AMDnn library.

Introductions.

TBD

Design goals.

TBD

Technical details.

# **Objects.**

There are 4 objects operated by AMDnn library.

They are alib, adata, anode, anet.

alib : represents a AMDnn library object, interchangeably called Lib.

adata represents a multi-dimensional data object called Data.

anode represents a Node object.

anet represents a Net object.

## ALIB object

An alib object has to be created and destroyed by the application.

A Lib object is used in every AMDnn API that creates other AMDnn objects.

When creating an alib object a host can send as a parameter the OpenCl context it has been already using or the context parameter might be 0. In the latter case the ANDnn starts with instantiating an OpenCl run-time and creation of its own context. The context could be retrieved by the host and be used in any regular OpenCl API. The internal alib context is destroyed together with the destruction of the alib object. After that it’s invalid.

A host also can use the utility interface to create an OpenCl command queue associated with any device its context connected to.

The devices’ ids can be inspected and obtained by the Inspect API.

## ADATA object

adata object is a major way of passing data between AMDnn Node objects and from the outside world to Node objects.

An adata object serves 2 purposes. It defines and provides a layout information associated with the data and it passes real system pointers and/or OpenCl buffer handles between a host application and Node objects and between Node objects.

A Data object has few predefined layouts from 1 to 5 dimensions (current - 4 dimensions). A Data object also can define a arbitrary layout up to 8 dimensions. Please, see adata object specific APIs.

## ANODE object.

Anode object is a major unit of the processing within AMDnn.

To create anode object a host has to provide to the Lib a node specific parameter structure.

The structure includes a node name and few other node specific parameters (see API specification).

Edge.

The major part of the node parameter structure is other structure called edge. There are 2 types of edges: input and output.

There are 8 input edges that can be attached to a node (the current AMDnn implementation uses 1 or 2 edges). A Node may have only a single output edge in the current implementation.

Both input and output edges should have a name. For naming rules see anet sections.

Both edges variations contain adata object to pass actual data and another adata object keeping a differential used in the SGD backward propagation pass.

An input edge is the most informative though. It keeps weights/bias adata objects and their differentials together with input filter parameters.

A filter can be multidimensional (up to 4 dimensions) with different parameters in each dimension (currently 2 dimensional and square).

A Node object can be constructed, built and run separately (on the building process see later paragraphs). Several anode objects that share edges can be combined into a pipeline driven by the host.

## ANET object

ANET object is created very much as an anode object. At minimum it has to have a name.

Any anode object can be added to the Net. To be part of the Net a Node has to have a name and one input edge of one of the Node from the Net has to have the same name. The first Node of the net has to have a unique input edge name – no Node should have the same name. The output edge name of a Node has to be the same as the Node name.

# Connect, construct, build and run.

A host has to make a few steps before start running a stand-alone Nodes or Nodes connected into a Net

## Connect.

If a host is going to run a Net it, first, has to call Connect API. The API connects all Nodes that have been added to the Net so far, verifies the Net correctness and returns error if it cannot connect Nodes correctly.

## Construct.

The Construct API prepares the Node or Net execution plan based on the Node parameters, Data layouts and other factors. After the Construct the host can inquire the total GPU memory required to run the Node, the set of stand-alone Nodes or the Net. The host also can inspect the execution plan to look at the build options, OpenCL kernel source code or other plan parameters (see API specification and samples). The host may reconfigure parameters, reallocate adata objects, makes other modifications based on that information.

## Build.

The Build API creates OpenCl kernel binaries and attempts to cache OpenCl kernel arguments. It is recommended for adata objects to allocate a real memory at this point to permit the Build to cache kernel arguments.

## Run.

Run, RunFwd, RunBwd APIs executes a single Node or Net iteration. It’s a host responsibly to run iteration in a loop. It’s also a host responsibility to update Nodes with data that have to be taken from the outside world at any moment of iteration loop. It can use run-time arguments for that purpose – see API specifications and samples.

