International Summer School on Deep Learning

July 2nd-6th 2018, Gdansk, Poland

Deep Learning
Inference with Movidius™ Neural Compute Stick

Jacek Czaja, Krzysztof Biniaś

Intel Corporation

Introduction to Intel Movidius C API (V2)

Jacek Czaja Krzysztof Binias

Intel Corporation

June 28, 2018

Table of contents

Intel Movidius Neural Compute Stick

NCS is a USB-thumb-drive-sized deep learning machine that you can use to learn AI programming at the edge

- Based on Myriad 2 processor (28 nm)
- 80-150 GFLOPS performance
- Consumes only 1W of power
- Connectivity: USB 3.0 Type-A
- Operating temperature: 0 - 40 C

Intel Movidius Neural Compute Stick

NCS is a USB-thumb-drive-sized deep learning machine that you can use to learn AI programming at the edge

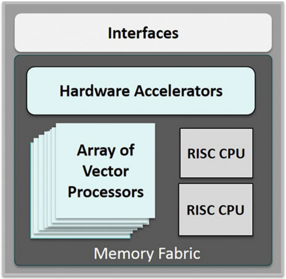
- Based on Myriad 2 processor (28 nm)
- 80-150 GFLOPS performance
- Consumes only 1W of power
- Connectivity: USB 3.0 Type-A
- Operating temperature: 0 - 40 C

- It consumes 1W of power which isl ower than raspberry pie (1.2 W minimum). Throughput of images (data from host) to NCS is limited by USB3 interface. There is operational temperature of 0 - 40 degrees. When temperature is too high frequnce of compute is reduced (throttling)

Intel Movidius Neural Compute Stick

The Intel Movidius Myriad 2 VPU

- An ultra-low power design
- Featuring 12 VLIW programmable SHAVE cores, dedicated vision accelerators and 2 CPUS
- 12 programmable SHAVE cores
- A small-area footprint
- Support for 16/32-bit floating point and 8/16/32-bit integer operations

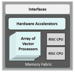


Myriad 2 Vision Processor Unit (VPU)

Intel Movidius Neural Compute Stick

The Intel Movidius Myriad 2 VPU

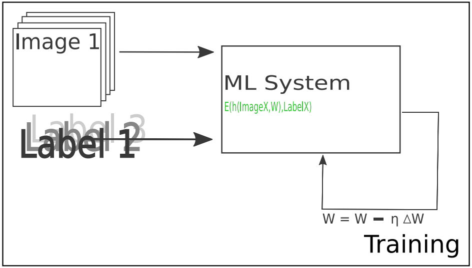
- An ultra-low power design
- Featuring 12 VLIW programmable SHAVE cores, dedicated vision accelerators and 2 CPUS
- 12 programmable SHAVE cores
- A small-area footprint
- Support for 16/32-bit floating point and 8/16/32-bit integer operations

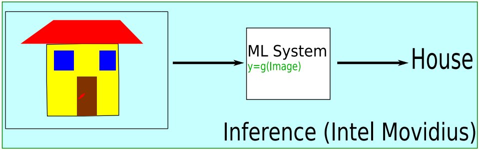


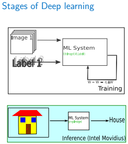
Myriad 2 Vision Processor Unit (VPU)

An ultra-low power design: For mobile and connected devices where battery life is critical, Intel's Myriad 2 VPU provides a way to combine advanced vision applications in a low power profile. This enables new vision applications in small form factors that could not exist before. A small-area footprint To conserve space inside mobile, wearable, and embedded devices, Intel's Myriad 2 VPU was designed with a very small footprint that can easily be integrated into existing products.

Stages of Deep learning







So there are two stages of deep learning one is process of tunnning of params of our model (Neural Network) which is called training. The other one is stage where we apply our trained model for previously not seen data. This second stage is called Inference. It is absolutly fine to train model on one device like CPU server and then perform inference on other device like Intel Movidius. In fact Intel Movidius does support only Inference of Deep learning models. It does work with Caffe[1] and Tensorflow[2] models.

