкциям мастера установки.

Для работы с программными модулями необходимо включить их в область "видимости" операционной системы. Одно из решений - создание переменной среды окружения, например, с именем *NMDLPLUS*, в которой записывается путь к каталогу с заголовочными и бинарными файлами, и добавление созданной переменной к переменной окружения *PATH*.

Для использования модуля *nmdl+* в C/C++ программах включите в исходный файл директиву #include "nmdlp.h" и свяжите программу с библиотекой *NMDL+*.

Если используется среда разработки MSVC++, то для раннего связывания подключите к проекту файл *nmdlp.lib*. Можно создать и использовать переменную окружения *NMDLPLUS* для задания пути к файлам *nmdlp.h* и *nmdlp.lib*. Таким же образом можно подключить модули *nmdlp_compiler* и *nmdlp_image_converter*.

3.2. Установка в Linux

Распространяется в виде .deb пакета. Поддерживаются только 64-х битовые системы семейства Debian.

Для установки используйте менеджер пакетов dpkg.

Например:

```
dpkg -i NmdlPlus-1.0.0-Linux.deb
```

3.3. Установка Python пакета

Python API для *NMDL+* распространяется в виде .whl архива, при этом все требования по системе и её разрядности аналогичны C/C++ версии.

Для установки рекомендуется использовать менеджер пакетов *pip*:

```
pip install nmdlplus-1.0.0-py3-none-linux_x86_64.whl
```

Требованием для установки является наличие установленного дистрибутива Python>=3.5, а также пакета numpy. При этом можно использовать актуальный дистрибутив Anaconda, что избавит от необходимости дополнительно загружать numpy.

Для корректной работы с вычислительными модулями требуются только драйвера, при этом симулятор можно использовать сразу.

Библиотека для Python используется независимо от основных программных модулей NMDL+.

После установки проверить работоспособность можно при помощи:

```
import nmdlplus as nm

print(nm.__version__)
```

3.4. Дополнительные компоненты

Вместе с программным модулем можно получить дополнительные компоненты для проверки и демонстрации работы *NMDL+*.

Компоненты распространяются в архивах (возможны изменения):

- python_examples.tar.gz - архив с примерами для демонстрации работы Python API
- mobilenet_v2_imagenet.tar.gz - архив для демонстрации работы сети MOBILENET V2
- alexnet_imagenet.tar.gz - архив для демонстрации работы сети ALEXNET
- inception_v3_imagenet.tar.gz - архив для демонстрации работы сети INCEPTION V3
- resnet_18_imagenet.tar.gz - архив для демонстрации работы сети RESNET18
- resnet_50_imagenet.tar.gz - архив для демонстрации работы сети RESNET50
- squeezenet_imagenet.tar.gz - архив для демонстрации работы сети SQUEEZENET
- yolo_v2_tiny_pascal_voc.tar.gz - архив для демонстрации работы сети YOLO V2 TINY
- yolo_v3_coco.tar.gz - архив для демонстрации работы сети YOLO V3
- yolo_v3_tiny_coco.tar.gz - архив для демонстрации работы сети YOLO V3 TINY
- yolo_v5s_coco.tar.gz - архив для демонстрации работы сети YOLO V5S

В каталогах с моделями содержатся файлы:

- frame.bmp - тестовое изображение
- Модель в двух возможных форматах:
  - model.onnx - модель в формате ONNX
  - model.cfg + model.weights - модель с весовыми коэффициентами в формате DARKNET
- description08.xml - файл описания модели для запуска в программе *nmdlp_gui*
- prepare[.sh, .cmd, .py] - Скрипты на различных языках для компиляции модели и подготовки тестового изображения

Для подготовки демонстрационных данных, которые будут обрабатываться на

вычислительном модуле необходимо распаковать их в каталог *ref_data*, выполнить компиляцию моделей и подготовку изображений так, как это описывается в разделах Компиляция модели и Подготовка изображений руководства.

Для удобства компиляции в архив включены скрипты *prepare.cmd*, *prepare.sh*, *prepare.py*.

В результате работы скрипта в каждом каталоге появятся файлы:

- frame08 - подготовленное для обработки изображение
- model.nm8 - скомпилированная модель для загрузки в режиме *"single unit"*
- model_mu.nm8 - скомпилированная модель для загрузки в режиме *"multi unit"*

3.5. Подготовка модулей MC127.05, NMCard, NMMezzo и NMQuad

Для работы с *NMDL+* необходимо установить плату в свободный слот PCIe (для *NMMezzo* в разъём PCIe несущей платы) и выполнить инсталляцию ПО поддержки модуля, входящее в комплект поставки изделия.

4. Компиляция модели

Нейронная сеть должна быть предварительно обучена с помощью нейросетевого пакета (например, PyTorch или TensorFlow), и преобразована к форматам *ONNX* или *DarkNet*.

Исходная модель в формате *ONNX*, как правило, хранится в файле со структурой Google Protocol Buffer и имеет расширение **.onnx*.

Модели для сетей типа *YOLO* могут храниться в формате *DarkNet*. В этом случае модель описывается двумя файлами - файл с расширением **.cfg* содержит описание графа обработки, файл **.weights* содержит веса для свёрток.

Файлы моделей *ONNX* и *DarkNet* являются результатом обучения нейронной сети и одновременно исходными данными для компилятора, который в результате обработки создаёт файлы с расширением *nm8*.

Если в модели ONNX входные и выходные тензоры имеют динамические размерности, то для компиляции модели необходимо сделать их статическими. Фиксацию можно сделать при помощи следующего примера на python.

Здесь предполагается, что размерности входного тензора динамические.
Мы фиксируем их к значениям [1, 3, 600, 640], где
1 - размер пакета (batch),
3 - каналы,
600 - высота,
640 - ширина.
Имя входного слоя - "input1",
имена выходных слоёв - "output1", "output2" и "output3".
Выходные размеры будут вычисляться автоматически:

```
# Model fixation script
import onnx
from onnx.tools import update_model_dims
from onnx import helper, shape_inference

model = onnx.load('path/to/the/dynamic_dim_model.onnx')

# update model dimensions
variable_length_model = update_model_dims.update_inputs_outputs_dims(model, \
{'input1': [1, 3, 600, 640]}, {'output1': ['1', 'C','H','W'], \
'output2': ['1', 'C','H','W'], 'output3': ['1', 'C','H','W']})

# out dim will be calculated automatically
inferred_model = shape_inference.infer_shapes(variable_length_model)

# Check the model
onnx.checker.check_model(inferred_model)

# Save the ONNX model
onnx.save(inferred_model, 'static_dim_model.onnx')
```

Модель в формате *nm8* является бинарным образом скомпилированной пользовательской модели. Её структура не документируется и зависит от версии компилятора.

Компилятор выполнен в виде динамически загружаемого модуля и может быть встроен в программу пользователя. Можно также воспользоваться консольной утилитой *nmdlp_compiler_console* для выполнения преобразования моделей из командной строки.

```
nmdlp_compiler_console NN_TYPE SRC_FILENAME DST_FILENAME IS_MULTI_UNIT
[WEIGHTS_FILENAME]
```

Аргументы командной строки:

- NN_TYPE - формат файла входной модели. Допустимые значения:
  - ONNX
  - DARKNET
- SRC_FILENAME - имя файла исходной модели
- DST_FILENAME - имя файла выходной модели
- IS_MULTI_UNIT
  - 0: модель будет обрабатываться в режиме "single unit"
  - 1: модель будет обрабатываться в режиме "multi unit"

  См. раздел "Режимы обработки"
- WEIGHTS_FILENAME - имя файла с весовыми коэффициентами модели. Нужен только для

моделей в формате DarkNet

Пример компиляции модели squeezenet для прогона на одном юните:

```
nmdlp_compiler_console ONNX squeezenet.onnx squeezenet.nm8 0
```

Пример компиляции модели yolo v2 tiny для прогона в режиме "multy unit":

```
nmdlp_compiler_console DARKNET yolo2t.cfg yolo2t.nm8 1 yolo2t.weights
```

4.1. Поддерживаемые операции

Компилятором поддерживается следующий набор операций:

- Abs
- Add
- AveragePool
  - AveragePool2x2, no pad, stride=1
  - AveragePool2x2, no pad, stride=2
  - AveragePool3x3, no pad, stride=2
  - AveragePool3x3, pad, stride=2
- BatchNormalization (при условии, что операция выполняется сразу после свёртки)
- Ceil
- Clip
- Concat (по осям каналов и ширины, количество входных тензоров - не более 7)
- Convolution
  - Conv1x1, stride=1
  - Conv1x1, stride=2
  - Conv3x3, no pad, all strides
  - Conv3x3, pad, stride=1
  - Conv3x3, pad, stride=2
  - Conv5x5, no pad, all strides
  - Conv5x5, pad, stride=1
  - Conv7x7, no pad, all strides
  - Conv7x7, pad, stride=2
  - Conv11x11, no pad, all strides
  - Conv7x1, pad_w, stride=1

    - Conv1x7, pad_h, stride=1
    - Conv3x1, pad_w, stride=1
    - Conv1x3, pad_h, stride=1
- ConvTranspose (kernel 2x2, stride 2x2)
- DepthwiseConv (Доступно только в режиме "single unit")
    - Conv3x3, pad=(1,1,1,1), stride=1x1
    - Conv3x3, pad=(1,1,1,1), stride=2x2
- Div
- Exp
- Floor
- GEMM
- GlobalAveragePool
- Leaky Relu
- Mat Mul
- MaxPool
    - MaxPool2x2, no pad, stride=1
    - MaxPool2x2, no pad, stride=2
    - MaxPool3x3, no pad, stride=2
    - MaxPool3x3, pad, stride=2
    - MaxPoolNxN (N - odd value), stride=1
- Mul
- Neg
- Pad
- PRelu
- Reciprocal
- Relu
- Reshape
- Resize
    - Resize nearest
    - Resize linear
    - Half fixel
    - Scale 2x2
- Sigmoid
- Softmax
- Slice

- Sub
- Sqrt
- Transpose
- Upsample nearest

Количество входных или выходных тензоров - не более 32.

5. Подготовка изображений

Обрабатываемые изображения необходимо предварительно сконвертировать в вещественный формат (тензор). Сделать это можно с помощью утилиты *nmdlp_image_converter_console*

```
nmdlp_image_converter_console SRC_FILE DST_FILE W H F DR DG DB AR AG AB
```

Аргументы командной строки:

- SRC_FILE - имя входного файла. Поддерживаемые форматы:
  - bmp
  - gif
  - jpeg
  - emf
  - png
  - tiff
  - ico
  - icns
  - tga
  - wbmp
  - webp
- DST_FILE - имя выходного файла
- W - ширина тензора изображения
- H - высота тензора изображения
- F - порядок следования цветовых каналов в тензоре изображения. Допустимые значения:
  - rgb
  - rbg
  - grb
  - gbr
  - brg
  - bgr
  - intensity - один канал градации серого
- DR - делитель для красного канала пикселя в выражении dst = src / D + A (float)
- DG - делитель для зелёного канала пикселя в выражении dst = src / D + A (float)
- DB - делитель для голубого канала пикселя в выражении dst = src / D + A (float)

- AR - слагаемое для красного канала пикселя в выражении dst = src / D + A (float)
- AG - слагаемое для зелёного канала пикселя в выражении dst = src / D + A (float)
- AB - слагаемое для голубого канала пикселя в выражении dst = src / D + A (float)

Для изображений с градацией серого, когда аргумент F задан intensity, используются только делитель DR и слагаемое AR. Остальные делители (DG и DB) и слагаемые (AG и AB) игнорируются и могут быть установлены в любые значения.

Программа масштабирует входное изображение относительно центра в соответствии с заданными аргументами.

Подготовленные изображения имеют пиксельный формат расположения элементов типа *float32*.

В пиксельном формате в файл записываются последовательно каналы пикселей изображения. Буфер можно представить как массив в стиле C/C++:

```
float image[HEIGHT][WIDTH][CHANNELS];
```

То есть самый быстро меняющийся индекс - это индекс по каналам.
Количество каналов - три (RGB каналы), или один для одноканальных изображений (градации серого).
В подготовленном буфере размер выравнивается по каналам изображения до ближайшего чётного значения.
Для изображений с чётным количеством каналов размер составит:
*size = width * height * channels* of *float32*.
Для изображений с нечётным количеством каналов размер составит:
*size = width * height * (channels + 1)* of *float32*.

