Abstract

The group develops analog neural network chips [5] [4] [2], which are able to emulate PyNN [1] modeled networks. As the realization of a PyNN modeled network on an analog neural network chip is not trivial, the usage of an appropriate software stack is inevitable. Especially the questions where the modeled neurons are placed on the chip and how the connections between them are realized by setting the available switches have to be resolved by software. This mapping process for the most recent chip (HICANN) is described in Jeltsch’s Ph.D. thesis [3]. This report describes where the model neuron placement information can be found in the code and how the access to this information was simplified. Furthermore the class structure and control flow from parts of the mapping software is presented in a comprehensible way.

1 Introduction

The BrainScales group in Heidelberg develops a PyNN [1] based Python interface (pyhmf) to emulate modeled networks on analog neural network chips user friendly. A software development goal is to uncouple the different tasks: modeling, mapping, running the experiment and getting the results on software level. For analysis purposes the user wants to know which electric circuits (denmen circuits) realize a distinguished model neuron. This information could not be found explicitly in the existing code and was added in an appropriate class.

This document explains firstly how the control flow for the neuron placement takes place in the mapping software (marocco). Second the reader gets an overview of the
placement result datastructure and where to find the neuron placement information. Third the implemented simplified access to the placement information is presented. In the end of the report problems of the existing mapping software are discussed.

2 Control Flow

The mapping software is written and structured in c++11 classes. Each class has a run member function, which is generally called to solve the problem the class was designed for. In figure 2 you can see in which run functions the displayed objects are created. The initialization of marocco begins by creating and running the object of class main. Objects of class MappingJob are created to run mapping processes in parallel. Therefore objects of class Mapper are created and run by objects of class MappingJob. Objects of class Mapper organize all concrete mapping processes like model neuron placement, merger tree routing and routing. The concepts of these steps are described in detail in Jeltsch’s Ph.D. thesis [3]. For this work the called object of class Placement is important, as this handles the actual assignment from model neurons to denmen circuits. This object creates and runs an object of class ReverseMapping, which provides the assignment from output spikes to model neurons. Therefore this object is passed from Placement → Mapper → MappingJob → Reader. The Reader gets the ReverseMapping and the ObjectStore, fetches the output spikes with low level hardware commands and writes the model network output to the ObjectStore. Objects of class ObjectStore contain various information about the PyNN modeled network for the user. Before the object of class Reader is created an object of class MappingJob creates and runs an object of class Control, which takes the hardware configuration, configures the hardware and runs the experiment on chip.

3 Placement Result Data Structure

In figure 3 you can see the class diagram of the result class for the mapping process. It is an incomplete diagram, thus it contains only the parts needed to understand the model neuron placement. Mostly the displayed classes are used for wrapping purposes. The important part is the std::unordered_map<graph_t::vertex_descriptor, assignment::Mapping>, as this class gives an assignment of populations to a vector of Hardware objects. Objects of class Hardware describe a certain amount of denmen circuitson one NeuronBlockOnHICANN with an offset (figure 3). The assignment of model neurons to denmen circuits is saved implicitly to be consecutive. This means for example that model neuron number one of a population is emulated by the first HardwareNeuronSize denmen circuitsfrom the head of std::vector<Hardware>.

4 The Implementation

The ReverseMapping class was choosen for modification, as the assignment from model neurons to denmen circuitsbelongs contextually to the ReverseMapping. Furthermore
Figure 1: Control Flow in marocco necessary for the placement process.
Figure 2: Class diagram of the mapping result data structure.
Figure 3: Schematic description what an object of class NeuronBlockOnHICANN describes.
Figure 4: Used coordinate system to identify a denmen circuit globally.

this assignment can easily be written to the ObjectStore, thus being available in the pyhmf interface. Moreover the class was renamed to LookupTable, as it contains from now on the assignment from L1Adresses to model neurons and from model neurons to denmen circuits. To identify a denmen circuit globally a structure denmen_id was added to LookupTable.h, which contains a global HICANN coordinate and a 2D vector. The 2D vector gives the denmen circuit’s position on a HICANN (fig. 4). In figure 4 you can see the part of the code, which was implemented in the run member function of class LookupTable. This code provides an easy access to the model neurons to denmen circuits assignment.

The implementation delivers correct results if a model neuron is emulated by $2 + 2n$ with $n \in [0, 31]$ denmen circuits. The denmen circuits, which emulate a model neuron, must be aligned consecutive such that a model neuron is emulated equally by denmen circuits from the first row and denmen circuits from the second row. Another constraint is that a HICANN must be divided exactly in eight equal NeuronBlockOnHICANN objects. As the mapping software actually has the same constraints, the implemented assignment is correct.
auto const& onm = get<0>(result); // get output neuron mapping
result
auto const& placement = onm.placement(); // get the distributed
placement map
auto unordMap = placement.getMap(); // get the std::unordered_map
for iteration
int bio_neuron_index;
// loop over all populations wrapped as graph vertex type
for (auto entries = unordMap.begin(); entries != unordMap.end(); ++entries) {
    auto const& mapping = entries->second;
    std::vector<assignment::Hardware> const& am = mapping.
    assignment();
    auto const& population = *popmap[entries->first];
    bio_neuron_index = 0; // set index of first bio neuron in
    a population to zero
    // each population has a vector with assigned denmen
circuits
    // wrapped in marocco/assignment/Hardware.h
    for (std::vector<assignment::Hardware>::const_iterator
        hardware = am.begin(); hardware != am.end(); ++
        hardware) {
        auto denmen_count = (*hardware).size();
        auto const& terminal_proxy = (*hardware).get();
        // get global HICANN Coord
        auto chip = terminal_proxy.chip;
        // the 256x2 array of denmen circuits on a HICANN
        is distributed into
        // 8 blocks of size 32x2
        auto block = terminal_proxy.block;
        int blockNr = block.value(); // in [0,7]
        auto offset = (*hardware).offset();
        auto hw_neuron_size = (*hardware).hw_neuron_size
        ();
        int bio_neurons_in_terminal = denmen_count/
        hw_neuron_size;
        // iterate over all bio neurons in this terminal
        for (int i = 0; i < bio_neurons_in_terminal; ++i) {
            bio_id bio {population.id(),
            bio_neuron_index};
            // iterate over the belonging denmen
circuits
            for (int d = 0; d < hw_neuron_size; ++d) {
                int x = blockNr * 32 + offset.x +
                i * hw_neuron_size / 2 + d / 2;
                int y = d % 2;
                Point2D point{x,y};
                denmen_id id {chip, point};
                mBio2DenmenMap[bi].push_back(id
                );
                ++bio_neuron_index;
            }
        }
    }
}

Figure 5: Implemented code for easy access to the model neurons to denmen circuitsas-
assignment in LookupTable.cpp's run routine.
5 Discussion and Outlook

To make the implemented assignment available in the pyhmf interface, one has to implement a transfer in the Reader class, where the assignment is given to the ObjectStore. Furthermore an appropriate command has to be built into the pyhmf interface, to get the assignment from the ObjectStore. Two problems of the mapping software are the overloaded class structure and the superfluous templated code. The first problem gets already obvious if you take a look at the class diagram in figure 3. One should keep in mind that this is only a class diagram with important parts for the placement process. Moreover the code contains a lot of unnecessary wrapping classes and typedefinitions (typedef), which make understanding the code more difficult.

The second problem was not mentioned yet. Many classes and functions are written in a generic way by using c++11 templates, although they are only used with one specific set of types. In such a case templates are not mandatory and have a negative effect on code readability.

References


