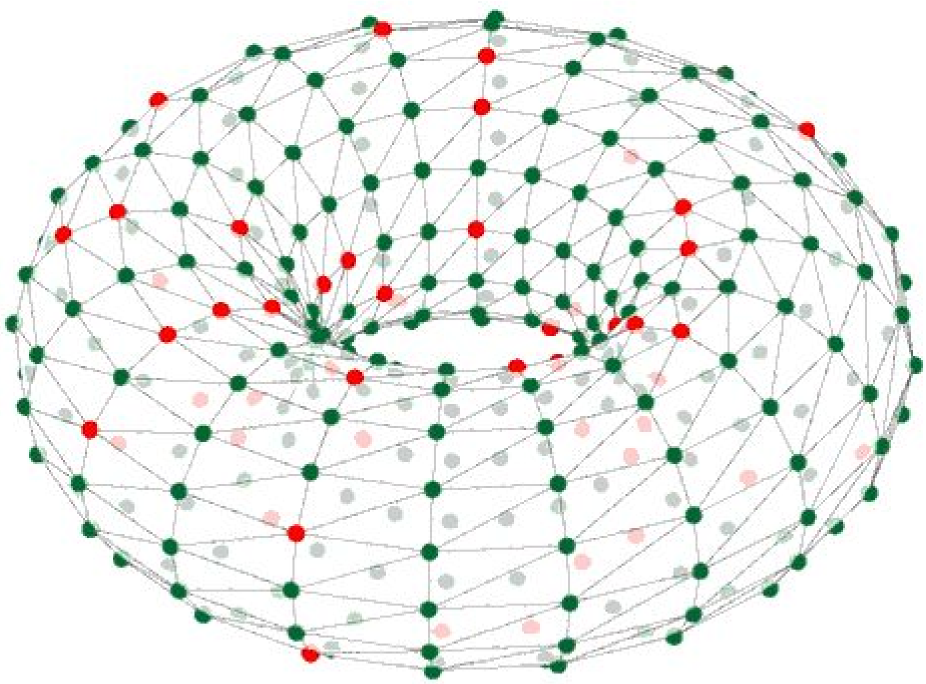
6th SpiNNaker Workshop

Proceedings

September 5th- 9th 2016



Manchester, UK

SpiNNaker

6th SpiNNaker Workshop

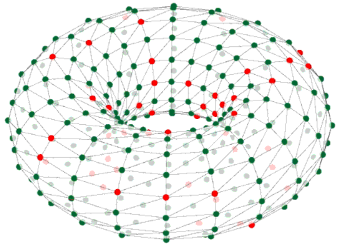
Day 1

September 5th 2016

Time	Session	Presenter
09:00	Registration	
10:00	Workshop Introduction & logistics	SD
10:15	SpiNNaker Hardware & Tools Overview	SD
11:00	Introductory Lab	AGR
12:00	Lunch	
13:00	SpiNNaker architecture and chip resources	ST
14:00	Running PyNN simulations on SpiNNaker	AGR
15:00	Coffee	
15:30	Lab time	
16:30	Close	

Manchester, UK

6th SpiNNaker Workshop



Welcome & Overview

SpiNNaker Workshop
September 2016

European Research Council

Human Brain Project

EPSRC

SpiNNaker

Sessions and Venues

- Two types of sessions:
  - Presentations (some optional!)
  - Lab work (with lab books)
- Three venues:
  - ***Collab 1*** (here) for labs and some presentations
  - ***Atlas-1*** (30m away) for some presentations
  - Area outside this room for lunch/drinks breaks
- Don't leave valuables here overnight!

Breakdown of each day

- Sessions:
  - Start at 9am – promptly!
  - Run to 5pm,except the last day (1pm)
- Breaks:
  - Drinks mid-morning and at 3pm daily (half hour)
  - Lunch at 12pm daily (one hour)
- Fire alarm test on Wednesday at 1pm

WI-FI Access

- *eduroam* is available as normal around the building
- UoM guest accounts available if you need one
  - Please ask!

Day 1 - Monday 5th

Time	Session	Presenter
09:00	Registration	
10:00	Workshop Introduction & logistics	SD
10:15	SpiNNaker Hardware & Tools Overview	SD
11:00	Introductory Lab	AGR
12:00	Lunch	
13:00	SpiNNaker architecture and chip resources	ST
14:00	Running PyNN simulations on SpiNNaker	AGR
15:00	Coffee	
15:30	Lab time	
16:30	Close	

Day 2 – Tuesday 6th

Time	Session	Presenter
09:00	SpiNNaker system software (SARK)	ST
10:00	Coffee (earlier than usual)	
10:30	SpiNNaker API + event driven simulation	LAP
11:30	Writing Applications on SpiNNaker - Overview	SD
12:00	Lunch	
13:00	Introduction to Graph Front End (GFE)	ABS (AGR)
14:00	ybug and gdb walk-through	ST
15:00	Coffee	
15:30	Lab time	
16:30	Close	

Day 3 – Wednesday 7th

Time	Session	Presenter
09:00	Simple data I/O and visualisation	ABS (SD)
10:00	Lab time (coffee at 10:30)	
11:00	Maths & fixed point libraries	MH
12:00	Lunch	
13:00	Adding new neuron models	AGR/MH
14:00	Connecting SpiNNaker to external devices	ABS (DRL)
14:30	Lab time (coffee at 15:00)	
16:30	Close	

Day 4 – Thursday 8th

Time	Session	Presenter
09:00	Adding new models of synaptic plasticity	JK
09:45	Graph Front End – further details	ABS (AGR)
10:30	Coffee	
11:00	Lab time	
12:00	Lunch	
13:00	Using big SpiNNaker machines remotely: The HBP portal	AGR
13:30	Lab time (coffee at 15:00)	
15:30	Demonstration of NENGO language and environment	TBC
16:30	Close	

Day 5 – Friday 9th

Time	Session	Presenter
09:00	Lab time	
10:30	Coffee	
11:00	Lab time	
12:00	Lunch and close	

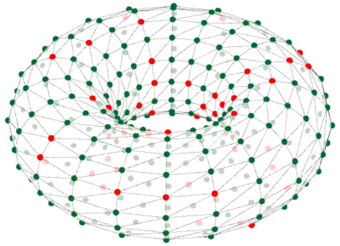
Loan board requests…

- 4-node boards can be loaned out
  - But supply is limited
- Please send an email with your project details to:
  simon.davidson@manchester.ac.uk
- Steve Temple will allocate and log board loans
  - Please don't just take one away….

Feedback…

- We'd appreciate some feedback on…
  - Your workshop experience
  - SpiNNaker hardware
  - SpiNNaker software
  - I'll email you in the next few weeks
  - We hope that you enjoy the workshop!

SpiNNaker Hardware & Software



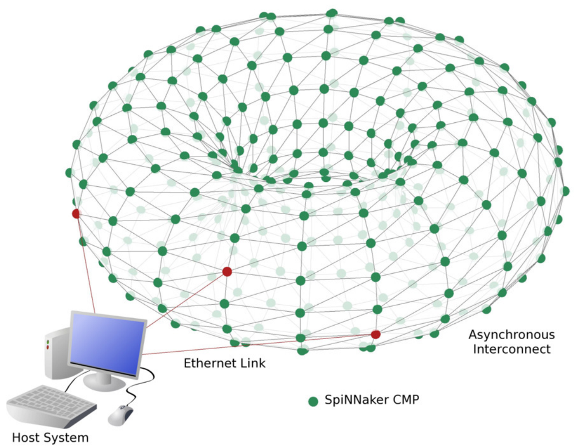
Overview

SpiNNaker Workshop
September 2016

Contents

- What is SpiNNaker?
- SpiNNaker at different scales
- SpiNNaker architecture: chip & system
- Using SpiNNaker

SpiNNaker Project

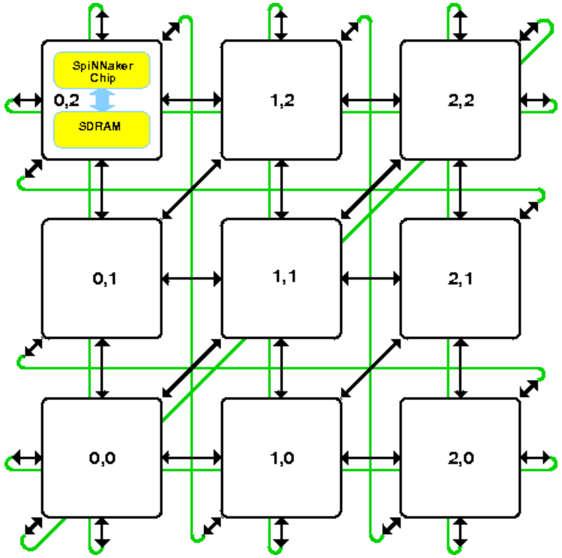


A million mobile phone processors in one computer
Able to model about 1% of the human brain…
…or 10 mice!

How is SpiNNaker Used?

- Some key user communities:
  - **Computational neuroscientists** to simulate large neural models and try to understand the brain
  - **Roboticists** to build advanced neural sensory and control systems
  - **Computer architects** to apply neural theories of computation to non-neural problems

SpiNNaker System



Chip-to-chip communications: Packet routing

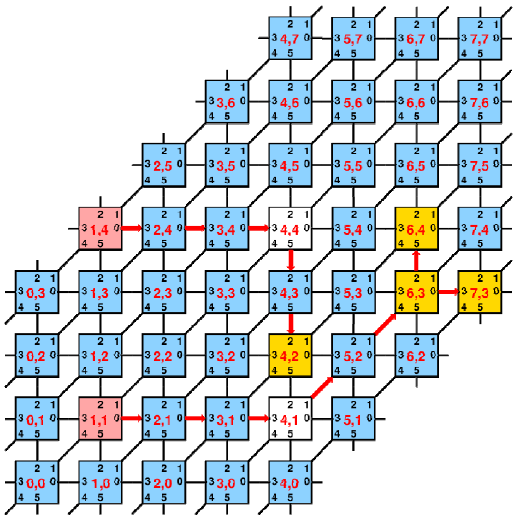
- No memory shared between chips!
- Communicate via simple messages called ***packets:***
  - *40 bit (no data) or*
  - *72 bit (includes 32-bit data word)*
- Four types of routing, most important (for you) is ***multicast***
- Packets used to communicate with the host and external peripherals:
  - Via Ethernet adapter for host comms.
  - Or via chip-to-chip SpiNNaker links for external devices

Routing Types

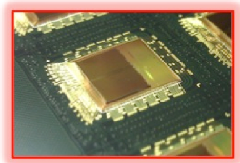
Nearest Neighbour
Point-to-Point
Multicast
Fixed Route

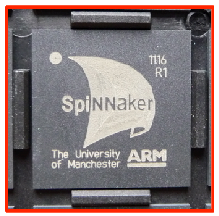
Multicast Routing

- Hardware router on each node
- Packets have a routing key
- Router has a look-up table of {*key, mask, data*} triplets
- If address matches a *key-mask* pair, the associated *data* tells router what to do with the packet



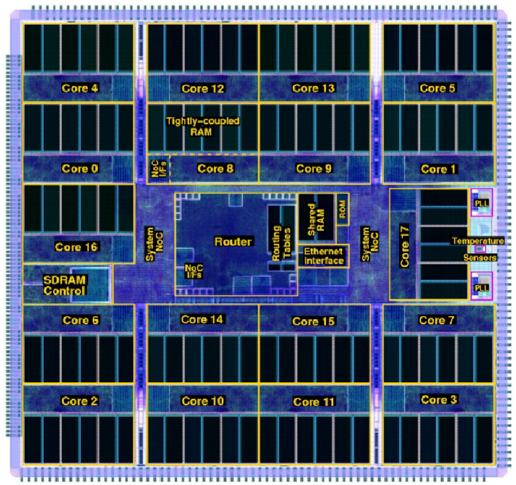
SpiNNaker Chip



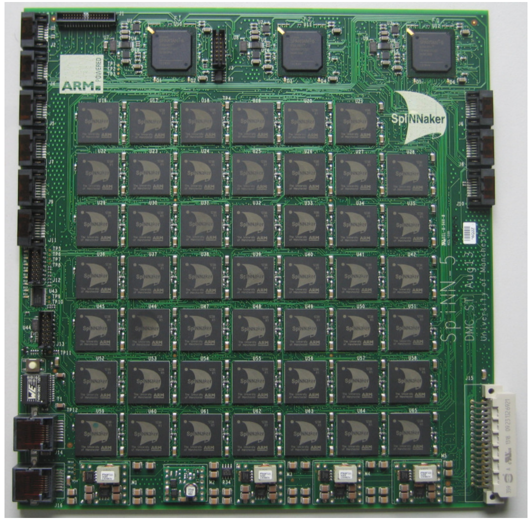


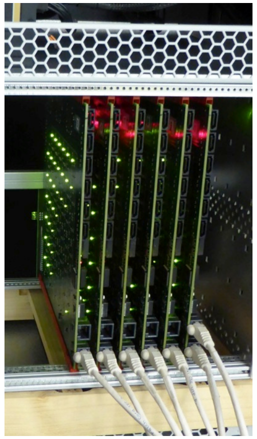
Multi-chip packaging by UNISEM Europe

unisem

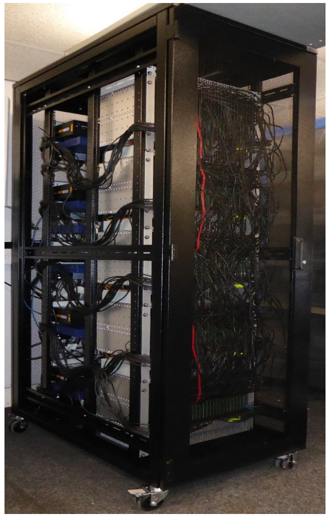


SpiNNaker Boards

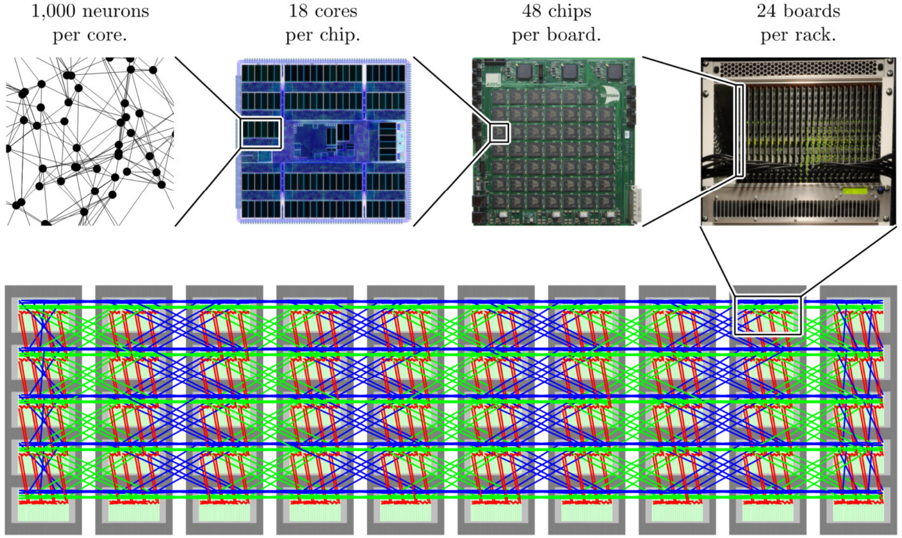




SpiNNaker Machines



Scaling to a billion neurons



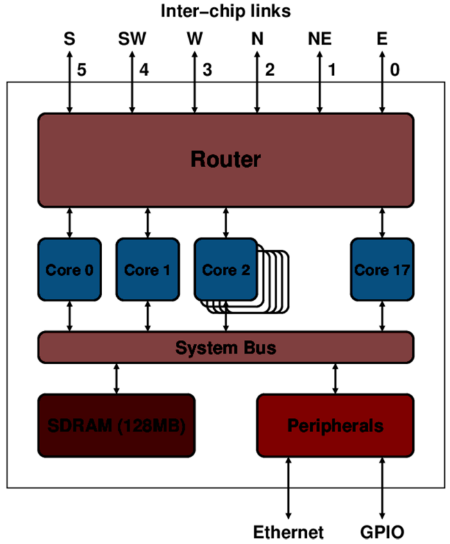
5 racks per cabinet, 10 cabinets.

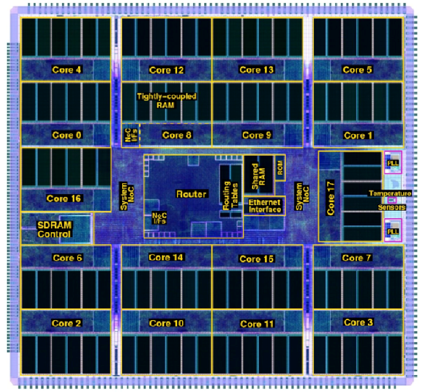
What Next for SpiNNaker?

- Five cabinet machine (500K ARM cores)
  - Now online and available!
  - Open to any research project, in principle
- SpiNNaker2 being developed within HBP
  - New systems by 2020?
- For further information contact: simon.davidson@manchester.ac.uk

Chip Architecture

SpiNNaker Node





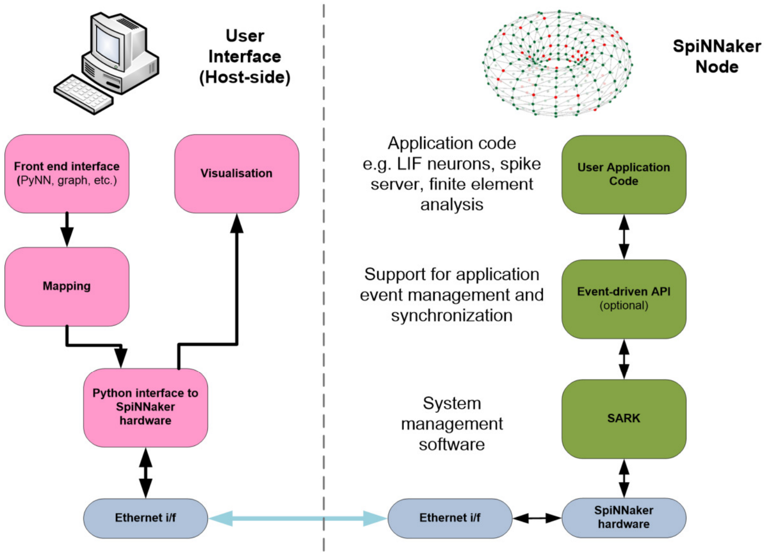
Chip Resources

- 18 cores on a chip:
  - 1 Monitor Processor
  - 16 Application processors
  - 1 fault-tolerant/yield spare
- Each core is an ARM968 processor
  - 200 MHz clock speed
  - No memory management or floating point!
  - Local memories:
    - 32K local code memory (ITCM), 64K local data (DTCM)
    - TCMs are visible only to local processor
- 128MByte SDRAM
  - Shared and visible to all processors on **same node**
- Router:
  - Directs flow of information from core-to-core across the machine

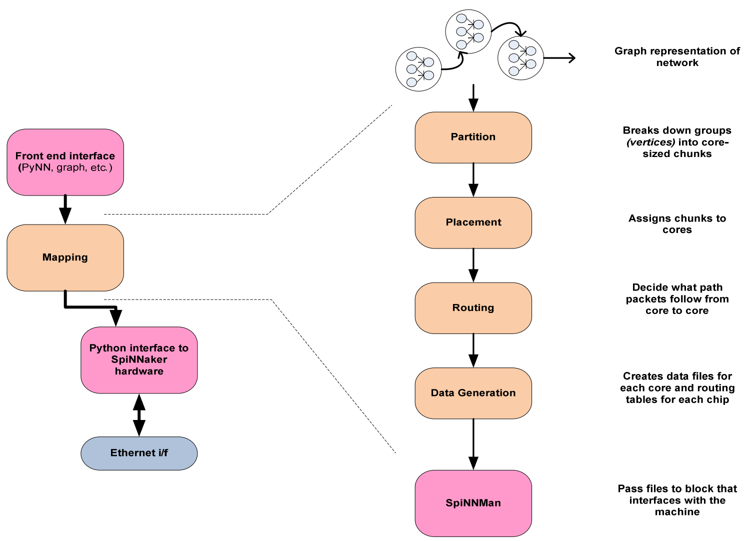
Using SpiNNaker:

The Software Stack

Software Stack



Mapping Process



What Files are Required for Simulation?

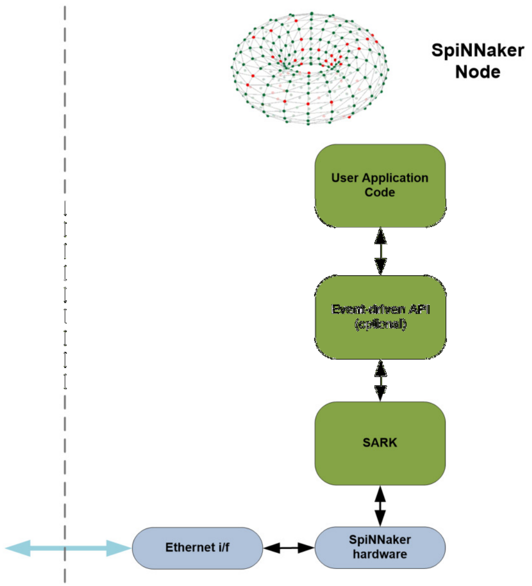
Files Required

For Each Core (16 per chip):

- Application C executable (32KB)
- Application local data (64KB)
- Application shared data (8MB)

For Each Chip:

- Router table
- Any shared data tables



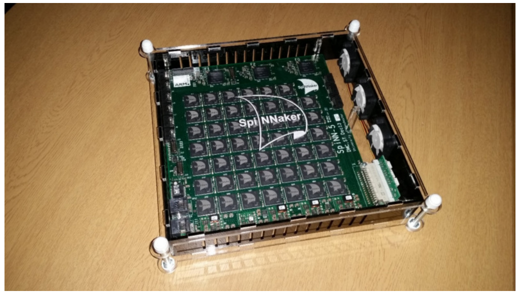
Order of Events (batch mode)

1. Compile network description
2. Map graph to machine
3. Generate data files
4. Load files
5. Synchronise the start on all cores!
6. Simulation runs to completion
7. Hands back control to host
8. Read back results and post-process

End of Overview!

- Much more detail on all of these topics
  - In the sessions to come….
- Any questions for now?
- Just one more thing to add….

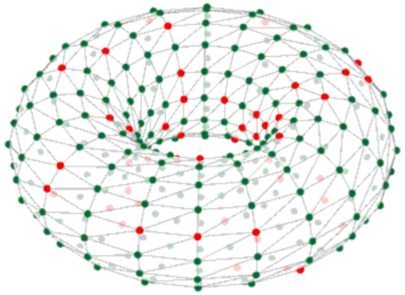
Buying SpiNNaker Hardware



- 48-node board now available for sale
- Non-commercial use only
- 4-node boards can only be loaned (currently!)

- For further information contact: simon.davidson@manchester.ac.uk

SpiNNaker Chip Resources



Steve Temple
SpiNNaker Workshop – Manchester – Sep 2016

European Research Council
Established by the European Commission

Human Brain Project

EPSRC

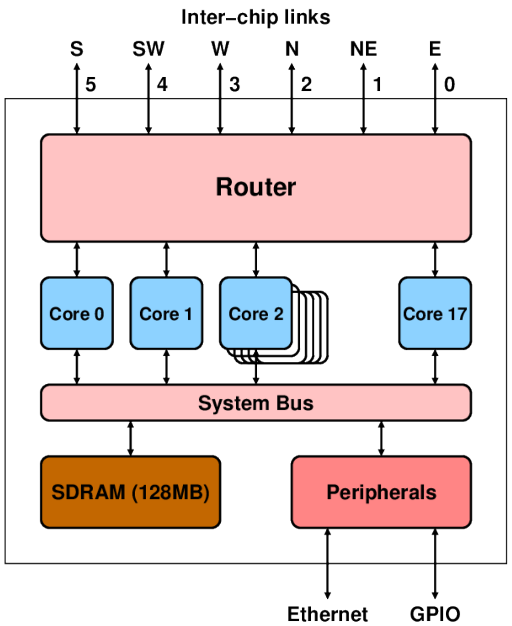
SpiNNaker

Overview

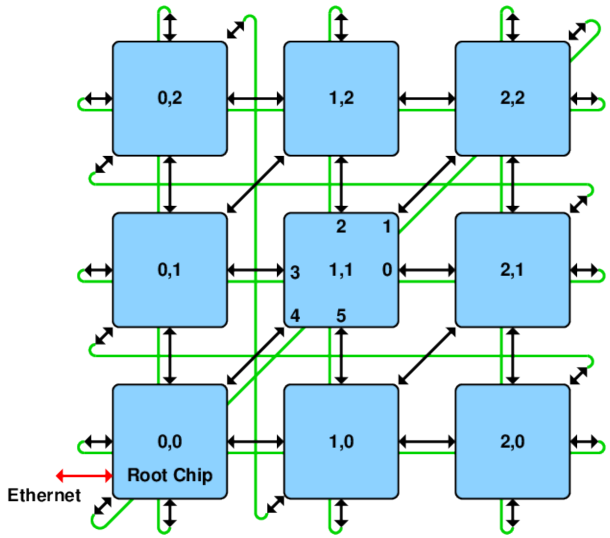
- Chip Architecture
- Core Architecture
- Low-level Communication
  - Packet formats
  - Multicast routing
- High-level Communication – SDP
- Hardware Limitations

Please interrupt if you have a question!

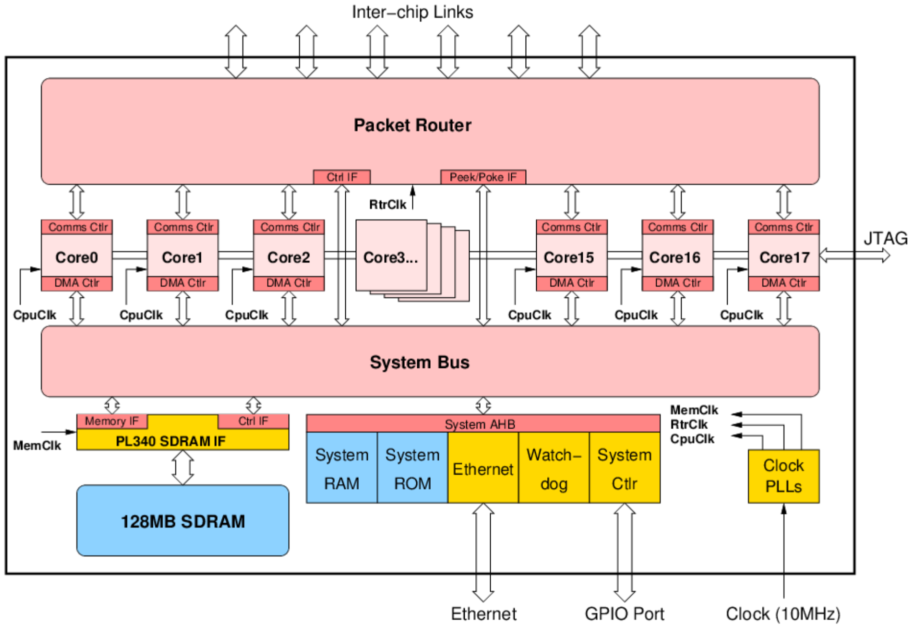
SpiNNaker Chip Outline



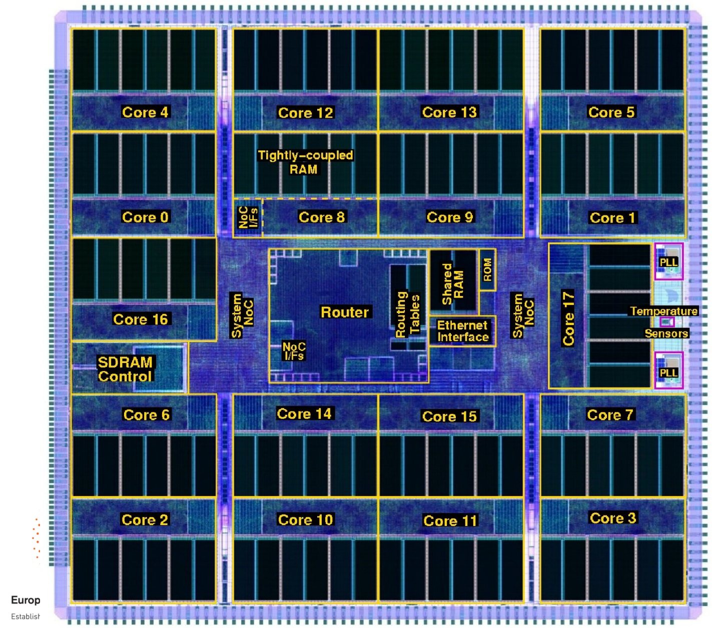
Chip Interconnect



SpiNNaker Chip Details

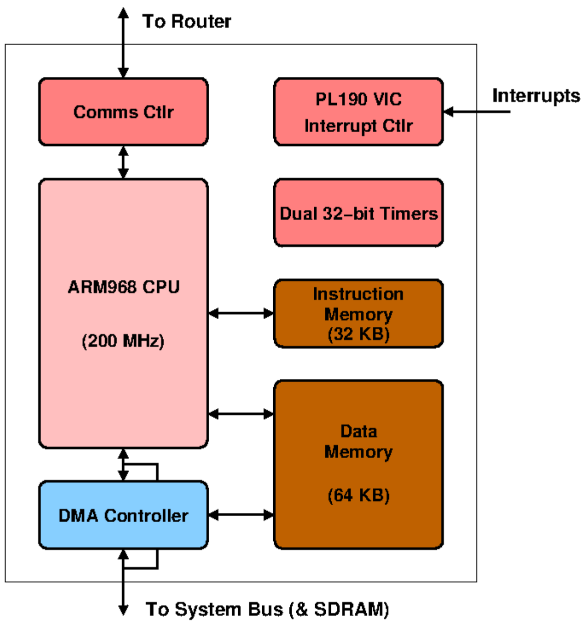


SpiNNaker Chip Layout



- 130nm process
- 10 x 10 mm
- 18 ARM cores with 96K SRAM
- Router
- SDRAM controller
- Asynchronous NoC