6. Демонстрационная программа

Для демонстрации функциональных возможностей библиотеки *NMDL+* была разработана утилита *nmdlp_gui*. В задачи утилиты входят:

- Инициализация библиотеки *NMDL+*
- Обнаружение и идентификация доступных ускорителей
  - *Simulator*
  - *MC127.05*
  - *NMCard*
  - *NMMezzo*
  - *NMQuad*
- Загрузка файла модели нейронной сети (*.nm8*)
- Загрузка файла описания нейронной сети (.xml)
- Предварительная обработка и загрузка изображений на вычислительный модуль
- Запуск обработки на вычислительном модуле
- Обработка и вывод результатов

При запуске программы появляется главное окно программы со строками меню, состояния и клиентской областью.

В меню расположены органы управления программой.
В клиентской области располагается изображение и результаты обработки.
В строке состояния выводится следующая информация:

- Выбранный ускоритель
  - *Simulator*
  - *MC127.05*
  - *NMCard*
  - *NMMezzo*
  - *NMQuad*
- Выбранная модель нейронной сети
- Выбранное описание нейронной сети
- Выбранное изображение
- Скорость обработки (кадров/с)

Внешний вид программы в процессе работы приведён на Рисунок 4.

Рисунок 4. Внешний вид программы в процессе работы

Выбор модуля осуществляется в диалоговом окне *Open Board*, внешний вид которого приведён на Рисунок 5. Окно содержит список доступных в системе ускорителей, кнопки выбора, а также возможность идентификации устройства посредством светодиодной индикации.

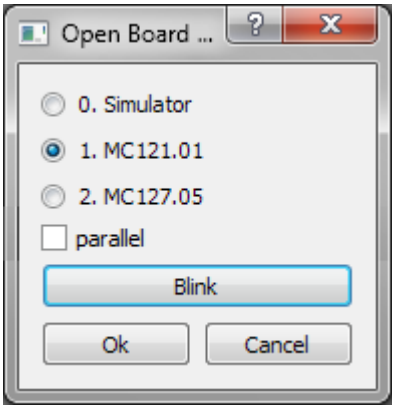

Рисунок 5. Выбор платы

Включение флага *"parallel"* обеспечивает обработку потока изображений в параллельном режиме. В этом режиме производится, по возможности, одновременная обработка нескольких кадров с использованием всех обнаруженных выбранных ускорителей в системе и всех юнитов выбранного ускорителя.
При достаточной пропускной способности канала, когда время передачи данных занимает существенно меньше времени инференса, производительность обработки одного кадра (один инференс) увеличивается кратно в соответствии с количеством ускорителей.
Например, при использовании четырёх ускорителей *MC127.05* в режиме *"single unit"* производительность увеличится примерно в 16 раз (четыре юнита в четырёх ускорителях), в режиме "multi unit" производительность увеличится приблизительно в четыре раза.
Параллельная обработка производится только при запуске в автоматическом режиме,

когда обрабатывается поток изображений - меню *"File→Run Auto"*.

Выбор описания нейронной сети осуществляется в диалоговом окне *Open Description*. Описание нейронной сети осуществляется в формате *XML* и содержит информацию о необходимом формате изображения, а также параметры интерпретации выходных параметров.

Параметр	Описание
model	Строка с названием файла модели нейронной сети в следующих форматах: • **.nm8* - проприетарный формат сконвертированной модели • **.onnx* - описание модели в формате ONNX Protobuf. Перед загрузкой на модуль модель предварительно конвертируется в **.nm8* • **.cfg* - описание модели в формате DARKNET. Файл с весовыми коэффициентами **.weights* должен размещаться в том же каталоге и иметь то же название, что и файл описания модели. Перед загрузкой на модуль модель предварительно конвертируется в **.nm8* Можно указать как абсолютный путь к файлу, так и путь, относительно размещения файла описания XML.
format	Формат, к которому будет преобразованы пиксели входного изображения перед обработкой. Может принимать значения: • rgb • rbg • grb • gbr • brg • bgr • intensity Это значение соответствует порядку следования каналов входного тензора.
divider	Масштабирующий коэффициент для каждой цветовой компоненты пикселя входного изображения. Используется при преобразовании изображения во входной тензор. См. раздел Подготовка изображений

Параметр	Описание
adder	Коэффициент смещения для каждой цветовой компоненты пикселя входного изображения. Используется при преобразовании изображения во входной тензор. См. раздел Подготовка изображений
neural_network	Выбранная нейронная сеть. Может принимать значения: • classifier - нейронная сеть - классификатор, например: ◦ ALEXNET ◦ RESNET ◦ SQUEEZENET • unet - нейронная сеть семейства U-Net • yolo2 - нейронная сеть семейства YOLO V2 • yolo3 - нейронная сеть семейства YOLO V3 • yolo5 - нейронная сеть семейства YOLO V5 Результатом работы классификатора является вектор вероятностей принадлежности изображения определённым классам. В демонстрационной программе классы и вероятности обнаружения выводятся во всплывающем окне. Сети YOLO в результате обработки выдают информацию о нескольких обнаруженных объектах и их расположении на исходном изображении. В демонстрационной программе обнаруженные объекты обозначаются непосредственно на исходном изображении в описывающих прямоугольниках.
yolo_anchors	Параметры начальной инициализации детектируемых прямоугольников (только для сетей *YOLO*).
yolo_confidence_threshold	Порог для отображения результатов детектирования (только для сетей *YOLO*).
yolo_iou_threshold	Порог пересекающихся прямоугольников детектирования (только для сетей *YOLO*)
labels	Список строк с названиями классов, которые будут отображаться в окне результата обработки.

Диалоговое окно *Open Picture* позволяет открыть одно или несколько изображений для обработки.

При наличии всех данных становится доступной кнопка *Run*, запускающая обработку.

Запуск можно инициировать с помощью комбинации *"Ctrl+R"*.
Для каждого переданного изображения осуществляется предварительная обработка в соответствии с заданным описанием нейронной сети для последующей обработки на ускорителе.
В результате обработки заполняется выходная структура с N тензорами.
В зависимости от нейронной сети, выходная структура интерпретируется различным образом. Для сетей классификации (таких как *SqueezeNet*) появляется дополнительное окно с классами и вероятностью.
Внешний вид этого окна приведён на Рисунок 6.

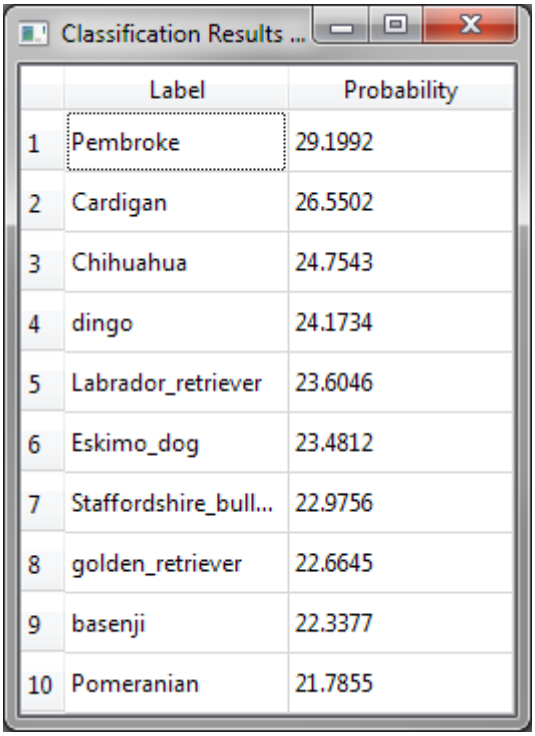

Рисунок 6. Внешний вид окна результатов классификации

Для сетей, которые выполняют местоопределение объекта (таких как *YOLO*), результат показывается непосредственно на исходном изображении.
После окончания работы ускорителя осуществляется наложение прямоугольников на обнаруженные объекты, приводится название класса и вероятность обнаружения.
Пример наложения результата на изображение приведён на Рисунок 7.

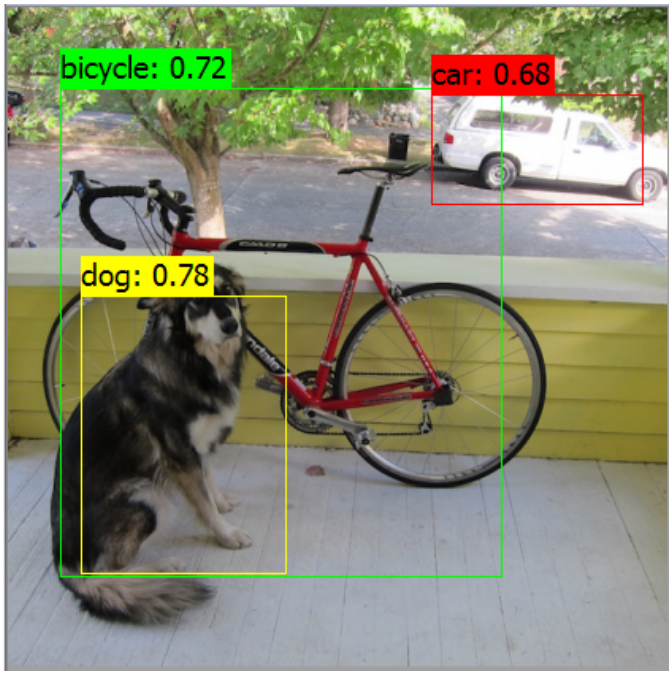

Рисунок 7. Отображение результатов детектирования

Программа позволяет обработать последовательность изображений.
Для этого необходимо при выборе изображений выделить несколько файлов.
После этого при запуске обработки (*Run*) будет обработано одно изображение.
При повторном запуске - второе и т. д.
Можно также запустить обработку в автоматическом режиме (*Run Auto*).
Для останова автоматической обработки нажмите клавишу "*пробел*".

Для отображения дополнительной информации используется строка состояния. В строку выводится название выбранного устройства, файла описания нейронной сети, файла изображения, а также измеренное программой значение fps с учётом пересылки кадра и получения результата. Скорость обработки без учёта времени пересылок выводится в скобках.

7. Пример использования NMDL+

В примере демонстрируется применение библиотеки *NMDL+*, программных модулей для компиляции модели и подготовки изображений. Пример состоит из файла исходного кода *example1.cpp* и скрипта сборки *CMakeLists.txt* в каталоге *"examples/example1"* установочной директории.

Предполагается, что исходные данные для обработки находятся в каталоге *"ref_data/squeezenet_imagenet"* в каталоге установки, это означает, что в примере демонстрируется обработка нейронной сети *squeezenet*.
Исходными данными являются:

- Модель нейронной сети в формате ONNX - файл "ref_data/squeezenet_imagenet/model.onnx"
- Обрабатываемое изображение в формате BMP - файл "ref_data/squeezenet_imagenet/frame.bmp"