# Initialization.

AMDnn provides an utility operator that allows to initialize a data object with few data sampled from several different probability distribution functions. The a data has to allocate memory before calling the initialization API.

# API Reference.

#define ADNN\_MAX\_INPUTS 8

#define ADNN\_MAX\_OUTPUTS 1

#define ADNN\_MAX\_TENSOR\_DIM 8

#define ADNN\_MAX\_FILTER\_DIM 4

/\*---------------------------------------------------------

MD data flags

----------------------------------------------------------\*/

#define ADNN\_MEM\_ALLOCSYS\_ONLY (1 << 24)

#define ADNN\_MEM\_ALLOCOCL\_ONLY (1 << 25)

#define ADNN\_MEM\_ACCESS\_WRITE\_DESTRUCT CL\_MAP\_WRITE\_INVALIDATE\_REGION

#define ADNN\_MEM\_ACCESS\_WRITE CL\_MAP\_WRITE

#define ADNN\_MEM\_ACCESS\_READ CL\_MAP\_READ

/\* library object \*/

typedef void \* alib\_obj;

/\* net object \*/

typedef void \* anet\_obj;

/\* node object \*/

typedef void \* anode\_obj;

/\* data object \*/

typedef void \* adata\_obj;

enum {

ADNN\_SUCCESS = 0,

ADNN\_GENERAL\_FAILURE = -1

};

/\*---------------------------------------------------------

ADNNLib lib object interface

----------------------------------------------------------\*/

/\*---------------------------------------------------------

ADNNLib lib object parameters and interfaces.

---------------------------------------------------------\*/

typedef enum {

ADNNLIB\_INSPECT\_CONTEXT, // OCL context

ADNNLIB\_INSPECT\_PARAMS, // current parameters

ADNNLIB\_INSPECT\_DEVICES, // accelareting devices

ADNNLIB\_INSPECT\_VERSION, // AMDnn libray version

ADNNLIB\_INSPECT\_TOTAL

} ADNNLIB\_INSPECT;

/\*---------------------------------------------------------

ADNNLib object initialization parameters

----------------------------------------------------------\*/

typedef struct \_adnn\_lib\_parameters {

cl\_context context; // library's context,

// if 0 - the library initilizes OCL run-time, creates internal context,

// otherwise the library assumes an external context and does not initilize

// the OCL run-time.

cl\_device\_type accel\_type; // type of accelerating device - current follows OCL protocol;

// default: CL\_DEVICE\_TYPE\_GPU.

const char \* accel\_platform; // accerating platform.

// default: "Advanced Micro Devices, Inc."

const char \*ocl\_kernels\_path; // OCL source/binary location

// default: current working directory

} adnn\_lib\_parameters;

/\*-----------------------------------------------------------------

creates a new AMDnn library object,

that is used in creating any other objects.

if context is 0,

initilizes OCL run-time,

creates context,

makes a list of accelareting devices and their properties.

otherwise,

keeps the context,

makes a list of accelareting devices and their properties.

a user can create any number of lib objects.

------------------------------------------------------------------\*/

alib\_obj ADNNLibCreate(const adnn\_lib\_parameters \* lib\_params); // pointer to the parameter structure.

// parameters are fully copied.

/\*-----------------------------------------------------------------

destroys previously a previously created library objects

------------------------------------------------------------------\*/

int ADNNLibDestroy(alib\_obj \* alib); // library object to be destroied

/\*-----------------------------------------------------------------

returns library object name.

------------------------------------------------------------------\*/

const char \*ADNNLibGetName(alib\_obj lib);

/\*-----------------------------------------------------------------

inspects AMDnn library internal state and parmeters.

------------------------------------------------------------------\*/

int ADNNLibInspect(alib\_obj alib,

ADNNLIB\_INSPECT cause, // reason for inspection,

size\_t \* size, // if data == NULL it returns the length

// of the inspected data, otherwise it

// passes the data buffer length;

void \* data); // pointer to a buffer to be filled in by

// the library for the inspection.