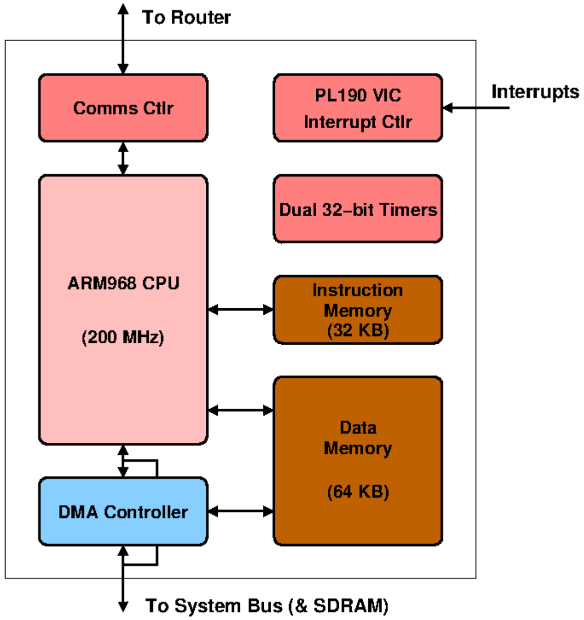
SpiNNaker Core



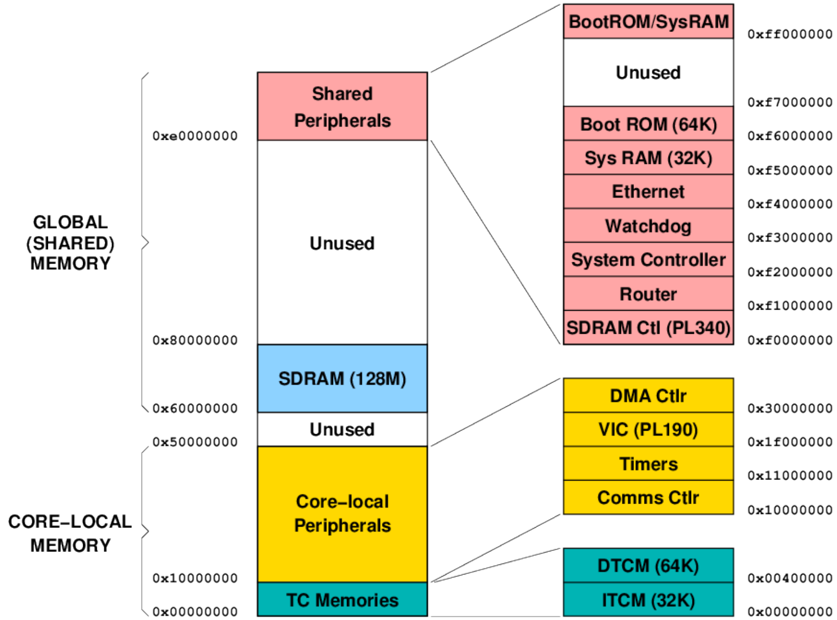
ARM968 CPU

- ARM9 CPU clocked at 200 MHz
- ARM v5TE architecture
  - Supports 32-bit ARM and 16-bit Thumb code
  - Some DSP instruction support - saturated arithmetic, extended multiplies
  - **No floating point hardware!**
- Two Tightly Coupled Memory (TCM) blocks
  - Single cycle (5 ns) access time
  - 32 KB Instruction TCM (ITCM)
  - 64 KB Data TCM (DTCM)
- DMA interface into both TCMs

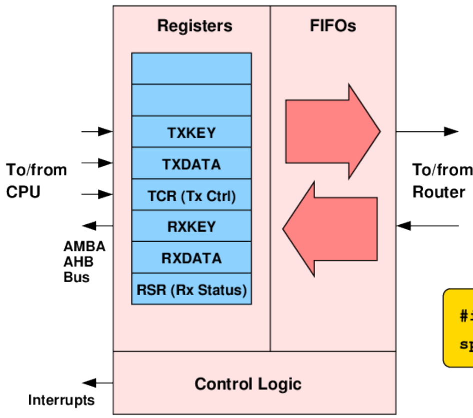
SpiNNaker Core



SpiNNaker Memory Map



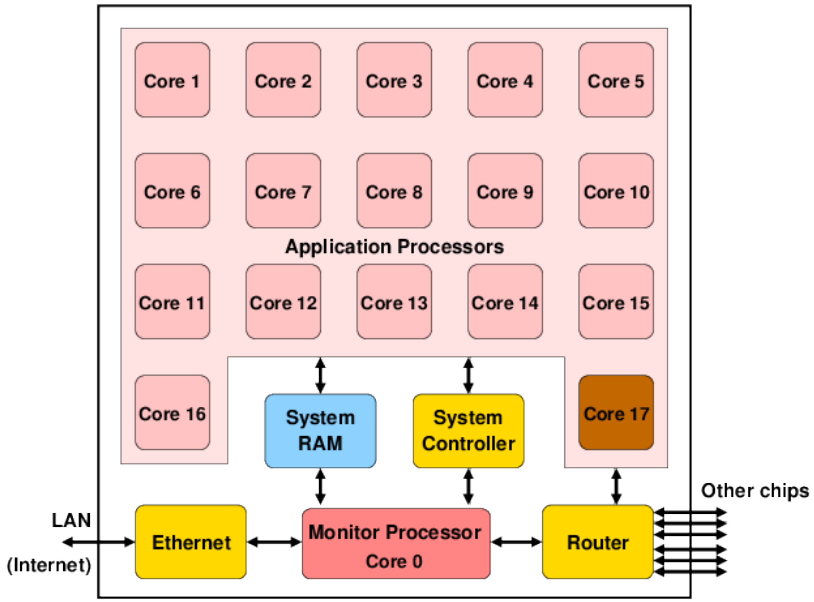
Communications Controller



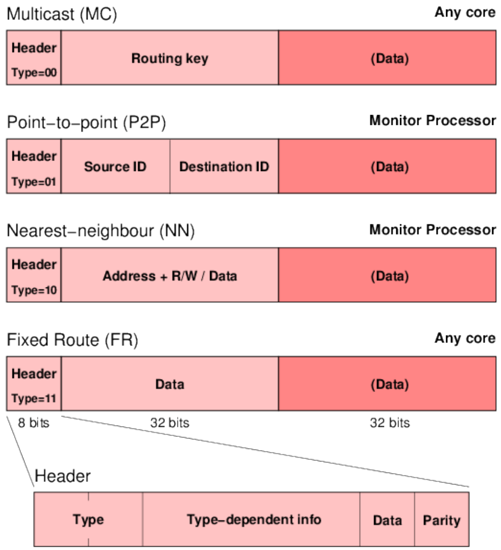

```
#include "spinnaker.h"

void send_mc_pkt (uint key, uint data)
{
  while (cc[TCR] & TX_FULL)
    continue;
  cc[TCR] = PKT_MC_PL;
  cc[TXDATA] = data;
  cc[TXKEY] = key;
}
```

```
#include "spin1_api.h"

spin1_send_mc_pkt (key, data, WITH_PAYLOAD);
```

Monitor Processor & Virtual Cores

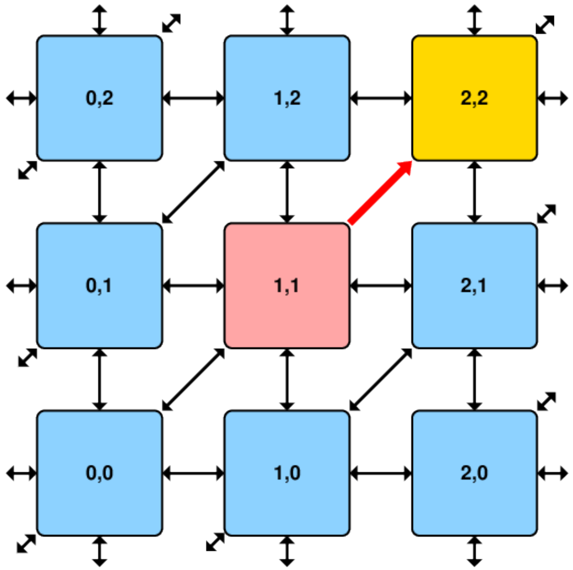


SpiNNaker Packet Types

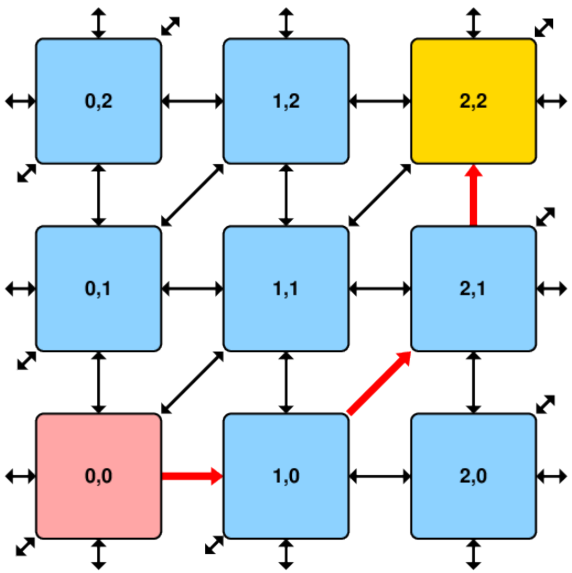



```
uint spin1_send_mc_pkt (uint key, uint data, uint payload);
```

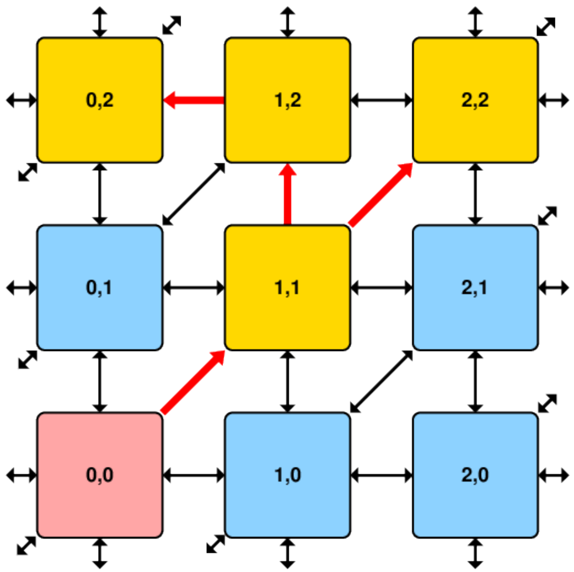
Nearest-neighbour packets



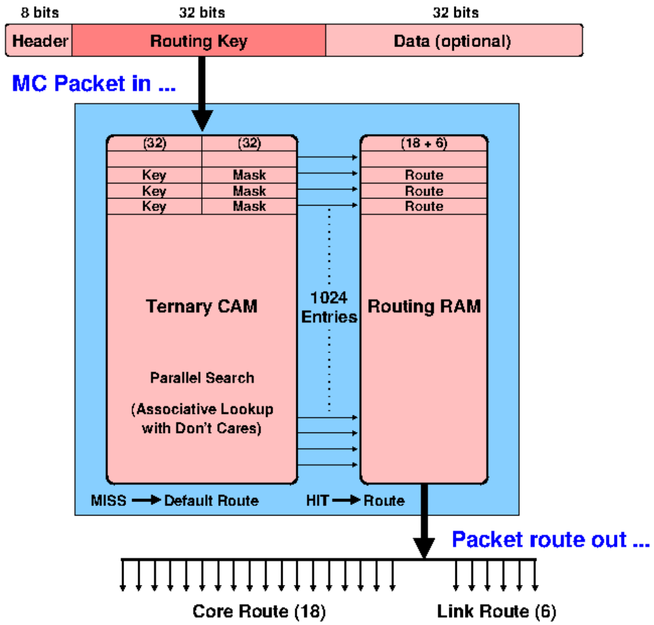
Point-to-point packets



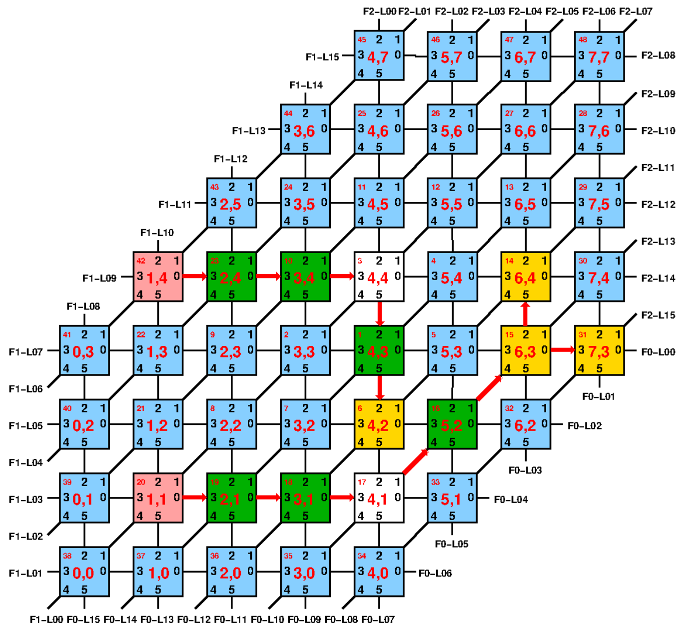
Multicast packets



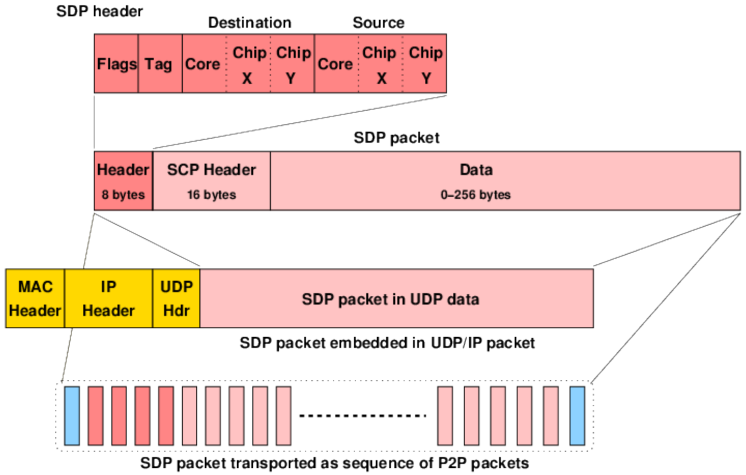
Multicast Packet Router



Multicast Packet Routing

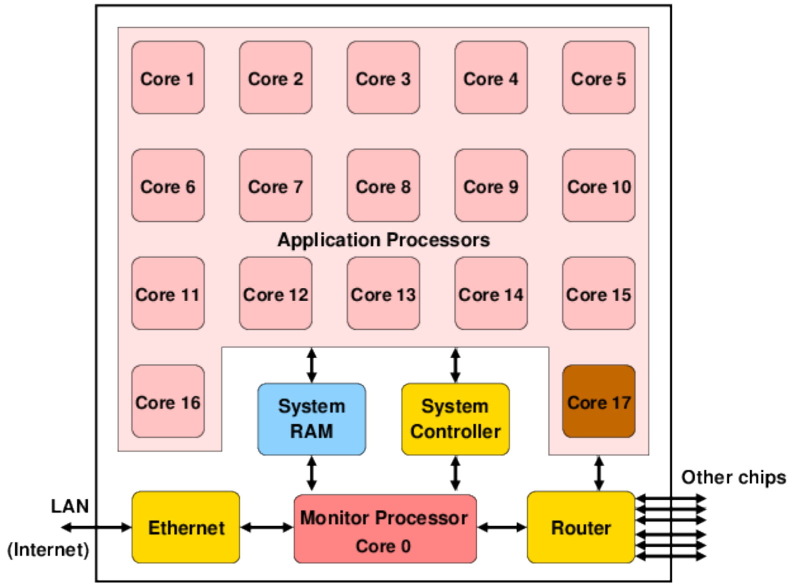


SpiNNaker Datagram Protocol




```
uint spin1_send_sdp_msg (sdp_msg_t *msg, uint timeout);
```

SDP Routing



MANCHESTER 1824

SpiNNaker Hardware Limits

- Processors – 16/17 per chip (but scalable to thousands of chips)
- ARM968 – ARM9 at 200MHz – 220 DMIPS
- Local memory – very limited
  - Instruction memory – 32K bytes
  - Data memory – 64K bytes
- Local Memory access time - 5 ns
- Per chip memory – 128M bytes (shared)
- Shared memory access time
  - Individual accesses - > 100 ns (NB write buffer)
  - DMA accesses ~ 15ns per word

SpiNNaker

MANCHESTER 1824

SpiNNaker Arithmetic Limits

- ARM968 has no floating point hardware
- Options
  - Soft Floating Point – slow and memory hungry
  - Fixed point – uses integer ops
    - Limited range before precision lost
    - Some GCC compiler support (but slowish)
    - Or hand code (C or assembly) for best performance (some libraries available)
- ARM968 has some DSP extensions
  - Saturation, MAC, double operations, CLZ
  - Accessible via compiler intrinsics

SpiNNaker

MANCHESTER 1824

SpiNNaker Packet Limits

- Packet payload is small – typically 32 bits
- Packet bandwidth is limited
- Chip-to-chip links ~ 250M bit/s (5 or 3 M pkt/s)
  - Currently 50% slower via board-to-board links
- CPU packet processing overhead typically 200-1000ns
- Packets can get lost (dropped) in case of congestion – can be “re-injected” in some cases
- Multicast router table is not infinite!

SpiNNaker

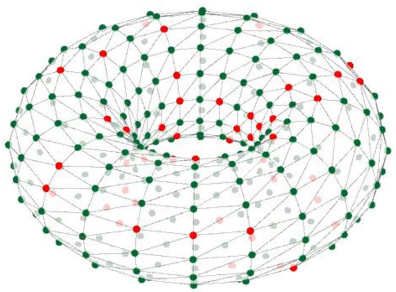
MANCHESTER 1824

SpiNNaker Bandwidth Limits

- Overall I/O bandwidth into the machine is limited
- Currently most external I/O is by 100 Mbit/s Ethernet (and only one interface per board)
- High level I/O via SDP is limited by software overheads
  - Around 10 Mbyte/s to Ethernet-attached chip
  - Around 2 Mbyte/s to 'unattached' chips (via P2P packets)
- Potential for higher I/O bandwidth via SATA links on FPGAs but currently unexploited

SpiNNaker

Running PyNN Simulations on SpiNNaker



Andrew Rowley, Alan Stokes

SpiNNaker Workshop, September 2016

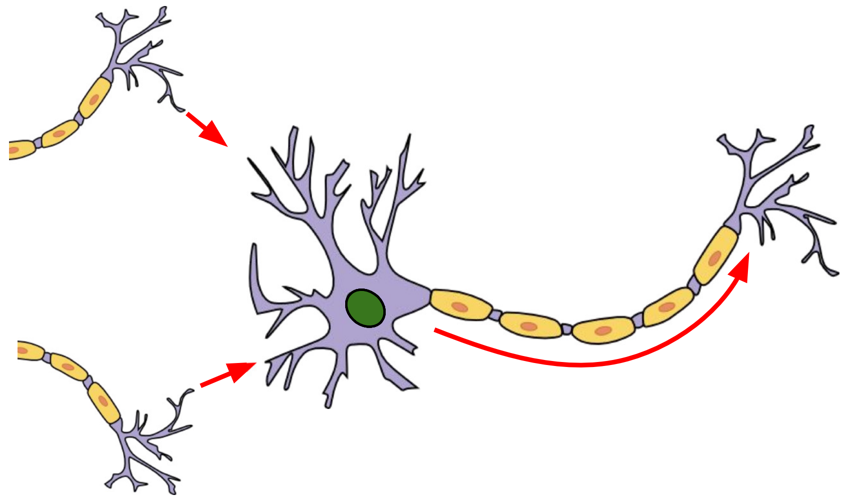
European Research Council
Established by the European Commission

Human Brain Project

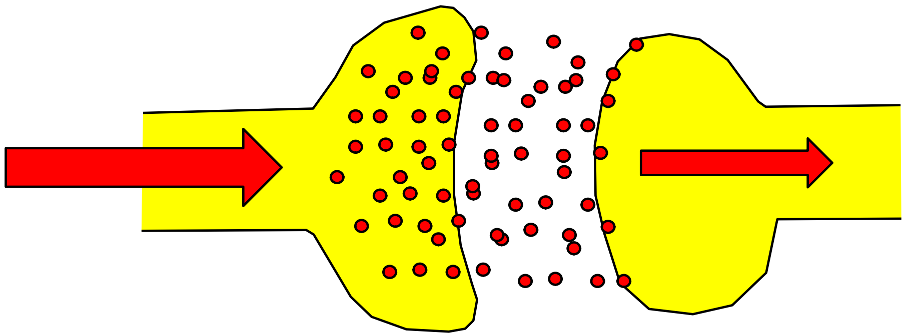
EPSRC

SpiNNaker

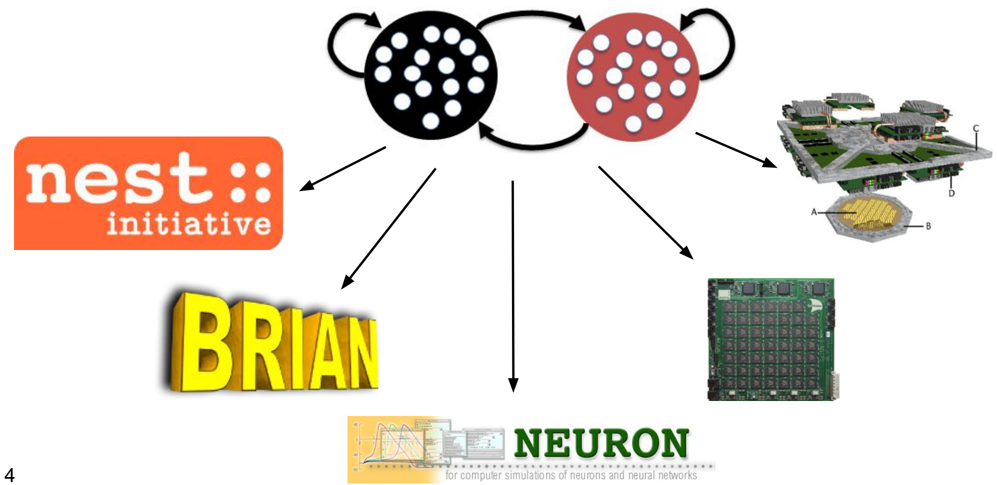
Spiking Neural Networks



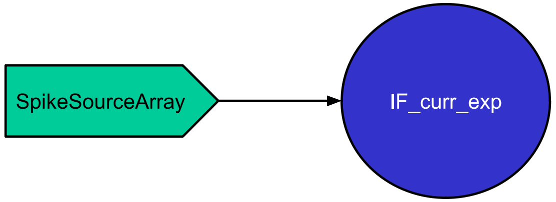
Spiking Neural Networks



What is PyNN?



A Simple PyNN Network



A Simple PyNN Network

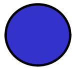
```
import pyNN.spiNNaker as p
```


A Simple PyNN Network

```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
```

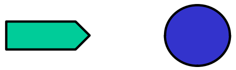

A Simple PyNN Network

```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
pop_1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
```



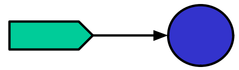
A Simple PyNN Network

```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
pop_1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
                     {'spike_times': [0]}, label="input")
```



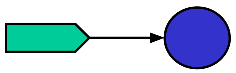
A Simple PyNN Network

```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
pop_1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
                     {'spike_times': [0]}, label="input")
input_proj = p.Projection(input, pop_1, p.OneToOneConnector(
    weights=5.0, delays=1))
```



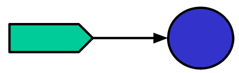
A Simple PyNN Network

```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
pop_1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
                     {'spike_times': [0]}, label="input")
input_proj = p.Projection(input, pop_1, p.OneToOneConnector(
    weights=5.0, delays=1))
pop_1.record()
pop_1.record_v()
```



A Simple PyNN Network

```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
pop_1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
                     {'spike_times': [0]}, label="input")
input_proj = p.Projection(input, pop_1, p.OneToOneConnector(
    weights=5.0, delays=1))
pop_1.record()
pop_1.record_v()
p.run(10)
```



Edit ~/.spynnaker.cfg

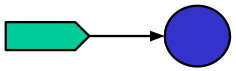
```
[Machine]
#-------
# Information about the target SpiNNaker board or machine:
# machineName:     The name or IP address of the target board
# version:         Version of the Spinnaker Hardware Board (1-5)
# machineTimeStep: Internal time step in simulations in usecs.
# timeScaleFactor: Change this to slow down the simulation time
#                       relative to real time.
#-------
machineName     = None
version         = None
#machineTimeStep = 1000
#timeScaleFactor = 1
```


Edit ~/.spynnaker.cfg

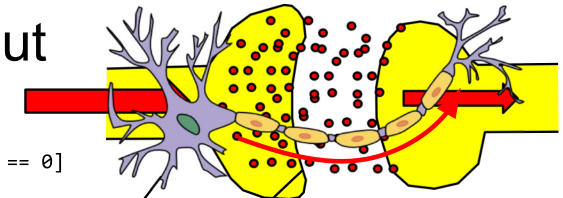
```
[Machine]
#-------
# Information about the target SpiNNaker board or machine:
# machineName:     The name or IP address of the target board
# version:         Version of the Spinnaker Hardware Board (1-5)
# machineTimeStep: Internal time step in simulations in usecs.
# timeScaleFactor: Change this to slow down the simulation time
#                       relative to real time.
#-------
machineName     = 192.168.240.253
version         = 3
#machineTimeStep = 1000
#timeScaleFactor = 1
```


A Simple PyNN Network

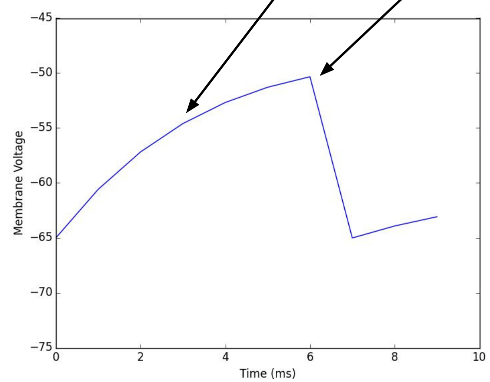
```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
pop_1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
                     {'spike_times': [0]}, label="input")
input_proj = p.Projection(input, pop_1, p.OneToOneConnector(
    weights=5.0, delays=1))
pop_1.record()
pop_1.record_v()
p.run(10)
spikes = pop_1.getSpikes()
v = pop_1.get_v()
```



Plotting Output

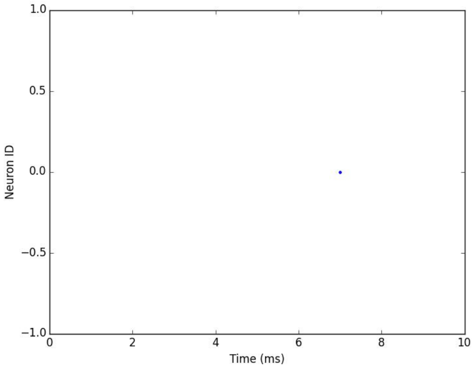


```
import pylab
time = [i[1] for i in v if i[0] == 0]
membrane_voltage = [i[2] for i in v if i[0] == 0]
pylab.plot(time, membrane_voltage)
pylab.xlabel("Time (ms)")
pylab.ylabel("Membrane Voltage")
pylab.axis([0, 10, -75, -45])
pylab.show()
```

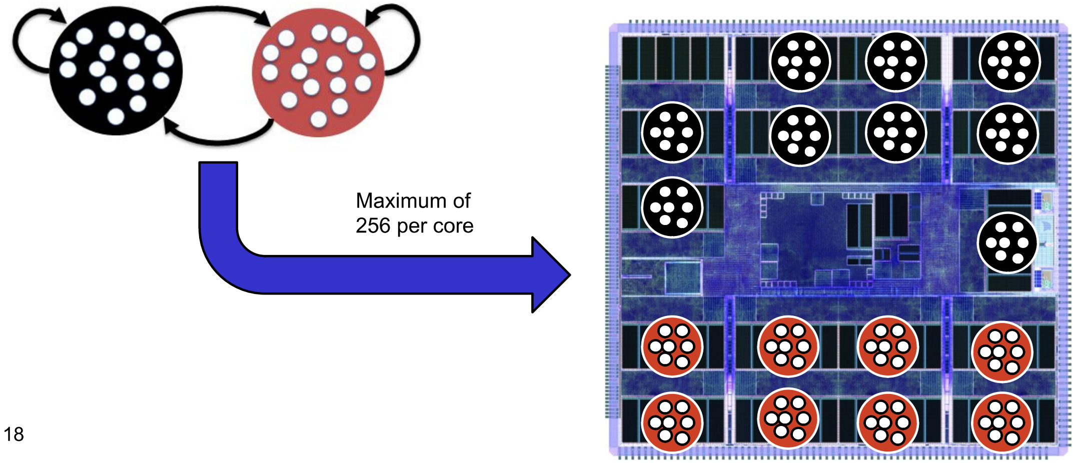


Plotting Output

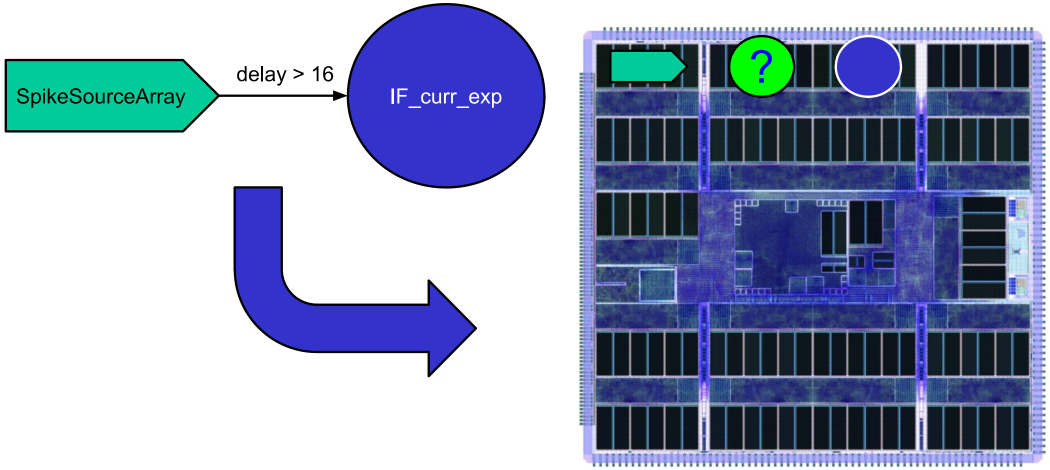
```
import pylab
spike_time = [i[1] for i in spikes]
spike_id = [i[0] for i in spikes]
pylab.plot(spike_time, spike_id, ".")
pylab.xlabel("Time (ms)")
pylab.ylabel("Neuron ID")
pylab.axis([0, 10, -1, 1])
pylab.show()
```



