PyGeNN: A Python library for GPU-enhanced neural networks

James C Knight [1,*], **Anton Komissarov** [2], **Thomas Nowotny** [1]

[1]*Centre for Computational Neuroscience and Robotics, School of Engineering and Informatics, University of Sussex, Brighton, United Kingdom*
[2]**(TODO: ANTON'S AFFILIATINO)**

Correspondence*:
James C Knight
J.C.Knight@sussex.ac.uk

ABSTRACT

For full guidelines regarding your manuscript please refer to Author Guidelines.

As a primary goal, the abstract should render the general significance and conceptual advance of the work clearly accessible to a broad readership. References should not be cited in the abstract. Leave the Abstract empty if your article does not require one, please see Summary Table for details according to article type.

Keywords: GPU, high-performance computing, parallel computing, benchmarking, computational neuroscience, spiking neural networks, Python

1 INTRODUCTION

A wide range of spiking neural network (SNN) simulators are available, each with their own application domains. NEST (Gewaltig and Diesmann, 2007) is widely used for large-scale point neuron simulations on distributed computing systems; NEURON (Carnevale and Hines, 2006) and Arbor (Akar et al., 2019) specialise in the simulation of complex multi-compartmental models; NeuroKernel (Givon and Lazar, 2016) is focused on emulating fly brain circuits using Graphics Processing Units (GPUs); and CARLsim (Chou et al., 2018), ANNarchy (Vitay et al., 2015), NeuronGPU (Golosio et al., 2020) and GeNN (Yavuz et al., 2016) use GPUs to accelerate point neuron models. For performance reasons, many of these simulators are written in C++ and, especially amongst the older simulators, users describe their models either using a Domain-Specific Language (DSL) or directly in C++. For programming language purists, a DSL may be an elegant way of describing an SNN network model and, for simulator developers, not having to add bindings to another language is convenient. However, both choices act as a barrier to potential users. Therefore, with both the computational neuroscience and machine learning communities gradually coalescing towards a Python-based ecosystem with a wealth of mature libraries for scientific computing (Hunter, 2007; Van Der Walt et al., 2011; Millman and Aivazis, 2011), exposing spiking neural network simulators to Python seems a pragmatic choice. NEST (Eppler et al., 2009), NEURON (Hines et al., 2009) and CARLsim (Balaji et al., 2020) have all taken this route and now offer a Python interface. Furthermore, newer simulators such as Arbor and Brian2 (Stimberg et al., 2019) have been designed from the ground up with a Python interface.

While we have recently demonstrated some very competitive performance results (Knight and Nowotny, 2018, 2020) using our GeNN simulator (Yavuz2016), it has so far not been usable directly from Python.

29 GeNN can already be used as a backend for the Python-based Brian2 simulator (Stimberg et al., 2019). In
30 brief, the Brian2GeNN interface (Stimberg et al., 2020) modifies the C++ backend "cpp_standalone" of
31 Brian 2 to generate C++ input files for GeNN. As for cpp_standalone, initialisation of simulations is mostly
32 done in C++ on the CPU and recording data is saved into binary files and re-imported into Python using
33 Brian 2's native methods. While Brian2GeNN allows Brian2 users to harness the performance benefits that
34 GeNN provides, it is not possible to expose all of GeNN's unique features to Python through the Brian2
35 API. Specifically, GeNN not only allows users to easily define their own neuron and synapse models but,
36 also 'snippets' for offloading the potentially costly initialisation of model parameters and connectivity
37 onto the GPU. Additionally, GeNN provides a lot of freedom for users to integrate their own code into the
38 simulation loop. In this paper we describe the implementation of PyGeNN – a Python package which aims
39 to expose the full range of GeNN functionality with minimal performance overheads. While implementing
40 new neuron and synapse models in the majority of other GPU simulators requires extending the underling
41 C++ code, using PyGeNN, models can be defined directly from Python. Finally, we demonstrate the
42 flexibility and performance of PyGeNN in two scenarios where minimising performance overheads is
43 particularly critical.

44 • In a simulation of a large, highly-connected model of a cortical microcircuit (Potjans and Diesmann,
45   2014) with small simulation timesteps. Here the cost of copying spike data off the GPU from a large
46   number of neurons every timestep can become a bottleneck.