В примере показаны вызовы функций библиотек в порядке, необходимом для осуществления корректной работы.

```
#include <array>
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
#include "nmdlp.h"
#include "nmdlp_compiler.h"
#include "nmdlp_image_converter.h"

// #define _USE_DARKNET_

namespace
{

    NMDLP_RESULT Call(NMDLP_COMPILER_RESULT result, const std::string &function_name)
    {
        switch (result)
        {
        case NMDLP_COMPILER_RESULT_OK:
            return NMDLP_RESULT_OK;
        case NMDLP_COMPILER_RESULT_MEMORY_ALLOCATION_ERROR:
            throw std::runtime_error(function_name + ": MEMORY_ALLOCATION_ERROR");
        case NMDLP_COMPILER_RESULT_MODEL_LOADING_ERROR:
            throw std::runtime_error(function_name + ": MODEL_LOADING_ERROR");
        case NMDLP_COMPILER_RESULT_INVALID_PARAMETER:
            throw std::runtime_error(function_name + ": INVALID_PARAMETER");
        case NMDLP_COMPILER_RESULT_INVALID_MODEL:
            throw std::runtime_error(function_name + ": INVALID_MODEL");
        case NMDLP_COMPILER_RESULT_UNSUPPORTED_OPERATION:
```

```
            throw std::runtime_error(function_name + ": UNSUPPORTED_OPERATION");
        default:
            throw std::runtime_error(function_name + ": UNKNOWN ERROR");
        }
    }

    NMDLP_RESULT Call(NMDLP_RESULT result, const std::string &function_name)
    {
        switch (result)
        {
        case NMDLP_RESULT_OK:
            return NMDLP_RESULT_OK;
        case NMDLP_RESULT_INVALID_FUNC_PARAMETER:
            throw std::runtime_error(function_name + ": INVALID_FUNC_PARAMETER");
        case NMDLP_RESULT_NO_LOAD_LIBRARY:
            throw std::runtime_error(function_name + ": NO_LOAD_LIBRARY");
        case NMDLP_RESULT_NO_BOARD:
            throw std::runtime_error(function_name + ": NO_BOARD");
        case NMDLP_RESULT_BOARD_RESET_ERROR:
            throw std::runtime_error(function_name + ": BOARD_RESET_ERROR");
        case NMDLP_RESULT_INIT_CODE_LOADING_ERROR:
            throw std::runtime_error(function_name + ": INIT_CODE_LOADING_ERROR");
        case NMDLP_RESULT_CORE_HANDLE_RETRIEVAL_ERROR:
            throw std::runtime_error(function_name + ": CORE_HANDLE_RETRIEVAL_ERROR");
        case NMDLP_RESULT_FILE_LOADING_ERROR:
            throw std::runtime_error(function_name + ": FILE_LOADING_ERROR");
        case NMDLP_RESULT_MEMORY_WRITE_ERROR:
            throw std::runtime_error(function_name + ": MEMORY_WRITE_ERROR");
        case NMDLP_RESULT_MEMORY_READ_ERROR:
            throw std::runtime_error(function_name + ": MEMORY_READ_ERROR");
        case NMDLP_RESULT_MEMORY_ALLOCATION_ERROR:
            throw std::runtime_error(function_name + ": MEMORY_ALLOCATION_ERROR");
        case NMDLP_RESULT_MODEL_LOADING_ERROR:
            throw std::runtime_error(function_name + ": MODEL_LOADING_ERROR");
        case NMDLP_RESULT_INVALID_MODEL:
            throw std::runtime_error(function_name + ": INVALID_MODEL");
        case NMDLP_RESULT_BOARD_SYNC_ERROR:
            throw std::runtime_error(function_name + ": BOARD_SYNC_ERROR");
        case NMDLP_RESULT_BOARD_MEMORY_ALLOCATION_ERROR:
            throw std::runtime_error(function_name + ":
BOARD_MEMORY_ALLOCATION_ERROR");
        case NMDLP_RESULT_NN_CREATION_ERROR:
            throw std::runtime_error(function_name + ": NN_CREATION_ERROR");
        case NMDLP_RESULT_NN_LOADING_ERROR:
            throw std::runtime_error(function_name + ": NN_LOADING_ERROR");
        case NMDLP_RESULT_NN_INFO_RETRIEVAL_ERROR:
            throw std::runtime_error(function_name + ": NN_INFO_RETRIEVAL_ERROR");
        case NMDLP_RESULT_MODEL_IS_TOO_BIG:
            throw std::runtime_error(function_name + ": MODEL_IS_TOO_BIG");
        case NMDLP_RESULT_NOT_INITIALIZED:
            throw std::runtime_error(function_name + ": NOT_INITIALIZED");
```

```
        case NMDLP_RESULT_INCOMPLETE:
            throw std::runtime_error(function_name + ": INCOMPLETE");
        case NMDLP_RESULT_UNKNOWN_ERROR:
            throw std::runtime_error(function_name + ": UNKNOWN_ERROR");
        default:
            throw std::runtime_error(function_name + ": UNKNOWN ERROR");
        };
    }

    template <typename T>
    std::vector<T> ReadFile(const std::string &filename)
    {
        std::ifstream ifs(filename, std::ios::binary | std::ios::ate);
        if (!ifs.is_open())
        {
            throw std::runtime_error("Unable to open input file: " + filename);
        }
        auto fsize = static_cast<std::size_t>(ifs.tellg());
        ifs.seekg(0);
        std::vector<T> data(fsize / sizeof(T));
        ifs.read(reinterpret_cast<char *>(data.data()), data.size() * sizeof(T));
        return data;
    }

    void ShowNMDLVersion()
    {
        std::uint32_t major = 0;
        std::uint32_t minor = 0;
        std::uint32_t patch = 0;
        Call(NMDLP_GetLibVersion(&major, &minor, &patch), "GetLibVersion");
        std::cout << "Lib version: " << major << "." << minor
                  << "." << patch << std::endl;
    }

    void CheckBoard(std::uint32_t required_board_type)
    {
        std::uint32_t boards = 0;
        std::uint32_t board_number = -1;
        Call(NMDLP_GetBoardCount(required_board_type, &boards), "GetBoardCount");
        std::cout << "Detected boards: " << boards << std::endl;
        if (!boards)
        {
            throw std::runtime_error("Board not found");
        }
    }

#ifdef _USE_DARKNET_
    std::vector<float> CompileModel(
        const std::string &config_filename,
        const std::string &weights_filename,
        bool is_multi_unit)
```

```
    {
        float *nm_model = nullptr;
        std::uint32_t nm_model_floats = 0u;
        auto config = ReadFile<char>(config_filename);
        auto weights = ReadFile<char>(weights_filename);
        Call(
            NMDLP_COMPILER_CompileDarkNet(
                is_multi_unit,
                config.data(),
                config.size(),
                weights.data(),
                weights.size(),
                &nm_model,
                &nm_model_floats
            ),
            "CompileONNX"
        );
        std::vector<float> result(nm_model, nm_model + nm_model_floats);
        NMDLP_COMPILER_FreeModel(nm_model);
        return result;
    }
#else
    std::vector<float> CompileModel(const std::string &model_filename, bool
is_multi_unit)
    {
        float *nm_model = nullptr;
        std::uint32_t nm_model_floats = 0u;
        auto model = ReadFile<char>(model_filename);
        Call(
            NMDLP_COMPILER_CompileONNX(
                is_multi_unit,
                model.data(),
                static_cast<std::uint32_t>(model.size()),
                &nm_model,
                &nm_model_floats),
            "CompileONNX");
        std::vector<float> result(nm_model, nm_model + nm_model_floats);
        NMDLP_COMPILER_FreeModel(nm_model);
        return result;
    }
#endif

    NMDLP_ModelInfo GetModelInformation(NMDLP_HANDLE nmdlp, std::uint32_t unit_num)
    {
        NMDLP_ModelInfo model_info;
        Call(NMDLP_GetModelInfo(nmdlp, unit_num, &model_info), "GetModelInfo");
        std::cout << "Input tensor number: " << model_info.input_tensor_num << std
::endl;
        for (std::size_t i = 0; i < model_info.input_tensor_num; ++i)
        {
            std::cout << "Input tensor " << i << ": " << model_info.input_tensors[i
```

```
].width << ", " << model_info.input_tensors[i].height << ", " << model_info
.input_tensors[i].depth << std::endl;
        }
        std::cout << "Output tensor number: " << model_info.output_tensor_num << std
::endl;
        for (std::size_t i = 0; i < model_info.output_tensor_num; ++i)
        {
            std::cout << "Output tensor " << i << ": " << model_info.output_tensors[
i].width << ", " << model_info.output_tensors[i].height << ", " << model_info
.output_tensors[i].depth << std::endl;
        }
        return model_info;
    }

    std::vector<float> PrepareInput(
        const std::string &filename,
        std::uint32_t width,
        std::uint32_t height,
        std::uint32_t color_format,
        const float rgb_divider[3],
        const float rgb_adder[3])
    {
        auto bmp_frame = ReadFile<char>(filename);
        std::vector<float> input(NMDLP_IMAGE_CONVERTER_RequiredSize(width, height,
color_format));
        auto result = NMDLP_IMAGE_CONVERTER_Convert(
            bmp_frame.data(),
            input.data(),
            static_cast<std::uint32_t>(bmp_frame.size()),
            width,
            height,
            color_format,
            rgb_divider,
            rgb_adder
        );
        if (result)
            throw std::runtime_error("Image conversion error");
        return input;
    }

    void WaitForOutput(NMDLP_HANDLE nmdlp, std::uint32_t unit_num, float *outputs[])
    {
        std::uint32_t status = NMDLP_PROCESS_FRAME_STATUS_INCOMPLETE;
        while (status == NMDLP_PROCESS_FRAME_STATUS_INCOMPLETE)
        {
            NMDLP_GetStatus(nmdlp, unit_num, &status);
        };
        double fps;

        Call(NMDLP_GetOutput(nmdlp, unit_num, outputs, &fps), "GetOutput");
        std::cout << "First four result values:" << std::endl;
```

```
        for (std::size_t i = 0; i < 4; ++i)
        {
            std::cout << outputs[0][i] << std::endl;
        }
        std::cout << "FPS:" << fps << std::endl;
    }

}

int main()
{
    const std::uint32_t BOARD_TYPE = NMDLP_BOARD_TYPE_SIMULATOR;
    // const uint32_t BOARD_TYPE = NMDLP_BOARD_TYPE_MC12705;
    // const uint32_t BOARD_TYPE = NMDLP_BOARD_TYPE_NMCARD;
    // const uint32_t BOARD_TYPE = NMDLP_BOARD_TYPE_NMMEZZO;
    // const uint32_t BOARD_TYPE = NMDLP_BOARD_TYPE_NMQUAD;
    const std::string ONNX_MODEL_FILENAME =
        "../ref_data/squeezenet_imagenet/model.onnx";
    const std::string BMP_FRAME_FILENAME =
        "../ref_data/squeezenet_imagenet/frame.bmp";
    const NMDLP_IMAGE_CONVERTER_COLOR_FORMAT IMAGE_CONVERTER_COLOR_FORMAT =
        NMDLP_IMAGE_CONVERTER_COLOR_FORMAT_BGR;
    const float NM_FRAME_RGB_DIVIDER[3] = {1.0f, 1.0f, 1.0f};
    const float NM_FRAME_RGB_ADDER[3] = {0.0f, 0.0f, 0.0f};
    const std::size_t BATCHES = 4;
    const std::size_t FRAMES = 5;

    NMDLP_HANDLE nmdlp = 0;

    try
    {
        std::cout << "Query library version..." << std::endl;
        ShowNMDLVersion();

        std::cout << "Board detection... " << std::endl;
        CheckBoard(BOARD_TYPE);

        std::cout << "NMDLP initialization... " << std::endl;
        Call(NMDLP_Create(&nmdlp), "Create");

        std::cout << "Use multi unit... " << std::endl;

        std::cout << "Compile model... " << std::endl;
#ifdef _USE_DARKNET_
        auto model = CompileModel(
            DARKNET_CONFIG_FILENAME,
            DARKNET_WEIGHTS_FILENAME,
            true);
#else
        auto model = CompileModel(ONNX_MODEL_FILENAME, true);
#endif
```

```
        std::array<const float *, NMDLP_MAX_UNITS> models = {model.data()};
        Call(NMDLP_Initialize(nmdlp, BOARD_TYPE, 0, 0, models.data()), "Initialize");

        std::cout << "Get model information... " << std::endl;
        auto model_info = GetModelInformation(nmdlp, 0);

        std::cout << "Prepare inputs... " << std::endl;
        auto input = PrepareInput(
            BMP_FRAME_FILENAME,
            model_info.input_tensors[0].width,
            model_info.input_tensors[0].height,
            IMAGE_CONVERTER_COLOR_FORMAT,
            NM_FRAME_RGB_DIVIDER,
            NM_FRAME_RGB_ADDER
        );
        std::array<const float *, 1> inputs = {input.data()};

        std::cout << "Reserve outputs... " << std::endl;
        std::vector<std::vector<float>> output_tensors(model_info.output_tensor_num);
        std::vector<float *> outputs(model_info.output_tensor_num);
        for (std::size_t i = 0; i < model_info.output_tensor_num; ++i)
        {
            output_tensors[i].resize(static_cast<std::size_t>(
                                        model_info.output_tensors[i].width) *
                                    model_info.output_tensors[i].height *
                                    model_info.output_tensors[i].depth);
            outputs[i] = output_tensors[i].data();
        }

        std::cout << "Process inputs... " << std::endl;
        // for(std::size_t i = 0; i < FRAMES; ++i) {
        //  Call(NMDLP_Process(nmdlp, 0, inputs.data()), "Process");
        //  WaitForOutput(nmdlp, 0, outputs.data());
        // }
        NMDLP_Release(nmdlp);

        std::cout << "Process batch... " << std::endl;

        std::cout << "Compile model... " << std::endl;
#ifdef _USE_DARKNET_
        model = CompileModel(
            DARKNET_CONFIG_FILENAME,
            DARKNET_WEIGHTS_FILENAME,
            false
        );
#else
        model = CompileModel(ONNX_MODEL_FILENAME, false);
#endif
        models = {model.data(), model.data(), model.data(), model.data()};
        Call(NMDLP_Initialize(nmdlp, BOARD_TYPE, 0, 0, models.data()), "Initialize");
```

```
        // std::uint32_t unit_num = 0;

        // for(int i = 0; i < 10; i++){
        //  Call(NMDLP_Process(nmdlp, unit_num, inputs.data()), "ProcessFrame");
        //  WaitForOutput(nmdlp, unit_num, outputs.data());
        // }
        std::uint32_t cnt_in = 0;
        std::uint32_t cnt_out = 0;
        for (auto i = 0u; i < BATCHES; ++i)
        {
            Call(NMDLP_Process(nmdlp, (cnt_in++) % BATCHES, inputs.data()),
                 "ProcessFrame");
        }
        for (auto i = BATCHES; i < FRAMES; ++i)
        {
            WaitForOutput(nmdlp, (cnt_out++) % BATCHES, outputs.data());
            Call(NMDLP_Process(nmdlp, (cnt_in++) % BATCHES, inputs.data()),
                 "ProcessFrame");
        }
        for (auto i = 0u; i < BATCHES; ++i)
        {
            WaitForOutput(nmdlp, (cnt_out++) % BATCHES, outputs.data());
        }
    }
    catch (std::exception &e)
    {
        std::cerr << e.what() << std::endl;
    }
    NMDLP_Release(nmdlp);
    NMDLP_Destroy(nmdlp);

    return 0;
}
```