Limitations of PyNN on SpiNNaker:
Neurons Per Core



Limitations of PyNN on SpiNNaker:
Number of cores available



SpiNNaker-Specific PyNN

```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
p.set_number_of_neurons_per_core(p.IF_curr_exp, 100)
pop_1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
```

Configuration with spynnaker.cfg

```
[Machine]
machineName      = None
version          = None
timeScaleFactor  = 1
```

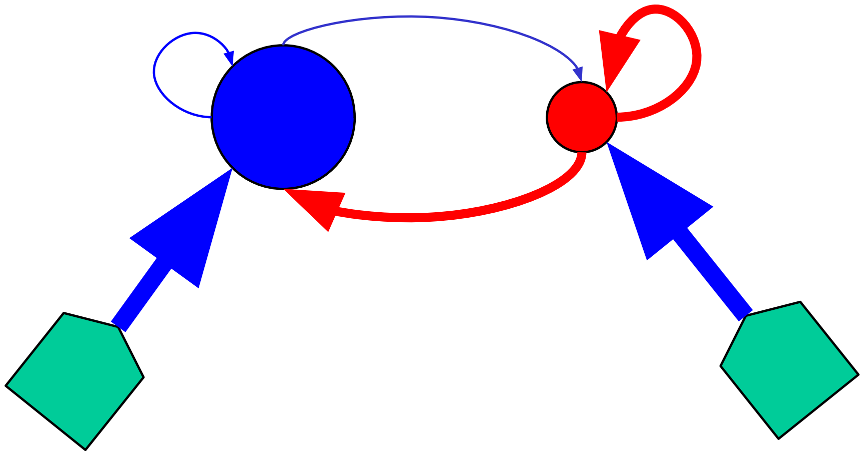
Real Time (ms)	0	10	20	30
Sim Time (ms)	0	10	20	30

Configuration with spynnaker.cfg

```
[Machine]
machineName      = None
version          = None
timeScaleFactor  = 10
```

Real Time (ms)	0	10	20	30
Sim Time (ms)	0	1	2	3

Balanced Random Network

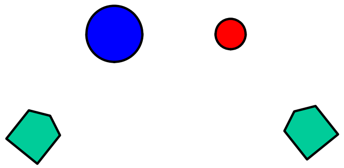


Balanced Random Network

```
import pyNN.spiNNaker as p
import pylab
from pyNN.random import RandomDistribution

p.setup(timestep=0.1)
n_neurons = 1000
n_exc = int(round(n_neurons * 0.8))
n_inh = int(round(n_neurons * 0.2))
```

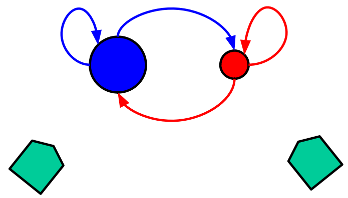
Balanced Random Network

```
pop_exc = p.Population(n_exc, p.IF_curr_exp, {},
                       label="Excitatory")
pop_inh = p.Population(n_inh, p.IF_curr_exp, {},
                       label="Inhibitory")
stim_exc = p.Population(n_exc, p.SpikeSourcePoisson,
                        {"rate": 10.0}, label="Stim_Exc")
stim_inh = p.Population(n_inh, p.SpikeSourcePoisson,
                        {"rate": 10.0}, label="Stim_Inh")
```



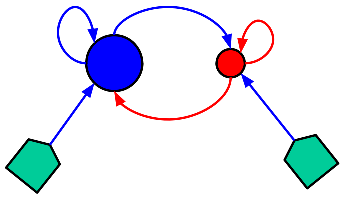
Balanced Random Network

```
conn_exc = p.FixedProbabilityConnector(0.1, weights=0.2,
                                       delays=2.0)
conn_inh = p.FixedProbabilityConnector(0.1, weights=-1.0,
                                       delays=2.0)
p.Projection(pop_exc, pop_exc, conn_exc, target="excitatory")
p.Projection(pop_exc, pop_inh, conn_exc, target="excitatory")
p.Projection(pop_inh, pop_inh, conn_inh, target="inhibitory")
p.Projection(pop_inh, pop_exc, conn_inh, target="inhibitory")
```



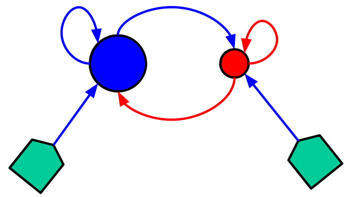
Balanced Random Network

```
delays_stim = RandomDistribution("uniform", [1.0, 1.6])
conn_stim = p.OneToOneConnector(weights=2.0,
                                delays=delays_stim)
p.Projection(stim_exc, pop_exc, conn_stim, target="excitatory")
p.Projection(stim_inh, pop_inh, conn_stim, target="excitatory")
```



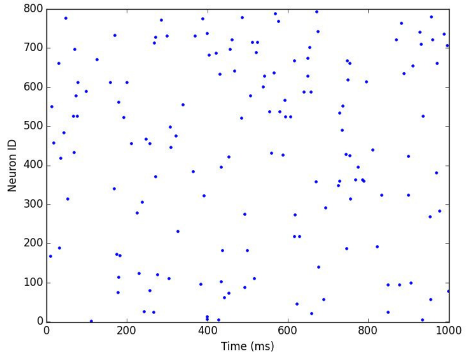
Balanced Random Network

```
pop_exc.initialize("v", RandomDistribution("uniform",
                                           [-65.0, -55.0]))
pop_inh.initialize("v", RandomDistribution("uniform",
                                           [-65.0, -55.0]))
pop_exc.record()
p.run(1000)
```

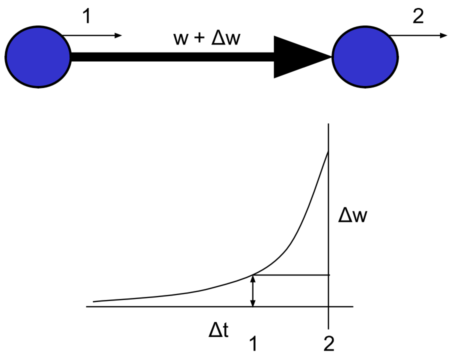


Balanced Random Network

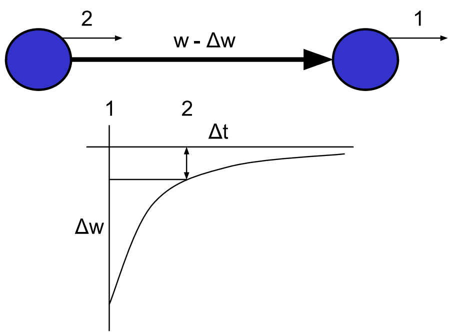
```
spikes = pop_exc.getSpikes()
pylab.plot([i[1] for i in spikes], [i[0] for i in spikes], ".")
pylab.xlabel("Time (ms)")
pylab.ylabel("Neuron ID")
pylab.axis([0, 1000, -1, n_exc + 1])
pylab.show()
```



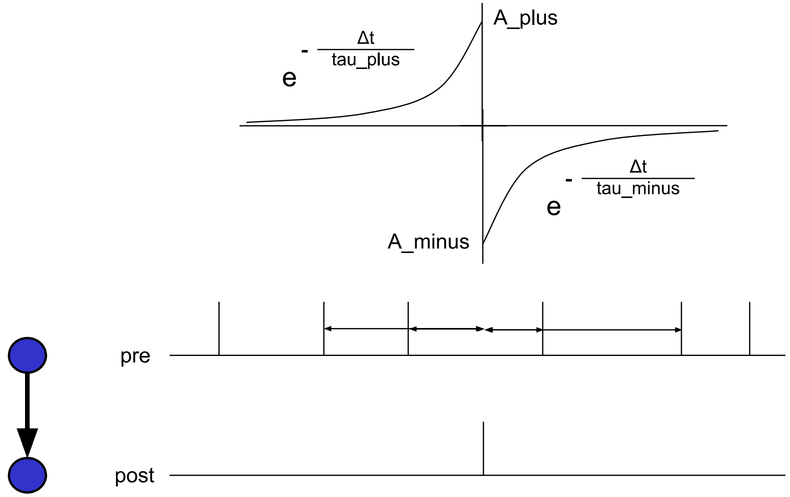
Spike Time Dependent Plasticity: Potentiation



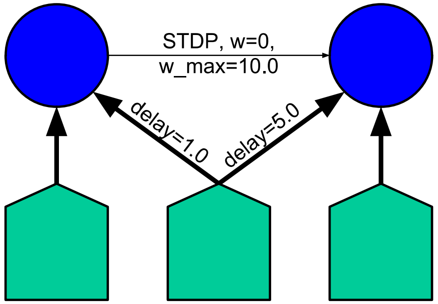
Spike Time Dependent Plasticity: Depression



STDP Rules

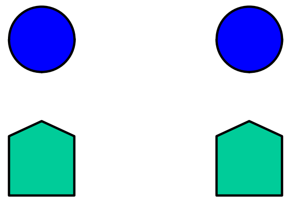


STDP in PyNN



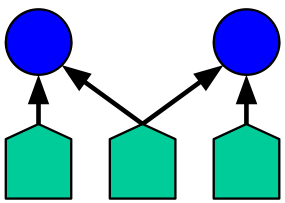
STDP in PyNN

```
import pyNN.spiNNaker as p
import pylab

p.setup(timestep=1.0)
n_neurons = 100

pre_pop    = p.Population(n_neurons, p.IF_curr_exp, {}, label="Pre")
post_pop   = p.Population(n_neurons, p.IF_curr_exp, {}, label="Post")
pre_noise  = p.Population(
    n_neurons, p.SpikeSourcePoisson, {"rate": 10.0}, label="Noise_Pre")
post_noise = p.Population(
    n_neurons, p.SpikeSourcePoisson, {"rate": 10.0}, label="Noise_Post")

pre_pop.record()
post_pop.record()
```



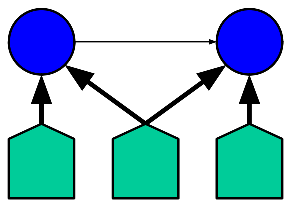
STDP in PyNN

```
training = p.Population(
    n_neurons, p.SpikeSourcePoisson,
    {"rate": 10.0, "start": 2000.0, "duration": 1000.0},
    label="Training")

p.Projection(pre_noise,  pre_pop,  p.OneToOneConnector(weights=2.0))
p.Projection(post_noise, post_pop, p.OneToOneConnector(weights=2.0))

p.Projection(training, pre_pop,  p.OneToOneConnector(weights=5.0, delays=1.0))
p.Projection(training, post_pop, p.OneToOneConnector(weights=5.0, delays=10.0))
```



STDP in PyNN

```
timing_rule = p.SpikePairRule(tau_plus=20.0, tau_minus=20.0)
weight_rule = p.AdditiveWeightDependence(
    w_max=5.0, w_min=0.0, A_plus=0.5, A_minus=0.5)

stdp_model = p.STDPMechanism(
    timing_dependence=timing_rule, weight_dependence=weight_rule)

stdp_projection = p.Projection(
    pre_pop, post_pop, p.OneToOneConnector(weights=0.0, delays=5.0),
    synapse_dynamics=p.SynapseDynamics(slow=stdp_model))
```



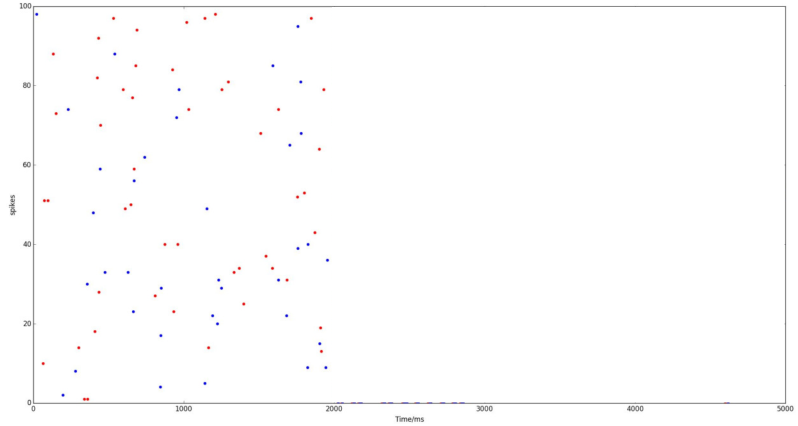
STDP in PyNN

```
p.run(5000)

pre_spikes = pre_pop.getSpikes()
post_spikes = post_pop.getSpikes()

print stdp_projection.getWeights()

p.end()
```

```
[ 4.83886719  5.          4.27246094  5.          3.80957031  4.87060547
  5.          5.          5.          5.          5.          5.
  4.25048828  5.          5.          4.57763672  5.          5.          5.
  5.          5.          5.          5.          5.          5.          5.
  5.          3.76953125  5.          5.          5.          5.          5.
  5.          5.          5.          5.          5.          5.          5.
  4.81591797  5.          5.          5.          5.          5.
  2.73339844  5.          5.          5.          5.          5.          5.
  5.          5.          5.          5.          5.          5.          5.
  5.          5.          5.          5.          4.98046875  5.          5.
  5.          5.          5.          5.          4.23388672  5.          5.
  5.          5.          5.          5.          4.69433594  5.          5.
  5.          5.          5.          5.          5.          5.          5.
  3.85400391  5.          5.          5.          4.07617188  5.          5.
  5.          5.          5.          5.          5.        ]
```


STDP in PyNN

```
pylab.figure()
pylab.xlim((0, 5000))
pylab.plot([i[1] for i in pre_spikes],  [i[0] for i in pre_spikes],  "r.")
pylab.plot([i[1] for i in post_spikes], [i[0] for i in post_spikes], "b.")
pylab.xlabel('Time/ms')
pylab.ylabel('spikes')
pylab.show()
```



6th SpiNNaker Workshop

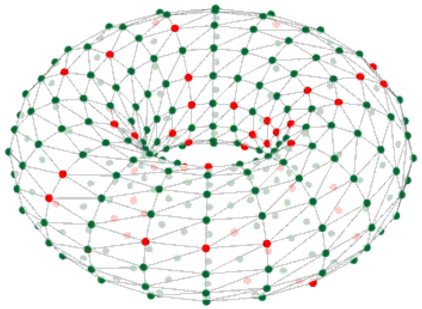
Day 2

September 6th 2016

	Session	Presenter
09:00	SpiNNaker system software (SARK)	ST
10:00	Coffee (earlier than usual)	
10:30	SpiNNaker API + event driven simulation	LAP
11:30	Writing Applications on SpiNNaker - Overview	SD
12:00	Lunch	
13:00	Introduction to Graph Front End (GFE)	ABS (AGR)
14:00	ybug and gdb walk-through	ST
15:00	Coffee	
15:30	Lab time	
16:30	Close	

Manchester, UK

SpiNNaker System Software



Steve Temple
SpiNNaker Workshop – Manchester – Sep 2016

European Research Council
Established by the European Commission

Human Brain Project

EPSRC

SpiNNaker

Overview

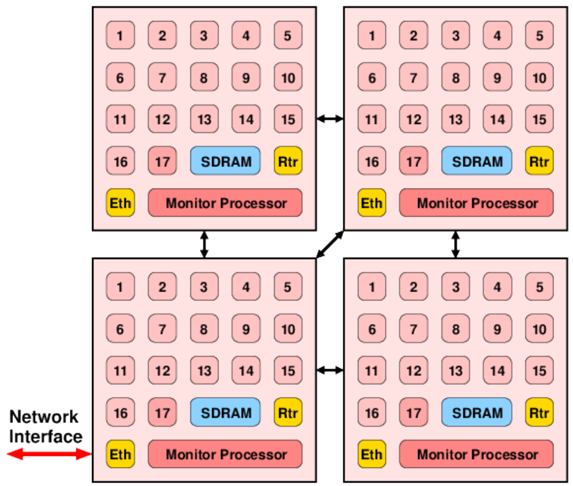
- SpiNNaker applications and their environment
- SC&MP, *ybug* and application loading
- SARK (SpiNNaker Application Runtime Kernel)
  - Application start-up
  - SARK function library
  - Examples
  - Documentation

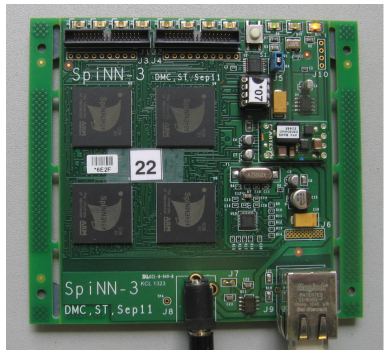
Please interrupt if you have a question!

Building Applications

- Languages – mostly C with bits of assembler
- Toolchain choice
  - ARM tools – RVDS 4 and DS-5 (free for academics)
  - GCC – GNU ARM Embedded Toochain (free)
- Library support
  - Toolchain libraries – C library functions, maths, etc
  - SARK – low-level SpiNNaker support library
  - Spin1 API – event-based application library
- Linking – support libs + application code
  - Creates application to be loaded
  - Application file format is APLX

Execution Environment (1)

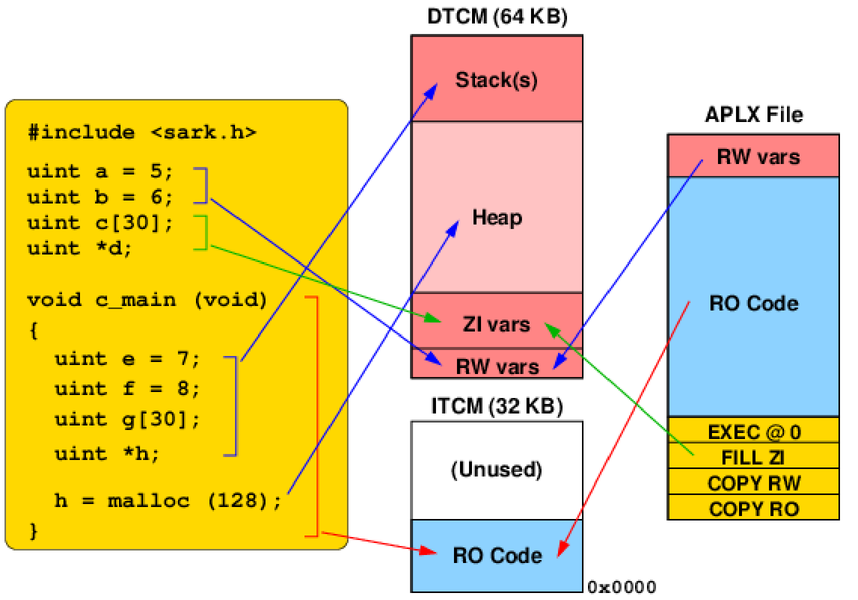




Execution Environment (2)

- One application per core
- Executable code (instructions) in ITCM (32 KB)
- Data (variables, stacks, heap) in DTCM (64 KB)
- Bulk and/or shared data in SDRAM (128 MB)
- Code/data access from ITCM/DTCM is fast (5 ns)
- Data access to SDRAM is slow (> 100 ns) and subject to contention
- DMA controller in each core can move bulk data between I/DTCM and SDRAM faster (~ 15 ns/word) without requiring CPU

Mapping Program to Memory



SC&MP

- "SpiNNaker Control & Monitor Program"
- Loaded onto all Monitor Processors during bootstrap
- Communicates with host computer using SCP (SpiNNaker Command Protocol) over SDP
- Supervises operation of a single chip
- Allows program loading to Application Cores
- Acts as router for SDP packets between any pair of cores or with external Internet endpoints
- Flashes the LED!

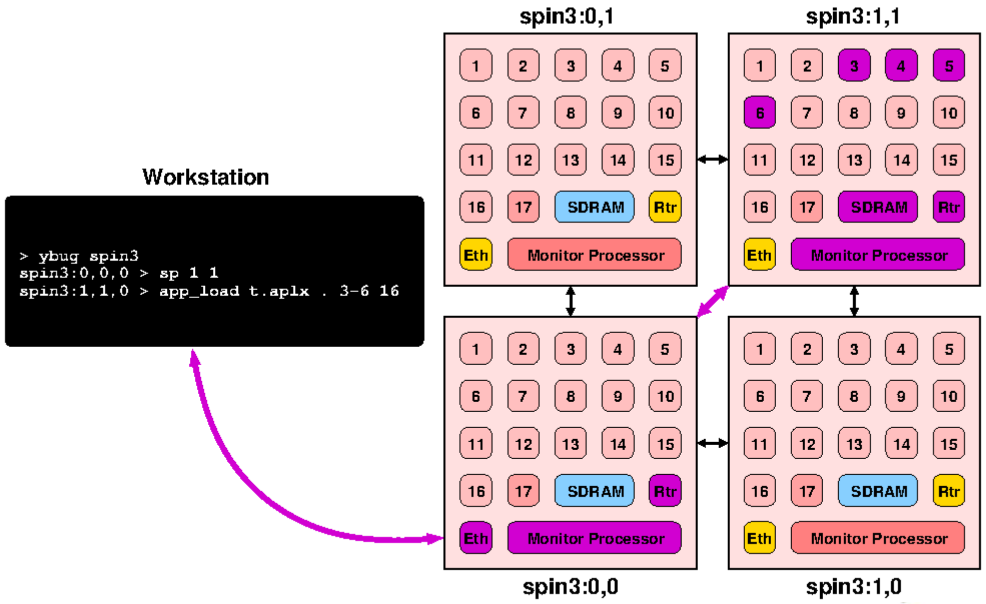
SC&MP, SCP and *ybug*

- SC&MP provides command interface via SCP
  - `Ver` – give S/W version, etc
  - `Read (addr, length)` – read SpiNNaker memory
  - `Write (addr, length, data)` – write SpiNNaker memory
  - `Reset (core_mask)` – reset Application Cores
- Host (workstation) embeds SCP/SDP in UDP/IP to talk to SpiNNaker Monitor Processor on the Root Chip
- *ybug* is a simple command-line tool which runs on a workstation and provides an interface to SC&MP for application loading and debug

Application Loading (1)

- *ybug* sends the application APLX to the relevant SpiNNaker chips.
- The APLX image is copied to a known place in shared memory
- *ybug* requests that the relevant Application cores are reset.
- The reset code is an *APLX loader* which loads the image according to instructions in the APLX header
- This usually results in the application being copied into ITCM and entered at address zero (the ARM reset vector)

Application Loading (2)



SARK

- SpiNNaker Application Runtime Kernel
- Three main functions
  1) Application start-up
  2) Library of useful functions
  3) Communication via SDP with Monitor Processor (and hence rest of system)
- SARK is automatically linked with applications when they are built
- Occupies around 3 KB in the image

Application Start-Up

- Start-up code at start of ITCM is SARK
  - Configures stacks for 4 ARM execution modes
  - Initialises Heap and SDP message buffers in DTCM
  - Initialises shared-memory data structure (VCPU)
  - Calls a function to do pre-application set-up
  - Calls the function **`c_main`**, the application entry point
  - Calls a function to do post-application clean-up
  - Goes to sleep!
- Some applications will never terminate
- SARK provides SDP communications with the application

SARK Library (1)

- CPU control
  - Interrupt disabling and enabling
  - Entering low power (sleep) mode
- Memory manipulation
  - Memory copy and fill (small footprint)
  - SDP message copying
- Pseudo-Random number generation (32-bit)
- SDP messaging
  - Message allocation in DTCM and shared memory
  - SDP message transmission

SARK Library (2)

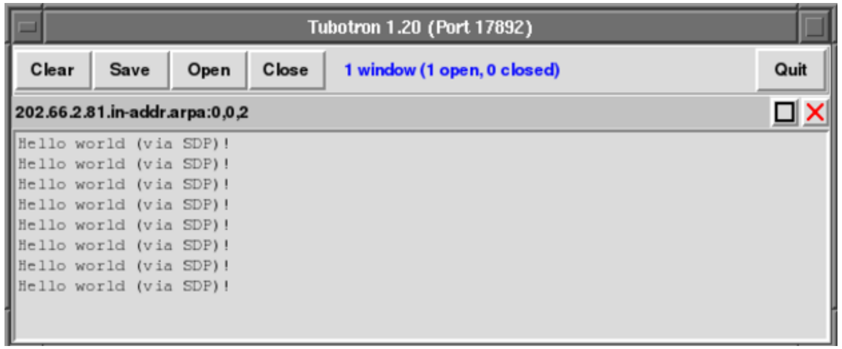
- Text output via “printf”
  - Text sent to a host system using SDP packets
  - Text buffered in SDRAM
- Hardware locks and semaphores
- Memory management
  - `malloc/free` for DTCM heap
  - `malloc/free` for shared memories (eg SDRAM) with locking
  - `malloc/free` for router MC routing table
- Environment queries
  - What is my core ID, chipID, etc

SARK Library (3)

- Hardware interfaces
  - LED control
  - Router control – setting MC and P2P table entries
  - VIC control – allocating interrupt handlers to specific hardware interrupts
- Timer management
  - Routines to schedule/cancel events at some time in the future
- Event management
  - Routines to associate events with interrupts
  - Management of priority event queues

SARK – Example 1

```
#include <sark.h>

void c_main (void)
{
  io_printf (IO_STD, "Hello world (via SDP)!\n");
  io_printf (IO_BUF, "Hello world (via SDRAM)!\n");
}
```



SARK - Example 2

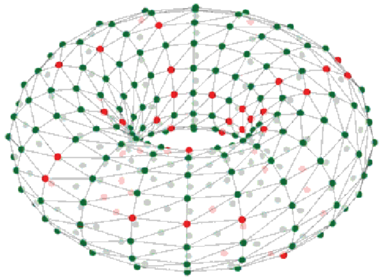
```
#include <sark.h>

INT_HANDLER timer_int_han (void)
{
  tc[T1_INT_CLR] = (uint) tc;      // Clear interrupt in timer
  sark_led_set (LED_FLIP (1));     // Flip a LED
  vic[VIC_VADDR] = (uint) vic;     // Tell VIC we're done
}

void timer_setup (uint period)
{
  tc[T1_CONTROL] = 0xe2;               // Set up count-down mode
  tc[T1_LOAD] = sark.cpu_clk * period; // Load time (us)
  sark_vic_set (SLOT_0, TIMER1_INT, 1, timer_int_han);
}

void c_main ()
{
  io_printf (IO_STD, "Timer interrupt example\n");
  timer_setup (500000);      // (0.5 secs)
  cpu_sleep ();              // Send core to sleep
}
```

Documentation & Help

- SARK – notes in SpiNNaker Tools - `docs/sark.pdf`
- *ybug* – user guide in SpiNNaker Tools – `docs/ybug.pdf`
- `"spinnaker.h"` - describes the SpiNNaker hardware – memory maps, peripheral registers...
- `"sark.h"` describes all SARK data structures and functions. Commented in Doxygen style.
- All source code is provided...
- If desperate, talk to us!

