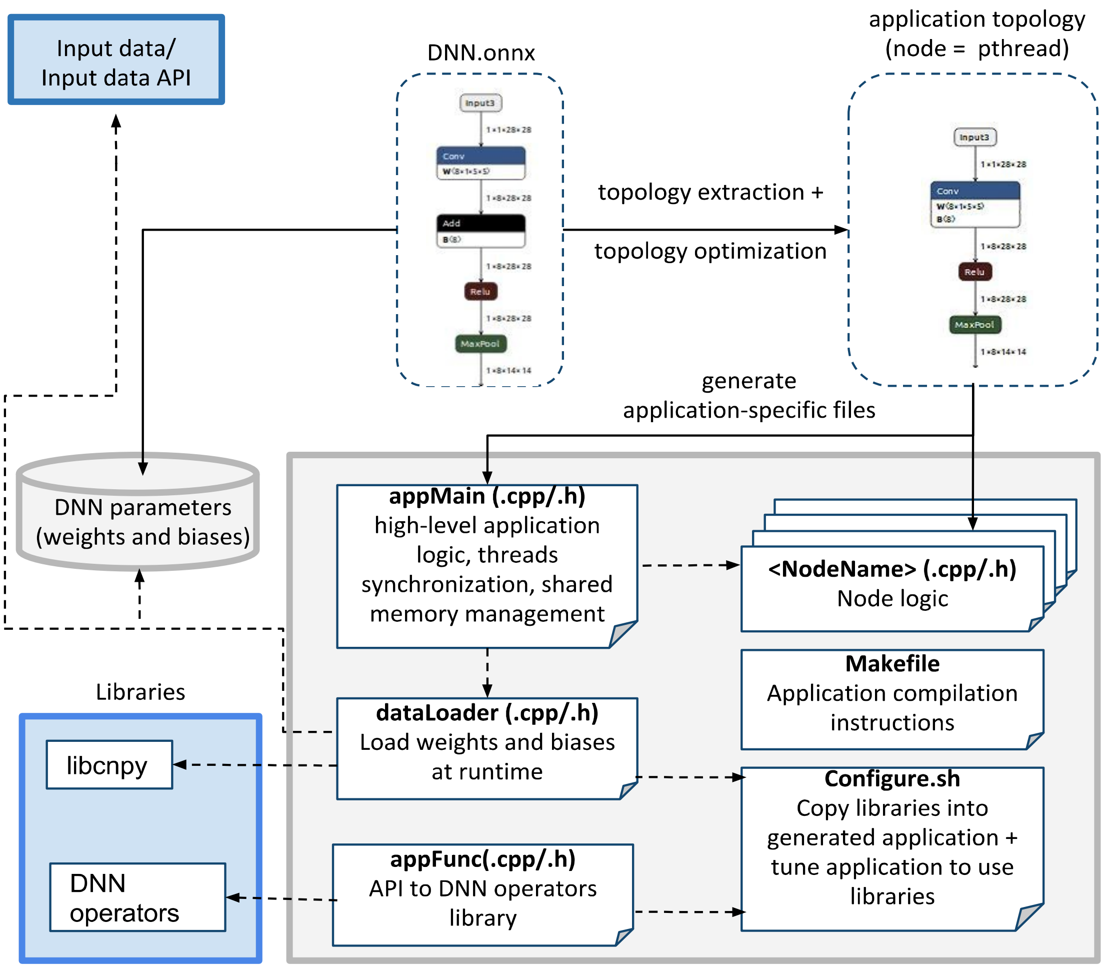
EspamAI-pthread manual

This is a manual for espamAI pthread applications generation module, illustrated in picture below. The module accepts as input path to a DNN model in .onnx format and a range of options, and provides on output components of the pthread application. The components of pthread application incude:

1. Application-specific files:
   - appMain .cpp/ .h files, contatining application definition and high-level logic
   - appFunc .cpp/ .h files, contatining API to DNN operators and some standard functions, used by DNN nodes.
   - .cpp / .h file for every DNN node, containing description and functionality of the corresponding DNN node
   - configure.sh application configuration file
   -makefile
2. DNN parameters (weights and biases), stored as numpy arrays.
3. Libraries: libcnpy (https://github.com/rogersce/cnpy) and custom DNN operators library.
4. Input data, stored as numpy arrays or input data API files.