/\*-------------------------------------------------------------------------------

utility that allows a user to create an OCL command queue to one of the

device libraries.

Note:

the created queue is managed by the library and will be released at the

library destruction time.

------------------------------------------------------------------------------\*/

int ADNNLibCreateDeviceQueue(alib\_obj alib // library object,

cl\_device\_id deviceId, // device ID obtain with

// ADNNLibInspect interface

const cl\_command\_queue\_properties \*prop, // OCL

//command queue prioperty

cl\_command\_queue \* new\_queue); // new cl\_command\_queue

/\*---------------------------------------------------------

ADNNData parameters and interfaces

----------------------------------------------------------\*/

/\*---------------------------------------------------------

ADNNData object data initilizing uiility

----------------------------------------------------------\*/

typedef enum {

ADNN\_WD\_NONE, // no initialization

ADNN\_WD\_CONSTANT, // initialize with a constant

ADNN\_WD\_GAUSSIAN, // initialize with a gaussain distribution

ADNN\_WD\_UNIFORM, // initialize with a uniform distribution

ADNN\_WD\_CATEGORIES, // initialize with random 1s in each row (testing only)

ADNN\_WD\_TOTAL

} ADNN\_DATA\_INIT\_DISTR;

typedef struct \_adnn\_data\_init\_parameters {

ADNN\_DATA\_INIT\_DISTR init\_distr; // distribution type

double mean; // distribution mean, also serves a const value for the

// CONSTANT distribution

double std; // standrad deviation or sigma

} adnn\_data\_init\_parameters;

/\*---------------------------------------------------------

ADNNData object

----------------------------------------------------------\*/

/\*---------------------------------------------------------

element format

----------------------------------------------------------\*/

typedef enum {

ADNN\_DF\_FP32,

ADNN\_DF\_FP64,

ADNN\_DF\_FP16,

ADNN\_DF\_UI8,

ADNN\_DF\_I8,

ADNN\_DF\_UI16,

ADNN\_DF\_I16,

ADNN\_DF\_UI32,

ADNN\_DF\_I32,

ADNN\_DF\_TOTAL

} ADNN\_DATA\_FORMAT;

/\*---------------------------------------------------------

fixed dimension data set layout

N - numbert of "batchs"

C - nuber of channels (feature maps)

H - height

W - width

----------------------------------------------------------\*/

typedef enum {

ADNN\_BF\_NCHW,

ADNN\_BF\_NHW,

ADNN\_BF\_NW,

ADNN\_BF\_HW,

ADNN\_BF\_W,

ADNN\_BF\_WHCN,

ADNN\_BF\_WHN,

ADNN\_BF\_WN,

ADNN\_BF\_WH,

ADNN\_BF\_TOTAL

} ADNN\_DATA\_BATCH\_FORMAT;

/\*---------------------------------------------------------

ADNNData object parameters

----------------------------------------------------------\*/

#define ADNN\_DATA\_CTL\_VERTIAL 1 // vertial data buffer - descriptor only

#define ADNN\_DATA\_CTL\_FORCELAYOUT 2 // force layout - accept layout as it's

// sent, not allow opimization-related modifications

typedef struct \_adnn\_data\_parameters {

ADNN\_DATA\_FORMAT data\_format; // element format

ADNN\_DATA\_BATCH\_FORMAT batch\_format; // fixed dimension data layout

int n\_dims; // diminsion of a arbitarry layout - less or

// eq ADNN\_MAX\_TENSOR\_DIM

size\_t dims[ADNN\_MAX\_TENSOR\_DIM]; // dimension length

size\_t strides[ADNN\_MAX\_TENSOR\_DIM]; // dimension stride (in elements)

size\_t size; // total size in elements considering all strides

size\_t size\_bytes; // total size in bytes considering all strides.

void \* sys\_mem; // syst pointers (when valid)

cl\_mem ocl\_mem; // OCL buffer handle (when valid)

unsigned int control\_bits; // control bits for what ??