Prerequisities

- Raspbian is officially supported
- Ubuntu 16.04 Linux is officially supported
  - native installation
  - Virtual machine
  - docker
- Other Linux flavours are not supported
  - Python3 is needed

Prerequisities

- Raspbian is officially supported
- Ubuntu 16.04 Linux is officially supported
  - native installation
  - Virtual machine
  - docker
- Other Linux flavours are not supported
  - Python3 is needed

What do we need to have NCS running? Currently Intel Movidius does support Ubuntu and Raspberry pie eg. Raspbian. It does not have to be nativly installed supported linux. Virtual machine (host Windows) will work as well. As for not supported configurations. They may work after some tunning. I personally had C API of Movidius working on Fedora 23. But to save some troubles, use supported configurations.

Configuring and Building project that uses C API - Commandline

Example Commandline :

```
g++ main.cpp -I<dir with mvnc.h> -lmvnc -L<dir with libmvnc.so> -o hello-movidius
```

Example Commandline from Fedora Linux :

```
g++ main.cpp -I/usr/local/include -lmvnc -L/usr/local/lib -o hello-movidius
```

Configuring and Building project that uses C API - Commandline

Example Commandline :

```
g++ main.cpp -I<dir with mvnc.h> -lmvnc -L<dir with libmvnc.so> -o hello-movidius
```

Example Commandline from Fedora Linux :

```
g++ main.cpp -I/usr/local/include -lmvnc -L/usr/local/lib -o hello-movidius
```

How can we build our project that is using Intel Movidius C API? For very simple project we can build it from commandline. To have it build we need to give compiler a path to where movidius headers are as well as movidius library and directory where library is installed. Second example shows how it can be done my custom (not supported) Fedora OS.

Configuring and Building of project using C API – cmake

```
cmake_minimum_required(VERSION 2.8)
project(task1)

# ---[ Find Intel Movidius header
find_path(NCS_INCLUDE_DIR NAMES "mvnc.h"
        HINTS "/usr/local"
        PATHS "/usr/local"
        PATH_SUFFIXES "include" )

# ---[ Find  Intel Movidius library
find_library(NCS_LIBRARY
        NAMES mvnc
        PATHS /usr/local/lib)
if(NCS_LIBRARY)
  message(STATUS "Found Movidius NCS (include: ${NCS_INCLUDE_DIR}, lib: ${NCS_LIBRARY}")
  include_directories(${NCS_INCLUDE_DIR})
else()
  message(FATAL " Intel NCS not located properly")
endif()

add_executable(task2 main.cpp )
target_link_libraries(task1 ${NCS_LIBRARY} )
```

Configuring and Building of project using C API – cmake

```
cmake_minimum_required(VERSION 2.8)
project(task1)

# ---[ Find Intel Movidius header
find_path(NCS_INCLUDE_DIR NAMES "mvnc.h"
        HINTS "/usr/local"
        PATHS "/usr/local"
        PATH_SUFFIXES "include" )

# ---[ Find Intel Movidius library
find_library(NCS_LIBRARY
        NAMES mvnc
        PATHS /usr/local/lib)
if(NCS_LIBRARY)
  message(STATUS "Found Movidius NCS (include: ${NCS_INCLUDE_DIR}, lib: ${NCS_LIBRARY}")
  include_directories(${NCS_INCLUDE_DIR})
else()
  message(FATAL " Intel NCS not located properly")
endif()

add_executable(task2 main.cpp )
target_link_libraries(task1 ${NCS_LIBRARY} )
```

Building from commandline is not very convenient for more complex projects. So we can use Cmake as building system. On shown example we try to locate mvnc.h and pass directory where this header is to linker. Then we try to locate where libmvnc.so is located to pass it to linker. After we have detected MVNC headers and library we let know compiler which source files should be compiled to build our target(binary) and then to this target we also pass information on libraries to be used. If no libmvnc.so is found then an error is reported and configuration of project is stopped.