The application-specific fiels and DNN parameters (shown as grey boxes) are automatically generated by espamAI from an input DNN model. The libraries and input data should be provided by an end user. Each application component is discussed in more details below.

Note: To run apllication on edge you should move on the edge all four components!

Requirements & Installation:

To *generate* pthread application, the *espamAI* installation needed (see README.md at the espamAI root folder or at https://git.liacs.nl/lerc/espam)
To *run* pthread application on edge the *pthread* and *C++11* support is needed on edge.

- cnpy library : numpy arrays load to/from C++. pthread applications use cnpy library https://github.com/rogersce/cnpy for weights loading. Before you can use the pthread application you will need to download the cnpy library and compile it for the platform, where you want to run your pthread application.

- python onnx library and python json library. This libraries are used by espamAI to extract weights and biases from onnx model. The json library should be provided in standard python package. To install the onnx library run:

```
$ pip install onnx
```

Execution

Execute espamAI from command line
See espamAIMan.pdf located in the same folder as this document (espam/src/espam/docs)

Input: Path to a DNN model in .onnx format + options (see below)
Output: C++ pthread application components.

Tool running progress:
(1) *Model reading*: read input DNN model, provided in ONNX format.

(2) *Model conversion and optimization*:
- (2.1) convert input DNN model into internal DNN model
- (2.2) optimize the internal DNN model for inference
- (2.3) convert the internal DNN model into CSDF (application) model

(3) *Output files generation:* generate pthread application files.

Generated application execution:
to run generated application:
1.Move all application files on edge
2. cd folder/with/application (../DNN_name/pthread/app)
3. sh ./configure.sh -b path/to/cnpy-build -s path/to/cnpy-source [-f /path/to/DNN/operators/library]
4. make
5 ./run

Troubleshooting

Sometimes the generated pthread application shows a **segmentation fault (core dumped) error.** Most likely, the reason is the lack of heapsize.The standard heapsize for a custom C++ aplication in Linux is 1GB. However as pthread applications, generated for large DNNs might require much more memory, the heapsize should be extended. To extend the heapsize within one Ubuntu terminal use command

```
$ ulimit -s unlimited
```

In case, this did not solve the problem ,or in case any other problems, please contact:
s.minakova@liacs.leidenuniv.nl

Command-line options

Note: The table below contains only the options, related to Pthread application generation. For all other options see espamAIMan.pdf

option	abbr	arguments	example
General options			
--generate	-g	Path to DNN/CSDFG model	--generate ./tests/lenet.json
Model representation options (by default --layer-based option is set)			
--layer-based	-lb	none	-lb
--block-based	-bb	number of blocks	--bb 20
--split-step [optional], for -bb models only	none	Number of child nodes, obtainbed after one layer splitting	--split-step 4
--opt-fi [optional]	none	Level of optimization of application graph for inference 0 – do not optimizie 1 – remove Dropout and Reshape nodes 2 – (default) – remove Dropout and Reshape nodes + incapsulate adding of biases in predcessing nodes	--opt-fi 2
Hardware specification options			
--cores	none	Number of cores for application (*should be used only if no mapping is provided*)	--cores 5
--mapping	-m	Path to XML mapping file	--mapping ./mapping1.xml
--platform	-p	Path to ESPAM plaform specification (in .xml format)	-p ./my_platform.xml
--na-arch	none	Path to NEURAGHE plaform specification (in .json format)	--na-arch ./my_platform.json

Command-line flags (The command-line options that are either present or not).

flag	abbr	description
General flags		
--help	-h	Printout help
--debug	none	Generate debug information in pthread application
--verbose	-V	Printout program progress information

Application generation flags		
--pthread	none	Generate pthread application
--weights	none	Extract weights in numpy format
--map-xml	none	Print mapping in .xml format
DNN operators library (dnnFunc)		
Default internal library		
--libCPU	none	Generate pthread application + dnnFunc for CPU cores
--libGPU	none	Generate pthread application + dnnFunc for CPU+GPU cores
Cagliari University internal library		
--libNA	none	Generate pthread application adopted for Neuraghe DNN operators lib