MANCHESTER 1824

SpiNNaker API and event-driven simulation



Luis Plana

SpiNNaker Workshop, September 2016

European Research Council

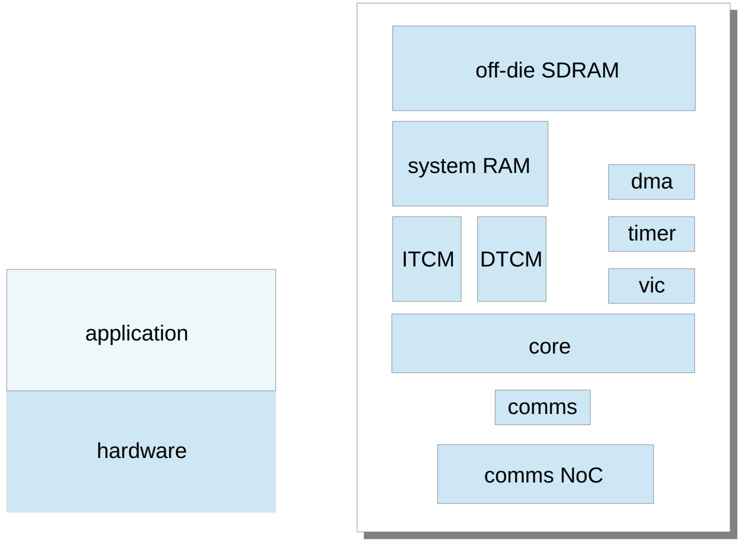
Human Brain Project

EPSRC

SpiNNaker

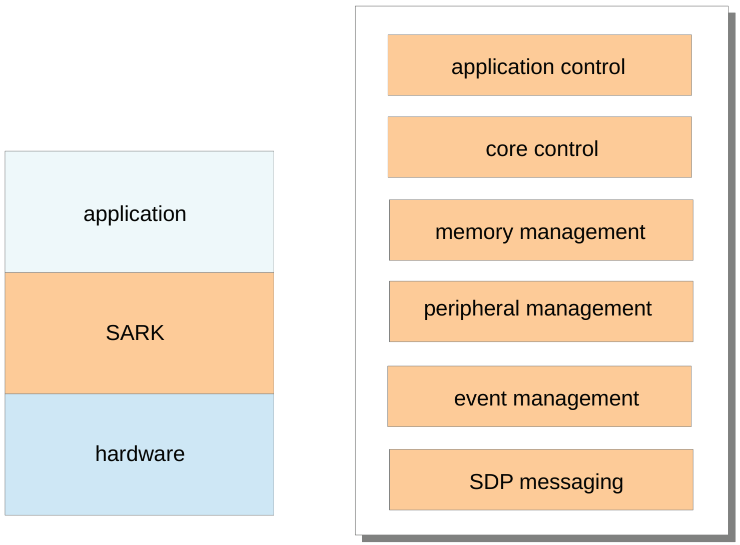
MANCHESTER 1824

hardware resources



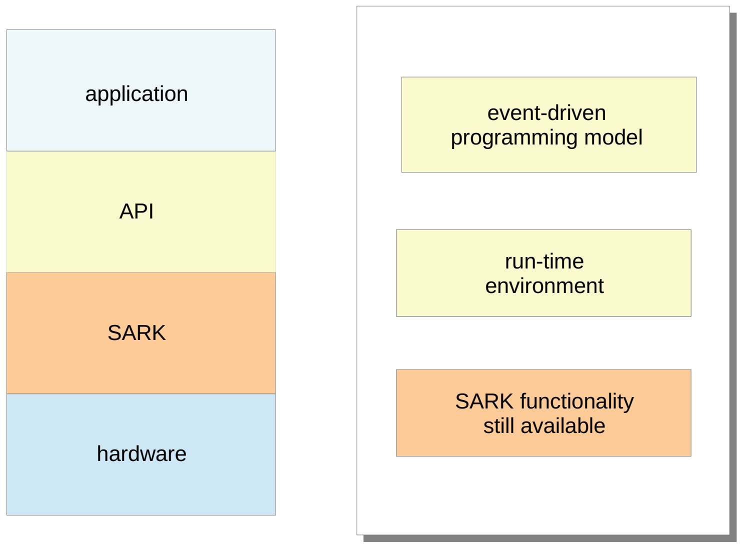
MANCHESTER 1824

SARK: low-level software



MANCHESTER 1824

API: run-time environment



MANCHESTER 1824

event-driven model

applications do not control execution flow

applications indicate functions to be executed when events of interest occur

API controls execution and schedules application functions when appropriate

application functions are known as callbacks

MANCHESTER 1824

events and callbacks

event	trigger
timer tick	periodic event has occurred
multicast packet received	multicast packet has arrived
DMA transfer done	scheduled DMA transfer completed successfully
SDP packet received	SDP packet has arrived
user event	application-triggered event has occurred

event	first argument	second argument
timer tick	simulation time (ticks)	null
MCP w/o payload received	key	0
MCP with payload received	key	payload
DMA transfer done	transfer ID	tag
SDP packet received	*mailbox	destination port
user event	arg0	arg1

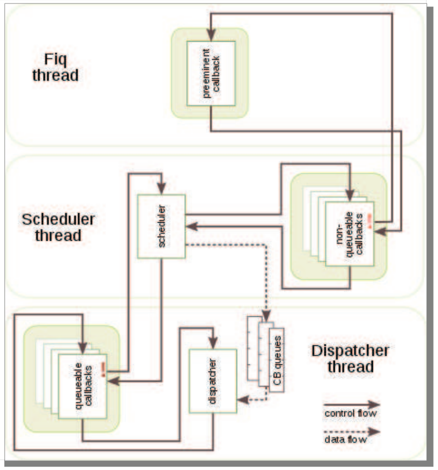
MANCHESTER 1824

priorities

priority level = -1
only one callback
cannot be pre-empted

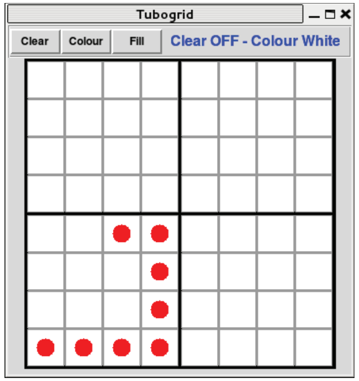
priority level = 0
can only be pre-empted
by priority -1 callback

priority level > 0
can be pre-empted
by priority <= 0 callbacks
scheduled in priority order



MANCHESTER 1824

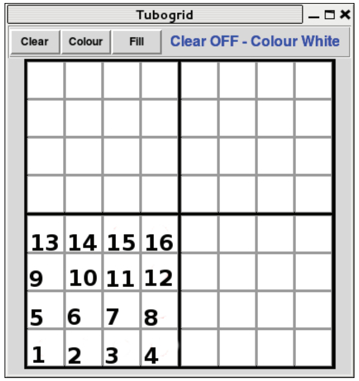
first program



first

```
// circle sequence
uint circle_pos[] =
{
  1, 2, 3, 4, 8, 12, 16, 15,
  14, 13, 9, 5, 6, 7, 11, 10
};

// iterate over 16 positions
for (uint i = 0; i < 16; i++)
{
  // update display,
  print_circle (circle_pos[i]);

  // and delay next circle
  for (uint j = 0; j < BIG_NUM; j++)
  {
    continue;
  }
}
```

distributed program



each core

```
// circle sequence
uint circle_pos[] =
{
  1, 2, 3, 4, 8, 12, 16, 15,
  14, 13, 9, 5, 6, 7, 11, 10
};

// this core's id
id = spin1_get_core_id();

// delay my circle,
for (uint j = 0; j < (id * BIG_NUM); j++)
{
  continue;
}

// and update display
print_circle (circle_pos[id]);
```

event-driven program

c_main

```
// 0.125s tick period (in microseconds)
#define TIMER_TICK_PERIOD  125000

void c_main()
{
  // initialize variables and state
  // ----------------------------------------
  id = spin1_get_core_id();
  my_state = OFF;
  old_state = my_state;

  // prepare for execution
  // ----------------------------------------
  // set timer tick value
  spin1_set_timer_tick (TIMER_TICK_PERIOD);

  // register callbacks
  spin1_callback_on (
    MC_PACKET_RECEIVED, packet, -1);

  spin1_callback_on (
    TIMER_TICK, timer, 0);

  // go
  // ----------------------------------------
  spin1_start(SYNC_WAIT);
}
```

packet callback

```
void packet (uint pkt_key, uint pkt_payload)
{
  // update my state
  my_state = ON;
}
```

timer callback

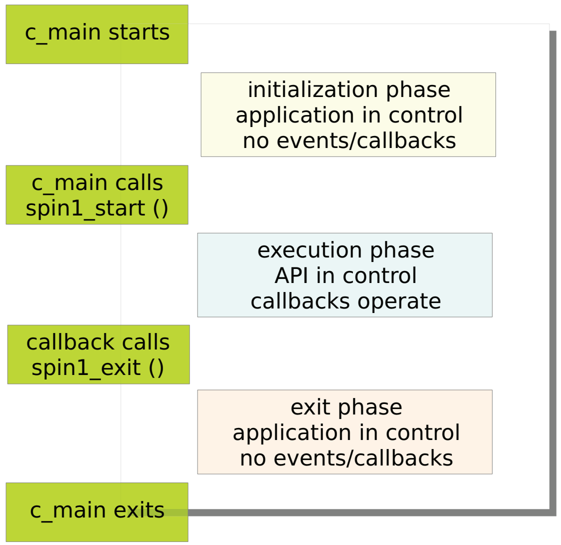
```
void timer (uint ticks, uint b)
{
  // check if state changed
  if (my_state != old_state)
  {
    // update display,
    print_circle (circle_pos[id]);

    // send a packet to next core in the chain,
    spin1_send_mc_packet(my_key, 0, NO_PAYLOAD);

    // and remember state
    old_state = my_state;
  }
}
```

additional support

function	use
start/stop execution	start and stop simulation
set timer period	real-time or periodic callback
send multicast packet	inter-core communications
send SDP packet	host or I/O peripheral communications
start DMA transfer	software-managed cache
trigger user event	start a callback with priority <= 0
schedule callback	start a callback with priority > 0
enable/disable interrupts	critical section access (inter-thread control)
provide chip address and core ID	find out who you are
configure multicast routing table	setup routing entries

see API documentation for complete list

program structure



MANCHESTER 1824

synchronization barrier

c_main starts

application initializes variables and may set up routing table

c_main calls spin1_start ()

core goes to sleep and waits for *sync0* signal

host can check that cores are waiting!

host sends *sync0* signal

core receives signal and starts execution

MANCHESTER 1824

to think about: pitfalls

asynchronous operation and communications

multicast packets can be dropped due to congestion

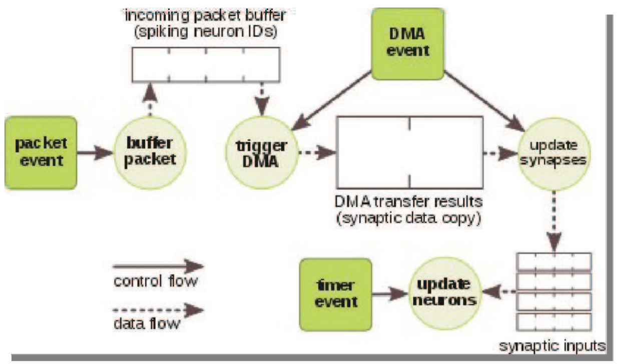
UDP-based I/O *not guaranteed!*

no floating-point support use fixed-point arithmetic

no globally-shared resources use message passing

MANCHESTER 1824

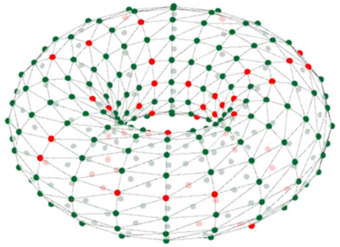
example: spiking neural network



what is a sensible choice of priorities?

MANCHESTER 1824

Writing an Application for SpiNNaker - Introduction



Simon Davidson, Alan Stokes, Andrew Rowley

SpiNNaker Workshop
September 2016

erc European Research Council Established by the European Commission — Human Brain Project — EPSRC — SpiNNaker

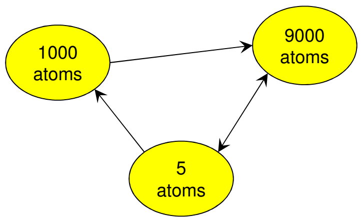
MANCHESTER 1824

Contents

MANCHESTER 1824

View of an application distributed across parallel processors

- Two main activities:
- Computation
- Communication
- Think of the problem as a graph:
- Vertex = computation node
- Edge = flow of information between nodes
- Node can hold a collections of objects of the same type, which we call ***atoms***
- e.g. Many spiking neurons in one population



MANCHESTER 1824

Design Process for New Applications

- **The application designer** creates components (nodes and communication types)
- These components plug into our tool chain
- A **user** can then invoke the Graph Front End (GFE) to create and run their own networks on SpiNNaker
- Input is textual, like a PyNN script, in which the user instantiates the components created by the application designer
- Graph Front End is NOT a Graphical Interface - No GUI!

MANCHESTER 1824

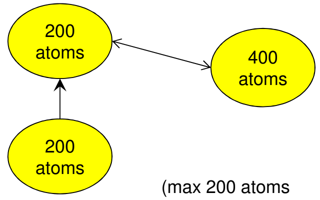
Example script: Conway's Game of Life

```
import spinnaker_graph_front_end as front_end
import sys

# set up the front end and ask for a machine with 48 chips
front_end.setup()

cell_1 = MyCell()
cell_2 = MyCell()
edge = MachineEdge(cell_1, cell_2)
front_end.add_machine_vertex_instance(cell_1)
front_end.add_machine_vertex_instance(cell_2)
front_end.add_machine_edge_instance(edge, "STATE")

# run the simulation for 5 seconds
front_end.run(5000)

# clean yp the machine for the next application.
front_end.stop()
```

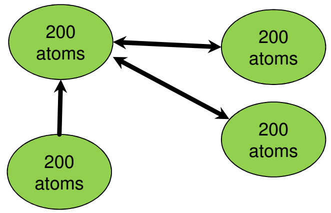

MANCHESTER 1824

Design Considerations I: Finite resources per core

- The user's graph will be mapped to the cores of the SpiNNaker machine
- Each core has finite resource:
- Compute power
- Local memory
- Share of SDRAM capacity & bandwidth
- Communications bandwidth for packets
- Where each vertex represents many atoms we ***partition*** each one into smaller pieces, so that one piece fits on one core:
- **Application graph** maps to **Machine Graph**
- Edges also split to maintain correct connectivity
- Merging of vertices NOT currently supported!



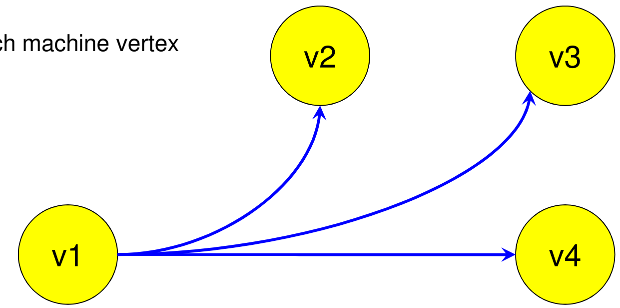
(max 200 atoms per core)



MANCHESTER 1824

Design Considerations II: Dataflow between Vertices

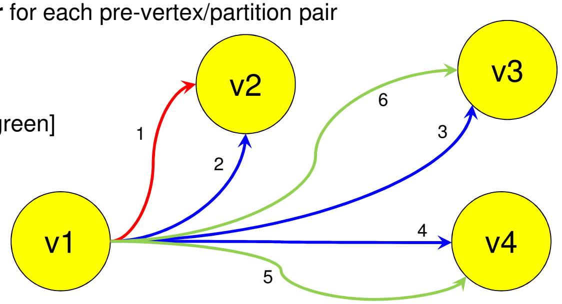
- Consider the pattern of messages flowing from each vertex:
- Case 1: Messages always go to the same set of targets
- Case 2: Messages go to different targets at different times
- Case 1: Homogeneous data flow
- e.g. spikes in neural simulation
- One **identifier** for each machine vertex



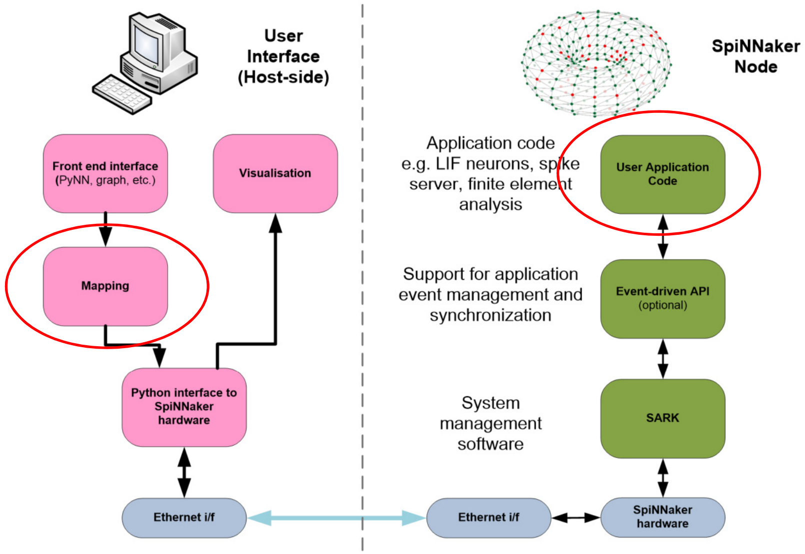
MANCHESTER 1824

Design Considerations II: Dataflow between Vertices

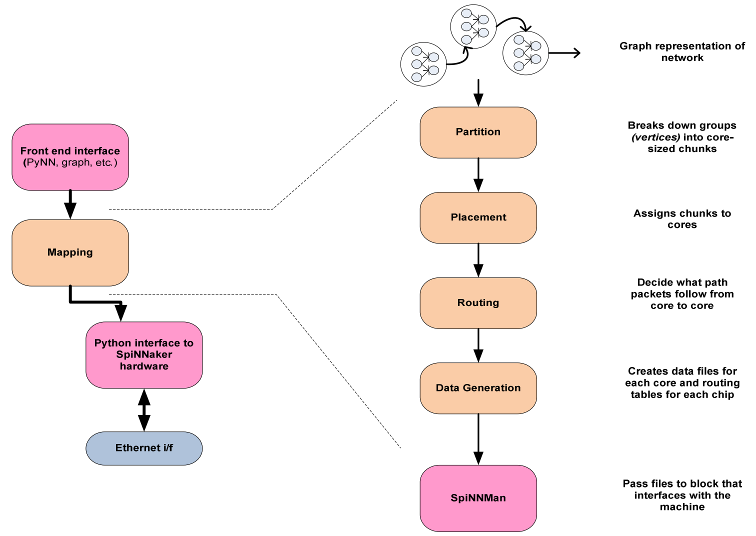
- Case 2: Data send to different targets at different times:
- e.g. multi-layered perceptron, with forward and backward data flow
- Useful when there are different modes of operation
- Group edges so that those in same mode are together
- A grouping is called a **partition**
- Assign a separate **identifier** for each pre-vertex/partition pair
- Six edges [1, 2, .., 6]
- Three partitions [red, blue, green]



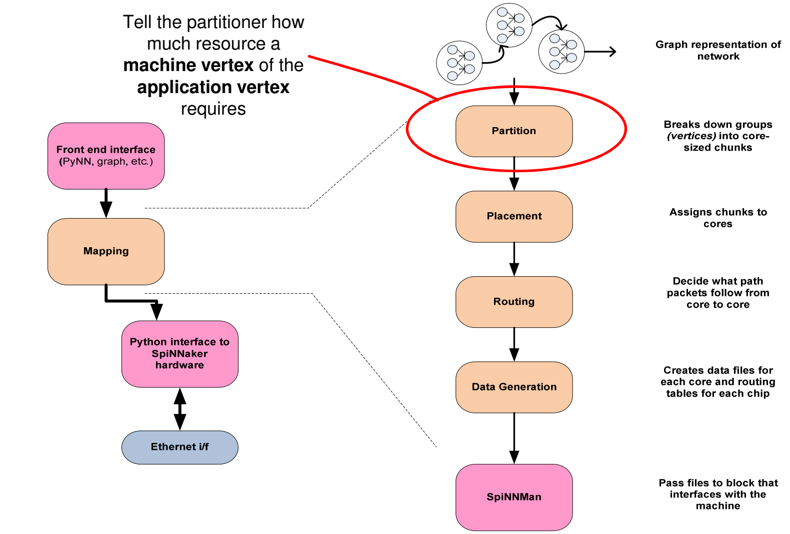
Software Stack



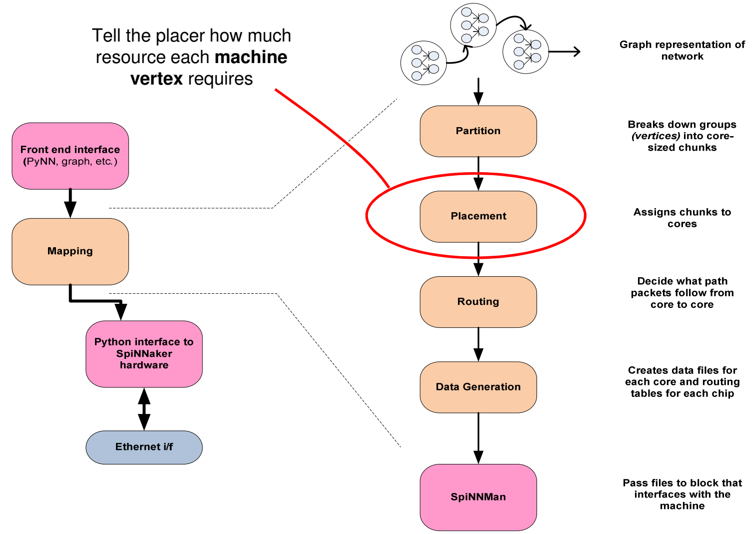
Where do you need to supply new information?



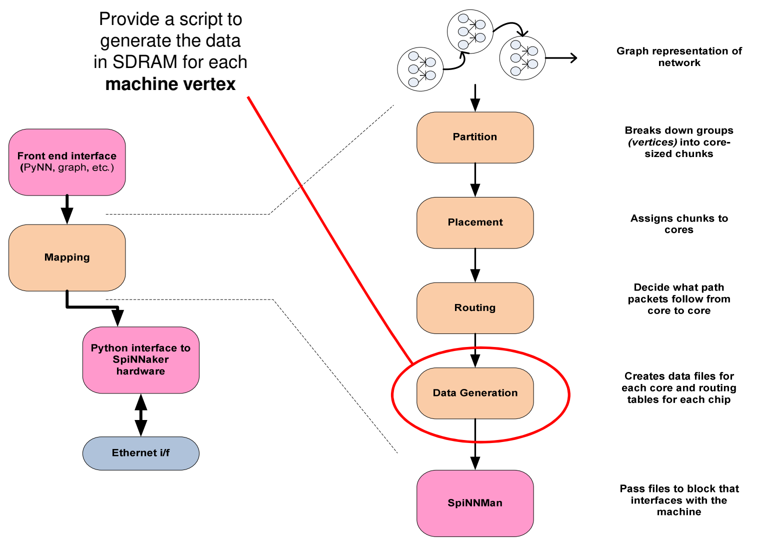
Where do you need to supply new information?



Where do you need to supply new information?



Where do you need to supply new information?



Data Spec. and Data Generation

- Each core running your application needs to generate its local data before it starts simulation
- We provide a simple ***virtual machine*** in which you can execute simple programs to generate this data
- This is the **Data Spec Executor (DSE)**
- The tools run code called the **Data Spec Generator (DSG)** that create a program (the specification or **spec**) for each core that is run by the DSG to generate its data

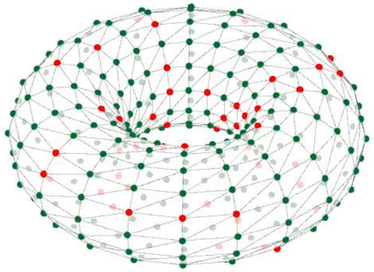
Summary

- It is useful to abstract any parallel application into the form of a graph with:
- Centres of computation (vertices)
- Connected by communication pathways (edges)

- **Application designer** must describe the computational elements and the communication types and plug those into our tools:
- Executables to run on SpiNNaker (typically written in C)
- Data specification, used to create each nodes data
- Describe resource requirements to allow tools to map networks to cores

- **User** can then specify application networks and run them using the Graph Front End.The tools handle :
- Mapping
- Routing table generation
- Data generation
- Loading
- Simulation
- Results gathering
- And other stuff ….

Introduction to the Graph Front End Functionality



Alan Stokes, Andrew Rowley

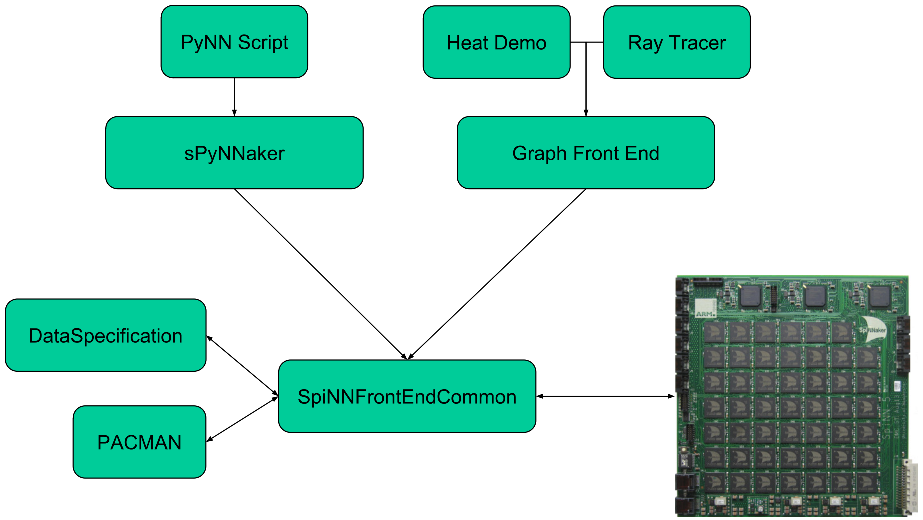
SpiNNaker Workshop
September 2016

European Research Council
Human Brain Project
EPSRC
SpiNNaker

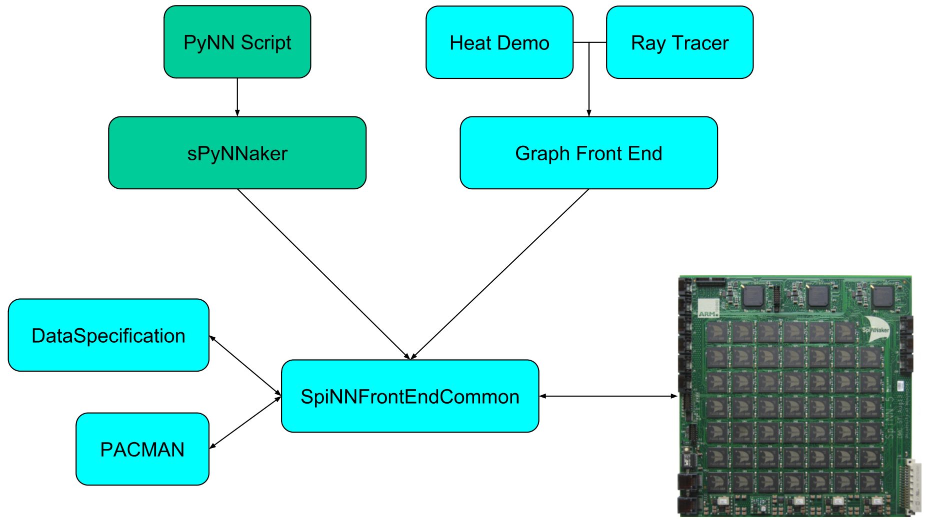
Contents

- The Graph Front End (GFE) interface
- Simple Usage of the GFE
- The Graph in the GFE
- Data Generation
- Binary Specification
- Writing and building simple C code

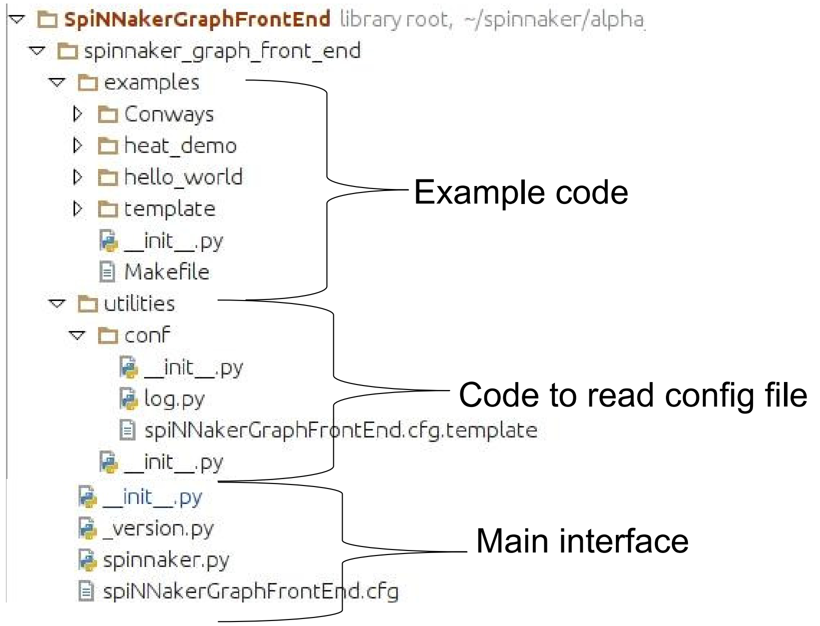
Software modules



Software modules covered here



GFE structure



Skeleton Functionality

```
import spinnaker_graph_front_end as front_end

# set up the front end
front_end.setup()

# run the simulation for 5 seconds
front_end.run(5000)

# clean up the machine for the next application
front_end.stop()
```

ConfigurationException:
There needs to be a graph which contains at least one vertex for the tool chain to map anything.

Main interface functions

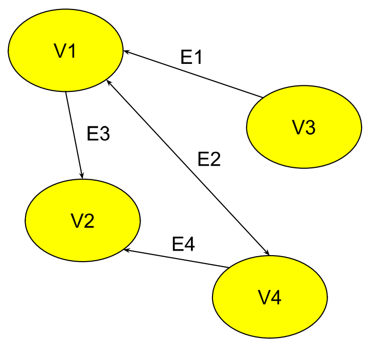
import spinnaker_graph_front_end **as** p

p.setup() Sets up the software stack so that it has read the configuration file and created whatever data objects are required.

p.run(duration) Runs the simulation for a given time period (microseconds).

p.stop() Closes down the application that is running on the SpiNNaker machine and does any housekeeping needed to allow the next application to run correctly.

PACMAN Graph



Has a 1:1 ratio between vertices and SpiNNaker core.

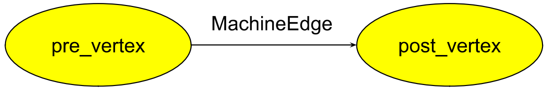
Basic script to add machine vertices into the graph

```
import spinnaker_graph_front_end as front_end

from spinnaker_graph_front_end.examples.Conways.conways_cell import \
  ConwayMachineCell

# set up the front end
front_end.setup()

for count in range(0, 800):
   front_end.add_machine_vertex_instance(
      ConwayMachineCell(label="cell{}".format(count)))

# run the simulation for 5 seconds
front_end.run(5000)

# clean up the machine for the next application
front_end.stop()
```


Creating a new type of machine vertex

```
from pacman.model.graphs.machine.impl.machine_vertex import MachineVertex
from pacman.model.resources.resource_container import ResourceContainer

class ConwayMachineCell(MachineVertex):
  """ Cell which represents a cell within the 2D grid
  """

  def __init__(self, label):

     # construct the resources this cell uses and instantiate superclass
     resources = ResourceContainer()
     MachineVertex.__init__(self, resources, label)
```


Adding edges to the machine graph



1. The main edge type available is MachineEdge.
2. This can be extended to add application specific data into.
3. Most important inputs are:
   i. **pre_vertex**: The source of the edge.
   ii. **post_vertex**: The destination of the edge.
4. Every edge in a graph is associated with a **partition_id**.