} adnn\_data\_parameters;

/\*---------------------------------------------------------

ADNNData interface

----------------------------------------------------------\*/

/\*---------------------------------------------------------

creates a new AMDnn Data object.

----------------------------------------------------------\*/

adata\_obj ADNNDataCreate(alib\_obj library, // library object,

const adnn\_data\_parameters \* data\_params) // data foramat and layout descriptor

/\*-------------------------------------------------------------------------

create a new data object with the same parameters as the original.

has to be allocated and destroyed independently from the original.

--------------------------------------------------------------------------\*/

adata\_obj ADNNDataClone(adata\_obj original, // original data object

bool no\_strides // the same data format, layout and dimensions, but not strides

);

/\*---------------------------------------------------------

destroys a Data object.

an object is going to be permenently destroyed together with underlaying memory,

if its reference counter == 0.

----------------------------------------------------------\*/

int ADNNDataDestroy(adata\_obj \* data); // pointer to a previously created object.

/\*---------------------------------------------------------

allocates a real block of memory for the data object

----------------------------------------------------------\*/

int ADNNDataAllocate(adata\_obj da // data object

int alloc\_flags); // allocation flags, follows OCL protocol + 2 others

/\*---------------------------------------------------------

increases a reference counter

----------------------------------------------------------\*/

int ADNNDataRetain(adata\_obj data); // data object

/\*---------------------------------------------------------

decreases a reference counter

----------------------------------------------------------\*/

int ADNNDataRelease(adata\_obj data); // data object

/\*---------------------------------------------------------

makes the data object accessible by host

----------------------------------------------------------\*/

int ADNNDataAccess(adata\_obj data, // data object

cl\_command\_queue non\_default\_queue, // if set the queue has been used

// to transfer data from/to GPU

// memory, if 0, the defualt queue is going to be used

int access\_flags, // memory access flags

adnn\_data\_parameters \* data\_params); // data object parameters

// structure with a non-NULL system pointer accesible

// by host and an actual memory layout.

/\*---------------------------------------------------------

makes the data object inaccessibe by host,

transfers data to the accelerator memory if necessary.

----------------------------------------------------------\*/

int ADNNDataCommit(adata\_obj data); // data object

/\*-------------------------------------------------------------------------

inspects the data object actual layout.

sytem pointer and cl\_mem fields do not represent any meaningful data.

--------------------------------------------------------------------------\*/

int ADNNDataInspect(adata\_obj data, // data object

adnn\_data\_parameters \* data\_params); // data object parameters with an actual memory layout.

/\*-------------------------------------------------------------------------

create a new data object with the same parameters as the original.

has to be allocated and destroyed independently from teh original.

--------------------------------------------------------------------------\*/

adata\_obj ADNNDataClone(adata\_obj original, // original data object

bool no\_strides); // the same data format, layout

// and dimensions, but not strides

/\*-------------------------------------------------------------------------

utility function

intializes a data object memory according to init parameters.

--------------------------------------------------------------------------\*/

int ADNNDataInit(adata\_obj data, // data object

const adnn\_data\_init\_parameters \* data\_init); // init parameters.

/\*---------------------------------------------------------

ADNNode parameters and interfaces.

----------------------------------------------------------\*/

/\*---------------------------------------------------------

ADNNode types

----------------------------------------------------------\*/

typedef enum{

ADNN\_NODE\_GENERIC,

ADNN\_NODE\_NET, // net

ADNN\_NODE\_FULLY\_CONNECT, // fully connected

ADNN\_NODE\_CONV, // convoluitional

ADNN\_NODE\_CONV\_LOCAL, // local convolutional

ADNN\_NODE\_POOLING, // pooling

ADNN\_NODE\_RESP\_NORM, // response normalization

ADNN\_NODE\_NEURON, // neuron

ADNN\_NODE\_DROPOUT, // dropout

ADNN\_NODE\_SOFTMAX, // softmax

ADNN\_NODE\_SOFTMAX\_COST\_CROSSENTROPY, // cost binomial cross - entropy

ADNN\_NODE\_COST\_SOFTMAX, // cost softmax

ADNN\_NODE\_COST\_LOGREG, // cost log regression

ADNN\_NODE\_ELEMSUM,

ADNN\_NODE\_ELEMMAX,

ADNN\_LAYER\_TOTAL

} ADNN\_NODE\_TYPE;