Application components

The general structure of application, ready for compilation and then running has the following structure:

```
<app_name>
        -/weights_npz //folder with weights and biases  in .npy array.
        -/data [optional] //folder with I/O data represented as .npy arrays
                -/input  //input .npy arrays that have names input0, input1,...,inputN
                -/output [optional] //output .npy arrays, that have names output0, output1,...,outputN
        -/pthread/app  //pthread application folder with .cpp files, .h files and .sh configuration file
```

Application-specific files

The application-specific C++ files, are automatically generated from ONNX DNN model and include:
1. appMain .cpp/ .h files, contatining application high-level logic.
2. appFunc .cpp/ .h files, contatining API to DNN operators and some standard functions, used by DNN nodes.
3. .cpp / .h file for every DNN node, containing description and functionality of the corresponding DNN node

Weights and biases

Pthread application uses weights and biases, stored separately from the code in .npy (numpy array) format. The weights are extracted from ONNX model and saved in .npy arrays by espamAI and loaded into pthread application by dataLoader file, automatically generated by espamAI. The relative path from the pthread application sources to the weights and biases is *../../weights_npz*. This path should not be changed. The weights are stored in linearizes tensors. The order of dimensions is [OFM * IFM * H * W] for Convolutional nodes, where OFM is number of output feature maps, IFM is number of input feature maps, H and W are height and width and [OFM * input_size] for Dense (fully connected) nodes, where input size is the total number of input values for a dense node.

Weights and biases partitioning.

Due to the large size of DNN weights, they are stored and loaded to pthread application by partitions. Each partition has name node_name_w0, node_name_w1,..., node_name_wN. For Conv nodes one partition = one convolutional filter weights tensor. For dense layers one partition = up to 100 dense neurons' weights.

Extract weights and biases from the ONNX model.

To extract weights and biases from the ONNX DNN model, use --onnx-weights generation flag.
**NOTE:* For weights extraction you need python + python onnx + python json installed (see Requirements & Installation).
Example: Extract weights for ../src/espam/examples/DNN/onnx/mnist.onnx DNN model.

```
1. $ cd espam/bin
2. $ ./espam --generate ../src/espam/examples/DNN/onnx/mnist.onnx --onnx-weights
```

Expected output: file directory output_models/CNTKGraph/weights_npz containing .npy arrays with CNTKGraph dnn model weights and biases.

Reuse weights and biases

Weights and biases can be generated once for an application and then reused for the same model with any level of exploited parallelism. E.g. weights should NOT be regenerated for the same DNN model with 20 and 30 blocks.

Load weights and biases into the pthread application.

The loading of weights and biases into pthread application is done by the dataLoader class, automatically generated by espamAI for every pthread application.

**NOTE:* dataLoader utilizes CNPY library to load weights into pthread application. To use the dataLoader, you need to compile CNPY library as descibed in paragraph CNPY library compilation before running the pthread application.

Input data

By default, pthread application uses the input data in .npy (numpy array) format, located in ../../data/inputs folder (from pthread application sources). The ../../data/inputs folder contains npy arrays input0.npy, input1.npy,...,inputN.npy, where every input*.npy corresponds to one DNN input sample. *This arrays should be obtained manually*. The examples of numpy input data, are provided in ../espam/src/espam/examples/DNN_data

NOTE: If the defaule data folder is empty, you will see a message: *application repetitions computation error: null* . In this case there are 2 possible solutions:

Numpy data

1) 1.1. copy manually obtained inputs examples into ../../data/inputs folder
   1.2 open appMain.cpp of the generated pthread application and find the loop over the input samples:
   `for(int inp_img =0; inp_img < 0; inp_img++)`
   1.3. Change the number of input samples to the number of samples you copied, e.g.,
   `for(int inp_img =0; inp_img < 10; inp_img++)`
   1.4. make and ./run your pthread application

Alternative data

2) Open appMain.cpp file and add your own data loaduing after the comment "//TODO: load your data here."