Configuring and Building project that uses C API - task

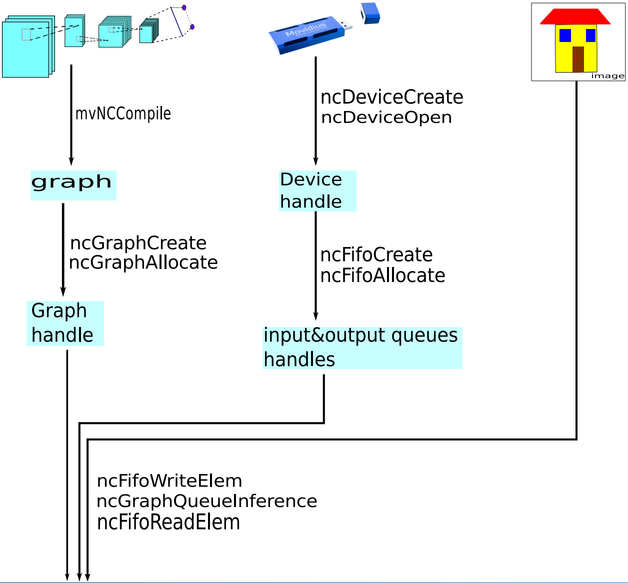
Example cmake commandline :

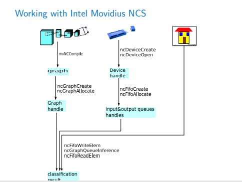
```
mkdir build; cd build; cmake ../
```

Task1:

1. Use cmake to build code in directory: task1
2. Run created binary without NCS inside
3. Run created binary with NCS

Working with Intel Movidius NCS





So how does the model of cooperation with NCS looks like? As mentioned in introduction only inference is supported in NCS. So we need to have CNN model already trained in Caffe or Tensorflow then with mvNCCompil we process it to the form that can be deployed to NCS. Also we need to locate NCS in our system and initialize it for work eg. create resources for NCS handling and open communication with NCS. This is done with ncDeviceCreate and ncDeviceOpen. After we have NCS initialized as a result we will get Device handle . Then we create input and output queses which are used for sending and recieving data from NCS.

To have actual inference we send an image into input queue (using ncFifoWriteElem) and then start inference with ncGraphQueueInference Result is to be acquired by reading from output queue eg. ncFifoReadElem.

When we want to conclude work we have to free resources eg. queues, graphs.

Locating NCS devices in a system

```
index = 0;  // Index of device to query for
ncStatus_t ret = ncDeviceCreate(index,&deviceHandle);
if (ret == NC_OK) {
  std::cout << "Found NCS named: " << index++ << std::endl;
}

// If not devices present the exit
if (index == 0) {
  throw std::string("Error: No Intel Movidius identified in a system!\n");
}
```

- can be called in multiple times (until an error is returned) to create device handles for multiple NCS devices.

Locating NCS devices in a system

```
index = 0;  // Index of device to query for
ncStatus_t ret = ncDeviceCreate(index,&deviceHandle);
if (ret == NC_OK) {
  std::cout << "Found NCS named: " << index++ << std::endl;
}

// If not devices present the exit
if (index == 0) {
  throw std::string("Error: No Intel Movidius identified in a system!\n");
}
```

- can be called in multiple times (until an error is returned) to create device handles for multiple NCS devices.

So to detect NCS in a system we call ncDeviceCreate. IF function succeed for given NCS name (from 0 onwords) then struct describing given device is allocated. Typical usage is to call this function in a loop until we have error returned . That way we initialize all NCS existing in a system. Corressponding function to release device handle is ncDeviceDestroy (example code is on later slides)

Opening communication with NCS devices

```
ret = ncDeviceOpen(deviceHandle);
if(ret != NC_OK) {
  // If we cannot open communication with device then clean up resources
  destroy();
  throw std::string("Error: Could not open NCS device: ") + std::to_string(index-1) ;
}
```

- ncDeviceOpen can be called for each initialized NCS

Opening communication with NCS devices

```
ret = ncDeviceOpen(deviceHandle);
if(ret != NC_OK) {
  // If we cannot open communication with device then clean up resources
  destroy();
  throw std::string("Error: Could not open NCS device: ") + std::to_string(index-1) ;
}
```