47 • In a simulation of a much smaller model of Pavlovian conditioning (Izhikevich, 2007) where learning
48   occurs over $1\,\text{h}$ of biological time and stimuli are delivered – following a complex scheme – throughout
49   the simulation. Here any overheads are multiplied by a large number of timesteps and copying stimuli
50   to the GPU can become a bottleneck.

51 Using the facilities provided by PyGeNN, we show that both scenarios can be simulated from Python with
52 only minimal overheads over a pure C++ implementation.

2 MATERIALS AND METHODS

2.1 GeNN

54 GeNN (Yavuz et al., 2016) is a library for generating CUDA code for the simulation of spiking neural
55 network models. GeNN handles much of the complexity of using CUDA directly as well as automatically
56 performing device-specific optimizations so as to to maximize performance.

57 GeNN consists of a main library – implementing the API used to define models as well as the generic
58 parts of the code generator – and an additional library for each backend (currently there is a reference C++
59 backend for generating CPU code and a CUDA backend. An OpenCL backend is under development).
60 Users describe their model by implementing a `modelDefinition` function within a C++ file. For example,
61 a model consisting of 4 Izhikevich neurons with heterogeneous parameters, driven by a constant input
62 current might be defined as follows:

```
63  void modelDefinition(ModelSpec &model)
64  {
65      model.setDT(0.1);
66      model.setName("izhikevich");
67
68      NeuronModels::IzhikevichVariable::VarValues popInit(
69          -65.0, -20.0, uninitialisedVar(), uninitialisedVar(),
```

```
70          uninitialisedVar(), uninitialisedVar());
71
72      model.addNeuronPopulation<NeuronModels::IzhikevichVariable>(
73          "Pop", 4, {}, popInit);
74
75      model.addCurrentSource<CurrentSourceModels::DC>(
76          "CS", "Pop", {10.0}, {});
77  }
```

The *genn-buildmodel* command line tool is then used to compile this file; link it against the main GeNN
library and the desired backend library; and finally run the resultant executable to generate the source code
required to build a simulation dynamic library (a .dll file on Windows or a .so file on Linux and Mac).
This dynamic library can then either be statically linked against a simulation loop provided by the user or
dynamically loaded by the user's simulation code. To demonstrate this latter approach, this example uses
the `SharedLibraryModel` helper class supplied with GeNN to dynamically load the previously defined
model, initialise the heterogenous neuron parameters and print each neuron's membrane voltage every
timestep:

```
86  #include "sharedLibraryModel.h"
87
88  int main()
89  {
90      SharedLibraryModel<float> model("./", "izhikevich");
91      model.allocateMem();
92      model.initialize();
93      float *aPop = model.getScalar<float>("a");
94      float *bPop = model.getScalar<float>("b");
95      float *cPop = model.getScalar<float>("c");
96      float *dPop = model.getScalar<float>("d");
97      aPop[0] = 0.02; bPop[0] = 0.2;  cPop[0] = -65.0;   dPop[0] = 8.0;  // RS
98      aPop[1] = 0.1;  bPop[1] = 0.2;  cPop[1] = -65.0;   dPop[1] = 2.0;  // FS
99      aPop[2] = 0.02; bPop[2] = 0.2;  cPop[2] = -50.0;   dPop[2] = 2.0;  // CH
100     aPop[3] = 0.02; bPop[3] = 0.2;  cPop[3] = -55.0;   dPop[3] = 4.0;  // IB
101     model.initializeSparse();
102
103     float *vPop = model.getScalar<float>("VPop");
104     while(model.getTime() < 200.0f) {
105         model.stepTime();
106         model.pullVarFromDevice("Pop", "V");
107         printf("%f, %f, %f, %f, %f\n", t, VPop[0], VPop[1], VPop[2], VPop[3]);
108     }
109     return EXIT_SUCCESS;
110 }
```

2.2 SWIG

In order to use GeNN from Python, both the model creation API and the `SharedLibraryModel`
functionality need to be 'wrapped' so they can be called from Python. While this is possible using
the API built into Python itself, a wrapper function would need to be manually implemented for each