CNPY library

The DNN parameters (weights and biases) as well as I/O data are stored in .npy files and loaded in pthread application during runtime. To load weights/biases/input data into pthread applications the cnpy library https://github.com/rogersce/cnpy is used. Before you can use the pthread application you need to download the cnpy library and compile it for the platform, where you want to run your pthread application (use your specific compiler, if needed). To use pthread application you should have both cnpy-master (sources) and cnpy-build (.so) copied on your board. For examples, given below, the cnpy-master and cnpy_build are located in espam/lib directory and installed as following:

1. Download cnpy from https://github.com/rogersce/cnpy
2. cd to /espam/lib and unzip cnpy-master there
3. $mkdir cnpy_build
4. open README in cnpy-master and follow them

DNN operators libraries

Currently pthread application provides Leiden University custom DNN operators library (--libCPU option). However, this library is not optimal and should not be used in real application. It is strongly recommended to use you can use your own DNN operators. To use your own DNN operators:

1. Create a folder, containing your own DNN operators sources
2. Use -n parameter of configure.sh file, automatically generated by espamAI to copy your library sources into the generated pthread application sources
3. Open appFunc.cpp and write your operators calls in appFunc::execute function
4. Make and run pthread application.

Configure.sh

Every pthread application containts automatically generated *Configure.sh* file. This file serves to add external source files to the generated pthread application and add libcnpy.so to the LD_LIBRARY_PATH. The external files include some sources of CNPY library (**-s**), compiled CNPY library (**-b**) and [optionally] DNN operators libraries (**-f**).

Addition of external sources

Example1: Use of configuration file with no DNN operators libraries:

```
$sh ./configure.sh -b ../cnpy_build -s ../cnpy-master
```

Expected output:

```
configuration status: success
```

This command will copy files from cnpy_build and cnpy_master folders and add them to the pthread application sources. The copied files are used for processing application weights, biases and I/O data, stored in .npy arrays.

Example 2: Use of configuration file with DNN operators library sources, located in ../DNN_func_libs/CPU_lib

```
$sh ./configure.sh -b ../cnpy_build -s ../cnpy-master -f ../DNN_func_libs/CPU_lib
```

Expected output:

```
configuration status: success
```

This command will copy files from cnpy_build and cnpy_master folders as well as every file in ../DNN_func_libs/CPU_lib folder.

Addition of libcnpy.so to LD_LIBRARY_PATH

If configuration status was ‘success, but after the running the generated pthread application (./run) you see “*./run: error while loading shared libraries: libcnpy.so*: *cannot open shared object file: No such file or directory*”, then manually perform the commands:

```
$ LD_LIBRARY_PATH=./:LD_LIBRARY_PATH
$ export LD_LIBRARY_PATH
$./run
```

Examples

**NOTE: Hereinafter all examples are performed from .../espam/bin directory.* To perform examples from arbitrary directory *my_dir*

- Instead of ./ in espam call provide absolute path to espam/bin or relative path to espam/bin from *my_dir*
- Provide absolute path(s) to input file(s) or relative path(s) to input file(s) from *my_dir*

The data, used for examples, provided below is copied from ../espam/src/espam/examples/DNN_data/CNTKGraph to CNTKGraph folder, already containing weights. Therefore, before Examples 1-5 the CNTKGraph folder already looks like:

```
<CNTKGraph>
        -/weights_npz //folder with weights and biases  in .npy array.
        -/data //folder with I/O data represented as .npy arrays
                -/input  //input .npy arrays that have names input0, input1,...,inputN
                -/output  //output .npy arrays, that have names output0, output1,...,outputN
```

Example 1: Simple application

Description:
Generate application for ../src/espam/examples/DNN/onnx/mnist.onnx DNN model, map application nodes randomly on 4 cores. Use CPU DNN operators library. Make and run the generated application.