- ncDeviceOpen can be called for each initialized NCS

After we detected NCS devices we need to open them for communication. This function can be called for each NCS detected in a system Function that closes communication with the device is ncDeviceClose

Creating and Training model

- caffe or tensorflow are supported
- classification and detection is supported
- No batch processing mode for inference.
- not all corner cases of caffe functionality is supported

Creating and Training model

- caffe or tensorflow are supported
- classification and detection is supported
- No batch processing mode for inference.
- not all corner cases of caffe functionality is supported

NCS supports classification and detection and its purpose is computer vision inference. Model can be created and trained using Caffe or Tensorflow. But not every caffe model is supported as not for all layers full range of functionality is supported. For example InnerProduct layer can accept many inputs in Caffe, but Movidius caffe supports one input. So before engaging into training it could be good to compile created model into NCS graph , just to check if our model is supported by NCS.

Conversion of model for NCS

Example Commandline for converting GoogleNet model prepared with Caffe:

```
mvNCCompile -w bvlc_googlenet.caffemodel deploy.prototxt -s 12 -o myGoogleNet
```

Actions performed by mvNCCompile:

- convert model's data layout from ZYX to YXZ
- convert data format : floating point 16
- merge(fuse) layers eg. Relu+BatchNorm

Supported DNN frameworks:

- Caffe
- Tensorflow

Conversion of model for NCS

Example Commandline for converting GoogleNet model prepared with Caffe:

```
mvNCCompile -w bvlc_googlenet.caffemodel deploy.prototxt -s 12 -o myGoogleNet
```

Actions performed by mvNCCompile:

- convert model's data layout from ZYX to YXZ
- convert data format : floating point 16
- merge(fuse) layers eg. Relu+BatchNorm

Supported DNN frameworks:

- Caffe
- Tensorflow

After we have model trained we need to convert model to the form that can be used by NCS. We used a mvNCCompile program to generate graph to be loaded into NCS. There is a number of actions that mvNCCompile performs. Model is analyzed then conversion from floating point to fp16 is performed. Data and model is rearranged so data layout from ZYX to YXZ . Also model is analyzed and some of layers are fused , for example Relu with Batch Normalization. I'm talking here on Caffe but it same program mvNCCompile is used for conversion of tensorflow model.

Conversion of model for NCS – task

Examples of conversions :

```
mvNCCompile -w bvlc_googlenet.caffemodel deploy.prototxt -s 1 -o myGoogleNet-shave1
mvNCCompile -w bvlc_googlenet.caffemodel deploy.prototxt -s 12 -o myGoogleNet-shave12
```

Task2:

1. find and enter task2 directory
2. convert trained googlenet model to graph suited for one shave
3. convert trained googlenet model to graph suited for 12 shaves

Loading NCS graph

```
// Creation of  graph resources
unsigned int graphSize = 0;
loadGraphFromFile(graphFile, graphFileName, &graphSize);
ncStatus_t ret = ncGraphCreate(graphFileName.c_str(),&graphHandlePtr);
if (ret != NC_OK) {
    throw std::string("Error: Graph Creation failed!");
}
// Allocate graph on NCS
ret = ncGraphAllocate(deviceHandle,graphHandlePtr,graphFile.get(),graphSize);
if (ret != NC_OK) {
    destroy();
    throw std::string("Error: Graph Allocation failed!");
}
unsigned int optionSize = sizeof(inputDescriptor);
ret = ncGraphGetOption(graphHandlePtr,
        NC_RO_GRAPH_INPUT_TENSOR_DESCRIPTORS,
        &inputDescriptor,
        &optionSize);
if (ret != NC_OK) {
  destroy();
  throw std::string("Error: Unable to create input FIFO!");
}
```

- Many graphs can be loaded to single NCS
- Each graph can be send to only one NCS.