Здесь выполняются следующие действия:

- Подключение заголовочных файлов. *"nmdlp.h"* - описание библиотеки нейросетевой обработки, *"nmdlp_compiler.h"* - описание библиотеки для компиляции моделей, *"nmdlp_image_converter.h"* - описание библиотеки для подготовки изображений.
- Функция-обёртка для вызовов функций библиотеки компиляции моделей. В случае ошибки формирует вывод об ошибке и инициирует исключение.
- Функция-обёртка для вызовов функций библиотеки нейросетевой обработки. В случае ошибки формирует вывод об ошибке и инициирует исключение.
- Универсальная функция для чтения данных из файла в вектор.
- Функция вывода версии NMDLP NMDLP_GetLibVersion.
- Функция проверки наличия модуля заданного типа. Фактически производится запрос количества обнаруженных модулей заданного типа - NMDLP_GetBoardCount.

- Функция компиляции исходной модели. Вызывается функция NMDLP_COMPILER_CompileONNX. Здесь происходит выделение памяти внутри функции компиляции, поэтому после работы выделенная память освобождается вызовом NMDLP_COMPILER_FreeModel, а результат компиляции копируется в возвращаемый вектор *float*. В целевой программе можно выполнить предварительную компиляцию модели с помощью утилиты *nmdlp_compiler_console* так, как описано в разделе Компиляция модели.
- Функция получения и вывода информации о параметрах входных и выходных тензоров. Используется вызов NMDLP_GetModelInfo.
- Функция подготовки кадра. Здесь выполняется чтение изображения из файла, его декодирование и предобработка (NMDLP_IMAGE_CONVERTER_Convert). Описание предобработки приводится в разделе Подготовка изображений.
- Функция ожидания обработки кадра. Принимает номер кластера на котором производится обработка (*unit_num*) и буфер-вектор для хранения результата обработки. Функция блокируется в цикле запроса статуса обработки (NMDLP_GetStatus). После получения статуса *NMDLP_PROCESS_FRAME_STATUS_FREE* производится копирование результата вызовом NMDLP_GetOutput.
- Задание типа модуля ускорителя. В примере используется симулятор модуля MC127.05. Для работы с другими типами раскомментируйте соответствующую строку.
- Задаются имена файлов с исходными моделью и изображением. Задаются параметры для подготовки изображения. Для исходной модели требуется подготовить изображение с пикселями в формате blue-green-red. Каждый пиксель модифицируется по формуле Y = X / 1.0 - 114. Это преобразование необходимо для обработки модели *squeezenet*. Для других моделей необходимо использовать соответствующие параметры, которые определяются при создании модели и являются исходными данными, привязанными к конкретной модели. Подробнее о подготовке изображений см. в разделе Подготовка изображений.
- Задаётся количество кластеров при пакетной обработке.
- Задаётся количество обрабатываемых кадров. В примере производится многократная обработка одного кадра.

Далее в главной функции выполняется последовательный вызов описанных вспомогательных функций:

- Инициализация NMDLP. Посредством вызова функции NMDLP_Create осуществляется инициализация внутренних структур библиотеки NMDLP.
- Пример последовательной обработки кадров в режиме разделения данных по кластерам *"multi unit"* (см. раздел Режимы обработки).
- Компиляция модели.
- инициализация. В функции NMDLP_Initialize производится загрузка скомпилированной модели в выбранный ускоритель.
- формирование входного тензора.
- резервирование буферов для выходных тензоров.
- обработка и получение результата.

- Пример последовательной обработки кадров в режиме пакетной обработки *"batch mode"* (см. раздел Режимы обработки).
- При завершении работы освобождаются выделенные ресурсы (NMDLP_Release и NMDLP_Destroy).

8. Описание идентификаторов, функций и структур NMDL+

8.1. Идентификаторы и структуры

8.1.1. NMDLP_BOARD_TYPE

Типы модулей.

```
typedef enum tagNMDLP_BOARD_TYPE {
    NMDLP_BOARD_TYPE_SIMULATOR,
    NMDLP_BOARD_TYPE_MC12705,
    NMDLP_BOARD_TYPE_NMCARD,
    NMDLP_BOARD_TYPE_NMMEZZO,
    NMDLP_BOARD_TYPE_NMQUAD
} NMDLP_BOARD_TYPE;
```

- *NMDLP_BOARD_TYPE_SIMULATOR* - симулятор модуля *MC127.05*
- *NMDLP_BOARD_TYPE_MC12705* - модуль *MC127.05*. Представляет собой серверный вычислитель со *СБИС К1879ВМ8Я*, подключаемый к стандартным портам PCI-E на материнской плате.
- *NMDLP_BOARD_TYPE_NMCARD* - модуль *NMCard*. Представляет собой спецвычислитель со *СБИС К1879ВМ8Я*, подключаемый в слот расширения PCIe на материнской плате компьютера.
- *NMDLP_BOARD_TYPE_NMMEZZO* - модуль *NMMezzo*. Представляет собой спецвычислитель со *СБИС К1879ВМ8Я*, подключаемый по шине PCIe к несущей плате пользователя.
- *NMDLP_BOARD_TYPE_NMQUAD* - модуль *NMQuad*. Представляет собой спецвычислитель со *СБИС К1879ВМ8Я*, подключаемый в слот расширения PCIe на материнской плате компьютера.

8.1.2. NMDLP_ModelInfo

Структура с информацией о модели.

```
typedef struct tagNMDLP_ModelInfo {
    unsigned int input_tensor_num;
    NMDLP_Tensor input_tensors[NMDLP_MAX_INPUT_TENSORS];
    unsigned int output_tensor_num;
    NMDLP_Tensor output_tensors[NMDLP_MAX_OUTPUT_TENSORS];
} NMDLP_ModelInfo;
```

- *input_tensor_num* - количество входных тензоров

- *input_tensors* - входные тензоры
- *output_tensor_num* - количество выходных тензоров
- *output_tensors* - выходные тензоры

NMDLP_MAX_INPUT_TENSORS - максимальное количество входных тензоров.

NMDLP_MAX_OUTPUT_TENSORS - максимальное количество выходных тензоров.

8.1.3. NMDLP_PROCESS_FRAME_STATUS

Идентификатор статуса обработки кадра.

```
typedef enum tagNMDLP_PROCESS_FRAME_STATUS {
    NMDLP_PROCESS_FRAME_STATUS_FREE,
    NMDLP_PROCESS_FRAME_STATUS_INCOMPLETE
} NMDLP_PROCESS_FRAME_STATUS;
```

- *NMDLP_PROCESS_FRAME_STATUS_FREE* - обработка кадра не производится
- *NMDLP_PROCESS_FRAME_STATUS_INCOMPLETE* - выполняется обработка кадра

8.1.4. NMDLP_RESULT

Возвращаемый результат.

```
typedef enum tagNMDLP_RESULT {
    NMDLP_RESULT_OK,
    NMDLP_RESULT_INVALID_FUNC_PARAMETER,
    NMDLP_RESULT_NO_BOARD,
    NMDLP_RESULT_BOARD_RESET_ERROR,
    NMDLP_RESULT_INIT_CODE_LOADING_ERROR,
    NMDLP_RESULT_CORE_HANDLE_RETRIEVAL_ERROR,
    NMDLP_RESULT_FILE_LOADING_ERROR,
    NMDLP_RESULT_MEMORY_WRITE_ERROR,
    NMDLP_RESULT_MEMORY_READ_ERROR,
    NMDLP_RESULT_MEMORY_ALLOCATION_ERROR,
    NMDLP_RESULT_MODEL_LOADING_ERROR,
    NMDLP_RESULT_INVALID_MODEL,
    NMDLP_RESULT_BOARD_SYNC_ERROR,
    NMDLP_RESULT_BOARD_MEMORY_ALLOCATION_ERROR,
    NMDLP_RESULT_NN_CREATION_ERROR,
    NMDLP_RESULT_NN_LOADING_ERROR,
    NMDLP_RESULT_NN_INFO_RETRIEVAL_ERROR,
    NMDLP_RESULT_MODEL_IS_TOO_BIG,
    NMDLP_RESULT_NOT_INITIALIZED,
    NMDLP_RESULT_INCOMPLETE,
    NMDLP_RESULT_UNKNOWN_ERROR
} NMDLP_RESULT;
```

- *NMDLP_RESULT_OK* - нет ошибок
- *NMDLP_RESULT_INVALID_FUNC_PARAMETER* - неверный параметр
- *NMDLP_RESULT_NO_BOARD* - нет модуля
- *NMDLP_RESULT_BOARD_RESET_ERROR* - ошибка сброса модуля
- *NMDLP_RESULT_INIT_CODE_LOADING_ERROR* - ошибка загрузки кода инициализации библиотеки загрузки и обмена
- *NMDLP_RESULT_CORE_HANDLE_RETRIEVAL_ERROR* - ошибка получения идентификатора вычислителя
- *NMDLP_RESULT_FILE_LOADING_ERROR* - ошибка загрузки программного образа
- *NMDLP_RESULT_MEMORY_WRITE_ERROR* - ошибка записи в память модуля
- *NMDLP_RESULT_MEMORY_READ_ERROR* - ошибка чтения из памяти модуля
- *NMDLP_RESULT_MEMORY_ALLOCATION_ERROR* - ошибка выделения памяти
- *NMDLP_RESULT_MODEL_LOADING_ERROR* - ошибка загрузки модели нейронной сети
- *NMDLP_RESULT_INVALID_MODEL* - ошибка в модели
- *NMDLP_RESULT_BOARD_SYNC_ERROR* - ошибка синхронизации с модулем
- *NMDLP_RESULT_BOARD_MEMORY_ALLOCATION_ERROR* - ошибка выделения памяти на модуле
- *NMDLP_RESULT_NN_CREATION_ERROR* - ошибка создания модели на модуле
- *NMDLP_RESULT_NN_LOADING_ERROR* - ошибка загрузки модели нейронной сети
- *NMDLP_RESULT_NN_INFO_RETRIEVAL_ERROR* - ошибка запроса информации о модели
- *NMDLP_RESULT_MODEL_IS_TOO_BIG* - модель не может быть размещена в памяти модуля
- *NMDLP_RESULT_NOT_INITIALIZED* - библиотека функций NMDLP не инициализирована
- *NMDLP_RESULT_INCOMPLETE* - устройство находится в состоянии обработки кадра
- *NMDLP_RESULT_UNKNOWN_ERROR* - неизвестная ошибка

8.1.5. NMDLP_Tensor

Структура, описывающая тензор.

```
typedef struct tagNMDLP_Tensor {
    unsigned int width;
    unsigned int height;
    unsigned int depth;
} NMDLP_Tensor;
```

- *width* - ширина тензора
- *height* - высота тензора
- *depth* - глубина тензора

8.2. Функции

8.2.1. NMDLP_Blink

Светодиодная индикация для идентификации модуля.

```
NMDLP_RESULT NMDLP_Blink(
    unsigned int board_type,
    unsigned int board_number
);
```

- *board_type* [input] - тип модуля, на котором вызывается процедура светодиодной индикации. Одно из значений перечисления *NMDLP_BOARD_TYPE*
- *board_number* [input] - порядковый номер модуля, на котором вызывается процедура светодиодной индикации

8.2.2. NMDLP_Create

Создание экземпляра NMDLP и получение идентификатора экземпляра *NMDLP*.

```
NMDLP_RESULT NMDLP_Create(
    NMDLP_HANDLE *nmdlp
);
```

- *nmdlp* [output] - идентификатор экземпляра NMDLP

После работы с экземпляром NMDLP необходимо освободить выделенные ресурсы вызовом NMDLP_Release

8.2.3. NMDLP_Destroy

Удаление экземпляра NMDLP.

```
void NMDLP_Destroy(
    NMDLP_HANDLE nmdlp
);
```

- *nmdlp* [input] - идентификатор экземпляра NMDLP

Функция вызывается для освобождения ресурсов, выделенных при вызовах NMDLP_Create и NMDLP_Initialize.