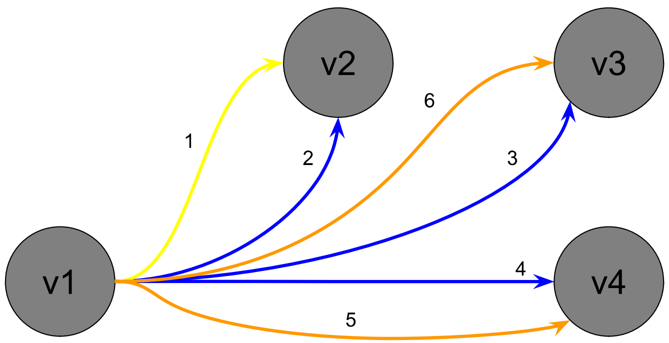
Basic Script adding edges

```
import spinnaker_graph_front_end as front_end
.......

# build and add vertices to the graph
vertices = list()
for count in range(0, 100):
  vertices.append(ConwayMachineCell("cell{}".format(count)))
  front_end.add_machine_vertex_instance(vertices[count])

# build an edge between two vertices
front_end.add_machine_edge_instance(
   MachineEdge(verts[0], verts[1]), "State")

front_end.run(5000)

front_end.stop()
```

Partition id

MANCHESTER 1824

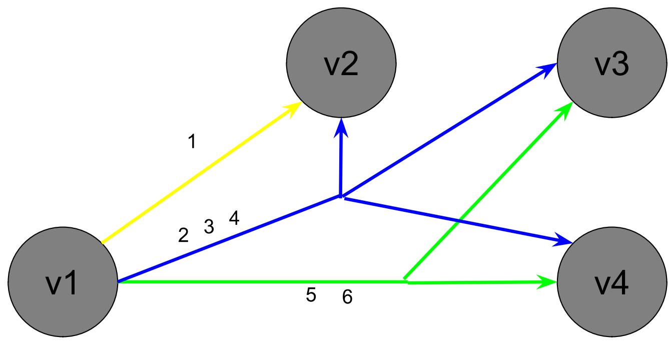
Adding edges to the application graph: Partitions



Edge 1 resides in partition A
Edges 2,3 and 4 reside in partition B
Edges 5 and 6 reside in partition C

MANCHESTER 1824

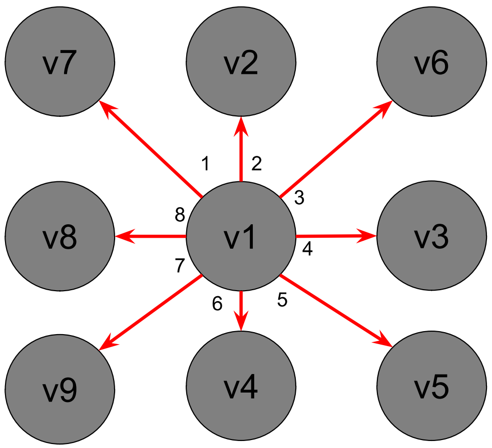
Adding edges to the application graph: Partitions



Edge 1 transmits information about **hotdogs**.
Edges 2,3 and 4 transmits information about **cats**.
Edges 5 and 6 transmits information about **bacon**.

MANCHESTER 1824

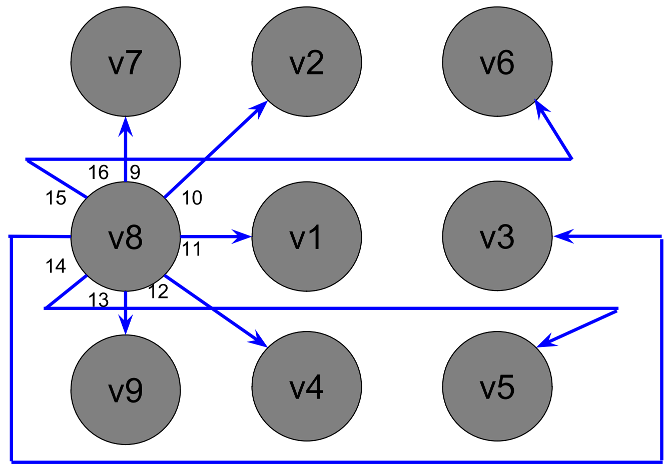
Conways: partitions.



Edges 1,2,3,4,5,6,7,8 transmits v1's **state** data.

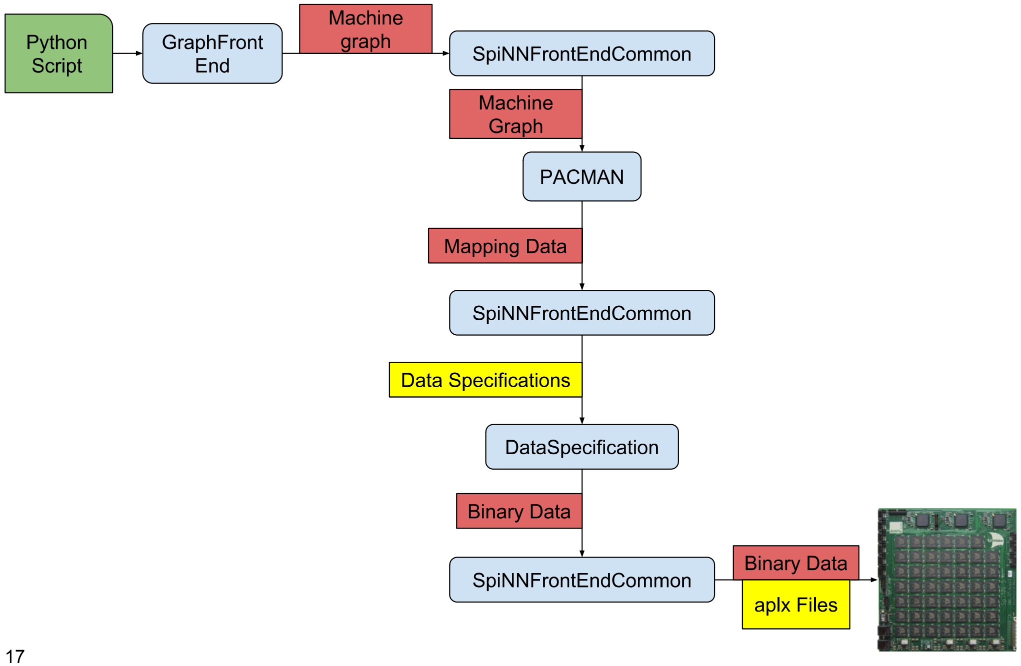
MANCHESTER 1824

Conways: partitions.



Edges 9,10,11,12,13,14,15,16 transmits v8's **state** data.

Workflow of the GFE



Data generation

1. Converts application data within a vertex into data stored on the SpiNNaker machine via SDRAM.
2. Supports separating the SDRAM into data regions
3. Supports writing data as scalars, arrays etc.
4. Common commands:
   i. reserve_memory_region()
   ii. switch_write_focus()
   iii. write_value()
   iv. write_array()
   v. comment()
   vi. close_spec()

Data generation

```
...
def generate_machine_data_specification(
        self,  spec, placement, machine_graph, routing_info, iptags,  reverse_iptags,
        machine_time_step, time_scale_factor):

# Reserve SDRAM space for memory areas:
spec.reserve_memory_region(
    region=0, size=constants.SYSTEM_BYTES_REQUIREMENT, label='system')
spec.reserve_memory_region(
    region=1, size=8, label="inputs")
```


Data generation

```
...
def generate_machine_data_specification(
        self,  spec, placement, machine_graph, routing_info, iptags,  reverse_iptags,
        machine_time_step, time_scale_factor):

# Reserve SDRAM space for memory areas:
spec.reserve_memory_region(
    region=0, size=constants.SYSTEM_BYTES_REQUIREMENT, label='system')
spec.reserve_memory_region(
    region=1, size=8, label="inputs")

# add simulation.c interface data
spec.switch_write_focus(0)
spec.write_array(simulation_utilities.get_simulation_header_array(
    self.get_binary_file_name(), machine_time_step, time_scale_factor))
```

Needed for simulation.c

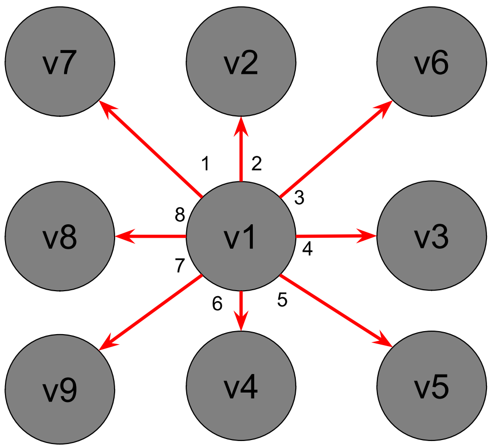
Data generation

```
...
def generate_machine_data_specification(
        self,  spec, placement, machine_graph, routing_info, Iptags,  reverse_iptags,
        machine_time_step, time_scale_factor):

# Reserve SDRAM space for memory areas:
spec.reserve_memory_region(
   region=0, size=constants.SYSTEM_BYTES_REQUIREMENT, label='system')
spec.reserve_memory_region(
   region=1, size=8, label="inputs")

# add simulation.c interface data
spec.switch_write_focus(0)
spec.write_array(simulation_utilities.get_simulation_header_array(
   self.get_binary_file_name(), machine_time_step, time_scale_factor))

# application specific data items
spec.switch_write_focus(region=1)
spec.comment("writing initial state for this conway element \n")
spec.write_value(data=self._state)
```


Data generation

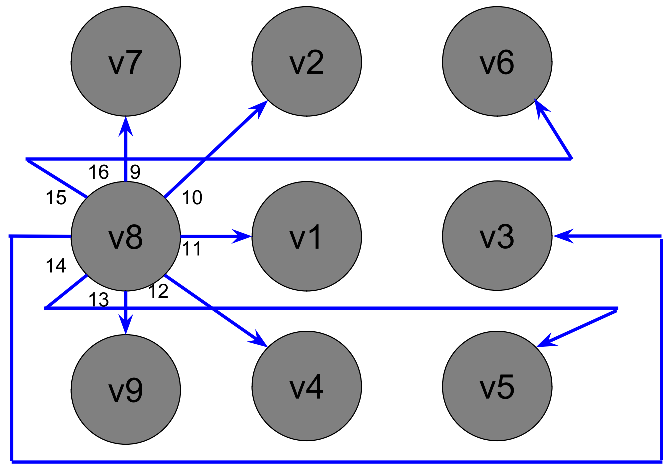
```
...
spec.comment("writing initial state for this conway element \n")
spec.write_value(data=self._state)

# write the routing key needed for my transmission
spec.comment("writing the routing key needed to transmit my state \n")
spec.write_value(data=routing_info.get_first_key_from_pre_vertex(self, "State"))

# close the spec
spec.comment("closing the spec \n")
spec.close_spec()
```


Conways: partitions.



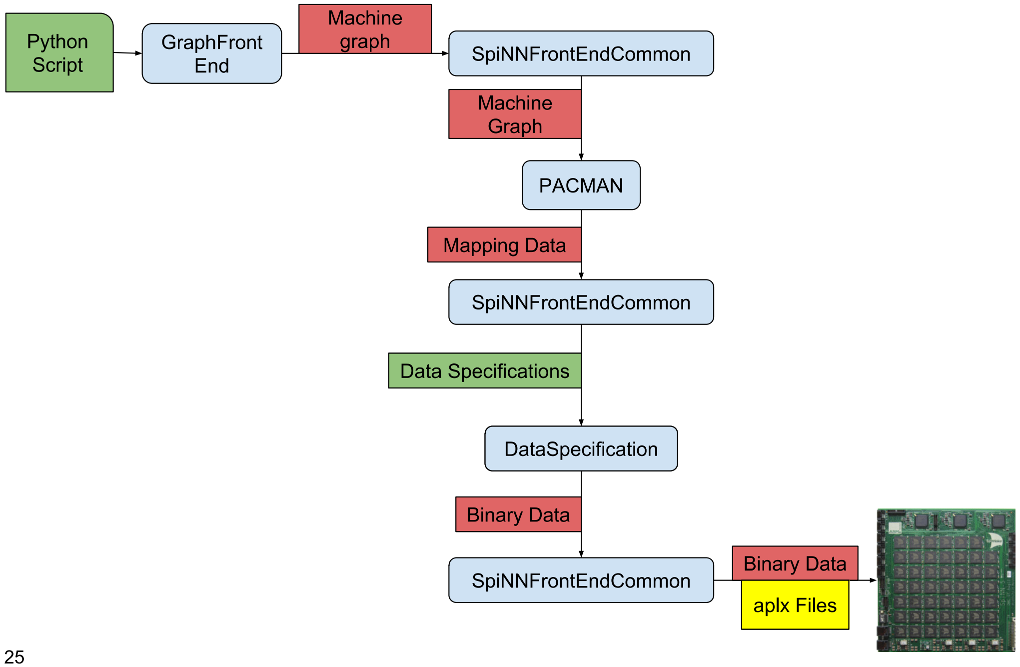
Edges 1,2,3,4,5,6,7,8 transmits with routing key 0.

Conways: partitions.



Edges 9,10,11,12,13,14,15,16 transmit with routing key 1.

Workflow of the GFE



Binary Executables

AbstractHasAssociatedBinary

```
def get_binary_file_name(self)
```

AbstractStartsSynchronized

- After loading binary, CPU state will be in SYNC0

AbstractBinaryUsesSimulationRun(AbstractStartsSynchronized)

- The binary uses the simulation environment provided by the tools

Binary Executables

```
from pacman.model.graphs.machine.impl.machine_vertex import MachineVertex
from pacman.model.resources.resource_container import ResourceContainer
from spinn_front_end_common.abstract_models.abstract_has_associated_binary\
   import AbstractHasAssociatedBinary
from spinn_front_end_common.abstract_models.abstract_binary_uses_simulation_run\
   import AbstractBinaryUsesSimulationRun

class ConwayMachineCell(
        MachineVertex, AbstractHasAssociatedBinary,
        AbstractBinaryUsesSimulationRun):
    """ Cell which represents a cell within the 2D grid
    """

    def __init__(self, label):

        # construct the resources this cell uses and instantiate superclass
        resources = ResourceContainer()
        MachineVertex.__init__(self, resources, label)

    def get_binary_file_name(self):

        # return the binary name of this vertex
        return "conways.aplx"
```


Linking Aplx files to Python

1. Compiled version of c code.
2. This code runs on SpiNNaker.
3. Is linked to your python classes through get_binary_file_name(self) of the vertex
4. How to write event driven C code for SpiNNaker is discussed in **Event Driven Simulations**.
5. We will cover the interfaces provided by the SpiNNFrontEndCommon (SFEC) module for the c code.

Example c code

```
static uint32_t timer_period, simulation_ticks, infinite_run = 0;
static uint32_t time;
static uint32_t SDP_PRIORITY = 1, TIMER_PRIORITY = 2;

void c_main(void) {

    // get address of simulation data
    address_t address = data_specification_get_data_address();
```

```
# gets the address where all the data for this core is stored.
address_t data_specification_get_data_address();
```


Example c code

```
static uint32_t timer_period, simulation_ticks, infinite_run = 0;
static uint32_t time;
static uint32_t SDP_PRIORITY = 1, TIMER_PRIORITY = 2;

void c_main(void) {

    // get address of simulation data
    address_t address = data_specification_get_data_address();

    // get the address of the system region
    address_t system_region = data_specification_get_region(0, address);
```

```
# gets the address of the start of a given data region
address_t data_specification_get_region(uint32_t region, address_t data_address)
```


Example c code

```
static uint32_t timer_period, simulation_ticks, infinite_run = 0;
static uint32_t time;
static uint32_t SDP_PRIORITY = 1, TIMER_PRIORITY = 2;

void c_main(void) {
    ...
    if (!simulation_initialise(
            system_region, APPLICATION_NAME_HASH,
            &timer_period, &simulation_ticks,
            &infinite_run, SDP_PRIORITY,
            NULL, NULL)) {
        log_error("Error in initialisation - exiting!");
        rt_error(RTE_SWERR);
    }
```

```
# sets up the system to interact with the SFEC simulation control functionality.
bool simulation_initialise(
        address_t address, uint32_t expected_application_hash,
        uint32_t* timer_period, uint32_t *simulation_ticks_pointer,
        uint32_t *infinite_run_pointer, int sdp_packet_callback_priority,
        prov_callback_t provenance_function, address_t provenance_data_address)
```


Example c code

```
static uint32_t timer_period, simulation_ticks, infinite_run = 0;
static uint32_t time;
static uint32_t SDP_PRIORITY = 1, TIMER_PRIORITY = 2;

void c_main(void) {
    ...

    // Set timer_callback period
    spin1_set_timer_tick(timer_period);

    // Set timer_callback
    spin1_callback_on(TIMER_TICK, timer_callback, TIMER_PRIORITY);

    // Set time to UINT32 MAX to wrap around to 0 on the first timestep
    time = UINT32_MAX;

    simulation_run();
}
```

```
# main entrance for the event driven nature of the SpiNNaker machine
void simulation_run()
```

Example c code

```
// Callbacks
void timer_callback(uint unused0, uint unused1) {

    // check if the simulation has run to completion
    if ((infinite_run != TRUE) && ((time + 1) >= simulation_ticks)) {
        simulation_exit();
    }
    time++;
}
```

```
# Used once you have finished your simulation
void simulation_exit()
```


Makefile

```
MAKEFILE_PATH := $(abspath $(lastword $(MAKEFILE_LIST)))
CURRENT_DIR := $(dir $(MAKEFILE_PATH))
```


Makefile

```
MAKEFILE_PATH := $(abspath $(lastword $(MAKEFILE_LIST)))
CURRENT_DIR := $(dir $(MAKEFILE_PATH))

SOURCE_DIR := $(abspath $(CURRENT_DIR))
SOURCE_DIRS += $(SOURCE_DIR)
```


Makefile

```
MAKEFILE_PATH := $(abspath $(lastword $(MAKEFILE_LIST)))
CURRENT_DIR := $(dir $(MAKEFILE_PATH))

SOURCE_DIR := $(abspath $(CURRENT_DIR))
SOURCE_DIRS += $(SOURCE_DIR)

APP_OUTPUT_DIR := $(abspath $(CURRENT_DIR))/

BUILD_DIR = build/
```

Makefile

```
MAKEFILE_PATH := $(abspath $(lastword $(MAKEFILE_LIST)))
CURRENT_DIR := $(dir $(MAKEFILE_PATH))

SOURCE_DIR := $(abspath $(CURRENT_DIR))
SOURCE_DIRS += $(SOURCE_DIR)

APP_OUTPUT_DIR := $(abspath $(CURRENT_DIR))/

BUILD_DIR = build/

APP = conways

SOURCES = conways.c
```


Makefile

```
MAKEFILE_PATH := $(abspath $(lastword $(MAKEFILE_LIST)))
CURRENT_DIR := $(dir $(MAKEFILE_PATH))

SOURCE_DIR := $(abspath $(CURRENT_DIR))
SOURCE_DIRS += $(SOURCE_DIR)

APP_OUTPUT_DIR := $(abspath $(CURRENT_DIR))/

BUILD_DIR = build/

APP = conways

SOURCES = conways.c

include $(SPINN_DIRS)/make/Makefile.SpiNNFrontEndCommon
```


Summary

1. How to use the GFE interface.
2. The machine graph supported by the GFE.
3. Adding vertices, edges and partitions to the machine graph.
4. Data Specification for the graph.
5. Binary Specification.
6. Building and making basic C code.

6th SpiNNaker Workshop

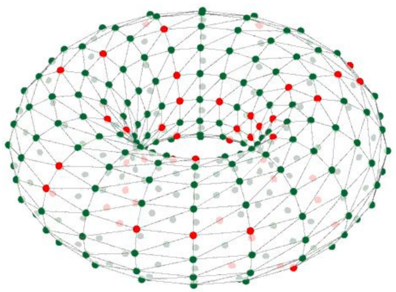
Day 3

September 7th 2016

Time	Session	Presenter
09:00	Simple data I/O and visualisation	ABS (SD)
10:00	Lab time (coffee at 10:30)	
11:00	Maths & fixed point libraries	MH
12:00	Lunch	
13:00	Adding new neuron models	AGR/MH
14:00	Connecting SpiNNaker to external devices	ABS (DRL)
14:30	Lab time (coffee at 15:00)	
16:30	Close	

Manchester, UK

Simple Data I/O and visualisation



Alan Stokes, Andrew Rowley

SpiNNaker Workshop
September 2016

European Research Council
Established by the European Commission

Human Brain Project

EPSRC

SpiNNaker

Contents

Summaries

- Standard PyNN support summary.

External Device Plugin

- What is it, why we need it?
- Usage caveats.

Input

- Injecting spikes into a executing PyNN script.

Output

- Live streaming of spikes from a PyNN script.

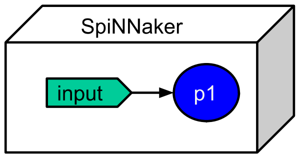
Visualisation

- Live visualisation.

Standard PyNN support (Summary)

- Supports post execution gathering of certain attributes:
  - aka transmitted spikes, voltages etc.

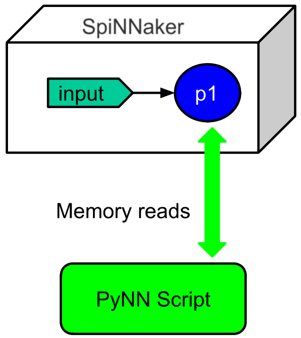
```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
p1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
            {'spike_times': [0]}, label="input")
input_proj = p.Projection(input, p1, p.OneToOneConnector(
  weights=5.0, delays=1))
p1.record()
p1.record_v()
```



Standard PyNN support (Summary)

- Supports post execution gathering of certain attributes:
  - aka transmitted spikes, voltages etc.

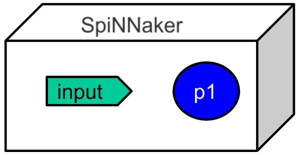
```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
p1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
            {'spike_times': [0]}, label="input")
input_proj = p.Projection(input, p1, p.OneToOneConnector(
  weights=5.0, delays=1))
p1.record()
p1.record_v()
p.run(5000)
spikes = p1.getSpikes()
v = p1.get_v()
```



Standard PyNN support (Summary)

- Supports spike sources of:
  - Spike Source Array, Spike source poisson.

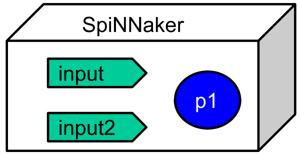
```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
p1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
            {'spike_times': [0]}, label="input")
```



Standard PyNN support (Summary)

- Supports spike sources of:
  - Spike Source Array, Spike source poisson.

```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
p1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
            {'spike_times': [0]}, label="input")
input2 = p.Population(1, p.SpikeSourcePoisson,
            {'rate':100, 'duration':50}, label='input2')
```

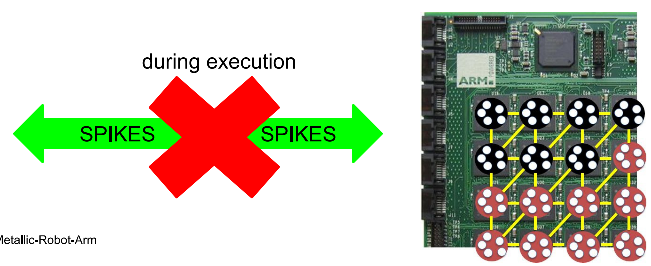


Standard PyNN support (Summary)

Restrictions

1. Recorded data is stored on SDRAM on each chip.
2. Data to be injected has to be known up-front, or rate based.
3. No support for closed loop execution with external devices.

http://www.conrad.com/ce/en/product/191516/Arexx-RA1-PRO-Metallic-Robot-Arm

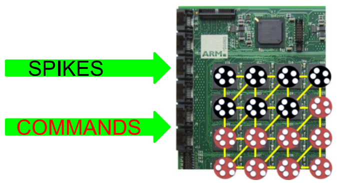


External Device Plugin

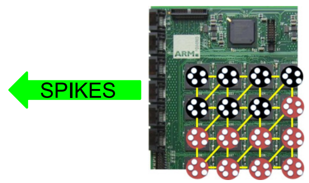
Why? what?

1. Contains functionality for PyNN scripts.
2. Not official PyNN!!!

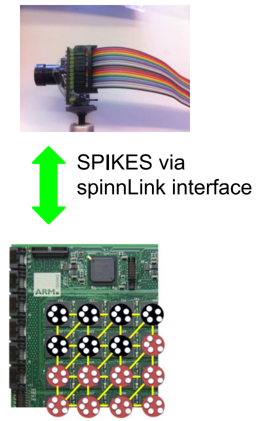
What does it Includes?



1. Live injection of events and commands into SpiNNaker



2. Live streaming of events from populations.



3. External devices support: Covered at 14:00

External Device Plugin

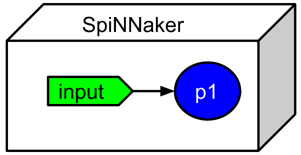
Caveats:

- Injection and live output currently only usable only with the ethernet connection,
- Limited bandwidth of:
  - A small number of spikes per millisecond time step, per ethernet,
  - Shared with both injection and live output,
- Best effort communication,
- Has a built in latency,
- Spinnaker commands not supported by other simulators,
- Loss of cores for injection and live output support,
- You can only feed a live population to one place.

Injecting spikes into PyNN scripts

PyNN script changes

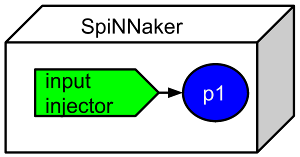
```
import pyNN.spiNNaker as p

p.setup(timestep=1.0)
p1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
                          {'spike_times': [0]}, label="input")
input_proj = p.Projection(input, p1, p.OneToOneConnector(
   weights=5.0, delays=1))
# loop(synfire connection)
loop_forward = list()
for i in range(0, n_neurons - 1):
        loop_forward.append((i, (i + 1) % n_neurons, weight_to_spike, 3))
Frontend.Projection(pop_forward, pop_forward, Frontend.FromListConnector(loop_forward))
```



Injecting spikes into PyNN scripts

PyNN script changes: Declaring an injector population

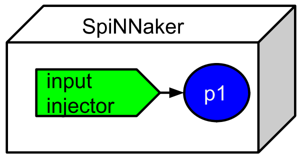
```
import pyNN.spiNNaker as p
import spynnaker_external_devices_plugin.pyNN as ExternalDevices
p.setup(timestep=1.0)
p1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input_injector = p.Population(1, ExternalDevices.SpikeInjector,
                          {'port':95768}, label="injector)
input_proj = p.Projection(input_injector, p1, p.OneToOneConnector(
   weights=5.0, delays=1))
# loop(synfire connection)
loop_forward = list()
for i in range(0, n_neurons - 1):
        loop_forward.append((i, (i + 1) % n_neurons, weight_to_spike, 3))
Frontend.Projection(pop_forward, pop_forward, Frontend.FromListConnector(loop_forward))
```

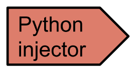


Injecting spikes into PyNN scripts

PyNN script changes: Setting up python injector

```
............
# create python injector
def send_spike(label, sender):
        sender.send_spike(label, 0, send_full_keys=True)
```

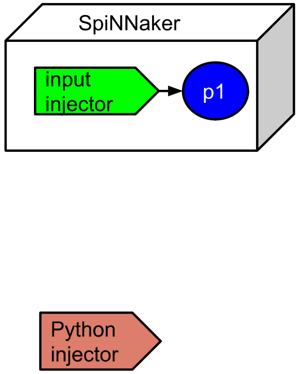




Injecting spikes into PyNN scripts

PyNN script changes: Setting up python injector

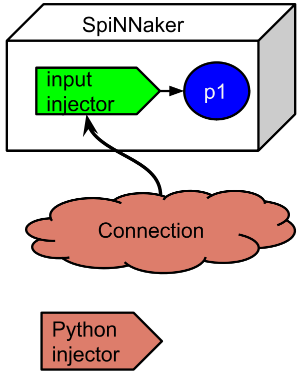
```
............
# create python injector
def send_spike(label, sender):
        sender.send_spike(label, 0, send_full_keys=True)
# import python injector connection
from spynnaker_external_devices_plugin.pyNN.connections.\
spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
```



Injecting spikes into PyNN scripts

PyNN script changes: Setting up python injector

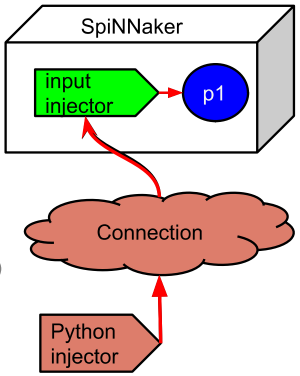
```
............
# create python injector
def send_spike(label, sender):
        sender.send_spike(label, 0, send_full_keys=True)
# import python injector connection
from spynnaker_external_devices_plugin.pyNN.connections.\
spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
# set up python injector connection
live_spikes_connection = SpynnakerLiveSpikesConnection(
   receive_labels=None, local_port=19996, send_labels=["spike_sender"])
```



Injecting spikes into PyNN scripts

PyNN script changes: Setting up python injector

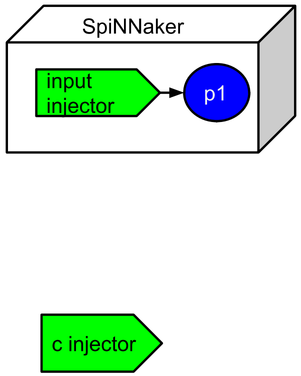
```
............
# create python injector
def send_spike(label, sender):
        sender.send_spike(label, 0, send_full_keys=True)
# import python injector connection
from spynnaker_external_devices_plugin.pyNN.connections.\
spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
# set up python injector connection
live_spikes_connection = SpynnakerLiveSpikesConnection(
        receive_labels=None, local_port=19996, send_labels=["spike_sender"])
# register python injector with injector connection
live_spikes_connection.add_start_callback("spike_sender",  send_spike)
p.run(500)
```



Injecting spikes into PyNN scripts

PyNN script changes: Setting up c injector

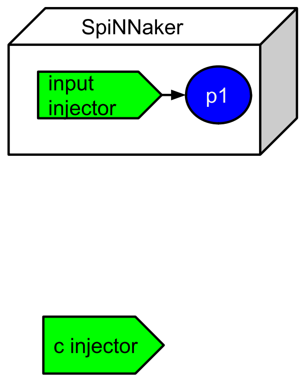
```
............
# create c injector
void send_spike(str label, SpynnakerLiveSpikeConnection sender){
        sender.send_spike(label, 0, send_full_keys=True)  }
```



Injecting spikes into PyNN scripts

PyNN script changes: Setting up c injector

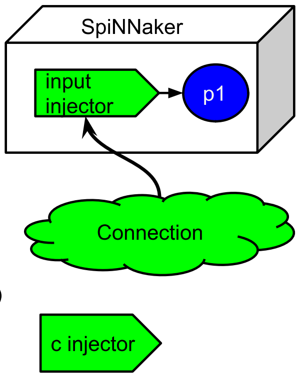
```
…………
# create c injector
void send_spike(str label, SpynnakerLiveSpikeConnection sender){
        sender.send_spike(label, 0, send_full_keys=True)  }
# import c injector connection
#include<SpynnakerLiveSpikeConnection.h>
```



Injecting spikes into PyNN scripts

PyNN script changes: Setting up c injector

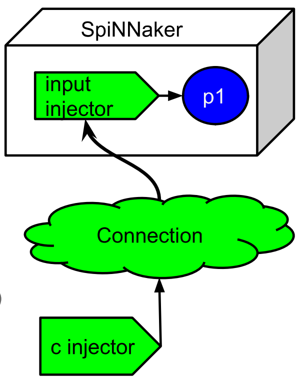
```
…………
# create c injector
void send_spike(str label, SpynnakerLiveSpikeConnection sender){
        sender.send_spike(label, 0, send_full_keys=True)  }
# import c injector connection
#include<SpynnakerLiveSpikeConnection.h>
# set up c injector connection
SpynnakerLiveSpikesConnection live_spikes_connection =

SpynnakerLiveSpikesConnection(

        receive_labels=None, local_port=19996, send_labels=["spike_sender"])
```



Injecting spikes into PyNN scripts

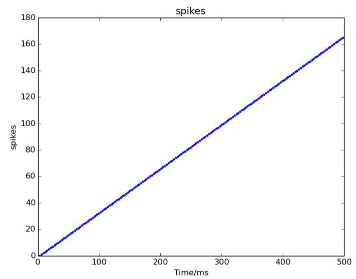
PyNN script changes: Setting up c injector

```
…………
# create c injector
void send_spike(str label, SpynnakerLiveSpikeConnection sender){
        sender.send_spike(label, 0, send_full_keys=True)  }
# import c injector connection
#include<SpynnakerLiveSpikeConnection.h>
# set up c injector connection
SpynnakerLiveSpikesConnection live_spikes_connection =

SpynnakerLiveSpikesConnection(

        receive_labels=None, local_port=19996, send_labels=["spike_sender"])
# register c injector with injector connection
live_spikes_connection.add_start_callback("spike_sender",  send_spike)
```



Injecting spikes into PyNN scripts

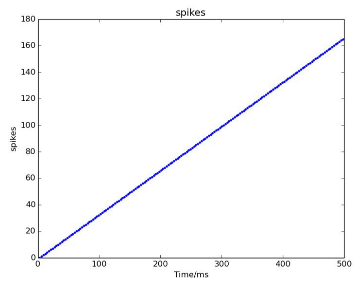
Behaviour with (SpikeSourceArray)



SAME!!!!!