Loading NCS graph

```
// Creation of  graph resources
unsigned int graphSize = 0;
loadGraphFromFile(graphFile, graphFileName, &graphSize);
ncStatus_t ret = ncGraphCreate(graphFileName.c_str(),&graphHandlePtr);
if (ret != NC_OK) {
    throw std::string("Error: Graph Creation failed!");
}
// Allocate graph on NCS
ret = ncGraphAllocate(deviceHandle,graphHandlePtr,graphFile.get(),graphSize);
if (ret != NC_OK) {
    destroy();
    throw std::string("Error: Graph Allocation failed!");
}
unsigned int optionSize = sizeof(inputDescriptor);
ret = ncGraphGetOption(graphHandlePtr,
        NC_RO_GRAPH_INPUT_TENSOR_DESCRIPTORS,
        &inputDescriptor,
        &optionSize);
if (ret != NC_OK) {
  destroy();
  throw std::string("Error: Unable to create input FIFO!");
}
```

- Many graphs can be loaded to single NCS
- Each graph can be send to only one NCS.

After we have NCS graph created from trained model we put it on the NCS. This done by first create handle to graph via ncGraphCreate. Then we allocate graph on NCS using ncGraphAlocate.Allocated graph is ready to ready to perform inferences. Then with ncGraphGetOption we can acquire dimensions of input to be used by graph that later on will be used (those dimensions) when initializing NCS queues.

Loading NCS graph

```
void loadGraphFromFile(std::unique_ptr<char[]>& graphFile, const std::string& graphFileNa
{
  std::ifstream ifs;
  ifs.open(graphFileName, std::ifstream::binary);
  if (ifs.good() == false) {
    throw std::string("Error: Unable to open graph file: ") + graphFileName;
  }

  // Get size of file
  ifs.seekg(0, ifs.end);
  *graphSize = ifs.tellg();
  ifs.seekg(0, ifs.beg);

  graphFile.reset(new char[*graphSize]);
  ifs.read(graphFile.get(),*graphSize);
  ifs.close();
}
```

Loading NCS graph

```
void loadGraphFromFile(std::unique_ptr<char[]>& graphFile, const std::string& graphFileNa
{
  std::ifstream ifs;
  ifs.open(graphFileName, std::ifstream::binary);
  if (ifs.good() == false) {
    throw std::string("Error: Unable to open graph file: ") + graphFileName;
  }

  // Get size of file
  ifs.seekg(0, ifs.end);
  *graphSize = ifs.tellg();
  ifs.seekg(0, ifs.beg);

  graphFile.reset(new char[*graphSize]);
  ifs.read(graphFile.get(),*graphSize);
  ifs.close();
}
```

How to load graph into memory. Well converted graph is stored on harddrive. So we read file into memory and pointer to that memory is given to ncGraphAllocate

Creating Queues (FIFO)

```
// Create input FIFO
ncStatus_t ret = ncFifoCreate("input1",fifotype, &FIFO_);
  if (ret != NC_OK) {
    throw std::string("Error: Unable to create FIFO!");
  }

....
  ret = ncFifoAllocate(input.FIFO_, deviceHandle, &inputDescriptor, 2);
  if (ret != NC_OK) {
    destroy();
    throw std::string("Error: Unable to allocate input  FIFO! on NCS");
  }
```

- It is nessesery to create at least one input and at least one output queues

Creating Queues (FIFO)

```
// Create input FIFO
ncStatus_t ret = ncFifoCreate("input1",fifotype, &FIFO_);
  if (ret != NC_OK) {
    throw std::string("Error: Unable to create FIFO!");
  }

....

  ret = ncFifoAllocate(input.FIFO_, deviceHandle, &inputDescriptor, 2);
  if (ret != NC_OK) {
    destroy();
    throw std::string("Error: Unable to allocate input  FIFO! on NCS");
  }
```

- It is nessesery to create at least one input and at least one output queues

Next we need to need to create a queues for communication with NCS device . One for input data and one for output data. This is done with ncFifoCreate. Next created Fifos should be allocated of NCS devices . This is done with ncFifoAllocate.