8.2.4. NMDLP_GetBoardCount

Запрос количества обнаруженных модулей заданного типа.

```
NMDLP_RESULT NMDLP_GetBoardCount(
    unsigned int board_type,
    unsigned int *boards
);
```

- *board_type* [input] - тип опрашиваемых модулей. Одно из значений перечисления *NMDLP_BOARD_TYPE*
- *boards* [output] - количество обнаруженных модулей

Для симулятора *MC127.05* (тип *NMDLP_BOARD_TYPE_SIMULATOR*) всегда обнаруживается один модуль.

***ВНИМАНИЕ!* При определении количества модулей производится обращение к ним со сбросом. Запущенные на модуле программы прекратят свою работу.**

8.2.5. NMDLP_GetLibVersion

Запрос версии библиотеки *NMDL+*.

```
NMDLP_RESULT NMDLP_GetLibVersion(
    unsigned int *major,
    unsigned int *minor,
    unsigned int *patch
);
```

- *major* [output] - старший номер версии
- *minor* [output] - младший номер версии
- *patch* [output] - номер патча

8.2.6. NMDLP_GetModelInfo

Запрос информации о модели.

```
NMDLP_RESULT NMDLP_GetModelInfo(
    NMDLP_HANDLE nmdlp,
    unsigned int unit_num,
    NMDLP_ModelInfo *model_info
);
```

- *nmdlp* [input] - дескриптор NMDLP
- *unit_num* [input] - номер юнита, для которого запрашивается информация
- *model_info* [output] - информация о модели

8.2.7. NMDLP_GetOutput

Запрос результата обработки.

```
NMDLP_RESULT NMDLP_GetOutput(
    NMDLP_HANDLE nmdlp,
    unsigned int unit_num,
    float *outputs[],
    double *fps
);
```

- *nmdlp* [input] - идентификатор экземпляра NMDLP
- *unit_num* [input] - номер юнита, для которого запрашивается результат обработки
- *outputs* [output] - массив указателей на буферы для выходных тензоров
    - Выходные тензоры нумеруются друг за другом в порядке их появления в графе обработки, то есть упорядоченные по уровням в графе
    - Память под буферы выделяется пользователем
    - Размер каждого тензора вычисляется как произведение его ширины, высоты и глубины (каналы)
    - Количество и параметры тензоров определяются в структуре *NMDLP_ModelInfo*, которую можно получить вызовом *NMDLP_GetModelInfo*
- *fps* [output] - производительность обработки (кадров в секунду). Может принимать значение 0.

Пример вызова *NMDLP_GetOutput* с выделением памяти под результат на C++.

```
NMDLP_ModelInfo model_info;
NMDLP_GetModelInfo(nmdlp_handle, unit_num, &model_info);
std::vector<std::vector<float>> output_tensors(model_info.output_tensor_num);
std::vector<float*> outputs(model_info.output_tensor_num);
for(std::size_t i = 0; i < model_info.output_tensor_num; ++i) {
    output_tensors[i].resize(static_cast<std::size_t>(
            model_info.output_tensors[i].width) *
            model_info.output_tensors[i].height *
            model_info.output_tensors[i].depth);
    outputs[i] = output_tensors[i].data();
        }
double fps;
NMDLP_GetOutput(nmdlp_handle, unit_num, outputs.data(), &fps);
```

8.2.8. NMDLP_GetStatus

Запрос статуса обработки. Функция вызывается для проверки окончания обработки кадра.

```
NMDLP_RESULT NMDLP_GetStatus(
    NMDLP_HANDLE nmdlp,
    unsigned int unit_num,
    unsigned int *status
);
```

- *nmdlp* [input] - идентификатор экземпляра NMDLP
- *unit_num* [input] - номер кластера, для которого запрашивается статус обработки
- *status* [output] - состояное кластера в момент вызова функции. Одно из значений перечисления *NMDLP_PROCESS_FRAME_STATUS*

Пример использования *NMDLP_GetStatus* для организации поллинга. Блокирующая функция для ожидания окончания обработки:

```
auto WaitForOutput(NMDLP_HANDLE nmdlp, std::uint32_t unit_num, float *output[]) {
    std::uint32_t status = NMDLP_PROCESS_FRAME_STATUS_INCOMPLETE;
    while(status == NMDLP_PROCESS_FRAME_STATUS_INCOMPLETE) {
        NMDLP_GetStatus(nmdlp, unit_num, &status);
    };
    double fps;
    NMDLP_GetOutput(nmdlp, unit_num, output, &fps);
    return fps;
}
```

8.2.9. NMDLP_Initialize

Инициализация NMDLP.

```
NMDLP_RESULT NMDLP_Initialize(
    NMDLP_HANDLE nmdlp,
    unsigned int board_type,
    unsigned int board_number,
    unsigned int proc_number,
    const float *model[NMDLP_MAX_UNITS],
);
```

- *nmdlp* [input] - идентификатор экземпляра *NMDL+*
- *board_type* [input] - тип инициализируемого модуля. Одно из значений перечисления *NMDLP_BOARD_TYPE*
- *board_number* [input] - порядковый номер инициализируемого модуля
- *proc_number* [input] - порядковый номер инициализируемого процессора. Для симулятора необходимо установить 0.
- *model* [input] - массив указателей на буферы, содержащие образы скомпилированных моделей

Для устройств на базе процессора *К1879ВМ8Я* - *MC127.05*, *NMCard*, *NMMezzo*, *NMQuad* и симуляторе возможно задать несколько моделей, так как здесь можно использовать до четырёх юнитов.
Подробнее о режимах обработки См. раздел Режимы обработки.

Например:

```
// NMDLP_HANDLE nmdlp - library descriptor created in NMDLP_Create.
// std::vector<float> model_0 - unit 0 compiled model data.
// std::vector<float> model_1 - unit 1 compiled model data.
// std::vector<float> model_2 - unit 2 compiled model data.
// std::vector<float> model_3 - unit 3 compiled model data.
std::array<const float*, NMDLP_MAX_UNITS> models = {
    model_0.data(), model_1.data(), model_2.data(), model_3.data()
};
NMDLP_Initialize(nmdlp, NMDLP_BOARD_TYPE_NMCARD, 0, models.data());
```

Если модель скомпилирована для устройств на базе процессора *К1879ВМ8Я* для прогона в режиме *"multi unit"*, то нужно задать только одну модель - эта модель содержит в себе данные для инициализации всех четырёх юнитов.
Подробнее о режимах обработки См. раздел Режимы обработки.

Например:

```
// NMDLP_HANDLE nmdlp - library descriptor created in NMDLP_Create.
// std::vector<float> model - compiled model data  to run in "multi unit" mode.
std::array<const float*, NMDLP_MAX_UNITS> models = {
    model.data()
};
NMDLP_Initialize(nmdlp, NMDLP_BOARD_TYPE_NMCARD, 0, models.data());
```

После работы с функциями NMDLP необходимо освободить выделенные ресурсы вызовом функции деинициализации NMDLP_Release.

8.2.10. NMDLP_Process

Обработка входных тензоров.

```
NMDLP_RESULT NMDLP_Process(
    NMDLP_HANDLE nmdlp,
    unsigned int unit_num,
    const float *frame[]
);
```

- *nmdlp* [input] - идентификатор экземпляра *NMDL+*
- *unit_num* [input] - номер юнита на котором запускается обработка. В режиме "multi unit"

необходимо установить 0

- *frame* [input] - массив, содержащий указатели на буферы со входными тензорами

Количество и геометрия входных тензоров должно соответствовать загруженной модели нейронной сети.

Пример вызова:

```
// NMDLP_HANDLE nmdlp - library descriptor created in NMDLP_Create.
// std::vector<float> input_tensor_0 - first input tensor data.
// std::vector<float> input_tensor_1 - second input tensor data.
std::array<const float*, 2> input_tensors = {
    input_tensor_0.data(), input_tensor_1.data()
};
NMDLP_Process(nmdlp, 0, input_tensors.data());
```

8.2.11. NMDLP_Release

Деинициализация NMDLP.

```
void NMDLP_Release(
    NMDLP_HANDLE nmdlp
);
```

- nmdlp [input] - идентификатор экземпляра *NMDL+*

9. Описание идентификаторов и функций nmdlp_compiler

9.1. Идентификаторы

9.1.1. NMDLP_COMPILER_RESULT

Возвращаемый результат.

```
typedef enum tagNMDLP_COMPILER_RESULT {
    NMDLP_COMPILER_RESULT_OK,
    NMDLP_COMPILER_RESULT_MEMORY_ALLOCATION_ERROR,
    NMDLP_COMPILER_RESULT_MODEL_LOADING_ERROR,
    NMDLP_COMPILER_RESULT_INVALID_PARAMETER,
    NMDLP_COMPILER_RESULT_INVALID_MODEL,
    NMDLP_COMPILER_RESULT_UNSUPPORTED_OPERATION
} NMDLP_COMPILER_RESULT;
```

- *NMDLP_COMPILER_RESULT_OK* - нет ошибок
- *NMDLP_COMPILER_RESULT_MEMORY_ALLOCATION_ERROR* - ошибка выделения памяти
- *NMDLP_COMPILER_RESULT_MODEL_LOADING_ERROR* - ошибка загрузки модели нейронной сети
- *NMDLP_COMPILER_RESULT_INVALID_PARAMETER* - неверный параметр
- *NMDLP_COMPILER_RESULT_INVALID_MODEL* - ошибка в модели
- *NMDLP_COMPILER_RESULT_UNSUPPORTED_OPERATION* - модель содержит неподдерживаемую операцию

9.2. Функции

9.2.1. NMDLP_COMPILER_CompileDarkNet

Компиляция исходной модели в формате DarkNet.

```
NMDLP_COMPILER_RESULT NMDLP_COMPILER_CompileDarkNet(
    unsigned int is_multi_unit,
    const char* src_model,
    unsigned int src_model_size,
    const char* src_weights,
    unsigned int src_weights_size,
    float** dst_model,
    unsigned int* dst_model_floats
);
```

- *is_multi_unit* [input] - флаг использования режима обработки *"multi unit"* (см. раздел Режимы обработки)
  - 0 - не используется
  - 1 - используется
- *src_model* [input] - буфер исходной модели в формате DarkNet, предварительно считанной из файла *.cfg*
- *src_model_size* [input] - размер буфера исходной модели в байтах
- *src_weights* [input] - буфер коэффициентов, предварительно считанный из файла *.weights*
- *src_weights_size* [input] - размер буфера коэффициентов в байтах
- *dst_model* [output] - буфер компилированной модели. Выделение памяти происходит в функции.
- *dst_model_floats* - [output] размер буфера компилированной модели в ***float32***

9.2.2. NMDLP_COMPILER_CompileONNX

Компиляция исходной модели в формате ONNX.

```
NMDLP_COMPILER_RESULT NMDLP_COMPILER_CompileONNX(
    unsigned int is_multi_unit,
    const char* src_model,
    unsigned int src_model_size,
    float** dst_model,
    unsigned int* dst_model_floats
);
```

- *is_multi_unit* [input] - флаг использования режима обработки *"multi unit"* (см. раздел Режимы обработки)
  - 0 - не используется
  - 1 - используется
- *src_model* [input] - буфер исходной модели в формате ONNX, предварительно считанной из файла *.onnx*
- *src_model_size* [input] - размер буфера исходной модели в байтах
- *dst_model* [output] - буфер компилированной модели. Выделение памяти происходит в функции.
- *dst_model_floats* - [output] размер буфера компилированной модели в ***float32***

9.2.3. NMDLP_COMPILER_FreeModel

Освобождение выделенной при компиляции памяти.

```
NMDLP_COMPILER_RESULT NMDLP_COMPILER_FreeModel(char* dst_model);
```

- *dst_model* [input] - буфер освобождаемой памяти. Память для буфера была выделена при вызове *NMDLP_COMPILER_CompileDarkNet* или *NMDLP_COMPILER_CompileONNX*