1. $ ./espam --generate ../src/espam/examples/DNN/onnx/mnist.onnx --pthread --cores 4 --libCPU

Expected output:
espamAI-Pthread application generated in: ./output_models/CNTKGraph/pthread/app/

2.$ cd output_models/CNTKGraph/pthread/app/
$ sh ./configure.sh -b ../../../../../lib/cnpy_build -s ../../../../../lib/cnpy-master -f ../../../../../lib/DNN_func_libs/CPU_lib

Expected output:
configuration status: success

3. $ cd output_models/CNTKGraph/pthread/app/

4. Provide input data. *NOTE: this step is skipped in examples below, because it can be done only once for one DNN model.*

```
        $ cp -R ../../../../../src/espam/examples/DNN_data/CNTKGraph/data  ../../
        - open appMain.cpp and manually change
                for(int inp_img =0; inp_img < 0; inp_img++)
        into
                for(int inp_img =0; inp_img < 1; inp_img++)
```

```
5 $ make
```

Expected output:

```
g++ -std=c++11 -pthread  -c -g Input3.cpp
g++ -std=c++11 -pthread  -c -g Convolution28.cpp
........
g++ -std=c++11 -pthread  -o run Input3.o Convolution28.o ... fifo.o  -L./ -lcnpy -lz
```

```
6. $ ./run
```

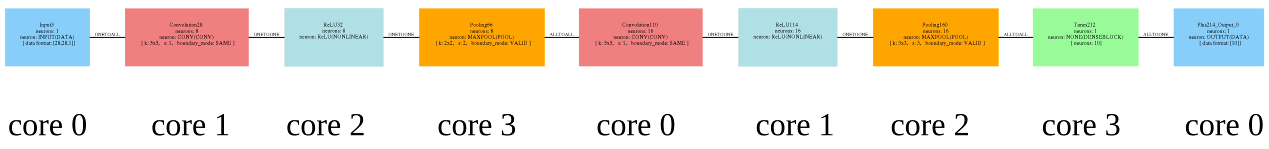
Expected output: // results for 1 input image

```
app output: 975.670288 -618.724304 6574.567871 668.028442 -917.270813 -1671.635620
-1952.760376 -61.549927 -777.176331 -1439.531494
```

NOTE: If after ./run you see "./run: error while loading shared libraries: libcnpy.so: cannot open shared object file: No such file or directory", performs commands:

```
$ LD_LIBRARY_PATH=./:LD_LIBRARY_PATH
$ export LD_LIBRARY_PATH
$./run
```

DNN topology and mapping:



Note: DNN topology can be generated as .dot file (see instructions for --dot option in espamAIMan.pdf)

Example 2: Debug mode

Description:
Generate application for ../src/espam/examples/DNN/onnx/mnist.onnx DNN model, map application nodes randomly on 4 cores. Use LU DNN operators lib. Print debug information.
    Make and run the generated application.

NOTE: Application assumes, that example input data is provided in ../../data folder (see Example1)

```
1. $ ./espam --generate ../src/espam/examples/DNN/onnx/mnist.onnx --pthread --cores 4 --debug
--libCPU
```

Expected output: as in Example 1.

```
2.$ cd output_models/CNTKGraph/pthread/app/
$ sh ./configure.sh -b ../../../../../lib/cnpy_build -s ../../../../../lib/cnpy-master -f
../../../../../lib/DNN_func_libs/CPU_lib
```

Expected output:

```
configuration status: success
```

```
3. $ make
```

Expected output: as in Example 1.

4. $ ./run

```
./../../weights_npz/Convolution28_w0.npy loaded
./../../weights_npz/Convolution28_w1.npy loaded
.....
./../../weights_npz/Plus214_b.npy loaded

Application topology is constructed!

../../data/inputs/input0.npy loaded
Joined with thread thread_Input3
Joined with thread thread_Convolution28
Successfully set thread 982300416 to cpu core 1
Input3 finished!
Joined with thread thread_ReLU32
Successfully set thread 973907712 to cpu core 2
Successfully set thread 965515008 to cpu core 3
Joined with thread thread_Pooling66
Successfully set thread 957122304 to cpu core 0
Joined with thread thread_Convolution110
Joined with thread thread_ReLU114
Joined with thread thread_Pooling160
Joined with thread thread_Times212
Successfully set thread 855627520 to cpu core 0
Joined with thread thread_Plus214_Output_0
Successfully set thread 847234816 to cpu core 1
 Convolution28 finished!
Successfully set thread 872412928 to cpu core 2
 ReLU32 finished!
Successfully set thread 864020224 to cpu core 3
 Pooling66 finished!
Successfully set thread 948729600 to cpu core 1
 Convolution110 finished!
 ReLU114 finished!
 Pooling160 finished!
 Times212 finished!
 Plus214_Output_0 finished!
Generated program run 0 is finished!

Input3 output
0 0 0 0 0 0 0 0 0 1
Convolution28 output
-0.16154 -0.16154 -0.16154 -0.16154 -0.16154 -0.16154 -2.20347 -13.6354 -9.81617 7.12668
ReLU32 output
0 0 0 0 0 0 0 0 0 7.12668
Pooling66 output
0 0 0 0 17.5782 3.6093 7.46961 8.82315 4.90488 20.1006
Convolution110 output
-0.185185 3.15493 5.96148 -48.6038 -135.497 -254.373 -621.933 -902.048 -753.847 -764.047
ReLU114 output
0 3.15493 5.96148 0 0 0 0 0 0 0
Pooling160 output
5.96148 0 310.082 773.373 0 0 0 1660.32 0 0
```

```
Times212 output
975.67 -618.724 6574.57 668.028 -917.271 -1671.64 -1952.76 -61.5499 -777.176 -1439.53
```