BUT BORING!!!!

Behaviour with Live injection!



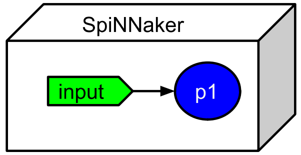
DEMO TIME!!! Injection

PYTHON DEMO!!!!

Live output from PyNN scripts

PyNN script changes: declaring live output population

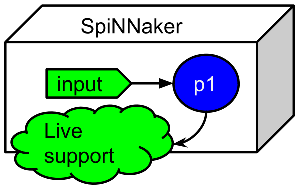
```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
p1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
              {'spike_times': [0]}, label="input")
input_proj = p.Projection(input, p1, p.OneToOneConnector(
   weights=5.0, delays=1))
```



Live output from PyNN scripts

PyNN script changes: declaring live output population

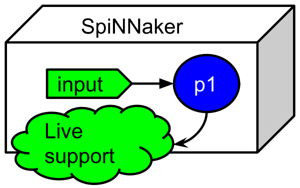
```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
p1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
              {'spike_times': [0]}, label="input")
input_proj = p.Projection(input, p1, p.OneToOneConnector(
   weights=5.0, delays=1))
# declare a live output for a given population.
import spynnaker_external_devices_plugin.pyNN as ExternalDevices
ExternalDevices.activate_live_output_for(p1)
```



Live output from PyNN scripts

PyNN script changes: python receiver

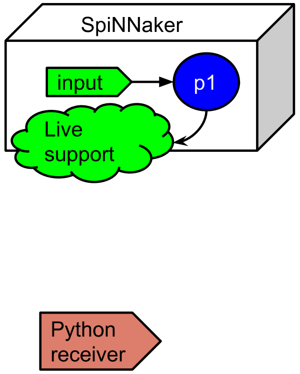
```
...............
# declare python code when received spikes for a timer tick
def receive_spikes(label, time, neuron_ids):
     for neuron_id in neuron_ids:
          print "Received spike at time {} from {}-{}"
               .format(time, label, neuron_id)
```



Live output from PyNN scripts

PyNN script changes: python receiver

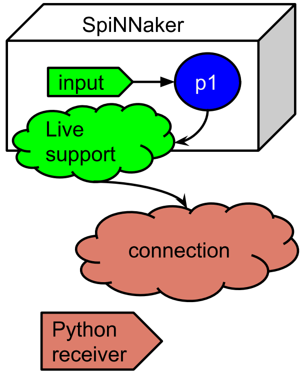
```
……………
# declare python code when received spikes for a timer tick
def receive_spikes(label, time, neuron_ids):
        for neuron_id in neuron_ids:
                print "Received spike at time {} from {}-{}"
                        .format(time, label, neuron_id)
# import python live spike connection
from spynnaker_external_devices_plugin.pyNN.connections.\
spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
```



Live output from PyNN scripts

PyNN script changes: python receiver

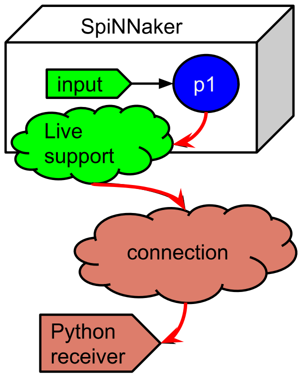
```
……………
# declare python code when received spikes for a timer tick
def receive_spikes(label, time, neuron_ids):
        for neuron_id in neuron_ids:
                print "Received spike at time {} from {}-{}"
                        .format(time, label, neuron_id)
# import python live spike connection
from spynnaker_external_devices_plugin.pyNN.connections.\
spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
# set up python live spike connection
live_spikes_connection = SpynnakerLiveSpikesConnection(
   receive_labels=["receiver"], local_port=19995, send_labels=None)
```



Live output from PyNN scripts

PyNN script changes: python receiver

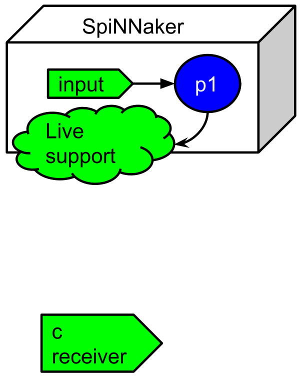
```
……………
# declare python code when received spikes for a timer tick
def receive_spikes(label, time, neuron_ids):
        for neuron_id in neuron_ids:
                print "Received spike at time {} from {}-{}"
                        .format(time, label, neuron_id)
# import python live spike connection
from spynnaker_external_devices_plugin.pyNN.connections.\
spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
# set up python live spike connection
live_spikes_connection = SpynnakerLiveSpikesConnection(
        receive_labels=["receiver"], local_port=19995, send_labels=None)
# register python receiver with live spike connection
live_spikes_connection.add_receive_callback("receiver", receive_spikes)
p.run(500)
```



Live output from PyNN scripts

PyNN script changes: c receiver

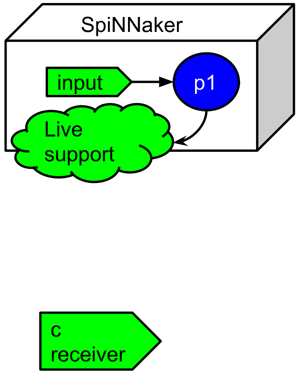
```
……………
# declare c code when received spikes for a timer tick
void receive_spikes(str label, int time, vector<int> neuron_ids){
        for (int index =0; index < neuron_ids.size(); index ++) {
                printf ("Received spike at time %d from %s-%d",
                time, label, neuron_id.next());   } }
```



Live output from PyNN scripts

PyNN script changes: c receiver

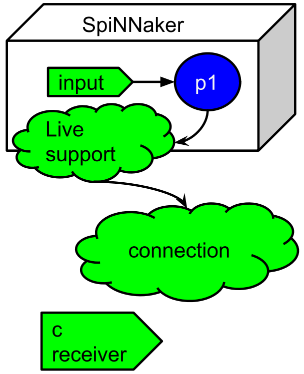
```
……………
# declare c code when received spikes for a timer tick
void receive_spikes(str label, int time, vector<int> neuron_ids){
        for (int index =0; index < neuron_ids.size(); index ++) {
                printf ("Received spike at time %d from %s-%d",
                time, label, neuron_id.next());   } }
# import c live spike connection
# include<SpynnakerLiveSpikesConnection.h>
```



Live output from PyNN scripts

PyNN script changes: c receiver

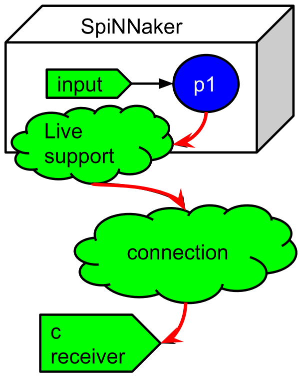
```
……………
# declare c code when received spikes for a timer tick
void receive_spikes(str label, int time, vector<int> neuron_ids){
        for (int index =0; index < neuron_ids.size(); index ++) {
                printf ("Received spike at time %d from %s-%d",
                time, label, neuron_id.next());   } }
# import c live spike connection
# include<SpynnakerLiveSpikesConnection.h>
# set up c live spike connection
SpynnakerLiveSpikesConnection live_spikes_connection =
 SpynnakerLiveSpikesConnection(
        receive_labels=["receiver"], local_port=19995, send_labels=None);
```



Live output from PyNN scripts

PyNN script changes: c receiver

```
……………
# declare c code when received spikes for a timer tick
void receive_spikes(str label, int time, vector<int> neuron_ids){
        for (int index =0; index < neuron_ids.size(); index ++) {
                printf ("Received spike at time %d from %s-%d",
                time, label, neuron_id.next());   } }
# import c live spike connection
# include<SpynnakerLiveSpikesConnection.h>
# set up c live spike connection
SpynnakerLiveSpikesConnection live_spikes_connection =
 SpynnakerLiveSpikesConnection(
        receive_labels=["receiver"], local_port=19995, send_labels=None);
# register c receiver with live spike connection
live_spikes_connection.add_receive_callback("receiver", receive_spikes);
```



DEMO TIME!!! receive live spikes

PYTHON DEMO!!!!

Visualisation

How current supported visualisations work:

1. Uses the live output functionality as discussed previously.
2. Uses the c based receiver and is planned to be open source for users to augment with their own special visuals.
3. Currently contains raster plot support.

Visualisation

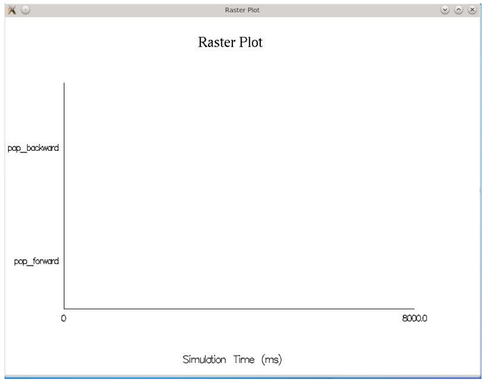
```
cspc277-visualiser-) make -f Makefile.linux
cspc277-visualiser-) .............
cspc277-visualiser-) ./vis -colour_map test_data/spikeio_colours
cspc277-visualiser-)
awaiting tool chain hand shake to say database is ready
```

Input parameters:

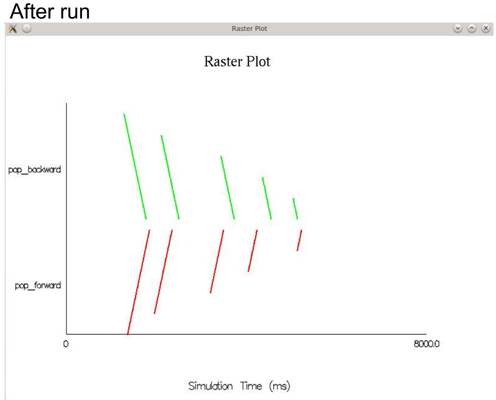
- -colour_map
  - Path to a file containing the population labels to receive, and their associated colours
- -hand_shake_port
  - optional port which the visualiser will listen to for database hand shaking
- -database
  - optional file path to where the database is located, if needed for manual configuration
- -remote_host
  - optional remote host, which will allow port triggering

Visualisation

Before run



After run



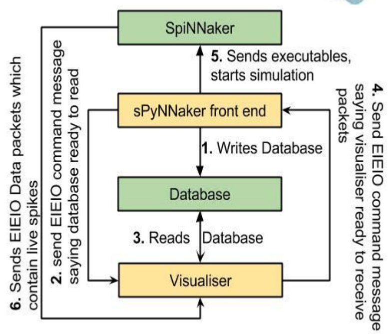
DEMO TIME!!! visualiser and injection of spikes

PYTHON DEMO!!!!

Technical Detail!!!

Notification protocol under the hood!

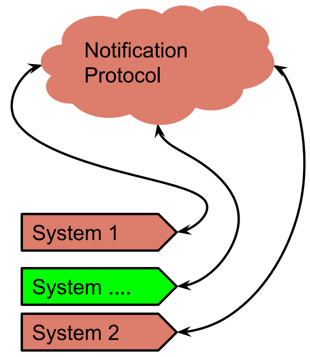
- Everything so far uses the notification protocol.
- It supplies data to translate spikes into population ids.
- If you have more than 1 system running to inject and/or receive, then you need to register this with the notification protocol.



Injecting spikes into PyNN scripts

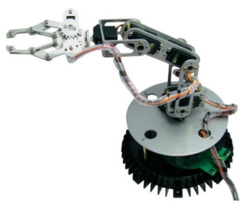
PyNN script changes:
registering a system to the notification protocol

```
............
# register socket addresses for each system
p.register_database_notification_request(
    hostname="local_host"
    notify_port=19990,
    ack_port=19992)
p.register_database_notification_request(
    hostname="local_host"
    notify_port=19993,
    ack_port=19987)
p.register_database_notification_request(
    hostname="local_host"
    notify_port=19760,
    ack_port=19232)
```

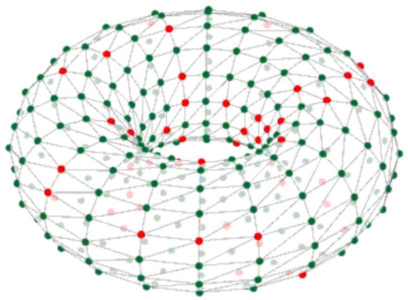


Thanks for listening

Any questions?!



Fixed-Point Maths and Libraries



Michael Hopkins

SpiNNaker Workshop, 7th September 2016

European Research Council
Established by the European Commission

Human Brain Project

EPSRC

SpiNNaker

Overview

1. Numerical calculation on SpiNNaker
2. ISO/IEC 18037 types and operations
3. A simple example
4. Some practical considerations
5. Libraries currently available
6. An example using the libraries
7. Using fixed-point to solve ODEs
8. Future directions

Numerical calculation on SpiNNaker

- No floating point hardware on SpiNNaker
- Software floating point available but too slow for most use cases (and larger binaries)
- Until recently, has needed hand-coded fixed point types and manipulations
- This approach not transparent so can be prone to maintenance issues & mysterious bugs
- More difficult than necessary for developers to translate algorithms into source code
- ISO draft 18037 for fixed point types and operations seen as a good solution

ISO 18037 types and operations

- Draft standard for native fixed point types & operations used like integer or floating point
- Currently only available on GNU toolchain >= 4.7 and ARM target architecture
- 8-, 16-, 32 and 64-bit precisions all available in (un-)saturated and (un-)signed versions
- *accum* type is 32-bit 'general purpose real'; we support io_printf() with s16.15 & u16.16
- *fract* type is 16-bit in [0,1]; we support io_printf() with s0.15 & u0.16

Operations supported are:

- prefix and postfix increment and decrement operators (++, --)
- unary arithmetic operators (+, -, !)
- binary arithmetic operators (+, -, *, /)
- binary shift operators (<<, >>)
- relational operators (<, <=, >=, >)
- equality operators (==, !=)
- assignment operators (+=, -=, *=, /=, <<=, >>=)
- conversions to and from integer, floating-point, or fixed-point types

A simple example

```
#include <stdfix.h>

#define REAL accum
#define REAL_CONST( x ) x##k

REAL  a, b, c = REAL_CONST( 100.001 );
accum d = REAL_CONST( 85.08765 );

int c_main( void )
{
   for( unsigned int i = 0; i < 50; i++ ) {

      a = i * REAL_CONST( 5.7 );

      b = a - i;

      if( a > d ) c  = a + b;
      else          c -= b;

      io_printf( IO_STD,
                 "\n i %u  a = %9.3k  b = %9.3k  c = %9.3k", i, a, b, c );
      }

   return 0;
}
```

Some practical considerations

- Range & precision e.g. for *accum* (s16.15) must have 0.000031 <= | x | <= 65536
- Still need to avoid divides in loops as these are slow on ARM architecture
- *saturated* types safe from overflow but significantly slower
- Need to remember that numerical precision is absolute rather than relative
- Literal constants require type suffix – simplest way is via macro REAL_CONST()
- Don't forget to `#include <stdfix.h>`
- Disciplined use of REAL and REAL_CONST() macros can parameterise entire code base
- Be careful to use the correct type suffix otherwise floating-point will be assumed

Libraries currently available - 1

<u>1) *random.h* – suite of pseudo random number generators by MWH</u>

Provides three high quality uniform generators of *uint32_t* values; Marsaglia's KISS 32 and KISS 64 and L'Ecuyer's WELL1024a.

- All three 'pass' the very stringent DIEHARD, dieharder and TestU01 test suites
- Trade-offs between speed, cycle length and equi-distributional properties
- Available in both simple-to-use form and with full user control over seeds

Have used these Uniform PRNGs as the basis for a set of Non-Uniform PRNGs including currently the following distributions:

- Gaussian
- Poisson (optimised for small rates at the moment)
- Exponential

...with more on the way. Let us know your requirements and we will try to help.

Libraries currently available - 2

<u>2) *stdfix-full-iso.h* & *stdfix-math.h* – ISO & transcendental functions by DRL</u>

Fill in the gaps in the GCC implementation of the ISO draft fixed point maths standard and some extensions:

- Standardised type conversions between fixed point representations
- Utility functions for all types i.e. abs(x), min(x), max(x), round(x), countls(x)
- Mechanism for automatically inferring the right argument type (uses GNU extension)

Fixed point replacements for essential floating point *libm* functions i.e. expk(x), sqrtk(x), logk(x), sink(x), cosk(x) and others such as atank(x), powk(x,y), 1/x on the way

- Hand-optimised for speed and accuracy on ARM architecture
- 10-30x faster than *libm* calls, hence feasible for use inside loops if necessary

An example using the libraries

```
accum           a, b, c, d;
uint32_t        r1;
unsigned fract  uf1;

init_WELL1024a_simp();  // need to initialise WELL1024a RNG before use

for( unsigned int i = 0; i < 22; i++ ) {

    r1 = WELL1024a_simp();                      // draw from Uniform RNG

    uf1 = (unsigned fract) ulrbits( r1 );       // convert to unsigned fract

// draw from Std Gaussian distribution using MARS64
    a = gaussian_dist_variate( mars_kiss64_simp, NULL );

// do some calculations on a and then log()
    b = logk( absk( a * REAL_CONST( 100.0 ) ) );

// sqrt() of value drawn from Exponential distribution using WELL1024a
    c = sqrtk( exponential_dist_variate( WELL1024a_simp, NULL ) );

    d = expk( (accum) ( i - 10 ) );         // exp() from -10 to 11

    io_printf( IO_STD, "\n i %4u
      uf1=[Uniform{*}]= %8.6R  a=[Gauss{*}]= %7.3k b=[ln(abs(100 a))]= %7.3k
      c=[sqrt(Exponential{*})]= %7.3k  d=[exp(i-10)]= %10.3k ", i, uf1, a, b, c, d );

    }
```

Using fixed-point to solve ODEs - 1

- Simulating neuron models usually means solving Ordinary Differential Equations (ODEs)
- This ranges from very easy (current input LIF has simple closed-form) solution to very challenging i.e. Hodgkin-Huxley with 4 state variables, nonlinear and very 'stiff' ODE
- Numerical calculations are required with a balance between accuracy & efficiency
- With care and attention to detail, fixed-point can be used to get very close to floating-point results. However, models with more complex behaviour are a significant challenge
- A new approach called *Explicit Solver Reduction* (ESR) makes this easier in many cases and is described in: Hopkins & Furber (2015), "Accuracy and Efficiency in Fixed-Point Neural ODE Solvers", *Neural Computation* **27**, 1–35
- Good results found for Izhikevich neuron at real-time simulation speed & 1 ms time step

Using fixed-point to solve ODEs - 2

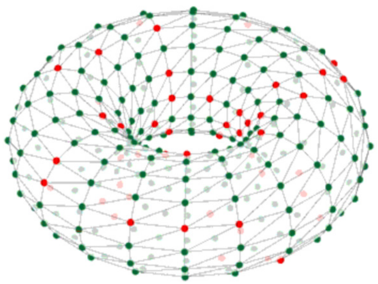
```
/*
   ESR algebraic reduction of the combination of Izhikevich neuron model and
   Runge-Kutta 2nd order midpoint method.  Hand-optimised interim variables and
   arithmetic ordering for balance between speed and accuracy.  See Neural Computation
   paper for more details.
*/
static inline void _rk2_kernel_midpoint( REAL h, neuron_pointer_t neuron,
                                          REAL input_this_timestep ) {
// to match Mathematica names
    REAL lastV1 = neuron->V;
    REAL lastU1 = neuron->U;
    REAL a = neuron->A;
    REAL b = neuron->B;

// generate common interim variables
    REAL pre_alph = REAL_CONST(140.0) + input_this_timestep - lastU1;
    REAL alpha = pre_alph
             + ( REAL_CONST(5.0) + REAL_CONST(0.0400) * lastV1 ) * lastV1;
    REAL eta = lastV1 + REAL_HALF( h * alpha );

// could be represented as a long fract but need efficient mixed-arithmetic functions
    REAL beta = REAL_HALF( h * ( b * lastV1 - lastU1 ) * a );

// update neuron state
    neuron->V += h * ( pre_alph - beta
                  + ( REAL_CONST(5.0) + REAL_CONST(0.0400) * eta ) * eta );

    neuron->U += a * h * ( -lastU1 - beta + b * eta );
}
```

Future directions

- Optimise operations on differing fixed point types i.e. *accum* * *long fract*
- Add to *stdfix-math* (e.g. new argument types and special functions)
- Add to *random* (e.g. longer cycle uniform PRNG and more non-uniform distributions)
- New libraries such as probability distributions to allow Bayesian inference tools
- io_printf() to be extended to more types such as *long fract*, *unsigned long fract*
- Linear Algebra operations such as matrix multiply, SVD and other decompositions
- SpiNNaker architecture potentially good choice for massively parallel algorithms e.g. MCMC

MANCHESTER 1824

Adding New Neuron Models



Andrew Rowley, Michael Hopkins

SpiNNaker Workshop, September 2016

European Research Council
Established by the European Commission

Human Brain Project

EPSRC

SpiNNaker

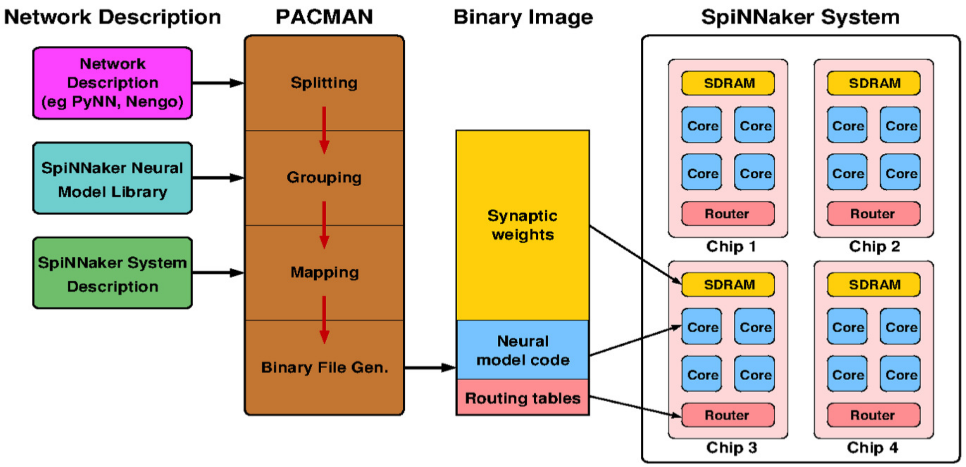
MANCHESTER 1824

Required code separation

- Any new neuron model requires both C and Python code
- C code makes the actual executable (on SpiNNaker), Python code configures the setup and load phases (on the host)
- These are separate but <u>must be</u> perfectly coordinated
- In almost all cases, the C code will be solving an ODE which describes how the neuron state evolves over time and in response to input

MANCHESTER 1824

Required code separation



We will first describe the C requirements...

MANCHESTER 1824

C Data Structures and Parameters

- The parameters and state of a neuron at any point in time need to be stored in memory
- For each neuron, the C header defines the ordering and size of each stored value
- The C types can be standard integer and floating-point, or ISO draft standard fixed-point, as required (see later talk *Maths & fixed-point libraries*)
- There is also one global data structure which services all neurons on a core

So here is an example using the Izhikevich neuron...

Specific neuron model – data structure

```
#include "neuron-model.h"

//  Izhikevic neuron data structure defined in neuron_model_izh_curr_impl.h

typedef struct neuron_t {

// 'fixed' parameters - abstract units
   REAL    A;
   REAL    B;
   REAL    C;
   REAL    D;

// variable-state parameters
   REAL    V;          // nominally in [mV]
   REAL    U;

// offset current [nA]
   REAL    I_offset;

// private variable used internally in C code
   REAL    this_h;

} neuron_t;

...
```

Global data structure

```
...

/*
   Global data structure defined in neuron_model_izh_curr_impl.h
*/

typedef struct global_neuron_params_t {

// Machine time step in milliseconds
   REAL    machine_timestep_ms;

} global_neuron_params_t;
```

Implementing the state update

- Neuron models are typically described as systems of initial value ODEs
- At each time step, the internal state of each neuron needs to be updated in response to inherent dynamics and synaptic input
- There are many ways to achieve this; there will usually be a 'best approach' (in terms of balance between accuracy & efficiency) for each neuron model
- A recently published paper gives a lot more detail: Hopkins & Furber (2015), "Accuracy and Efficiency in Fixed-Point Neural ODE Solvers" , *Neural Computation* **27**, 1–35
- The key function will always be *neuron_model_state_update*(); the other functions are mainly to support this and allow debugging etc.

Continuing the example by describing the key interfaces...