9.2.4. NMDLP_COMPILER_GetLastError

Возвращает константную строку с описанием последней ошибки.

```
const char *NMDLP_COMPILER_GetLastError();
```

10. Описание идентификаторов и функций nmdlp_image_converter

10.1. Идентификаторы и структуры

10.1.1. NMDLP_IMAGE_CONVERTER_COLOR_FORMAT

Идентификаторы формата пикселя. Описывает порядок следования цветовых компонент RGB в пикселе.

```
typedef enum tagNMDLP_IMAGE_CONVERTER_COLOR_FORMAT {
    NMDLP_IMAGE_CONVERTER_COLOR_FORMAT_RGB,
    NMDLP_IMAGE_CONVERTER_COLOR_FORMAT_RBG,
    NMDLP_IMAGE_CONVERTER_COLOR_FORMAT_GRB,
    NMDLP_IMAGE_CONVERTER_COLOR_FORMAT_GBR,
    NMDLP_IMAGE_CONVERTER_COLOR_FORMAT_BRG,
    NMDLP_IMAGE_CONVERTER_COLOR_FORMAT_BGR,
    NMDLP_IMAGE_CONVERTER_COLOR_FORMAT_INTENSITY,
} NMDLP_IMAGE_CONVERTER_COLOR_FORMAT;
```

10.2. Функции

10.2.1. NMDLP_IMAGE_CONVERTER_Convert

Подготовка изображения.

```
int NMDLP_IMAGE_CONVERTER_Convert(
    const char* src,
    float* dst,
    unsigned int src_size,
    unsigned int dst_width,
    unsigned int dst_height,
    unsigned int dst_color_format,
    const float rgb_divider[3],
    const float rgb_adder[3]
);
```

- *src* [input] - буфер с образом исходного изображения. Содержимое буфера соответствует содержимому файла изображения. Изображения могут быть представлены в форматах:
  - .bmp
  - .gif
  - .jpg
  - .png
- *dst* [output] - буфер для подготовленного изображения
  - Память для буфера должна быть предварительно выделена
  - Размер буфера должен быть не менше значения, возвращаемого функцией *NMDLP_IMAGE_CONVERTER_RequiredSize*
- *src_size* [input] - размер буфера исходного изображения в байтах
- *dst_width* [input] - ширина подготовленного изображения
- *dst_height* [input] - высота подготовленного изображения
- *dst_color_format* [input] - идентификатор формата пикселя подготовленного изображения. Одно из значений перечисления *NMDLP_IMAGE_CONVERTER_COLOR_FORMAT*
- *rgb_divider* [input] - делитель в выражении *dst = src / divider + adder*. Приведённая операция выполняется над каналами каждого пикселя исходного изображения:
  - *rgb_divider*[0] - для красного канала
  - *rgb_divider*[1] - для зелёного канала
  - *rgb_divider*[2] - для голубого канала
- *rgb_adder* [input] - слагаемое в выражении *dst = src / divider + adder*. Приведённая операция выполняется над каналами каждого пикселя исходного изображения:
  - *rgb_adder*[0] - для красного канала

- *rgb_adder*[1] - для зелёного канала
- *rgb_adder*[2] - для голубого канала

Для изображений с градацией серого, когда поле
dst_color_format = NMDLP_IMAGE_CONVERTER_COLOR_FORMAT_INTENSITY,
используются только делитель *rgb_divider*[0] и слагаемое *rgb_adder*[0].
Остальные делители (*rgb_divider*[1] и *rgb_divider*[2]) и слагаемые (*rgb_adder*[1] и *rgb_adder*[2]) игнорируются и могут быть установлены в любые значения.

Возвращаемое значение:

- 0 - нормальное завершение
- -1 - ошибка.

10.2.2. NMDLP_IMAGE_CONVERTER_RequiredSize

Возвращает размер буфера в элементах *float32* для хранения подготовленного изображения.

```
int NMDLP_IMAGE_CONVERTER_RequiredSize(
    unsigned int dst_width,
    unsigned int dst_height,
    unsigned int dst_color_format
);
```

- *dst_width* [input] - ширина подготовленного изображения
- *dst_height*[input] - высота подготовленного изображения
- *dst_color_format*[input] - идентификатор формата пикселя подготовленного изображения. Одно из значений перечисления *NMDLP_IMAGE_CONVERTER_COLOR_FORMAT*

11. Описание Python API

11.1. Высокоуровневые функции NMDLPCompiler

Предварительно обученная нейронная сеть в формате *ONNX* или *DarkNet* должна быть скомпилирована в бинарный образ в формате *nm8* для того, чтобы её можно было использовать на вычислительном модуле или симуляторе.

Необходимо отметить, что совместимость моделей, скомпилированных в различных версиях, не гарантируется.

По аналогии с компилятором для C/C++, размерности всех тензоров должны быть статическими. Скрипт для фиксации размерностей рассмотрен в главе Компиляция модели.

11.1.1. Compile

```
def Compile(
        model_type: str,
        is_multi_unit: int,
        model_filename: str,
        weights_filename: str = None
    ) -> numpy.ndarray:
```

- *model_type* - тип компилируемой модели *"Onnx"* или *"DarkNet"*, поддерживает любой регистр
- *is_multi_unit* - флаг использования режима *multi unit*
- *model_filename* - название файла с моделью
- *weights_filename* - название файла с весами модели, используется только при работе с моделью в формате DarkNet(по умолчанию задан None)

11.1.2. SaveToFile

```
def SaveToFile(model: numpy.ndarray, model_filename: str) -> None:
```

- *model* - бинарное представление модели в виде массива *numpy.ndarray*
- *model_filename* - название файла, в который модель будет сохранена

11.2. Высокоуровневые функции NMDLPImageConverter

11.2.1. ConvertImage

Возвращает массив типа *numpy.ndarray*, содержащий сконвертированное выходное

изображение (размер с выравниванием по чётному)

```
def ConvertImage(
        image: numpy.ndarray,
        dst_width: int,
        dst_height: int,
        dst_color_format: int,
        rgb_divider: Tuple[float, float, float],
        rgb_adder: Tuple[float, float, float]
    )-> numpy.ndarray:
```

- *image* - массив типа numpu.ndarray, содержащий исходное изображение
- *dst_width* - ширина выходного изображения
- *dst_height* - высота выходного изображения
- *dst_color_format* - формат цвета (*NMDLP_IMAGE_CONVERTER_COLOR_FORMAT*_)
- *rgb_divider* - кортеж, содержащий пределители для пикселов входного изображения
- *rgb_adder* - кортеж, содержащий дополнительное слагаемое для пикселов входного изображения

11.3. NMDLPSession

11.3.1. Инициализация

Инициализация сессиии для инференса.

```
def __init__(
        board_type : int,
        board_number : int,
        proc_number : int,
        models : List[numpy.ndarray]
    ) -> None:
```

- *board_type* - тип инициализируемого модуля. Одно из значений *NMDLP_BOARD_TYPE*_*
- *board_number* - порядковый номер инициализируемого модуля
- *proc_number* - порядковый номер инициализируемого процессора. Для симулятора необходимо установить 0.
- *models* - список скомпилированных моделей:
  - *models[0]* - загружается на 0 юнит
  - *models[1]* - загружается на 1 юнит
  - *models[2]* - загружается на 2 юнит
  - *models[3]* - загружается на 3 юнит

11.3.2. Process

Запуск обработки изображения на указанном юните.

```
def Process(unit_num : int, input : List[np.ndarray]) -> None:
```

- *unit_num* - номер юнита, на котором запускается обработка. В режиме "multi unit" необходимо установить 0
- *input* - набор тензоров, поступающий на вход сети

11.3.3. Status

Статус обработки на юните, True - finished, False - in process.

```
def Status(unit_num : int) -> bool:
```

- *unit_num* - номер юнита. В режиме "multi unit" необходимо установить 0

11.3.4. Wait

Ожидание окончания обработки на юните и получение результата.

```
def Wait(unit_num : int) -> List[np.ndarray]:
```

- *unit_num* - номер юнита. В режиме "multi unit" необходимо установить 0

11.3.5. Fps

Получение fps на юните, устанавливается функцией *Wait*.

```
def Fps(unit_num : int) -> float:
```

- *unit_num* - номер юнита. В режиме "multi unit" необходимо установить 0

11.3.6. Close

Остановка инференс сессии вне деструктора.

```
def Close() -> None:
```

12. Примеры использования высокоуровневого Python API

12.1. Пример компиляции

```
#!/usr/bin/env python3
from nmdlplus import NMDLPCompiler

nmdlp_compiler = NMDLPCompiler()

models_data_path = os.path.join("..", "models")
onnx_model_path = os.path.join(models_data_path, "squeezenet_imagenet", "model.onnx")
darknet_model_path = os.path.join(models_data_path, "yolo_v3_coco", "model.cfg")
darknet_weights_path = os.path.join(models_data_path, "yolo_v3_coco", "model.weights")

### ____________________________________________________________________ ###

onnx_model = nmdlp_compiler.Compile("Onnx", 0, onnx_model_path)
nmdlp_compiler.SaveToFile(onnx_model, "sqeezenet.nm8")
print("Converted ONNX model")

darknet_model = nmdlp_compiler.Compile("DarkNet", 0, darknet_model_path,
darknet_weights_path)
nmdlp_compiler.SaveToFile(darknet_model, "yolo_v3_coco.nm8")
print("Converted DarkNet model")
```

В данном случае модели в формате *ONNX* и *DarkNet* при помощи функции *Compile()* конвертируется в бинарный образ в формате nm8. Сама модель возвращается в формате *numpy.ndarray* и может быть сохранена в файл или же далее использоваться. В данном примере модели сохраняются в бинарные файлы с названиями *sqeezenet.nm8* и *yolo_v3_coco.nm8*.

12.2. Пример преобразования изображения

```
#!/usr/bin/env python3
from nmdlplus import NMDLPImageConverter
from nmdlplus import NMDLP_IMAGE_CONVERTER_COLOR_FORMAT_BGR

import os
import numpy

models_data_path = os.path.join("..", "models")

frame_path = os.path.join(models_data_path, "squeezenet_imagenet", "frame.bmp")
frame_width = 224
frame_height = 224
frame_format = NMDLP_IMAGE_CONVERTER_COLOR_FORMAT_BGR
frame_divider = (1.0, 1.0, 1.0)
frame_adder = (0.0, 0.0, 0.0)

nmdlp_image_converter = NMDLPImageConverter()

frame = numpy.fromfile(frame_path, dtype=numpy.byte)
converted_frame = nmdlp_image_converter.ConvertImage(
    frame,
    frame_width,
    frame_height,
    frame_format,
    frame_divider,
    frame_adder
)
print("Converted frame")
```

12.3. Пример полного инференса с использованием NMDLPSession

Загрузка двух разных моделей на плату *NMCard*, обработка в режиме single unit

- *onnx_compiled_model* - Сконвертированная ONNX модель
- *darknet_compiled_model* - Сконвертированная DarkNet модель
- *onnx_converted_frame* - Подготовленный фрейм для ONNX модели
- *darknet_converted_frame* - Подготовленный фрейм для DarkNet модели

```
from nmdlplus import NMDLPSession
from nmdlplus import NMDLP_BOARD_TYPE_NMCARD

board_type = NMDLP_BOARD_TYPE_NMCARD
board_number = 0
proc_number = 0
multi_unit = False

"""
        ...
        Model conversion
        ...

        ...
        Frame conversion
        ...
"""

models = [onnx_compiled_model, darknet_compiled_model]
nmdl_session = NMDLPSession(board_type, board_number, proc_number, models)
print("Started session")

onnx_inputs = [onnx_converted_frame]
nmdl_session.Process(0, onnx_inputs)

darknet_inputs = [darknet_converted_frame]
nmdl_session.Process(1, darknet_inputs)

output_onnx = nmdl_session.Wait(0)
fps_onnx = nmdl_session.Fps(0)
print("Finished ONNX")

output_darknet = nmdl_session.Wait(1)
fps_darknet = nmdl_session.Fps(1)
print("Finished DarkNet")

# Print first 4 floats in each output tensor
print("----- Output ONNX -----")
print("Fps: {0}".format(fps_onnx))
for tensor in range(len(output_onnx)):
    print("Tensor: {0}".format(tensor))
    for pos in range(4):
        print(output_onnx[tensor][pos])

print("----- Output DarkNet -----")
print("Fps: {0}".format(fps_darknet))
for tensor in range(len(output_darknet)):
    print("Tensor: {0}".format(tensor))
    for pos in range(4):
        print(output_darknet[tensor][pos])
```