115 GeNN function to be exposed which would result in a lot of maintenance overhead. Instead, we chose
116 to use SWIG (Beazley, 1996) to automatically generate wrapper functions and classes. SWIG generates
117 Python modules based on special interface files which can directly include C++ code as well as special
118 'directives' which control SWIG, for instance:

```
119 %module(package="package") package
120 %include "test.h"
```

121 where the `%module` directive sets the name of the generated module and the package it will be located in
122 and the `%include` directive parses and automatically generates wrapper functions for a C++ header file.
123 We use SWIG in this manner to wrap both the model building and `SharedLibraryModel` APIs described
124 in section 2.1. However, key parts of GeNN's API such as the `ModelSpec::addNeuronPopulation` method
125 employed in section 2.1, rely on C++ templates which are not directly translatable to Python. Instead, valid
126 template instantiations need to be given a unique name in Python using the `%template` SWIG directive:

```
127 %template(addNeuronPopulationLIF) ModelSpec::addNeuronPopulation<NeuronModels::LIF>;
```

128 Having to manually add these directives whenever a model is added to GeNN would be exactly the sort of
129 maintenance overhead we were trying to avoid by using SWIG. Instead, when building the Python wrapper,
130 we search the GeNN header files for the macros used to declare models in C++ and automatically generate
131 SWIG `%template` directives.

132   As previously discussed, a key feature of GeNN is the ease with which it allows users to define their
133 own neuron and synapse models as well as 'snippets' defining how variables and connectivity should be
134 initialised. Beneath the syntactic sugar described in our previous work (Knight and Nowotny, 2018), new
135 models can be defined in C++ by defining a new class derived from, for example, the `NeuronModels::Base`
136 class. The ability to extend this system to Python was a key requirement of PyGeNN and, by using SWIG
137 'directors', C++ classes can be made inheritable from Python using a single SWIG directive:

```
138 %feature("director") NeuronModels::Base;
```

139   2.3 PyGeNN

140   While GeNN *could* be used from Python via the wrapper generated using the techniques described in the
141 previous section, the resultant code would be unpleasant to use directly. For example, rather than being
142 able to specify neuron parameters using a native Python data structure such as a list or dictionary, one
143 would have to use a wrapped type such as `DoubleVector([0.25, 10.0, 0.0, 0.0, 20.0, 2.0, 0.5])`. To provide
144 a more user-friendly and pythonic interface, we have built PyGeNN on top of the wrapper generated by
145 SWIG. PyGeNN combines the separate model building and simulation stages of building a GeNN model
146 in C++ into a single API, likely to be more familiar to users of existing Python-based model description
147 languages such as PyNEST (Eppler et al., 2009) or PyNN (Davison et al., 2008). By combining the two
148 stages together, PyGeNN can provide a unified dictionary-based API for initialising homogeneous and
149 heterogeneous parameters as shown in this re-implementation of the previous example:

```
150 from pygenn import genn_wrapper, genn_model
151
152 model = genn_model.GeNNModel("float", "izhikevich")
153 model.dT = 0.1
154
155 izk_init = {"V": -65.0,
```

```
156            "U": −20.0,
157            "a": [0.02,      0.1,     0.02,    0.02],
158            "b": [0.2,       0.2,     0.2,     0.2],
159            "c": [−65.0,     −65.0,   −50.0,   −55.0],
160            "d": [8.0,       2.0,     2.0,     4.0]}
161
162 pop = model.add_neuron_population("Pop", 4, "IzhikevichVariable", {}, izk_init)
163 model.add_current_source("CS", "DC", "Pop", {"amp": 10.0}, {})
164
165 model.build()
166 model.load()
167
168 v = pop.vars["V"].view
169 while model.t < 200.0:
170     model.step_time()
171     model.pull_state_from_device("Pop")
172     print("%t, %f, %f, %f, %f" % (model.t, v[0], v[1], v[2], v[3]))
```

Initialisation of variables with homogeneous values – such as the neurons' membrane potential – is performed by GeNN and those with heterogeneous values – such as the a, b and c parameters – are initialised by PyGeNN when the model is loaded. While the PyGeNN API is more pythonic and, hopefully, more user-friendly than the C++ interface, it still provides users with the same low-level control over the simulation. Furthermore, by using SWIG's numpy (Van Der Walt et al., 2011) interface, the host memory allocated by GeNN can be accessed directly from Python using the pop.**vars**["V"].view syntax meaning that no potentially expensive additional copying of data is required.