Starting inference

```
ret = ncFifoWriteElem(input.FIFO_,tensor.get(),&inputLength, nullptr);
if (ret != NC_OK) {
  throw std::string("Error: Loading Tensor into input queue failed!");
}

ret = ncGraphQueueInference(graphHandlePtr,&(input.FIFO_), 1, &(output.FIFO_), 1);
if (ret != NC_OK) {
  throw std::string("Error:  Queing inference failed!");
}
```

- For each written element into input FIFO we need to call ncGraphQueueInference

Starting inference

```
ret = ncFifoWriteElem(input_FIFO_, tensor.get(),&inputLength, nullptr);
if (ret != NC_OK) {
  throw std::string("Error: Loading Tensor into input queue failed!");
}

ret = ncGraphQueueInference(graphHandlePtr,&(input_FIFO_), 1, &(output_FIFO_), 1);
if (ret != NC_OK) {
  throw std::string("Error:  Queing inference failed!");
}
```

- For each written element into input FIFO we need to call ncGraphQueueInference

Having graph loaded into NCS we can send input data (images) to NCS for execution. This is done with writting data to input FIFO ncFifoWriteElem and then enqueuing inference with ncGraphQueueInference. Currently for each element in a input FIFO we need to call ncGraphQueueInference

Preparing tensor – 1

```
void prepareTensor(std::unique_ptr<unsigned char[]>& input, std::string& imageName, unsig
{
  // load an image using OpenCV
  cv::Mat imagefp32 = cv::imread(imageName, -1);
  if (imagefp32.empty())
    throw std::string("Error reading image: ") + imageName;

  // Convert to expected format
  cv::Mat samplefp32;
  if (imagefp32.channels() == 4 && net_data_channels == 3)
    cv::cvtColor(imagefp32, samplefp32, cv::COLOR_BGRA2BGR);
  else if (imagefp32.channels() == 1 && net_data_channels == 3)
    cv::cvtColor(imagefp32, samplefp32, cv::COLOR_GRAY2BGR);
  else
    samplefp32 = imagefp32;

  // Resize input image to expected geometry
  cv::Size input_geometry(net_data_width, net_data_height);

  cv::Mat samplefp32_resized;
  if (samplefp32.size() != input_geometry)
    cv::resize(samplefp32, samplefp32_resized, input_geometry);
  else
    samplefp32_resized = samplefp32;

  ....
```

Preparing tensor – 1

```
void prepareTensor(std::unique_ptr<unsigned char[]>& input, std::string& imageName, unsig
{
  // load an image using OpenCV
  cv::Mat imagefp32 = cv::imread(imageName, -1);
  if (imagefp32.empty())
    throw std::string("Error reading image: ") + imageName;

  // Convert to expected format
  cv::Mat samplefp32;
  if (imagefp32.channels() == 4 && net_data_channels == 3)
    cv::cvtColor(imagefp32, samplefp32, cv::COLOR_BGRA2BGR);
  else if (imagefp32.channels() == 1 && net_data_channels == 3)
    cv::cvtColor(imagefp32, samplefp32, cv::COLOR_GRAY2BGR);
  else
    samplefp32 = imagefp32;

  // Resize input image to expected geometry
  cv::Size input_geometry(net_data_width, net_data_height);

  cv::Mat samplefp32_resized;
  if (samplefp32.size() != input_geometry)
    cv::resize(samplefp32, samplefp32_resized, input_geometry);
  else
    samplefp32_resized = samplefp32;

  ....
```

How an example of preparation of tensor looks like. OpenCV is used here but any other library is fine. Movidius ncsdk comes with stb_image which is public domain image loader. Ok. So how typical tensor preparation look like. We load image, then convert image to pixel format matching our graph eg. NN model. So if googlenet is requiring images 224x224 RGB then we perform such operation.