/\*---------------------------------------------------------

control and timing

----------------------------------------------------------\*/

typedef struct \_adnn\_control\_params {

bool per\_layer\_timing; // do timing per layer

int per\_layer\_iter; // # of per layer iterations for timing

bool per\_layer\_messages; // do per layer messges

int debug\_level; // verification and debug control

void \* monitor; // real time monitor callback

} adnn\_control\_params;

/\*---------------------------------------------------------

learning policy

----------------------------------------------------------\*/

typedef enum {

ADNN\_LP\_FIXED,

ADNN\_LP\_LINEAR,

ADNN\_LP\_EXP\_STEP,

ADNN\_LP\_EXP,

ADNN\_LP\_EXP\_INV,

ADNN\_LP\_TOTAL

} ADNN\_LEARNINGPOLICY;

/\*---------------------------------------------------------

learning policy arguments

----------------------------------------------------------\*/

typedef struct \_adnn\_lr\_policy\_params {

ADNN\_LEARNINGPOLICY policy;

double base;

union {

double gamma;

double slope;

};

union {

double step;

double power;

};

} adnn\_lr\_policy\_params;

/\*---------------------------------------------------------

SGD update arguments

----------------------------------------------------------\*/

typedef struct \_adnn\_update\_params{

adnn\_lr\_policy\_params weights\_lr; // weights learning policy

adnn\_lr\_policy\_params bias\_lr; // bias learning policy

double weights\_momentum; // weights momentun

double bias\_momentum; // bias momentum

double weights\_decay; // weights decay

double bias\_decay; // bias decay

} adnn\_update\_params;

/\*---------------------------------------------------------

neuron Node types

----------------------------------------------------------\*/

typedef enum {

ADNN\_NEURON\_PASTHRU, // x

ADNN\_NEURON\_LOGISTIC, // Sigmoid: 1 / (1 + e^-x)

ADNN\_NEURON\_TANH, // a \* tanh( b \* x)

ADNN\_NEURON\_RELU, // max(0, x)

ADNN\_NEURON\_BRELU, // min(a, max(0, x))

ADNN\_NEURON\_SOFTRELU, // bonomial normal log likelihood: log(1 + e^x)

ADNN\_NEURON\_ABS, // abs(x)

ADNN\_NEURON\_SQUARE, // x^2

ADNN\_NEURON\_SQR, // sqr(x)

ADNN\_NEURON\_LINEAR, // ax + b

ADNN\_NEURON\_POWER, // (a + b \* x ) ^power

ADNN\_NEURON\_TOTAL

} ADNN\_NEURON\_TYPE;

/\*---------------------------------------------------------

neuron Node parameters

----------------------------------------------------------\*/

typedef struct \_adnn\_neuron\_parameters {

ADNN\_NEURON\_TYPE type;

double alpha;

double beta;

double power;

} adnn\_neuron\_parameters;

/\*-----------------------------------------------------------

LRN Node parameters

------------------------------------------------------------\*/

/\*-----------------------------------------------------------

Norm region

------------------------------------------------------------\*/

typedef enum {

ADNN\_LRN\_WITHIN\_CHANNEL,

ADNN\_LRN\_ACROSS\_CHANNELS,

ADNN\_LRN\_TOTAL

} ADNN\_LRN\_REGION;

typedef struct \_adnn\_lrn\_parameters {

ADNN\_LRN\_REGION region;

int kernel\_sz;

double alpha;

double beta;

} adnn\_lrn\_parameters;