As illustrated in the previously-defined model, for convenience, PyGeNN allows users to access GeNN's built-in models. However, one of PyGeNN's most powerful features is that it enables users to easily define their own neuron and synapse models from within Python. For example, an Izhikevich neuron model (Izhikevich, 2003) can be defined using the create_custom_neuron_class helper function which provides some syntactic sugar over the model class inheritance described in the previous section:

```
185 izk_model = genn_model.create_custom_neuron_class(
186     "izk",
187
188     param_names=["a", "b", "c", "d"],
189     var_name_types=[("V", "scalar"), ("U", "scalar")],
190
191     sim_code=
192         """
193         $(V)+=0.5*(0.04*$(V)*$(V)+5.0*$(V)+140.0−$(U)+$(Isyn))*DT;
194         $(V)+=0.5*(0.04*$(V)*$(V)+5.0*$(V)+140.0−$(U)+$(Isyn))*DT;
195         $(U)+=$(a)*($(b)*$(V)−$(U))*DT;
196         """,
197     threshold_condition_code="$(V) >= 30.0",
198     reset_code=
199         """
200         $(V)=$(c);
201         $(U)+=$(d);
202         """)
```
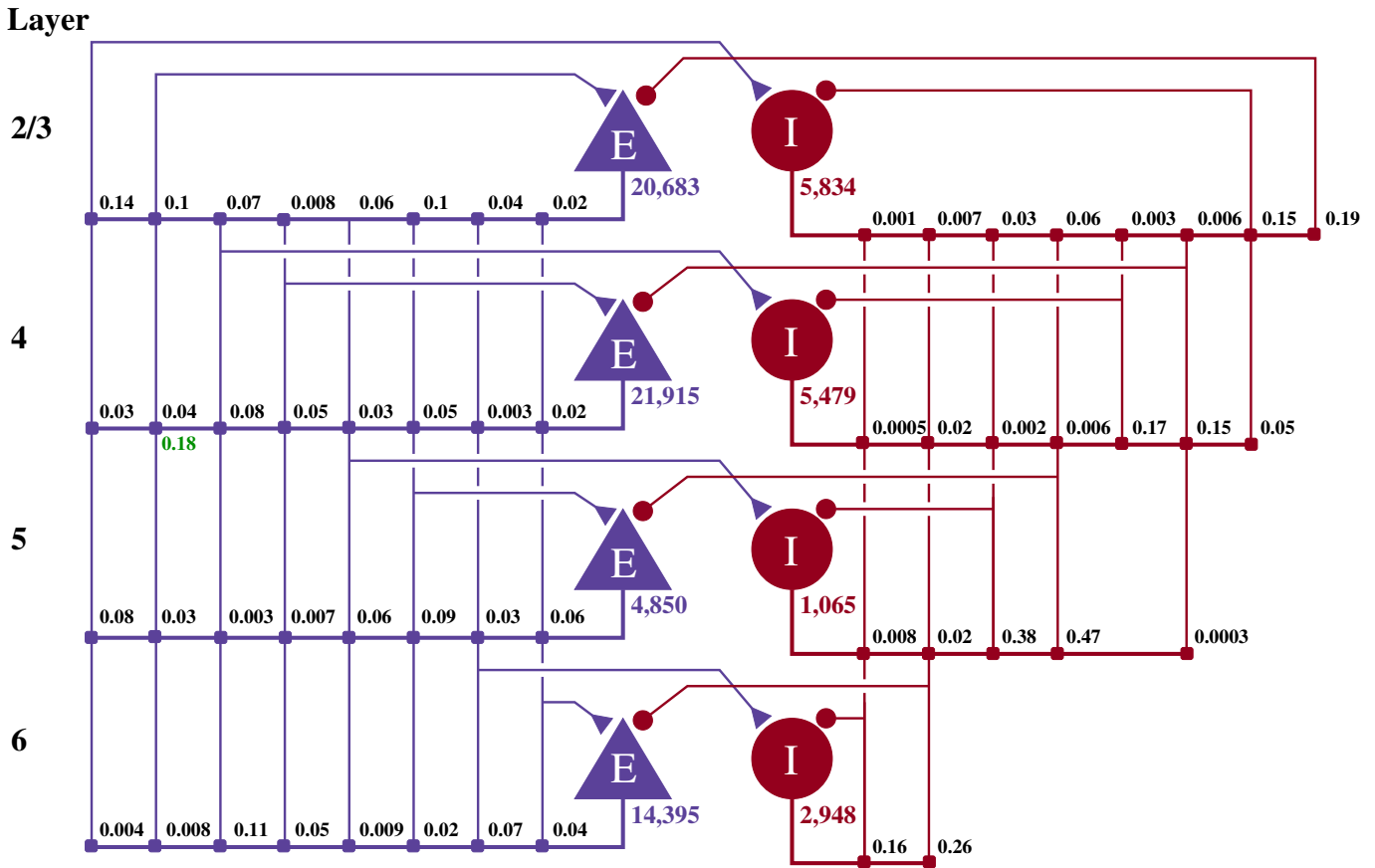
Layer