Neuron model API

```
// pointer to a neuron data type - used in all access operations
typedef struct neuron_t* neuron_pointer_t;


// set the global neuron parameters
void neuron_model_set_global_neuron_params( global_neuron_params_pointer_t params );


// key function in timer loop that updates neuron state and returns membrane voltage
state_t neuron_model_state_update(
             input_t exc_input, input_t inh_input, input_t external_bias,
             neuron_pointer_t neuron );


// return membrane voltage (= first state variable) for a given neuron
state_t neuron_model_get_membrane_voltage( restrict neuron_pointer_t neuron );


// update the neuron structure to take account of a spike
void neuron_model_has_spiked( neuron_pointer_t neuron );


// print out neuron definition and/or state variables (for debug)
void neuron_model_print_parameters( restrict neuron_pointer_t neuron );
void neuron_model_print_state_variables( restrict neuron_pointer_t neuron );
```

Specific neuron model – key functions

```
/* simplified version of Izhikevic neuron code defined in neuron_model_izh_curr_impl.c  */


// key function in timer loop that updates neuron state and returns membrane voltage
state_t neuron_model_state_update(
             input_t exc_input, input_t inh_input, input_t external_bias,
             neuron_pointer_t neuron ) {

// collect inputs
    input_t input_this_timestep =
        exc_input - inh_input + external_bias + neuron->I_offset;

// most balanced ESR update found so far
    _rk2_kernel_midpoint( neuron->this_h, neuron, input_this_timestep );
    neuron->this_h = global_params->machine_timestep_ms;

// return the value of the membrane voltage
    return neuron->V;
}


// make the discrete changes to state after a spike has occurred
void neuron_model_has_spiked( neuron_pointer_t neuron ) {
    neuron->V  = neuron->C;        // reset membrane voltage
    neuron->U += neuron->D;        // offset 2nd state variable
}
```

Threshold Models – Interface and Implementation

Interface

```
// Pointer to threshold data type - used to access all operations
typedef struct threshold_type_t;


// Main interface function - determine if the value is above the threshold
static inline bool threshold_type_is_above_threshold(
        state_t value, threshold_type_pointer_t threshold_type );
```

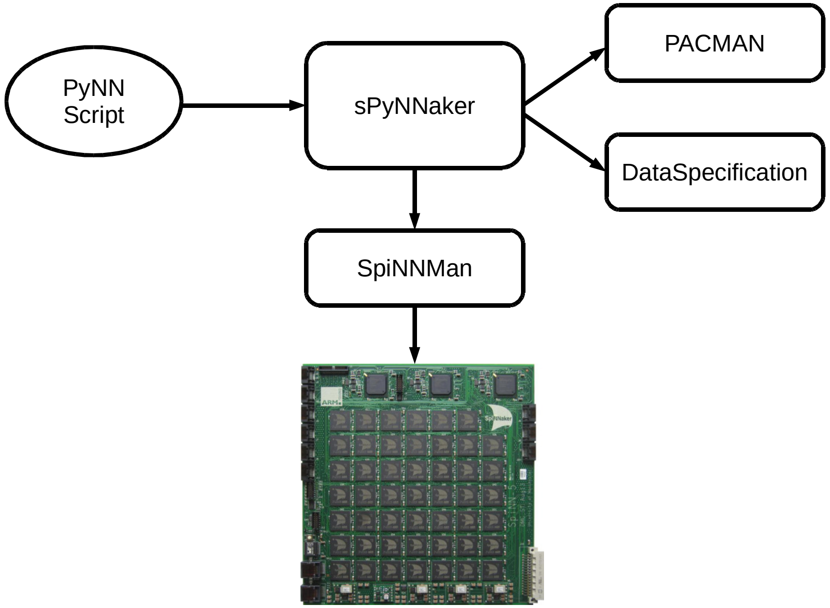
Static Threshold Implementation

```
typedef struct threshold_type_t {

    // The value of the static threshold
    REAL threshold_value;

} threshold_type_t;


static inline bool threshold_type_is_above_threshold(
        state_t value, threshold_type_pointer_t threshold_type ) {

    return REAL_COMPARE( value, >=, threshold_type•threshold_value );
}
```


Makefile

```
APP = my_model_curr_exp

# This is the folder where things will be built (this will be created)
BUILD_DIR = build/

# This is the neuron model implementation
NEURON_MODEL = $(EXTRA_SRC_DIR)/neuron/models/my_neuron_model_impl.c

# This is the header of the neuron model, containing the definition of neuron_t
NEURON_MODEL_H = $(EXTRA_SRC_DIR)/neuron/models/neuron_model_my_model_curr_exp.h

# This is the header containing the input type (current in this case)
INPUT_TYPE_H = $(SOURCE_DIR)/neuron/input_types/input_type_current.h

# This is the header containing the threshold type (static in this case)
THRESHOLD_TYPE_TYPE_H = $(SOURCE_DIR)/neuron/threshold_types/threshold_type_static.h

# This is the header containing the synapse shaping type (exponential in this case)
SYNAPSE_TYPE_H = $(SOURCE_DIR)/neuron/synapse_types/synapse_types_exponential_impl.h

# This is the synapse dynamics type (in this case static i.e. no synapse dynamics)
SYNAPSE_DYNAMICS = $(SOURCE_DIR)/neuron/plasticity/synapse_dynamics_static_impl.c

# This includes the common Makefile that hides away the details of the build
include ../Makefile.common
```

Python Interface – Why?



Python Interface

```
from spynnaker.pyNN.models.neuron.neuron_models.abstract_neuron_model \
    import AbstractNeuronModel

class NeuronModelIzh(AbstractNeuronModel):
    def __init__(self, n_neurons, a, b, c, d, v_init, u_init, i_offset):
        AbstractNeuronModel.__init__(self)
        self._n_neurons = n_neurons
```


Python Interface - Parameters

- Parameters can be:
  - Individual values
  - Array of values (one per neuron)
  - RandomDistribution
- Normalise Parameters
  - utility_calls.convert_param_to_numpy(
    param, n_neurons)

Python Interface – initializer

```
from spynnaker.pyNN.models.neuron.neuron_models.abstract_neuron_model \
    import AbstractNeuronModel

class NeuronModelIzh(AbstractNeuronModel):
    def __init__(self, n_neurons, a, b, c, d, v_init, u_init, i_offset):
        AbstractNeuronModel.__init__(self)
        self._n_neurons = n_neurons

        self._a = utility_calls.convert_param_to_numpy(a, n_neurons)
        self._b = utility_calls.convert_param_to_numpy(b, n_neurons)
        self._c = utility_calls.convert_param_to_numpy(c, n_neurons)
        self._d = utility_calls.convert_param_to_numpy(d, n_neurons)
        self._v_init = utility_calls.convert_param_to_numpy(v_init, n_neurons)
        self._u_init = utility_calls.convert_param_to_numpy(u_init, n_neurons)
        self._i_offset = utility_calls.convert_param_to_numpy(
            i_offset, n_neurons)
```


Python Interface – properties

```
class NeuronModelIzh(AbstractNeuronModel):
    ...

    @property
    def a(self):
        return self._a

    @a.setter
    def a(self, a):
        self._a = utility_calls.convert_param_to_numpy(a, self.n_atoms)

    @property
    def b(self):
        return self._b

    @b.setter
    def b(self, b):
        self._b = utility_calls.convert_param_to_numpy(b, self.n_atoms)

    ...
```

MANCHESTER 1824

Python Interface – state initializers

```
class NeuronModelIzh(AbstractNeuronModel):
    ...

    def initialize_v(self, v_init):
        self._v_init = utility_calls.convert_param_to_numpy(v_init, self.n_atoms)

    def initialize_u(self, u_init):
        self._u_init = utility_calls.convert_param_to_numpy(u_init, self.n_atoms)
```


MANCHESTER 1824

Python Interface – parameters

```
class NeuronModelIzh(AbstractNeuronModel):
    ...

    def get_n_neural_parameters(self):
        Return 8

    def get_parameters(self):
        return [
            # REAL a
            NeuronParameter(self._a, DataType.S1615),
            # REAL b
            NeuronParameter(self._b, DataType.S1615),
            # REAL c
            NeuronParameter(self._c, DataType.S1615),
            # REAL d
            NeuronParameter(self._d, DataType.S1615),
            # REAL v
            NeuronParameter(self._v_init, DataType.S1615),
            # REAL u
            NeuronParameter(self._u_init, DataType.S1615),
            # REAL I_offset
            NeuronParameter(self._i_offset, DataType.S1615),
            # REAL this_h
            NeuronParameter(self._machine_time_step / 1000.0, DataType.S1615)
        ]
```


MANCHESTER 1824

Python Interface – global params

```
class NeuronModelIzh(AbstractNeuronModel):
    ...

    def get_n_global_parameters(self):
        return 1

    @inject_items({"machine_time_step": "MachineTimeStep"})
    def get_global_parameters(self, machine_time_step):
        return [
            NeuronParameter(machine_time_step / 1000.0, DataType.S1615)
        ]
```


MANCHESTER 1824

Python Interface - Injection

```
@inject_items({"machine_time_step": "MachineTimeStep"})
def get_global_parameters(self, machine_time_step):
```

- Some items can be “injected” from the interface
  - Specify a dictionary of parameter name to “type” to inject
  - Parameter is in addition to the interface
- Common types include:
  - MachineTimeStep
  - TimeScaleFactor
  - TotalRunTime

Python Interface – CPU usage

```
class NeuronModelIzh(AbstractNeuronModel):
    ...

    def get_n_cpu_cycles_per_neuron(self):

        # A bit of a guess
        return 150
```


Python Interface – Threshold type

```
class ThresholdTypeStatic(AbstractThresholdType):
    """ A threshold that is a static value
    """

    def __init__(self, n_neurons, v_thresh):
        AbstractThresholdType.__init__(self)
        self._n_neurons = n_neurons
        self._v_thresh = utility_calls.convert_param_to_numpy(
            v_thresh, n_neurons)
```

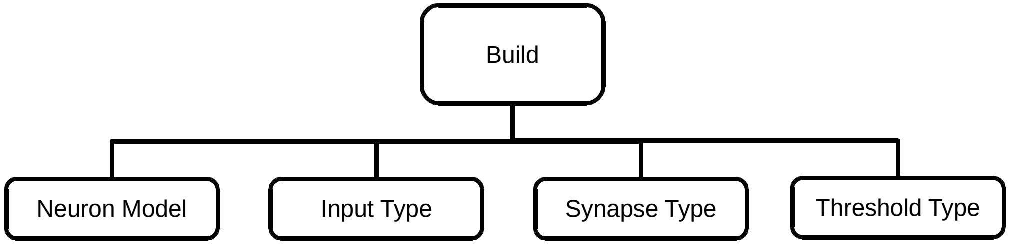

Python Interface – Threshold type

```
class ThresholdTypeStatic(AbstractThresholdType):
    """ A threshold that is a static value
    """
    ...

    @property
    def v_thresh(self):
        return self._v_thresh

    @v_thresh.setter
    def v_thresh(self, v_thresh):
        self._v_thresh = utility_calls.convert_param_to_numpy(
            v_thresh, self._n_neurons)
```


Python Interface – Threshold type

```
class ThresholdTypeStatic(AbstractThresholdType):
    """ A threshold that is a static value
    """
    ...

    def get_n_threshold_parameters(self):
        return 1

    def get_threshold_parameters(self):
        return [
            NeuronParameter(self._v_thresh, DataType.S1615)
        ]

    def get_n_cpu_cycles_per_neuron(self):

        # Just a comparison, but 2 just in case!
        return 2
```

Python Build



Python Build – Class Definition

```python
from spynnaker.pyNN.models.abstract_models.abstract_population_vertex import \
    AbstractPopulationVertex

class IzkCurrExp(AbstractPopulationVertex):
```


Python Build

```python
class IzkCurrExp(AbstractPopulationVertex):
    _model_based_max_atoms_per_core = 255
    default_parameters = {
        'a': 0.02, 'c': -65.0, 'b': 0.2, 'd': 2.0, 'i_offset': 0,
        'u_init': -14.0, 'v_init': -70.0, 'tau_syn_E': 5.0, 'tau_syn_I': 5.0}
```


Python Build – initializer

```python
class IzkCurrExp(AbstractPopulationVertex):
    def __init__(
            self, n_neurons, spikes_per_second=None, ring_buffer_sigma=None,
            incoming_spike_buffer_size=None, constraints=None, label=None,
            a=default_parameters['a'], b=default_parameters['b'],
            c=default_parameters['c'], d=default_parameters['d'],
            i_offset=default_parameters['i_offset'],
            u_init=default_parameters['u_init'],
            v_init=default_parameters['v_init'],
            tau_syn_E=default_parameters['tau_syn_E'],
            tau_syn_I=default_parameters['tau_syn_I']):

        neuron_model = NeuronModelIzh(
            n_neurons, a, b, c, d, v_init, u_init, i_offset)
        synapse_type = SynapseTypeExponential(
            n_neurons, tau_syn_E, tau_syn_I)
        input_type = InputTypeCurrent()
        threshold_type = ThresholdTypeStatic(n_neurons, _IZK_THRESHOLD)

        AbstractPopulationVertex.__init__(
            self, n_neurons=n_neurons, binary="IZK_curr_exp.aplx", label=label,
            max_atoms_per_core=IzkCurrExp._model_based_max_atoms_per_core,
            spikes_per_second=spikes_per_second,
            ring_buffer_sigma=ring_buffer_sigma,
            incoming_spike_buffer_size=incoming_spike_buffer_size,
            model_name="IZK_curr_exp", neuron_model=neuron_model,
            input_type=input_type, synapse_type=synapse_type,
            threshold_type=threshold_type, constraints=constraints)
```

Python Build – max atoms

```python
class IzkCurrExp(AbstractPopulationVertex):

    ...

    @staticmethod
    def set_model_max_atoms_per_core(new_value):
        IzhikevichCurrentExponentialPopulation.\
            _model_based_max_atoms_per_core = new_value

    @staticmethod
    def get_max_atoms_per_core():
        return IzkCurrExp._model_based_max_atoms_per_core
```


New Model Template

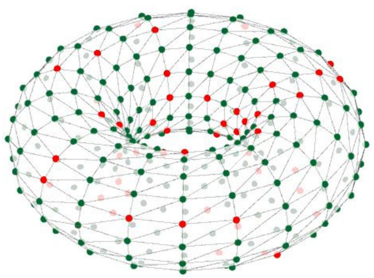
```
c_models
  src
    neuron
      additional_inputs
        my_additional_input.h
      builds
        my_model_curr_exp
          build
          Makefile
        my_model_curr_exp_my_additional_input
        my_model_curr_exp_my_threshold
        my_model_curr_exp_stdp_mad_my_timing_my_weight
        my_model_curr_my_synapse_type
        Makefile.common
      models
        my_neuron_model_impl.c
        my_neuron_model_impl.h
      plasticity
      synapse_types
        synapse_types_my_impl.h
      threshold_types
        my_threshold_type.h
      Makefile
    Makefile.common
  Makefile
```

```
examples
  __init__.py
  my_example.py
python_models
  connectors
  model_binaries
  neuron
    additional_inputs
      __init__.py
      my_additional_input.py
    builds
      __init__.py
      my_model_curr_exp_my_additional_input.py
      my_model_curr_exp_my_threshold.py
      my_model_curr_exp.py
      my_model_curr_my_synapse_type.py
    neuron_models
      __init__.py
      my_neuron_model.py
    plasticity
      stdp
      __init__.py
    synapse_types
      __init__.py
      my_synapse_type.py
    threshold_types
      __init__.py
      my_threshold_type.py
    __init__.py
```


Using Your Model

```python
import pyNN.spiNNaker as p
import python_models as new_models

my_model_pop = p.Population(
    1, new_models.MyModelCurrExp,
    {"my_parameter": 2.0,
     "i_offset": i_offset},
    label="my_model_pop")
```

External Devices



Alan Stokes, Andrew Rowley

SpiNNaker Workshop
September 2016

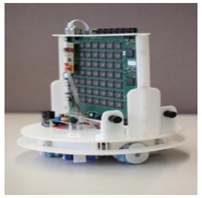
European Research Council

Human Brain Project

Contents

1. How to add external devices that communicate through the SpiNNaker Link into your PyNN scripts.
2. How to add external devices that communicate through the FPGA/SATA connector into your PyNN scripts.
3. How FPGA's are used within multi board systems.

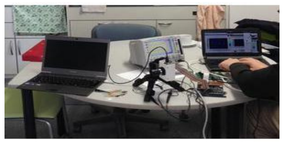
Real time systems?



SpOmnibot
(Retinas & Motors)

FPGA connector

A Retina

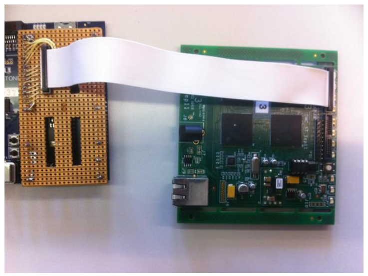


A Osaka retina

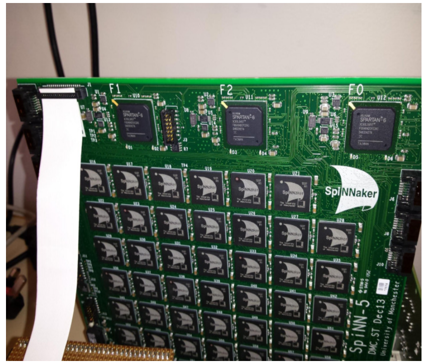
A Cochlea

How to connect devices to a spiNNaker board

Connect the device to the SpiNNaker link connector



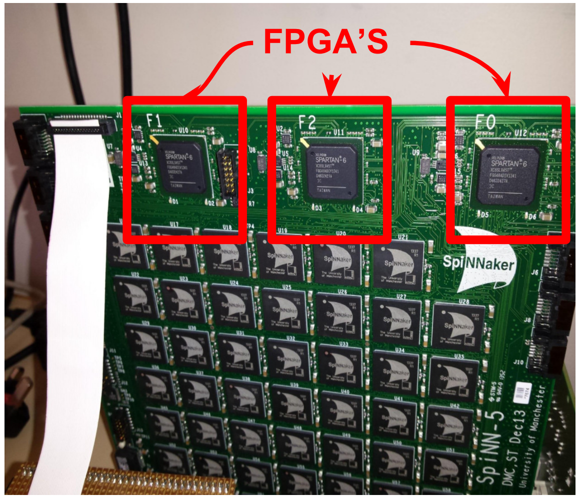
connecting to a
spinn-3 Board



connecting to a
spinn-5 Board

FPGA Programming

1. The FPGA's need repogramming to support external device plugin.
2. This reprogramming is not done by the tools to date.



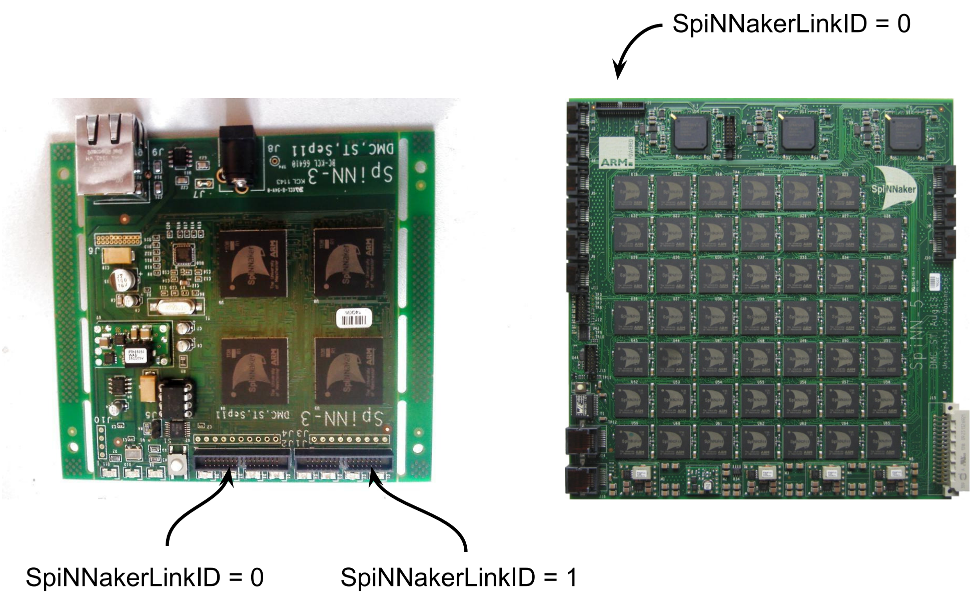
Using an external device: Calls from PYNN

```
import pynn.spinnaker as p
p.setup( timestep=1.0,
         min_delay = 1.0,
         max_delay = 32.0)

# set up populations
pop =  p.Population(
     1, p.IfCurExp, {} label='pop1))


# set populations to record spikes
pop.record()

# run the simulation for 10000 ms
p.run(10000)
```

Using an external device: Calls from PYNN

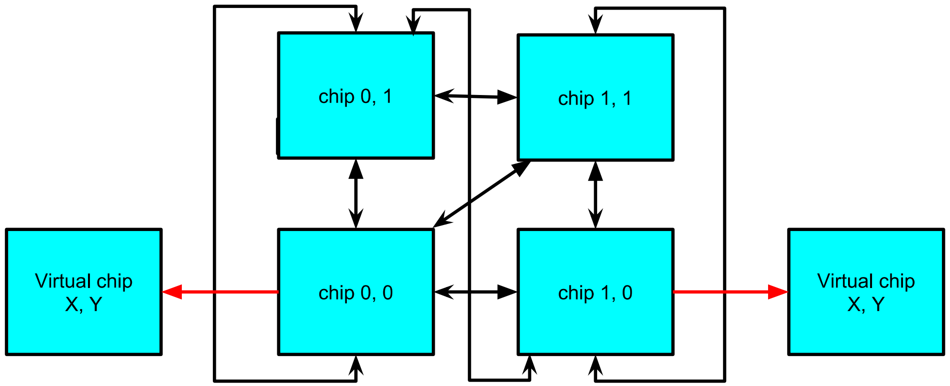
```
import pynn.spinnaker as p
p.setup( timestep=1.0,
         min_delay = 1.0,
         max_delay = 32.0)

# set up populations
pop =  p.Population(
     1, p.ExternalDevice, {
          'spinnaker_link':0,
           'board_address':None OR 192.168.0.253,} label='pop1))

# set populations to record spikes
pop.record()       External devices cant be recorded

# run the simulation for 10000 ms
p.run(10000)
```

Which SpiNNaker Link is which?



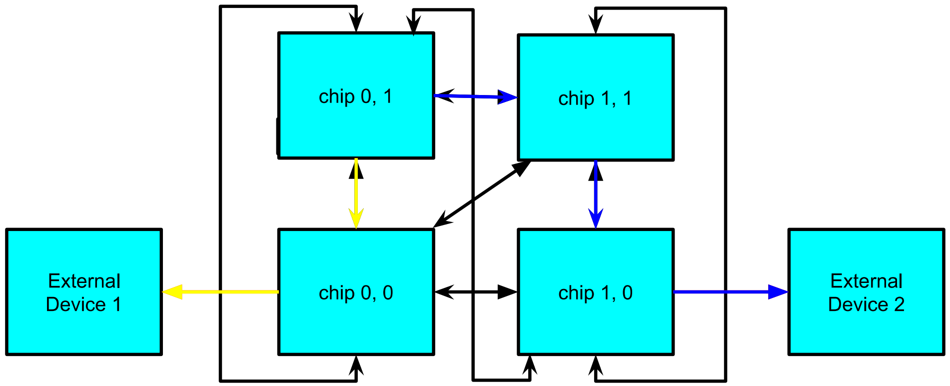
How this works in detail

1. Every Spinnaker link is defined as a link to a virtual chip
2. Your device vertex is then placed within this virtual chip.



Why does it matter?

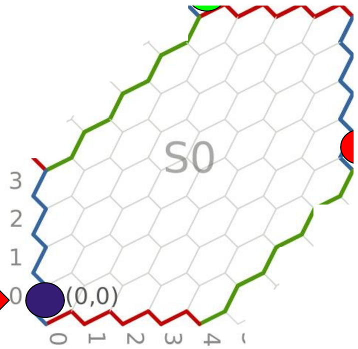
1. Routing won't work if mixed up



Board Address?

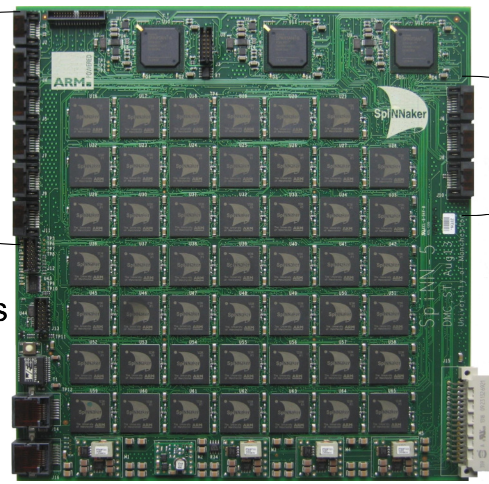
- 192.168.0.1
- 192.168.0.3
- 192.168.0.5

When set to None (default behaviour)



SATA Link connected devices!

torus SATA links



SATA links

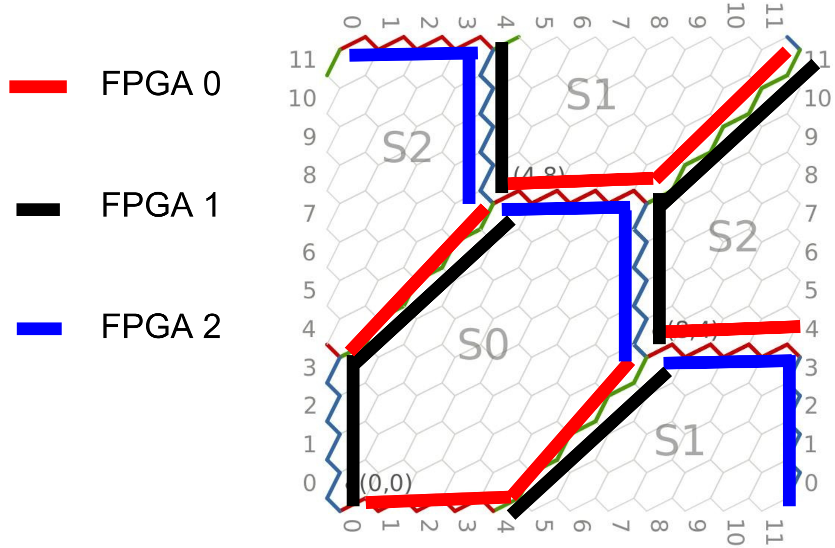
People who have devices

1. Bernabe Linares-Barranco
2. Jorg Conradt

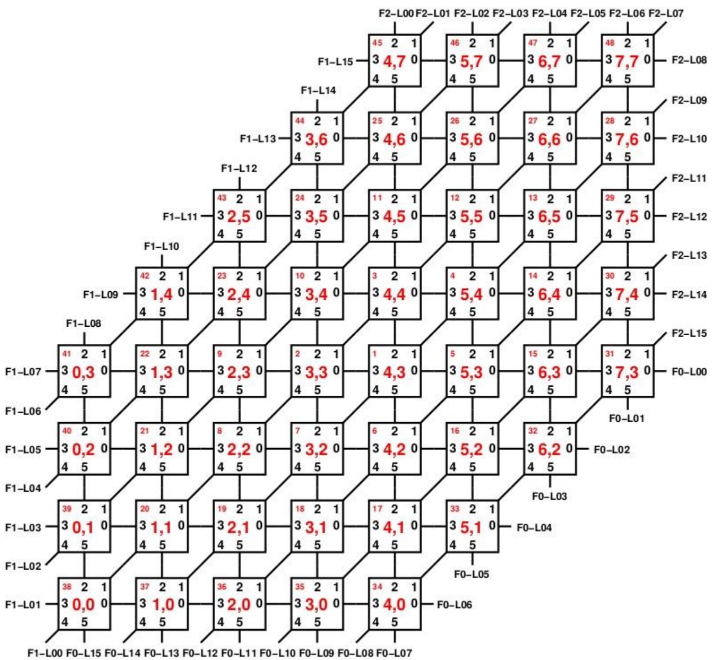
How to represent this in your PyNN scripts.

```
p.Population(2000, external_devices.ArbitraryFPGADevice,
             {
                'fpga_link_id': 12,
                'fpga_id': 1,
                'board_address': None OR 192.168.0.1
             },
           label='External sata thing')
```

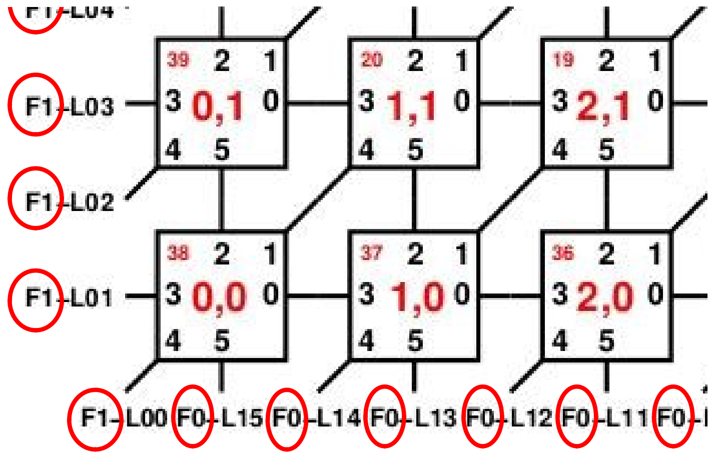

How do FPGA's work in Multi-board machines?



What the input parameters mean?

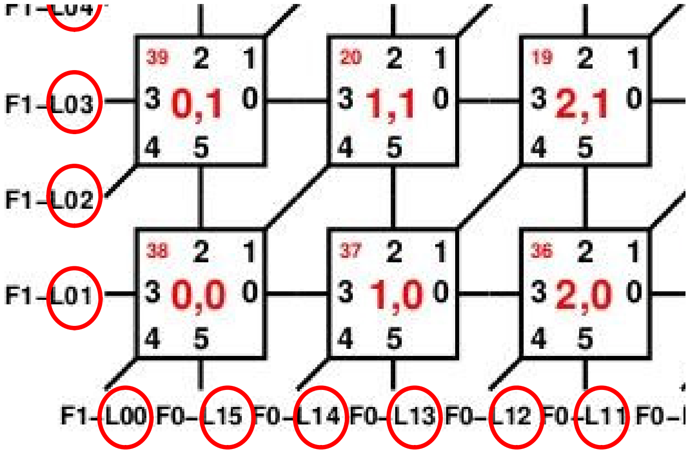


What the input parameters mean?



FPGA_ID =

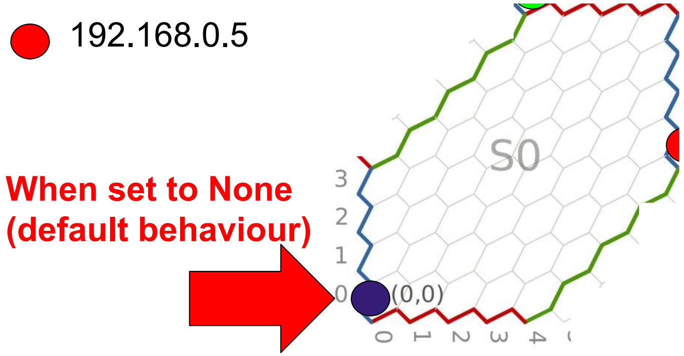
What the input parameters mean?



FPGA__link_ID =

Board Address again

- 192.168.0.1
- 192.168.0.3
- 192.168.0.5



What you need to do to get SATA links working for your device.