/\*---------------------------------------------------------

pooling method

----------------------------------------------------------\*/

typedef enum {

ADNN\_POOLING\_AVE, // average

ADNN\_POOLING\_MAX, // max

ADNN\_POOLING\_RAND, // random

ADNN\_POOLING\_TOTAL

} ADNN\_POOLING\_METHOD;

/\*---------------------------------------------------------

1D filter parameters

----------------------------------------------------------\*/

typedef struct \_adnn\_filter1D\_parameters {

int size; // filter size

short pad; // padding size

short stride; // subsamplinig stride

} adnn\_filter1D\_parameters;

/\*---------------------------------------------------------

MD filter parameters

----------------------------------------------------------\*/

typedef struct \_adnn\_filter\_parameters {

int n\_dims; // # of dimensions

adnn\_filter1D\_parameters filter[ADNN\_MAX\_FILTER\_DIM]; // fiter parameters per dimenstion

bool non\_sharedBiases; // if false biases are shared

bool correlation; // if flase - convolution, if true - correlation

} adnn\_filter\_parameters;

/\*---------------------------------------------------------

Node input edge type

----------------------------------------------------------\*/

typedef enum {

ADNN\_ED\_INTERNAL, // internal Net edge

ADNN\_ED\_SOURCE, // external net source edge

ADNN\_ED\_SINK, // external ner sink edge

ADNN\_ED\_TOTAL

} ADNN\_EDGE\_DIR\_TYPE;

/\*---------------------------------------------------------

Node (input or output) edge parameters

----------------------------------------------------------\*/

typedef struct \_adnn\_net\_edge\_parameters {

const char \* name; // name

ADNN\_EDGE\_DIR\_TYPE edge\_type; // edge type (inside Net)

adata\_obj data; // data

adata\_obj data\_diff; // data diff (backpropagation)

// input edge only

adata\_obj weights; // weights data

adata\_obj bias; // bias data

adata\_obj weights\_diff; // weights diff data(backpropagation)

adata\_obj bias\_diff; // bias diff data(backpropagation)

#if 0

adata\_obj weights\_history; // weights history (SGD)

adata\_obj bias\_history; // bias history (SGD)

#endif

adnn\_filter\_parameters filter\_params; // filter parameters

ADNN\_POOLING\_METHOD pooling\_method;

adnn\_lrn\_parameters lrn\_parameters;

long long update\_bits; // bit representing structure's field

} adnn\_net\_edge\_parameters;

/\*---------------------------------------------------------

causes of Node inspection

----------------------------------------------------------\*/

typedef enum {

ADNNODE\_INSPECT\_INPUT\_EDGE, // input edge

ADNNODE\_INSPECT\_OUTPUT\_EDGE, // output edge

ADNNODE\_INSPECT\_EXECUTION\_PLAN\_FWD, // execution plan forward propagation

ADNNODE\_INSPECT\_EXECUTION\_PLAN\_BWD, // execution plan backward propagation

ADNNODE\_INSPECT\_N\_SLOTS, // number of internal data slots

ADNNODE\_INSPECT\_SLOT\_LIST, // list of internal slot names, send list of char ptrs,

ADNNODE\_INSPECT\_SLOT, // internal data object assosiated with the slot; send ptr to the slot name, get ptr to the slot's anode\_obj

ADNNODE\_INSPECT\_TOTAL

} ADNNODE\_INSPECT;

/\*---------------------------------------------------------

Node execution plan parameters

----------------------------------------------------------\*/

typedef struct \_adnn\_node\_exe\_parameters {

const char \* kern\_src\_file; // OCL source file

const char \* kern\_nm; // OCL kernel name

const char \* kern\_src\_string; // OCL kernel source code string

const char \* kern\_build\_options; // OCL compiler build options

cl\_kernel kernel; // OCL kernel (when velid)

size\_t lcl\_sz[3]; // local work size

size\_t glb\_sz[3]; // global work size

cl\_command\_queue queue; // OCL command queue

cl\_event completion\_event; // complition event (whwn velid)

int n\_weit\_events; // # of wait events

const cl\_event \*wait\_events; // array of wait events

} adnn\_node\_exe\_parameters;