13. Описание низкоуровневого Python API

13.1. Низкоуровневые функции NMDLPCompiler

13.1.1. NMDLP_COMPILER_RESULT

Тип возвращаемого результата, соответствует C++ интерфейсу.

```
NMDLP_COMPILER_RESULT = {
    0 : "NMDLP_COMPILER_RESULT_OK",
    1 : "NMDLP_COMPILER_RESULT_MEMORY_ALLOCATION_ERROR",
    2 : "NMDLP_COMPILER_RESULT_MODEL_LOADING_ERROR",
    3 : "NMDLP_COMPILER_RESULT_INVALID_PARAMETER",
    4 : "NMDLP_COMPILER_RESULT_INVALID_MODEL",
    5 : "NMDLP_COMPILER_RESULT_UNSUPPORTED_OPERATION"
}
```

13.1.2. CompileONNX

Компиляция модели в формате *ONNX*.

```
def CompileONNX(
    is_multi_unit: ctypes.c_uint32,
    src_model: ctypes._Pointer[ctypes.c_byte],
    src_model_size: ctypes.c_uint32,
    dst_model: ctypes._Pointer[ctypes.c_float],
    dst_model_floats: ctypes._Pointer[ctypes.c_uint32]) -> NMDLP_COMPILER_RESULT:
```

- *is_multi_unit* - флаг использования режима *multi unit*
- *src_model* - буфер, содержащий исходную модель в формате *ONNX*
- *src_model_size* - размер исходной модели в числах, формата *float32*
- *dst_model* - буфер, выделенный для выходной модели
- *dst_model_floats* - размер выходной модели в числах, формата *float32*

13.1.3. CompileDarkNet

Компиляция модели в формате *DarkNet*.

```
def CompileDarkNet(
    is_multi_unit: ctypes.c_uint32,
    src_model: ctypes._Pointer[ctypes.c_byte],
    src_model_size: ctypes.c_uint32,
    src_weights: ctypes._Pointer[ctypes.c_byte],
    src_weights_size: ctypes.c_uint32,
    dst_model: ctypes._Pointer[ctypes.c_float],
    dst_model_floats: ctypes._Pointer[ctypes.c_uint32]) -> NMDLP_COMPILER_RESULT:
```

- *is_multi_unit* - флаг использования режима *multi unit*
- *src_model* - буфер, содержащий конфигурацию исходной модели в формате *DarkNet*
- *src_model_size* - размер конфигурации исходной модели в числах, формата *float32*
- *src_weights* - буфер, содержащий веса исходной модели в формате *DarkNet*
- *src_weights_size* - размер весов исходной модели в числах, формата *float32*
- *dst_model* - буфер, выделенный для выходной модели
- *dst_model_floats* - размер выходной модели в числах, формата *float32*

13.1.4. FreeModel

Освобождает ресурсы, выделенные под компиляцию модели.

```
def FreeModel(dst_model: ctypes._Pointer[ctypes.c_float]) -> NMDLP_COMPILER_RESULT:
```

- *dst_model* - буфер, содержащий выходную модель (указатель ctypes-style)

13.1.5. GetLastError

Возвращает сообщение об ошибке, в случае некорректного срабатывания компилятора.

```
def GetLastError() -> str:
```

13.2. Низкоуровневые функции NMDLPImageConverter

13.2.1. RequiredSize

Возвращает размер картинки в виде целого числа, равного количеству элементов типа *float32*.

```
def RequiredSize(
        dst_width: ctypes.c_uint32,
        dst_height: ctypes.c_uint32,
        dst_color_format: ctypes.c_uint32
    ) -> ctypes.c_int32:
```

- *dst_width* - ширина изображения
- *dst_height* - высота изображения
- *dst_color_format* - формат цвета (*NMDLP_IMAGE_CONVERTER_COLOR_FORMAT*_*)

13.2.2. ConvertFunc

Преобразует исходное изображение к внутреннему формату, возвращает 0 в результате корректного исполнения.

```
def ConvertFunc(
        src: ctypes._Pointer[ctypes.c_byte],
        dst: ctypes._Pointer[ctypes.c_float],
        src_size: ctypes.c_uint32,
        dst_width: ctypes.c_uint32,
        dst_height: ctypes.c_uint32,
        dst_color_format: ctypes.c_uint32,
        rgb_divider: ctypes._Pointer[ctypes.c_float],
        rgb_adder: ctypes._Pointer[ctypes.c_float]
    ) -> ctypes.c_int32:
```

- *src* - буфер, содержащий исходное изображение
- *dst* - буфер, выделенный под сконвертированное изображение
- *src_size* - размер буфера исходного изображения
- *dst_width* - ширина выходного изображения
- *dst_height* - высота выходного изображения
- *dst_color_format* - формат цвета
- *rgb_divider* - массив, содержащий делители для пикселов входного изображения
- *rgb_adder* - массив, содержащий дополнительное слагаемое для пикселов входного изображения

13.3. Низкоуровневые функции NMDLP

13.3.1. NMDLP_RESULT

Тип возвращаемого результата, соответствует C++ интерфейсу.

```
NMDLP_RESULT = {
    0 : "NMDLP_RESULT_OK",
    1 : "NMDLP_RESULT_INVALID_FUNC_PARAMETER",
    2 : "NMDLP_RESULT_NO_LOAD_LIBRARY",
    3 : "NMDLP_RESULT_NO_BOARD",
    4 : "NMDLP_RESULT_BOARD_RESET_ERROR",
    5 : "NMDLP_RESULT_INIT_CODE_LOADING_ERROR",
    6 : "NMDLP_RESULT_CORE_HANDLE_RETRIEVAL_ERROR",
    7 : "NMDLP_RESULT_FILE_LOADING_ERROR",
    8 : "NMDLP_RESULT_MEMORY_WRITE_ERROR",
    9 : "NMDLP_RESULT_MEMORY_READ_ERROR",
    10: "NMDLP_RESULT_MEMORY_ALLOCATION_ERROR",
    11: "NMDLP_RESULT_MODEL_LOADING_ERROR",
    12: "NMDLP_RESULT_INVALID_MODEL",
    13: "NMDLP_RESULT_BOARD_SYNC_ERROR",
    14: "NMDLP_RESULT_BOARD_MEMORY_ALLOCATION_ERROR",
    15: "NMDLP_RESULT_NN_CREATION_ERROR",
    16: "NMDLP_RESULT_NN_LOADING_ERROR",
    17: "NMDLP_RESULT_NN_INFO_RETRIEVAL_ERROR",
    18: "NMDLP_RESULT_MODEL_IS_TOO_BIG",
    19: "NMDLP_RESULT_NOT_INITIALIZED",
    20: "NMDLP_RESULT_INCOMPLETE",
    21: "NMDLP_RESULT_UNKNOWN_ERROR"
}
```

13.3.2. Blink

Светодиодная индикация вычислительного модуля.

```
def Blink(board_type: ctypes.c_uint32, board_number: ctypes.c_uint32) -> NMDLP_RESULT:
```

- *board_type* - тип вычислительного модуля
- *board_number* - номер вычислительного модуля

13.3.3. GetLibVersion

Запрос на получение версии библиотеки *NMDL+*.

```
def GetLibVersion(major: ctypes._Pointer, minor: ctypes._Pointer, patch: ctypes
._Pointer) -> NMDLP_RESULT:
```

- *major* - ctypes-указатель на старший номер версии
- *minor* - ctypes-указатель на младший номер версии
- *patch* - ctypes-указатель на номер патча

13.3.4. GetBoardCount

Запрос на получение количества вычислительных модулей определенного типа.

```
def GetBoardCount(board_type: ctypes.c_uint32, boards: ctypes._Pointer) ->
NMDLP_RESULT:
```

- *board_type* - тип вычислительного модуля
- *boards* - ctypes-указатель на количество вычислительных модулей

13.3.5. Create

Создание объекта типа *NMDLP*.

```
def Create(nmdlp: ctypes._Pointer) -> NMDLP_RESULT:
```

- *nmdlp* - идентификатор процесса работы с *NMDL+*

13.3.6. Destroy

Уничтожение объекта *NMDLP*.

```
def Destroy(nmdlp: NMDLP_HANDLE) -> None:
```

- *nmdlp* - идентификатор процесса работы с *NMDL+*

13.3.7. Release

Деинициализация объекта *NMDLP*.

```
def Release(nmdlp: NMDLP_HANDLE) -> None:
```

- *nmdlp* - идентификатор процесса работы с *NMDL+*

13.3.8. Initialize

Инициализация вычислительного модуля.

```
def Initialize(
        nmdlp: NMDLP_HANDLE,
        board_type: ctypes.c_uint32,
        board_number: ctypes.c_uint32,
        proc_number: ctypes.c_uint32,
        model: ctypes._Pointer
    ) -> NMDLP_RESULT:
```

- *nmdlp* - идентификатор процесса работы с *NMDL+*
- *board_type* - тип вычислительного модуля
- *board_number* - порядковый номер вычислительного модуля
- *proc_number* - порядковый номер инициализируемого процессора
- *model* - ctypes-массив указателей на буферы, содержащие сконвертированную модель

13.3.9. GetModelInfo

Запрос на получение информации о модели.

```
def GetModelInfo(
        nmdlp: NMDLP_HANDLE,
        unit_num: ctypes.c_uint32,
        model_info: ctypes._Pointer
    ) -> NMDLP_RESULT:
```

- *nmdlp* - идентификатор процесса работы с *NMDL+*
- *unit_num* - номер кластера
- *model_info* - ctypes-указатель на структуру, содержащую данные о модели

13.3.10. Process

Посылка входных данных на вычислительный модуль.

```
def Process(
        nmdlp: NMDLP_HANDLE,
        unit_num: ctypes.c_uint32,
        inputs: ctypes._Pointer
    ) -> NMDLP_RESULT:
```

- *nmdlp* - идентификатор процесса работы с *NMDL+*
- *unit_num* - номер кластера
- *inputs* - массив, содержащий входной тензор (в общем случае - входные тензора)

13.3.11. GetStatus

Запрос статуса обработки тензора.

```
def GetStatus(
        nmdlp: NMDLP_HANDLE,
        unit_num: ctypes.c_uint32,
        status: ctypes._Pointer
    ) -> NMDLP_RESULT:
```

- *nmdlp* - идентификатор процесса работы с *NMDL+*
- *unit_num* - номер кластера
- *status* - ctypes-указатель на переменную, содержащую флаг состояния обработки на вычислительном модуле

13.3.12. GetOutput

Запрос на получение данных с вычислительного модуля.

```
def GetOutput(
        nmdlp: NMDLP_HANDLE,
        unit_num: ctypes.c_uint32,
        outputs: ctypes._Pointer,
        fps: ctypes._Pointer
    ) -> NMDLP_RESULT:
```

- *nmdlp* - идентификатор процесса работы с *NMDL+*
- *unit_num* - номер кластера
- *outputs* - массив, содержащий выходной тензор (в общем случае - выходные тензора)
- *fps* - ctypes-указатель на переменную, содержащую количество кадров в секунду, затрачиваемых на обработку (При работе на симуляторе возвращает ноль)

14. Примеры для низкоуровневого Python API

Важно отметить, что Python API предоставляет возможность работы не только на низком уровне, с C-подобными функциями и типами данных, которые требуют приведения и дополнительного выделения памяти, но и с высокоуровневыми сессиями, которые предельно упрощают работу с данными путем незначительного урезания функционала и позволяют работать исключительно с объектами типа numpy.ndarray.