Preparing tensor – 2

```
...
  // Convert to float32
  cv::Mat samplefp32_float;
  samplefp32_resized.convertTo(samplefp32_float, CV_32FC3);

  // Mean subtract
  cv::Mat input;
  cv::Mat mean = cv::Mat(input_geometry, CV_32FC3, net_mean);
  cv::subtract(samplefp32_float, mean, input);

  *inputLength = sizeof(short)*net_data_width*net_data_height*net_data_channels;
}
```

- By default data is accepted in float (32 bit format type), but it is possible to deliver input in 16 bit floating point type

Preparing tensor – 2

```
...
  // Convert to float32
  cv::Mat samplefp32_float;
  samplefp32_resized.convertTo(samplefp32_float, CV_32FC3);

  // Mean subtract
  cv::Mat input;
  cv::Mat mean = cv::Mat(input_geometry, CV_32FC3, net_mean);
  cv::subtract(samplefp32_float, mean, input);

  *inputLength = sizeof(short)*net_data_width*net_data_height*net_data_channels;
}
```

- By default data is accepted in float (32 bit format type), but it is possible to deliver input in 16 bit floating point type

Next we convert image to float values, subtract mean of images if model require to do so. We may convert on our own image data from float32 to float16. If we do then NCS sdk does not have to do it.

Getting inference results

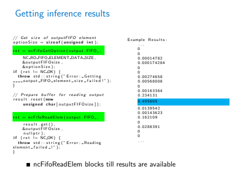
```
// Get size of outputFIFO element
optionSize = sizeof(unsigned int);
ret = ncFifoGetOption(output.FIFO_,
    NC_RO_FIFO_ELEMENT_DATA_SIZE,
    &outputFIFOsize,
    &optionSize);
if (ret != NC_OK) {
  throw std::string("Error: Getting
    output FIFO element size failed!");
}

// Prepare buffer for reading output
result.reset(new
    unsigned char[outputFIFOsize]);

ret = ncFifoReadElem(output.FIFO_,
    result.get(),
    &outputFIFOsize,
    nullptr);
if (ret != NC_OK) {
  throw std::string("Error: Reading
element failed !");
}
```

```
Example Results:
    ...
    0
    0
    0.00014782
    0.000174284
    0
    0
    0.00274658
    0.00568008
    0
    0.00163364
    0.234131
    0.495605
    0.0139542
    0.00143623
    0.162109
    0
    0.0288391
    0
    0
    ...
```

- ncFifoReadElem blocks till results are available



Getting outcome of execution eg. inference is an element in output FIFO(queue). This element can be read with ncReadElem . This function does block execution eg. wait till something to be read in output queue. To use this function we need to know the size of element in a output queue, which can be acquired using ncFifoGetOption . For classification returned result is a vector of floating point values that are interpreted as probabilities that given input(image) represented given category

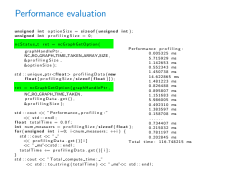
Performance evaluation

```
unsigned int optionSize = sizeof(unsigned int);
unsigned int profilingSize = 0;
ncStatus_t ret = ncGraphGetOption(
    graphHandlePtr,
    NC_RO_GRAPH_TIME_TAKEN_ARRAY_SIZE,
    &profilingSize,
    &optionSize);

std::unique_ptr<float> profilingData(new
    float[profilingSize/sizeof(float)]);
ret = ncGraphGetOption(graphHandlePtr,
    NC_RO_GRAPH_TIME_TAKEN,
    profilingData.get(),
    &profilingSize);

std::cout << "Performance profiling:"
    << std::endl;
float totalTime = 0.0f;
int num_measuers = profilingSize/sizeof(float);
for(unsigned int i=0; i<num_measuers; ++i) {
  std::cout << " "
    << profilingData.get()[i]
    << " ms"<<std::endl;
  totalTime += profilingData.get()[i];
}
std::cout << "Total compute time: "
    << std::to_string(totalTime) << " ms"<< std::endl;
```

```
Performance profiling:
        0.005325 ms
        5.715929 ms
        1.142653 ms
        0.552343 ms
        1.450738 ms
        14.622865 ms
        1.481223 ms
        0.826488 ms
        0.895807 ms
        1.151683 ms
        5.986005 ms
        0.492310 ms
        1.383597 ms
        0.158708 ms
        ......
        0.734407 ms
        0.215032 ms
        0.781197 ms
        0.202845 ms
Total time: 116.748215 ms
```