/\*---------------------------------------------------------

Node parameters

----------------------------------------------------------\*/

typedef struct \_adnn\_node\_parameters {

ADNN\_NODE\_TYPE type; // type

const char \* name; // name

int n\_input\_nodes; // number of input nodes(edges)

adnn\_net\_edge\_parameters inputs[ADNN\_MAX\_INPUTS]; // input edges

int n\_output\_nodes; // number of output nodes(edges)

adnn\_net\_edge\_parameters outputs[ADNN\_MAX\_OUTPUTS]; // output edges

adnn\_neuron\_parameters neuron\_params; // neuron parameters

adnn\_update\_params update\_params; // SGD update parameters

adnn\_control\_params control; // control and timing

} adnn\_node\_parameters;

/\*---------------------------------------------------------

ADNNode interface

----------------------------------------------------------\*/

/\*-------------------------------------------------------------------------

creates a Node of specific type.

--------------------------------------------------------------------------\*/

anode\_obj ADNNodeCreate(alib\_obj library, // library object

const adnn\_node\_parameters \* layer\_params); // node(layer) parameter structure.

/\*-------------------------------------------------------------------------

destroys a previously ctreated Node.

--------------------------------------------------------------------------\*/

int ADNNodeDestroy(anode\_obj \* alayer); // a pointer to an exiting node.

/\*-------------------------------------------------------------------------

updates Node's parmeters

--------------------------------------------------------------------------\*/

int ADNNodeUpdate(anode\_obj alayer, // ADNNode object

const adnn\_node\_parameters \* layer\_params); // updated set of parameters.

/\*-------------------------------------------------------------------------

return's Node's name

--------------------------------------------------------------------------\*/

const char \*ADNNodeGetName(anode\_obj alayer);

/\*-------------------------------------------------------------------------

defines build and run options for inference (forward propagation pass)

calculates memory reqierment

--------------------------------------------------------------------------\*/

int ADNNodeConstruct(anode\_obj node); // ADNNode object

/\*-------------------------------------------------------------------------

defines build and run options for inference (forward propagation pass) nad training (backward pass).

calculates memory reqierment

--------------------------------------------------------------------------\*/

int ADNNodeConstructTraining(anode\_obj node); // ADNNode object

/\*-------------------------------------------------------------------------

inference (forward pass)

compiles execution kernels,

caches (passes) OCL kernel arguments,

creates execution plan.

Notes:

it's preferable but not mandotary to have memory to be allocated at this point.

the actual OCL memory buffers can be passed at run-time.

--------------------------------------------------------------------------\*/

int ADNNodeBuild(anode\_obj node); // ADNNode object

/\*-------------------------------------------------------------------------

inference (forward pass) + training (backward pass)

compiles execution kernels,

caches (passes) OCL kernel arguments,

creates execution plan.

Notes:

it's preferable but not mandotary to have memory to be allocated at this point.

the actual OCL memory buffers can be passed at run-time.

--------------------------------------------------------------------------\*/

int ADNNodeBuildTraining(anode\_obj node); // ADNNode object

/\*-------------------------------------------------------------------------

runs a single forward iteration.

Notes:

at this point:

source data has to be uploaded,

weights have to be initilized or uploaded.

--------------------------------------------------------------------------\*/

int ADNNodeRunInference(anode\_obj node, // ADNNode object

const adnn\_node\_parameters \* running\_params); // node parameters structure passing run-time arguments if needed.

/\*-------------------------------------------------------------------------

runs a single backward iteration.

Notes:

at this point:

source data has to be uploaded,

weights have to be initilized or uploaded.

--------------------------------------------------------------------------\*/

int ADNNodeRunTraining(anode\_obj node, // ADNNode object

const adnn\_node\_parameters \* running\_params); // node parameters structure passing run-time arguments if needed.

/\*-------------------------------------------------------------------------

inspects node states and properties.