DNN topology and mapping: as in Example 1.

Example 3: Automated application parallelization

Description:
Generate application for ../src/espam/examples/DNN/onnx/mnist.onnx DNN model, map application nodes randomly on 4 cores. Split DNN graph by performance bottleneck, into<=30 blocks (for split algorithm see espamAIMan.pdf). Make and run the generated application.

NOTE: Application assumes, that example input data is provided in ../../data folder (see Example1)

1.
```
$ ./espam --generate ../src/espam/examples/DNN/onnx/mnist.onnx --pthread -bb 30 --cores 4
--libCPU
```

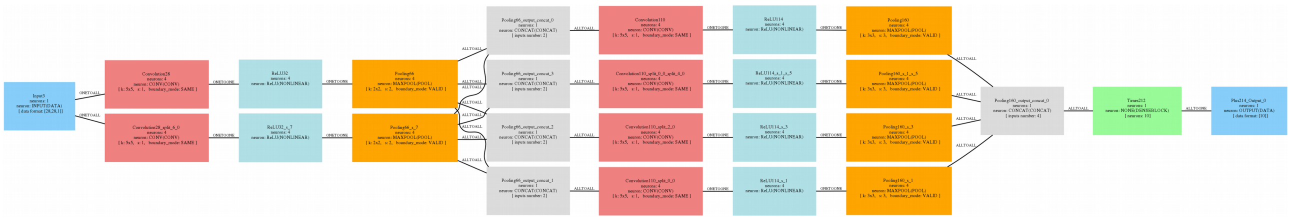
Expected output: as in Example 1.

2.
```
$ cd output_models/CNTKGraph/pthread/app/
sh ./configure.sh -b ../../../../../lib/cnpy_build -s ../../../../../lib/cnpy-master -f
../../../../../lib/DNN_func_libs/CPU_lib
```

Expected output:
```
configuration status: success
```

3.
```
$ make
$ ./run
```
Expected output:
```
app output: 226.048447 655.682007 1505.263794 449.827209 -1047.294556 -63.064754
-360.705994 -22.146065 -369.674591 -421.463135
```

DNN topology and mapping:



core 0	core 1	core 3	core 1	core 3	core 3	core 3	core 3	core 3	core 0	core 1
	core 2	core 0	core 2	core 0	core 0	core 0	core 0			
				core 1	core 1	core 1	core 1			
				core 2	core 2	core 2	core 2			

Example 4: Mapping file usage

A mapping file contains a simple description of proceesors assignment to the pthread application nodes. The mapping specification takes into account both DNN application and used platform and can be either written manually, or automatically obtained from input pair {application, platform}.