1. Reprogram the FPGA's to support the communication between device and PyNN related models.

2. **The reprogramming needs to result in a disconnected edge between two chips who's communication is done through the FPGA.**

3. Extend or use the ArbitaryFPGADevice vertex to represent any extra constraints you need.

Summary

1. Discussed External devices plugged in through the SpiNNaker Link.

2. Discussed External devices plugged in through the FPGA / SATA connector.

3. Discussed How the FPGA's interact in the communication fabric.

6th SpiNNaker Workshop

Day 4

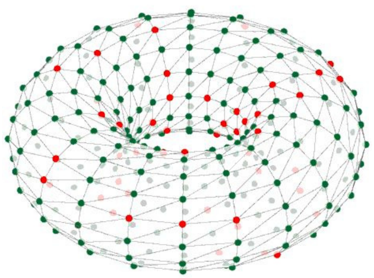
September 8th 2016

Time	Session	Presenter
09:00	Adding new models of synaptic plasticity	JK
09:45	Graph Front End – further details	ABS (AGR)
10:30	Coffee	
11:00	Lab time	
12:00	Lunch	
13:00	Using big SpiNNaker machines remotely: The HBP portal	AGR
13:30	Lab time (coffee at 15:00)	
15:30	Demonstration of NENGO language and environment	TBC
16:30	Close	

Manchester, UK

MANCHESTER 1824

Adding new models of synaptic plasticity



Jamie Knight

SpiNNaker Workshop
September 2016

erc European Research Council — Established by the European Commission

Human Brain Project

EPSRC

SpiNNaker

MANCHESTER 1824

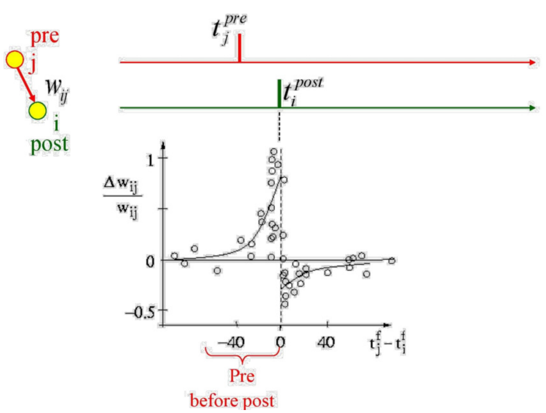
Outline

- Introduction to spike-timing dependent plasticity
- Simulating STDP
- Limitations of pair-based STDP
- Triplet STDP
- SpiNNaker implementation

MANCHESTER 1824

Introduction to spike-timing dependent plasticity

"Cells that fire together, wire together"



http://www.scholarpedia.org/article/Spike-timing_dependent_plasticity

MANCHESTER 1824

Simulating STDP - Traces

Pre-synaptic trace

$$\frac{dx_j}{dt} = -\frac{x_j}{\tau_x} + \sum_{t_j^f} \delta(t - t_j^f)$$

Post-synaptic trace

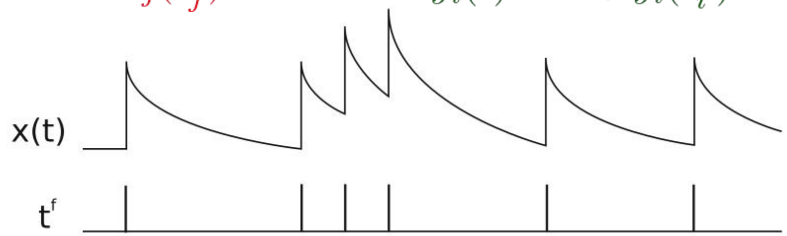
$$\frac{dy_i}{dt} = -\frac{y_i}{\tau_y} + \sum_{t_i^f} \delta(t - t_i^f)$$

At pre-synaptic spike time

$$x_j(t) = 1 + x_j(t_j^f)e^{-\frac{t - t_j^f}{\tau_x}}$$

At post-synaptic spike time

$$y_i(t) = 1 + y_i(t_i^f)e^{-\frac{t - t_i^f}{\tau_y}}$$



Morrison, A., Diesmann, M., & Gerstner, W. (2008). Phenomenological models of synaptic plasticity based on spike timing. Biological Cybernetics, 98, 459–478.

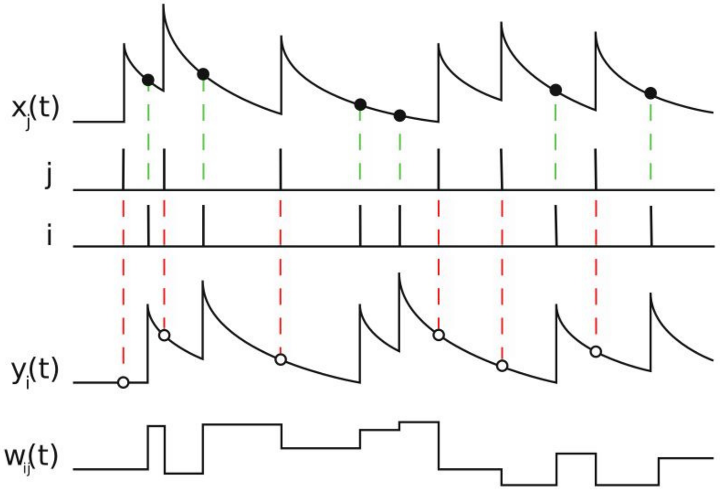
Simulating STDP - Weight update

Pre-synaptic weight update

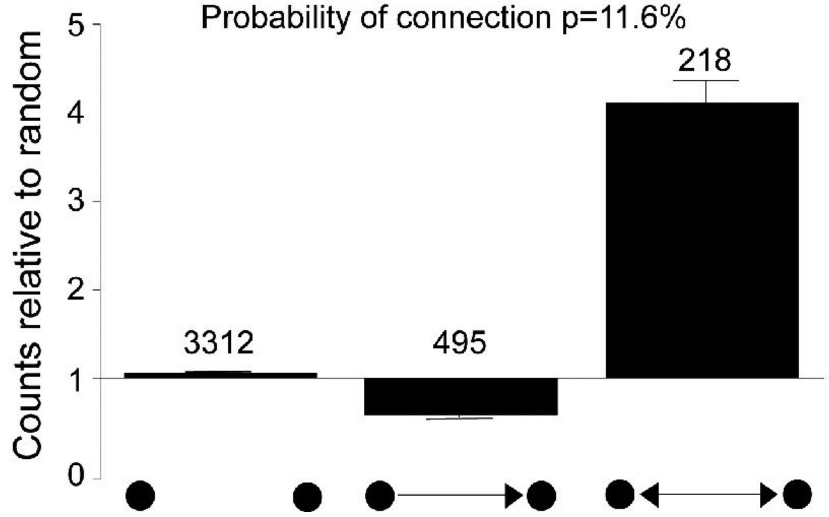
$$\Delta w_{ij}^- = F_-(w_{ij}) y_i(t_j^f)$$

Post-synaptic weight update

$$\Delta w_{ij}^+ = F_+(w_{ij}) x_j(t_i^f)$$

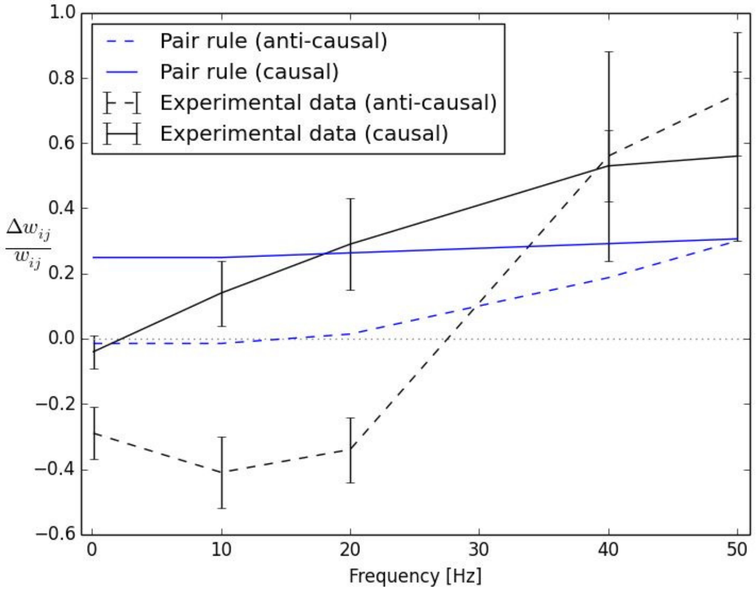


Limitations of pair-based STDP



Song, S., Sjöström, P. J., Reigl, M., Nelson, S., & Chklovskii, D. B. (2005). Highly nonrandom features of synaptic connectivity in local cortical circuits. PLoS Biology, 3(3), 0507–0519.

Limitations of pair-based STDP



Sjöström, P. J., Turrigiano, G. G., & Nelson, S. B. (2001). Rate, timing, and cooperativity jointly determine cortical synaptic plasticity. *Neuron*, *32*(6), 1149–64.

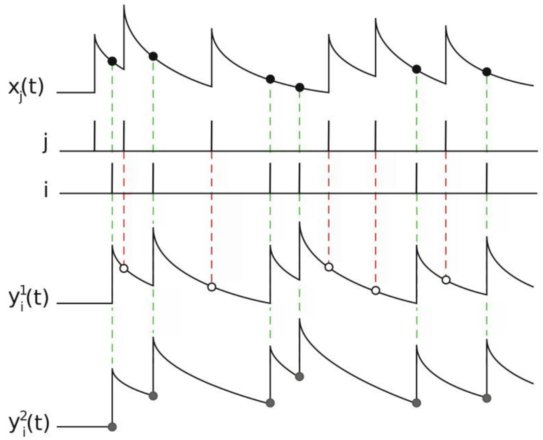
Triplet STDP

Slow post-synaptic trace

$$y_i^2(t) = \left(1 + y_i^2(t_i^f)\right) e^{-\frac{t-t_i^f}{\tau_y^2}}$$

Post-synaptic weight update

$$\Delta w_{ij}^+ = F_+(w_{ij}) x_j(t_i^f) y_i^2(t_i^{f-})$$



Pfister, J. P., & Gerstner, W. (2006). Triplets of spikes in a model of spike timing-dependent plasticity. The Journal of Neuroscience : The Official Journal of the Society for Neuroscience, 26(38), 9673–82.

SpiNNaker - Pair traces

timing_pair_impl.h
line 7

```
typedef int16_t post_trace_t;
```

timing_pair_impl.h
lines 46-49

```
static inline post_trace_t timing_get_initial_post_trace()
{
  return 0;
}
```


SpiNNaker - Triplet traces

timing_triplet_impl.h
line 7

```
typedef struct post_trace_t
{
  int16_t y1;
  int16_t y2;
} post_trace_t;
```

timing_triplet_impl.h
lines 46-49

```
static inline post_trace_t timing_get_initial_post_trace()
{
  return (post_trace_t){.y1 = 0, .y2 = 0};
}
```


SpiNNaker - Pair trace update

timing_pair_impl.h
lines 54-66

$$y_i(t) = 1 + y_i(t_i^f)e^{-\frac{t-t_i^f}{\tau_y}}$$

```
// Get time since last spike
uint32_t delta_time = time - last_time;

// Decay previous trace (y)
int32_t new_y = STDP_FIXED_MUL_16X16(last_trace,
  DECAY_TAU_Y(delta_time));

// Add energy caused by new spike to trace
new_y += STDP_FIXED_POINT_ONE;

log_debug("\tdelta_time=%d, y=%d\n", delta_time, new_y);

// Return new trace_value
return (post_trace_t)new_y;
```


SpiNNaker - Triplet trace update

timing_triplet_impl.h
lines 77-87

$$y_i^2(t) = \left(1 + y_i^2(t_i^f)\right) e^{-\frac{t-t_i^f}{\tau_y^2}}$$

```
// Y2 is sampled in timing_apply_post_spike BEFORE the spike
// Therefore, if this is the first spike, y2 must be zero
int32_t new_y2;
if(last_time == 0)
{
  new_y2 = 0;
}
// Otherwise, add energy of spike to last value and decay
else
{
  new_y2 = STDP_FIXED_MUL_16X16(
    last_trace.y2 + STDP_FIXED_POINT_ONE,
    DECAY_TAU_Y2(delta_time));
}
```

SpiNNaker - Pair weight update

timing_pair_impl.h
lines 136-150

$$\Delta w_{ij}^{+} = F_{+}(w_{ij}) x_j(t_i^f)$$

```
uint32_t delta_t = time - last_pre_time;

// If spikes are not co-incident
if (delta_t > 0)
{
  // Calculate x(time) = x(last_pre_time) * e^(-delta_t/tau_x)
  int32_t x = STDP_FIXED_MUL_16X16(last_pre_trace,
    DECAY_TAU_X(delta_t));

  log_debug("\t\t\tdelta_t=%u, x=%d\n",
    delta_t, x);

  // Apply potentiation to synapse state
  return weight_one_term_apply_potentiation(previous_state, x);
}
```

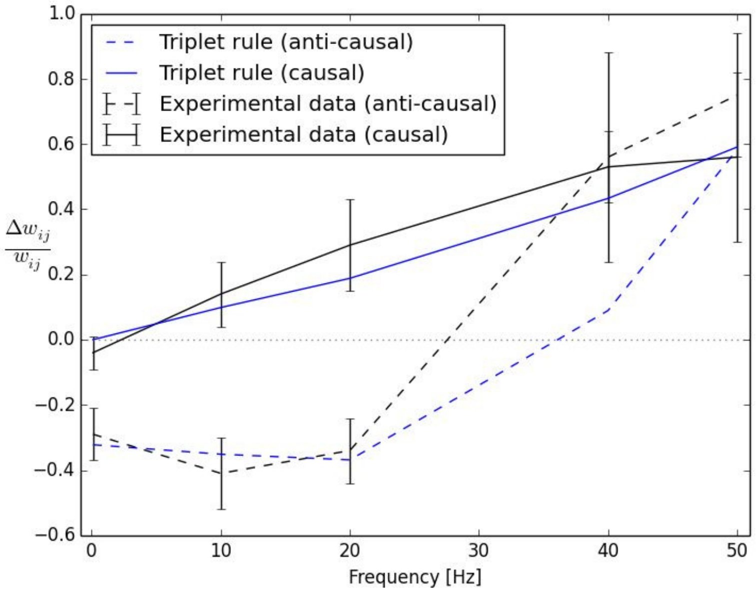

SpiNNaker - Triplet weight update

timing_triplet_impl.h
lines 165-179

$$\Delta w_{ij}^{+} = F_{+}(w_{ij}) x_j(t_i^f) y_i^2(t_i^{f-})$$

```
if (delta_t > 0)
{
  // Calculate x(time) = x(last_pre_time) * e^(-delta_t/tau_x)
  int32_t x = STDP_FIXED_MUL_16X16(last_pre_trace,
    DECAY_TAU_X(delta_t));

  // Multiply this by y2(time) to get triplet term
  int32_t x_y2 = STDP_FIXED_MUL_16X16(x, trace.y2);

  log_debug("\t\t\tdelta_t=%u, x=%d, y2=%d, x_y2=%d\n",
    delta_t, x, trace.y2, x_y2);

  // Apply potentiation to synapse state
  return weight_one_term_apply_potentiation(previous_state, x_y2);
}
```


Triplet model



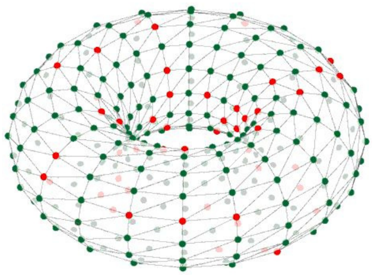
Thank you!

Any questions?

james.knight@manchester.ac.uk

MANCHESTER 1824

Graph Front End - Advanced



Alan Stokes, Andrew Rowley

SpiNNaker Workshop
September 2016

European Research Council
Human Brain Project
EPSRC
SpiNNaker

MANCHESTER 1824

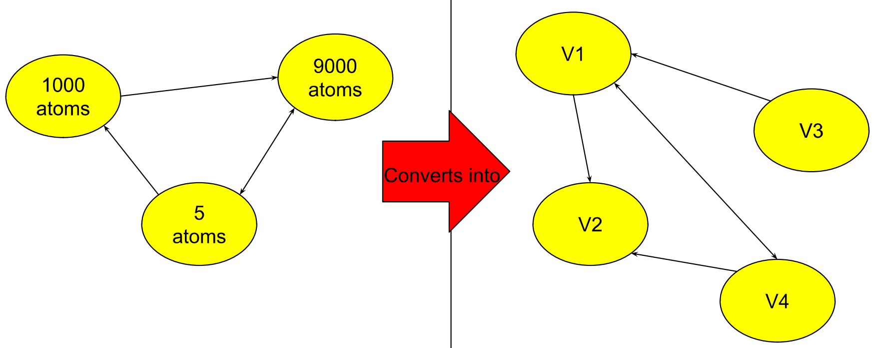
Contents

- Working with application graphs
- Buffered recordings
- Auto pause and resume
- Provenance data

MANCHESTER 1824

Supported graphs (PACMAN)

Application Graph | **Machine Graph**



Needs breaking down into core sized chunks

Already has a 1:1 ratio between vertices and core.

MANCHESTER 1824

Basic script to add application vertices into the graph

```
import spinnaker_graph_front_end as front_end

from spinnaker_graph_front_end.examples.Conways.conways_application_cell\
  import ConwayApplicationCell

# set up the front end and ask for the detected machines dimensions
front_end.setup()

front_end.add_application_vertex_instance(
     ConwayApplicationCell(800, "ConwayCells"))

# run the simulation for 5 seconds
front_end.run(5000)

# clean up the machine for the next application
front_end.stop()
```

Creating a new type of application vertex

```python
from pacman.model.graphs.application.impl.application_vertex  import ApplicationVertex
from pacman.model.resources.resource_container import ResourceContainer
from pacman.model.resources.cpu_cycles_per_tick_resource import CPUCyclesPerTickResource
from pacman.model.resources.dtcm_resource import DTCMResource
from pacman.model.resources.sdram_resource import SDRAMResource

class ConwayApplicationCell(ApplicationVertex):
    """ Represents a collection of cells within the 2D grid
    """
    def __init__(self, n_atoms, label):
        ApplicationVertex.__init__(self, label=label, max_atoms_per_core=200)
        self._n_atoms = n_atoms

    def get_resources_used_by_atoms(self, vertex_slice):
        resources = ResourceContainer(
            sdram=SDRAMResource(4 * vertex_slice.n_atoms),
            dtcm=DTCMResource(4 * vertex_slice.n_atoms),
            cpu_cycles=CPUCyclesPerTickResource(100 * vertex_slice.n_atoms)
        )
```


Creating a new type of application vertex

```python
def create_machine_vertex(
      self, vertex_slice, resources_required, label=None, constraints=None):

    # return a partitioned vertex that's designed to handle multiple atoms within it
    return ConwayMachineCell(
        label=label, resources_required=resources_required,
        constraints=constraints)

@property
def n_atoms(self):

    # return the atoms this vertex contains
    return self._n_atoms
```


Basic Script adding application edges

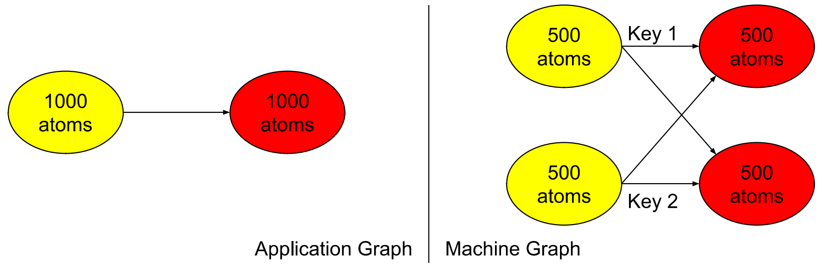
```python
import spinnaker_graph_front_end as front_end

# build and add application vertex
vertex = ConwayApplicationCell(800, "ConwayCells")
front_end.add_application_vertex_instance(vertex)

# build an application edge
front_end.add_application_edge_instance(
    ApplicationEdge(vertex, vertex), "State")

front_end.run(5000)

front_end.stop()
```

Partition id

Data generation

```python
...
def generate_application_data_specification(
      self,  spec, placement, graph_mapper, application_graph, machine_graph,
      routing_info, iptags,  reverse_iptags, machine_time_step, time_scale_factor):

# Reserve SDRAM space for memory areas:
spec.reserve_memory_region(
   region=0, size=constants.SYSTEM_BYTES_REQUIREMENT, label='system')
spec.reserve_memory_region(
   region=1, size=8, label="inputs")
...

# get slice of atoms for machine vertex
vertex_slice = graph_mapper.get_slice(placement.vertex)
```

Application vertex c code



Hints:

1. You need to be able to distinguish from the received key which atoms it effects on the core you are writing the data for
2. You need to execute your application c code for every atom on the core

Buffered Recordings

Problem

1. SDRAM is limited on the SpiNNaker machines.
2. Recording of data is more reliable on SDRAM than live transmissions.
3. Simulations run for long periods of time gathering data.

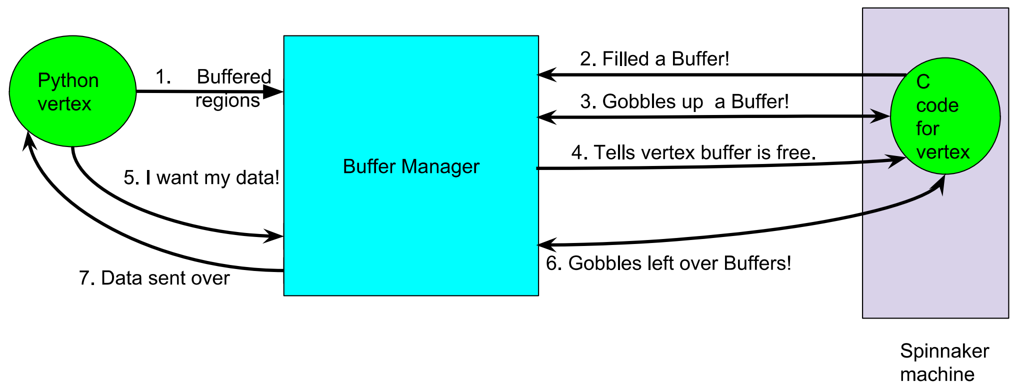
Buffered Recordings

Solution

1. Store data in small chunks called buffers
2. During simulation, or during a pause, extract the buffers

NOTE: This only works in tandem with the simulation.h and data_specification.h and python interfaces.

How does a extracted buffered data region work?



Buffered Recording - Python

```
class MyBufferedVertex(..., ReceiveBuffersToHostBasicImpl):

def __init__(...):
  ReceiveBuffersToHostBasicImpl.__init__(self)
  ...
```


Buffered Recording - Python

```
class MyBufferedVertex(..., ReceiveBuffersToHostBasicImpl):
  ...

  def generate_data_spec(...):
    ...
    spec.reserve_memory_region(
      region=2, size=self.get_recording_data_size(3), label="recording")
    ...
    spec.reserve_memory_region(
      region=6, size=self.get_buffer_state_region_size(3), label="state")
    ...

    ...
    self.reserve_buffer_regions(spec, 6, [4,5,7], [1000000, 1000000, 100000])
    ...
    spec.switch_write_focus(2)
    self.write_recording_data(spec, iptags, [1000000, 1000000, 100000], 16384)
    ...
```

Number of buffered regions

Extra region for storing buffered state

Buffered region ids

Allocated buffer sizes

IP Tags holder

Buffer size before request sent

Buffered Recording - C

```
static uint32_t recording_flags = 0;

void c_main(void) {
  ...
  address_t address = data_specification_get_data_address();
  address_t recording_region = data_specification_get_region(2, address);
  uint8_t *regions_to_record[] = {4,5,7};

  bool success = recording_initialize(
    3, regions_to_record, recording_region, 6, &recording_flags);

  ...
  simulation_run();
}
```

Buffered region ids (channels 0, 1 and 2)

Number of buffered regions

Extra region for storing buffered state

Buffered Recording - C

```
...
void timer_callback(uint unused0, uint unused1) {
  ...
  if ((infinite_run != TRUE) && ((time + 1) >= simulation_ticks)) {
    recording_finalize();
    ...
  }

if (recording_is_channel_enabled(recording_flags, 0)) {

    uint32_t data = 23;
    recording_record(0, &data, 4);
  }

  recording_do_timestep_update(time);
  ...
}
```

Recording channel number (= region 4)

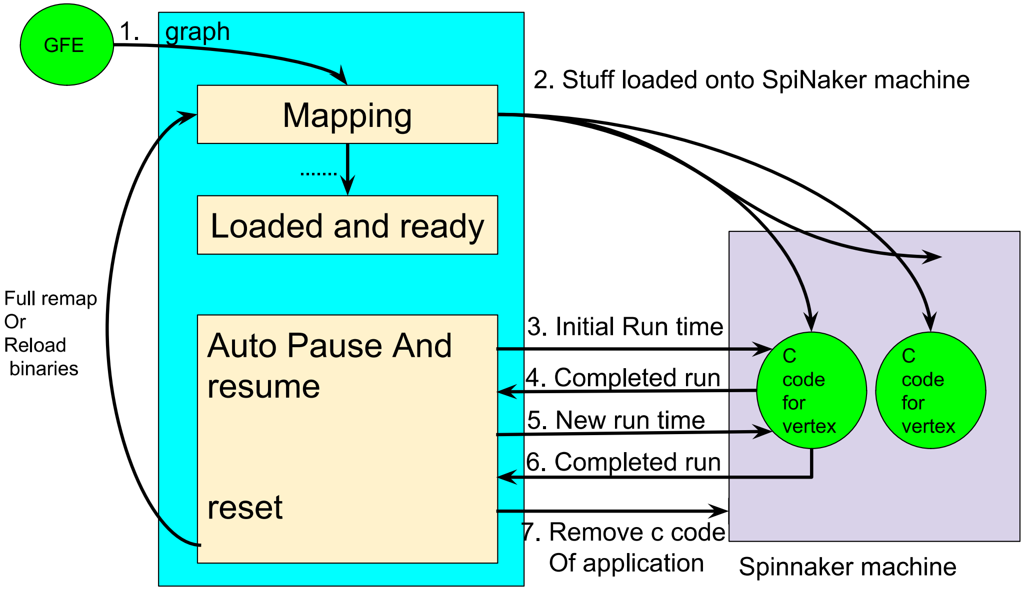
Pointer to data to record

Size of data to record in bytes

Auto pause and resume functionality

1. Provides the ability to run a simulation for multiple periods without remapping the application.
2. Provides the ability to extract buffers without affecting the running simulation.
3. Supports the ability to reset a simulation to the state at t=0.

How Auto Pause and Resume works.



Auto Pause and Resume - Python

```
class AbstractPopulationVertex(..., AbstractChangableAfterRun):
...
def __init__(......):
   AbstractChangableAfterRun.__init__(self)

   # bool for if state has changed.
   self._change_requires_mapping = True

@property
def requires_mapping(self):
   # determine if there are changes within which require a remapping
   return self._change_requires_mapping

def mark_no_changes(self):
   # restart the tracking of changes
   self._change_requires_mapping = False

def set_recording_spikes(self):
  self._change_requires_mapping = not self._spike_recorder.record
  self._spike_recorder.record = True
```


Auto Pause and Resume - C

```
...
void timer_callback(uint unused0, uint unused1) {
   ...

   if ((infinite_run != TRUE) && ((time + 1) >= simulation_ticks)) {
      simulation_exit();
      simulation_handle_pause_resume(resume_callback);
      ...
   }
}

void resume_callback() {

   // restart the recording just before resuming
   if (!initialise_recording()) {
      rt_error(RTE_SWERR);
   }
}
```

Provenance data gatherers

1. Data that can be used to prove 2 simulations are equivalent to each other.
2. Data that can also be used for debug purposes.
3. Is stored in XML and searched through for errors by the main tools.
4. Every vertex can provide its own provenance data.

Provenance example output

```
<provenance_data_items name="my_object">
  <provenance_data_items name="my_category">
    <provenance_data_item name="my_item">0</provenance_data_item>
    <provenance_data_item name="my_other_item">0</provenance_data_item>
  </provenance_data_items>
</provenance_data_items>

<provenance_data_items name="0_0_5_my_vertex">
  <provenance_data_items name="my_category">
    <provenance_data_item name="my_machine_value">0</provenance_data_item>
  </provenance_data_items>
</provenance_data_items>
```


Local Provenance Data - Python

```
class MyVertex(..., AbstractProvidesLocalProvenanceData):
    ...

    def get_local_provenance_data(self)
        self._data_items = list()

        # store data in a provenance data item
        self._data_items.append(
            ProvenanceDataItem(
                ["my_object" "my_category", "my_item"], my_value))
        self._data_items.append(
            ProvenanceDataItem(
                ["my_object", "my_category", "my_other_item"], my_other_value,
                report=(my_other_value > error_value),
                message="value {} was bigger than expected ({})".format(
                    my_value, error_value))
        ...

        # return provenance items
        return self._data_items
```

Hierarchy of categories and names used to group items in XML

debug arguments

Simulation Provenance Data - Python

```
class MyVertex(..., ProvidesProvenanceDataFromMachineImpl):
    ...
    def get_provenance_data_from_machine(self, transceiver, placement):
        provenance_data = self._read_provenance_data(transceiver, placement)

        # translate system specific provenance data items
        provenance_items = self._read_basic_provenance_items(
            provenance_data, placement)

        # translate application specific provenance data items
        provenance_data = self._get_remaining_provenance_data_items(
                provenance_data)
        my_value = provenance_data[0]
        label, x, y, p, names = self._get_placement_details(placement)

        # translate into provenance data items
        provenance_items.append(
            ProvenanceDataItem(
                self._add_names(names, ["my_category", "my_machine_value"]),
                my_value))

        return provenance_items
```

Simulation Provenance Data - Python

```
class MyVertex(..., ProvidesProvenanceDataFromMachineImpl):
    ...

    def __init__(self, …)
        ProvidesProveanceDataFromMachineImpl.__init__(self, 9, 1)
        ...

    def generate_data_spec(...):
        ...
        self.reserve_provenance_data_region(spec)
```

Provenance Region

Number of custom provenance data items

Simulation Provenance Data - C

```
static my_value = 0;

void c_main(void) {

    ...
    if (!simulation_initialise(
            system_region, APPLICATION_NAME_HASH,
            &timer_period, &simulation_ticks,
            &infinite_run, SDP,
            get_provenance_data,
            data_specification_get_region(9, address))) {
        log_error("Error in initialisation - exiting!");
        rt_error(RTE_SWERR);
    }
    …
}

void get_provenance_data(address_t provenance_data_address) {
    provenance_data_address[0] = my_value;
}
```

Provenance Region

Summary

1. Application graphs
2. Buffered recording
3. Auto pause and resume
4. Provenance data gathering

6th SpiNNaker Workshop

Day 5

September 9th 2016

Time	Session	Presenter
09:00	Lab time	
10:30	Coffee	
11:00	Lab time	
12:00	Lunch and close	

Manchester, UK