--------------------------------------------------------------------------\*/

int ADNNodeInspect(anode\_obj node, // ADNNode object

ADNNODE\_INSPECT cause, // cause of inspection

size\_t \* size, // if data == NULL it returns the length of the inspected data, otherwise it passes the data buffer length.

void \* data); // pointer to a buffer to be fiiled by the library for the inspection.

/\*---------------------------------------------------------

ADNN net parameters and interfaces.

----------------------------------------------------------\*/

/\*---------------------------------------------------------

ADNN net object parameters

----------------------------------------------------------\*/

typedef adnn\_node\_parameters adnn\_net\_parameters;

/\*---------------------------------------------------------

ADNN net interface

----------------------------------------------------------\*/

/\*-------------------------------------------------------------------------

creates a Net.

Note:

any number of Nets can be created.

--------------------------------------------------------------------------\*/

anet\_obj ADNNCreate(alib\_obj library, // library object.

const adnn\_net\_parameters \* net\_params); // net parameters structure.

/\*-------------------------------------------------------------------------

destroys a previously ctreated Net.

Note:

Nodes attached to the Net are not destroyed and have to be destroyed with

ADNNodeDestroy interface (see).

--------------------------------------------------------------------------\*/

int ADNNDestroy(anet\_obj \* anet); // pointer to an existing Net object.

/\*-------------------------------------------------------------------------

returns Net's name.

--------------------------------------------------------------------------\*/

const char \*ADNNGetName(anet\_obj net); // Net object.

/\*-------------------------------------------------------------------------

add an array of Nodes to the Net.

Note:

Nodes have to be created with the ADNNodeCreate interface.

each Node input edge's name has to have a pair Node name, except for the input Node.

any number of arrays can be added utill the ADNNConnect call.

After that any ADNNodesAdd call will be ignored.

--------------------------------------------------------------------------\*/

int ADNNodesAdd(anet\_obj net, // Net object.

int n\_nodes, // number of Nodes in the array.

const anode\_obj \* nodes); // array of already created Nodes.

/\*-------------------------------------------------------------------------

verifies the Net,

connects nodes.

--------------------------------------------------------------------------\*/

int ADNNConnect(anet\_obj net); // Net object.

/\*-------------------------------------------------------------------------

Construct inference only net.

defines build and run options for each Node,

creates execution plans,

calculates memory reqierment for each Node and total per a Net run.

--------------------------------------------------------------------------\*/

int ADNNConstruct(anet\_obj net); // Net object.

/\*-------------------------------------------------------------------------

Construct training net.

defines build and run options for each Node,

creates execution plans,

calculates memory reqierment for each Node and total per a Net run.

--------------------------------------------------------------------------\*/

int ADNNConstructTraining(anet\_obj net); // Net object.

/\*-------------------------------------------------------------------------

Build execution passes for inference only net

allocates memory for internal buffers,

compiles kernels,

finalizes execution plans per a Node,

considers global Net optimizations.

--------------------------------------------------------------------------\*/

int ADNNBuild(anet\_obj net); // Net object.

/\*-------------------------------------------------------------------------

Build execution passes for training net

allocates memory for internal buffers,

compiles kernels,

finalizes execution plans per a Node,

considers global Net optimizations.

--------------------------------------------------------------------------\*/

int ADNNBuildTraining(anet\_obj net); // Net object.

/\*-------------------------------------------------------------------------

runs one inference iteration,

accepting udates to a running parameters per each Node.

--------------------------------------------------------------------------\*/

int ADNNRunInference(anet\_obj net, // Net object,

int n\_running\_params, // number updating parameter structures,

const adnn\_node\_parameters \* running\_params); // array of updating parameter structures.

/\*-------------------------------------------------------------------------

runs one training iteration,

accepting udates to a running parameters per each Node.

--------------------------------------------------------------------------\*/

int ADNNRunTraining(anet\_obj net, // Net object,

int n\_running\_params, // number updating parameter structures,

const adnn\_node\_parameters \* running\_params); // array of updating parameter structures.