It is possible from C API to get times of execution of stages. Profiling info is done anyway so it just matter of acquiring it from NCS. This can be done using ncGraphGetOption with argument NC_RO_GRAPH_TIME_TAKEN . As a result we will get time in ms of stages being executed. To alocate memory for results we need to know size of performance measures to be returned. This size can acquired with ncGraphGetOption with NC_RO_GRAPH_TIME_TAKEN_ARRAY_SIZE parameter. Returned results are of float data type that is a reason we divide size returned of results by size of float data type. More detailed info can be taken using mvncProfile

Finishing work with NCS

```
// Deallocating graph
ncStatus_t ret = ncGraphDestroy(&graphHandlePtr);
if (ret != NC_OK)
  std::cout << "Error: Graph destroying failed!" << std::endl;

// Releasing queue
ncStatus_t ret = ncFifoDestroy(&FIFO_);
if (ret != NC_OK)
  std::cout << "Error: FIFO destroying failed!" << std::endl;

// Closing communication with NCS
ncStatus_t ret = ncDeviceClose(deviceHandle);
if (ret != NC_OK) {
  std::cerr << "Error: Closing of device: "
    <<  std::to_string(index-1) <<" failed!" << std::endl;
}

// Releasing resources for NCS handling
ncStatus_t ret = ncDeviceDestroy(&deviceHandle);
if (ret != NC_OK) {
  std::cerr << "Error: Freeing resources of device: "<<  std::to_string(index-1)
    <<" failed!" << std::endl;
}
```

- Lack of closing device may result in device not been available for some time

Finishing work with NCS

```cpp
// Deallocating graph
ncStatus_t ret = ncGraphDestroy(&graphHandlePtr);
if (ret != NC_OK)
  std::cout << "Error: Graph destroying failed!" << std::endl;

// Releasing queue
ncStatus_t ret = ncFifoDestroy(&FIFO_);
if (ret != NC_OK)
  std::cout << "Error: FIFO destroying failed!" << std::endl;

// Closing communication with NCS
ncStatus_t ret = ncDeviceClose(deviceHandle);
if (ret != NC_OK) {
  std::cerr << "Error: Closing of device: "
    << std::to_string(index-1) <<" failed!" << std::endl;
}

// Releasing resources for NCS handling
ncStatus_t ret = ncDeviceDestroy(&deviceHandle);
if (ret != NC_OK) {
  std::cerr << "Error: Freeing resources of device: "<< std::to_string(index-1)
    <<" failed!" << std::endl;
}
```

- Lack of closing device may result in device not been available for some time

When we end work we release graph allocated with ncGraphDestroy and then close NCS with ncDeviceClose and ncDeviceDestroy. Lack of closing device and releasing the graph may result in device not been available at least that is my observation that next time we call program using NCS NCS may not be able to be initialized properly

Performance evaluation - Task

Task3:

1. find and enter task3 directory
2. build main.cpp to use graph compiled for one shave

   ```
   mkdir build; cd build; cmake ../; make
   ```

3. execute program task3 using cat.jpg and note top-1 classification result

   ```
   cd ../; ./build/test-ncs-v2 cat.jpg -graph myGoogleNet-shave1
   ```

4. execute program task3 using cat.jpg with **profiling** and note performance results

   ```
   ./build/test-ncs-v2 cat.jpg -graph myGoogleNet-shave1 -profile
   ```

5. execute program task3 using cat.jpg with **profiling** but with graph compiled for 12 shaves. Note performance results

   ```
   ./build/test-ncs-v2 cat.jpg -graph myGoogleNet-shave12 -profile
   ```

References I

[1] Caffe* Optimized for Intel Architecture: Applying Modern Code Techniques. Improving the computational performance of a deep learning framework. Vadim Karpusenko, Ph.D. Andres Rodriguez, Ph.D. Jacek Czaja, Mariusz Moczala

[2] TensorFlow* Optimizations on Modern Intel Architecture. Elmoustapha Ould-Ahmed-Vall et el.