The mapping_mnist.xml, designed for a DNN model from Example 1 and a platform with 6 CPU cores and 1 GPU core:

```
<mapping name="myMapping">
            <processor name="cpu_0">
                        <process name="Input3"/>
             </processor><processor name="cpu_1">
                        <process name="ReLU32"/>
                         <process name="Pooling66"/>
            </processor>
            <processor name="cpu_2">
                        <process name="Convolution110"/>
                        <process name="ReLU114"/>
            </processor>
            <processor name="cpu_3">
                        <process name="Pooling160"/>
                        <process name="Times212"/>
            </processor>
            <processor name="cpu_4">
                        <process name="Plus214_Output_0"/>
                        <process name="Convolution28"/>
            </processor>
            <processor name="cpu_5">
            </processor>
            <processor name="gpu">
            </processor>
</mapping>
```

Manually obtained mapping

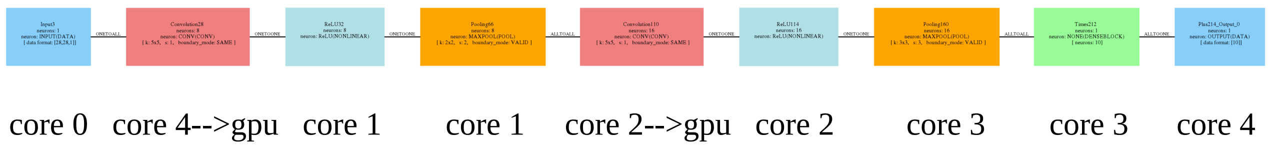
To apply a manually obtained mapping file, use --mapping argument:

NOTE: 1. The names of the processes in the mapping file should match the names of the processes in the input (DNN/CSDF) application.
2. NO processes should be assigned to GPU in a mapping file. The call of GPU is a subprocess performed by a CPU node. The call of GPU by CPU node is currently an application (not a mapping) property.

1. $ ./espam --generate ../src/espam/examples/DNN/onnx/mnist.onnx --pthread --mapping ../src/espam/examples/mapping/mapping_mnist.xml

 2 – 4. as in Example 1.

DNN topology and mapping:



Automatically obtained mapping

The mapping specification can be utomatically obtained from an input pair {application, platform}, provided to espamAI as input parameters. To use an automatically generated mapping file, provide an application and platform specification. Currently there are two types of platform specification supported:

1. ESPAM platform specification: see examples in .../espam/src/espam/examples/platform
1. NEURAGHE platform specification: see examples in .../espam/src/espam/examples/platform/neuraghe_arch

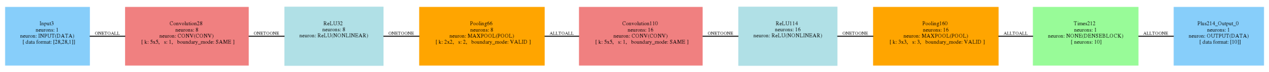
ESPAM platform

To use ESPAM platform specification utilize --platform or -p argument:

```
1. $ ./espam --generate ../src/espam/examples/DNN/onnx/mnist.onnx --pthread -p /
../src/espam/examples/platform/Jetson.xml
```

2 – 4. as in Example 1.

DNN topology and mapping:



core 0   core 4-->gpu   core 1   core 1   core 2-->gpu   core 2   core 3   core 3   core 4

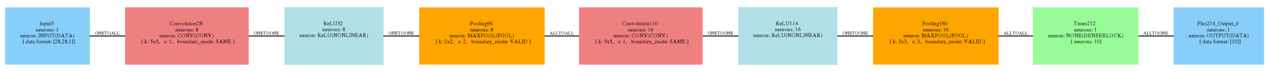
NEURAGHE platform

To use NEURAGHE platform specification utilize --na-arch argument:

```
1. $ ./espam --generate ../src/espam/examples/DNN/onnx/mnist.onnx --pthread --na-arch /
../src/espam/examples/platform/neuraghe_arch/NEURAghe_architectural_description.json
```

2 – 4. as in Example 1.

DNN topology and mapping:



ARM0   ARM1   ARM0   ARM1   ARM0   ARM1   ARM0   ARM1   ARM0

Print mapping file

To print automatically generated mapping in a file use –map-xml argurment:

```
$ ./espam --generate ../src/espam/examples/DNN/onnx/mnist.onnx --map-xml --na-arch /
../src/espam/examples/platform/neuraghe_arch/NEURAghe_architectural_description.json
```

Expected output:

```
Mapping file generated in: ./output_models/CNTKGraph/CNTKGraph_to_ZynqPS.xml
```

Work in progress

1. Data tiling

2. New GPU library support + opimal blocks number support