14.1. Запуск модели в single unit режиме

```
#!/usr/bin/env python3
from nmdlplus import *
import numpy
import ctypes
import os
import shutil

"""
    Simple inference example on single unit
"""

nmdlp = NMDLP()
nmdlp_image_converter = NMDLPImageConverter()
nmdlp_compiler = NMDLPCompiler()

models_data_path = os.path.join("..", "models")
models_data_path = os.path.abspath(models_data_path)
onnx_model_path = os.path.join(models_data_path, "squeezenet_imagenet", "model.onnx")

frame_name = os.path.join(models_data_path, "squeezenet_imagenet", "frame.bmp")
color_format = NMDLP_IMAGE_CONVERTER_COLOR_FORMAT_BGR
divider = (1.0, 1.0, 1.0)
addition = (0.0, 0.0, 0.0)

board_type = NMDLP_BOARD_TYPE_SIMULATOR
board_number = 0
proc_number = 0
unit_num = 0

### _________________________ Build & Save model _________________________ ###

rmdir = lambda dir : shutil.rmtree(dir) if os.path.exists(dir) and os.path.isdir(dir)
else None

rmdir("build")
os.mkdir("build")
os.chdir("build")
```

```
onnx_model = nmdlp_compiler.Compile("Onnx", 0, onnx_model_path)
nmdlp_compiler.SaveToFile(onnx_model, "model.nm8")
print("Converted ONNX model")

model_filename = os.path.abspath("model.nm8")

os.chdir("..")

### _________________________ Inference _________________________ ###

model = numpy.fromfile(model_filename, dtype=numpy.float32) ### loading model form nm8

nmdlp_handle = NMDLP_HANDLE(ctypes.c_int(0))                ### handler creation
nmdlp.Create(ctypes.byref(nmdlp_handle))                     ### handler
initiialization

board_type = ctypes.c_uint32(board_type)
board_number = ctypes.c_uint32(board_number)
proc_number = ctypes.c_uint32(proc_number)
unit_num = ctypes.c_uint32(unit_num)

### init model block
models = (ctypes.POINTER(ctypes.c_float) * NMDLP_MAX_UNITS)()
model0 = model
model1 = []
model2 = []
model3 = []
if len(model0): models[0] = (ctypes.c_float * len(model0))(*model0)
if len(model1): models[1] = (ctypes.c_float * len(model1))(*model1)
if len(model2): models[2] = (ctypes.c_float * len(model2))(*model2)
if len(model3): models[3] = (ctypes.c_float * len(model3))(*model3)
res = nmdlp.Initialize(nmdlp_handle, board_type, board_number, proc_number, models)
### initialized

### reecieving model info from NM module
model_info = NMDLP_ModelInfo()
res = nmdlp.GetModelInfo(nmdlp_handle, unit_num, ctypes.byref(model_info))
#nmdl_object.CheckResult(res)
print("----- Model info -----")
print(f"is_multi_unit: {model_info.is_multi_unit}")
print(f"input_tensor_num: {model_info.input_tensor_num}")
for tn in range(model_info.input_tensor_num):
    print(f"    input_tensor {tn}: w:{model_info.input_tensors[tn].width}
h:{model_info.input_tensors[tn].height} d:{model_info.input_tensors[tn].depth}")
print(f"output_tensor_num: {model_info.output_tensor_num}")
for tn in range(model_info.output_tensor_num):
    print(f"    output_tensor {tn}: w:{model_info.output_tensors[tn].width}
h:{model_info.output_tensors[tn].height} d:{model_info.output_tensors[tn].depth}")
### info loaded
```

```
### image block
image = numpy.fromfile(frame_name, dtype=numpy.byte)###loading image

### conversion to ctypes image parames
dst_width =  ctypes.c_uint32(model_info.input_tensors[0].width)
dst_height =  ctypes.c_uint32(model_info.input_tensors[0].height)
dst_color_format =  ctypes.c_uint32(color_format)
###calculation dst model length
requiredSize = nmdlp_image_converter.RequiredSize(dst_width, dst_height,
dst_color_format)

### conversion python to ctypes and memory allocation
image_ptr = image.ctypes.data_as(ctypes.POINTER(ctypes.c_byte))
result = (ctypes.c_float * requiredSize)()
rgb_divider = (ctypes.c_float * len(divider))(*divider)
rgb_adder = (ctypes.c_float * len(addition))(*addition)

res = nmdlp_image_converter.ConvertFunc(
    image_ptr,
    result,
    len(image),
    dst_width,
    dst_height,
    dst_color_format,
    ctypes.byref(rgb_divider),
    ctypes.byref(rgb_adder)
)
convertedImage = numpy.frombuffer(result, dtype=numpy.float32)
### pic converted

### data processing
input_data = convertedImage.ctypes.data_as(ctypes.POINTER(ctypes.c_float))
res = nmdlp.Process(nmdlp_handle, unit_num, ctypes.byref(input_data))
### output tensor memory allocation
lengths = []
for i in range(model_info.output_tensor_num):
    current_len = model_info.output_tensors[i].width *\
        model_info.output_tensors[i].height * model_info.output_tensors[i].depth
    lengths.append(current_len) ### getting ouput tensors lengths

outputTensorList = []
outputTensors = (ctypes.POINTER(ctypes.c_float) * model_info.output_tensor_num)()
for iIT in range(model_info.output_tensor_num):
    lng = lengths[iIT]
    outputTensor = (ctypes.c_float * lng)()
    outputTensors[iIT] = outputTensor
    outputTensorList.append(outputTensor)

### getting status for data receiving
status = ctypes.c_uint32(NMDLP_PROCESS_FRAME_STATUS_INCOMPLETE)
while status.value == NMDLP_PROCESS_FRAME_STATUS_INCOMPLETE:
```

```
    res = nmdlp.GetStatus(nmdlp_handle, unit_num, ctypes.byref(status))
fps = ctypes.c_double(0)

res = nmdlp.GetOutput(nmdlp_handle, unit_num, outputTensors, ctypes.byref(fps))

print("----- Output -----")
for T in range(len(outputTensorList)):### by range of output tensors
    print("Num of OutputTensor:", T)
    #for i in range(len(outputTensorList[T])):
    for i in range(4):###first 4
        print(outputTensorList[T][i])

### deinit handler
nmdlp.Destroy(nmdlp_handle)
nmdlp.Release(nmdlp_handle)
```

Здесь выполняются следующие действия:

- Инициализация набора основных параметров для обработки изображения нейронной сетью и конвертером
- Компиляция модели в формате ONNX
- Приведение типов к формату ctypes, совестимому с основным C/C++ программным модулем
- Инициализация модели путем создания массива размером 4 (по количеству кластеров) и загрузки модели на 1 кластер при помощи функции *Initialize()* из пакета nmdlp
- Получение данных о модели с использованием команд *GetModelInfo()* и выведение её в консоль
- Загрузка изображения из файла, приведение переменных к типу ctypes, выделение памяти и конверсия картинки с использованием функции *ConvertFunc()* из пакета *image_converter*
- Конверсия изображения в битовый формат и посылка на вычислительный модуль (в данном случае - симулятор) - *Process()*
- Вычисление размеров выходных тензоров (выходного тензора), а также выделение памяти для него - флаг ожидания status выставляется в единицу (INCOMPLETE) и в цикле вызывается функция *GetStatus()*, которая опрашивает устройство, пока оно не завершит работу и не выставит *status* в ноль
- Ожидание окончания обработки на вычислительном модуле и получение результата при помощи функции *GetOutput()*, вывод его в консоль

14.2. Запуск модели в multi unit режиме

```
#!/usr/bin/env python3
from nmdlplus import *

import numpy
```

```
import ctypes
import os

"""
    Multi unit example with compilation
"""

def CreateOutputTensor(output_tensor_num, lengths): ### creating output tensor
    outputTensorList = []
    outputTensors = (ctypes.POINTER(ctypes.c_float) * output_tensor_num)()
    for iIT in range(output_tensor_num):
        lng = lengths[iIT]
        outputTensor = (ctypes.c_float * lng)()
        outputTensors[iIT] = outputTensor
        outputTensorList.append(outputTensor)

    return outputTensors, outputTensorList

def ProcessNumpy(nmdl_object, nmdlp_handle, unit_num, inputTensor): ### processing pic
    unit_num = ctypes.c_uint32(unit_num)
    input_data = inputTensor.ctypes.data_as(ctypes.POINTER(ctypes.c_float))
    res = nmdl_object.Process(nmdlp_handle, unit_num,  ctypes.byref(input_data))

def WaitForOutput(nmdl_object, nmdlp_handle, unit_num, outputTensors): ### blocking
func
    unit_num = ctypes.c_uint32(unit_num)

    fps = ctypes.c_double(0)
    status = ctypes.c_uint32(NMDLP_PROCESS_FRAME_STATUS_INCOMPLETE)
    while status.value == NMDLP_PROCESS_FRAME_STATUS_INCOMPLETE:
        res = nmdl_object.GetStatus(nmdlp_handle, unit_num, ctypes.byref(status))

    res = nmdl_object.GetOutput(nmdlp_handle, unit_num, outputTensors, ctypes.byref
(fps))

    return outputTensors, fps

if __name__ == "__main__":
    models_data_path = os.path.join("..", "models")
    models_data_path = os.path.abspath(models_data_path)
    onnx_model_filename = os.path.join(models_data_path, "squeezenet_imagenet",
"model.onnx")
    frame_name = os.path.join(models_data_path, "squeezenet_imagenet", "frame.bmp")

    nmdl_object = NMDLP()
    image_converter = NMDLPImageConverter()
    nmdlp_compiler = NMDLPCompiler()

    color_format = NMDLP_IMAGE_CONVERTER_COLOR_FORMAT_BGR
    divider = (1.0, 1.0, 1.0)
    addition = (0.0, 0.0, 0.0)
```

```
    is_multi_unit = 1

    board_type = NMDLP_BOARD_TYPE_SIMULATOR
    board_number = 0
    proc_number = 0
    unit_num = 0

    ###
------------------------------------------------------------------------------ ###

    ### model compilation
    model = nmdlp_compiler.Compile("Onnx", is_multi_unit, onnx_model_filename)

    nmdlp_handle = NMDLP_HANDLE(ctypes.c_int(0))### handler creation
    nmdl_object.Create(ctypes.byref(nmdlp_handle))### handler initiialization

    board_type = ctypes.c_uint32(board_type)
    board_number = ctypes.c_uint32(board_number)
    proc_number = ctypes.c_uint32(proc_number)

    ### init model block
    models = (ctypes.POINTER(ctypes.c_float) * NMDLP_MAX_UNITS)()
    model0 = model
    model1 = []
    model2 = []
    model3 = []
    if len(model0): models[0] = (ctypes.c_float * len(model0))(*model0)
    if len(model1): models[1] = (ctypes.c_float * len(model1))(*model1)
    if len(model2): models[2] = (ctypes.c_float * len(model2))(*model2)
    if len(model3): models[3] = (ctypes.c_float * len(model3))(*model3)
    res = nmdl_object.Initialize(nmdlp_handle, board_type, board_number, proc_number,
models)
    ### initialized


    ### reecieving model info from NM module
    model_info = NMDLP_ModelInfo()
    res = nmdl_object.GetModelInfo(nmdlp_handle, unit_num, ctypes.byref(model_info))
    #nmdl_object.CheckResult(res)
    print("----- Model info -----")
    print(f"is_multi_unit: {model_info.is_multi_unit}")
    print(f"input_tensor_num: {model_info.input_tensor_num}")
    for tn in range(model_info.input_tensor_num):
        print(f"    input_tensor {tn}: w:{model_info.input_tensors[tn].width}
h:{model_info.input_tensors[tn].height} d:{model_info.input_tensors[tn].depth}")
    print(f"output_tensor_num: {model_info.output_tensor_num}")
    for tn in range(model_info.output_tensor_num):
        print(f"    output_tensor {tn}: w:{model_info.output_tensors[tn].width}
h:{model_info.output_tensors[tn].height} d:{model_info.output_tensors[tn].depth}")
    ### info loaded
```

```
    ### image block
    image = numpy.fromfile(frame_name, dtype=numpy.byte)###loading image
    convertedImage = image_converter.ConvertImage(image, model_info.input_tensors[0]
.width, model_info.input_tensors[0].height,\
                                                  color_format, divider, addition
)###image conversion with High Level func

    ### output tensor memory allocation
    lengths = []
    for i in range(model_info.output_tensor_num):
        current_len = model_info.output_tensors[i].width *\
            model_info.output_tensors[i].height * model_info.output_tensors[i].depth
        lengths.append(current_len) ### getting ouput tensors lengths

    outputTensors, outputTensorList = CreateOutputTensor(model_info.output_tensor_num,
lengths)

    ### pic converted

    ### multi unit prcoessing
    res = ProcessNumpy(nmdl_object, nmdlp_handle, unit_num, convertedImage)
    outputTensors, fps = WaitForOutput(nmdl_object, nmdlp_handle, unit_num,
outputTensors)

    print("----- Output -----")
    for T in range(len(outputTensorList)):### by range of output tensors
        print("Num of OutputTensor:", T)
        #for i in range(len(outputTensorList[T])):
        for i in range(4):###first 4
            print(outputTensorList[T][i])
    print(f"fps:{fps.value}")

    ### deinit handler
    nmdl_object.Destroy(nmdlp_handle)
    nmdl_object.Release(nmdlp_handle)
```

АО НТЦ "Модуль"
А/Я 166, Москва, 125190, Россия
Тел: +7 **(499) 152-9698**
Факс: +7 **(499) 152-4661**
E-Mail: **rusales@module.ru**
WWW: **http://www.module.ru**