2/3

4

5

6

E 20,683   I 5,834

0.14 0.1 0.07 0.008 0.06 0.1 0.04 0.02   0.001 0.007 0.03 0.06 0.003 0.006 0.15 0.19

E 21,915   I 5,479

0.03 0.04 0.08 0.05 0.03 0.05 0.003 0.02   0.0005 0.02 0.002 0.006 0.17 0.15 0.05
0.18

E 4,850   I 1,065

0.08 0.03 0.003 0.007 0.06 0.09 0.03 0.06   0.008 0.02 0.38 0.47 0.0003

E 14,395   I 2,948

0.004 0.008 0.11 0.05 0.009 0.02 0.07 0.04   0.16 0.26

Figure 1. Illustration of the microcircuit model. Blue triangles represent excitatory populations, red circles represent inhibitory populations and the numbers beneath each symbol shows the number of neurons in each population. Connection probabilities are shown in small bold numbers at the appropriate point in the connection matrix. All excitatory synaptic weights are normally distributed with a mean of $0.0878\,\text{nA}$ (unless otherwise indicated in green) and a standard deviation of $0.008\,78\,\text{nA}$. All inhibitory synaptic weights are normally distributed with a mean of $0.3512\,\text{nA}$ and a standard deviation of $0.035\,12\,\text{nA}$.

203 The `param_names` list defines the real-valued parameters that are constant across the whole population of
204 neurons and the `var_name_types` list defines the model state variables and their type (the `scalar` type is
205 an alias for single or double-precision floating point, depending on the precision passed to the `GeNNModel`
206 constructor). The behaviour of the model can then be defined using a number of code strings. Unlike
207 in tools like Brian 2 (Stimberg et al., 2019), these code strings are specified in a C-like language rather
208 than in terms of differential equations. This allows expert users to choose their own solver for models
209 described in terms of differential equations and to programatically define models such as spike sources.
210 For example, in our example model, we chose to implement this neuron using the idiomatic forward Euler
211 integration scheme employed by Izhikevich (2003). Finally, the `threshold_condition_code` expression
212 defines *when* the neuron will spike whereas the `reset_code` code string defines how the state variables
213 should be reset after a spike.

2.4 Spike recording system

214

215   Internally, GeNN stores the spikes emitted by a neuron population during one simulation timestep in an
216 array containing the indices of the neurons that spiked alongside a counter of how many spikes have been
217 emitted. Previously, recording spikes in GeNN was very similar to the recording of voltages shown in the

218 previous example code – the array of neuron indices was simply copied from the GPU to the CPU every
219 timestep. However, especially when simulating models with a small simulation timestep, such frequent
220 synchronization between the CPU and GPU is costly – especially if a higher-level language such as Python
221 is involved. Furthermore, biological neurons typically spike at a low rate (in the cortex, the average firing
222 rate is only around $3\,\mathrm{Hz}$ (Buzsáki and Mizuseki, 2014)) meaning that the amount of spike data transferred
223 every timestep is typically very small. To address both of these sources of inefficiency, we have added a
224 new data structure to GeNN which stores spike data for many timesteps on device. To reduce the memory
225 required for this data structure and to make its size independent of neural activity, the spikes emitted by a
226 population of N neurons in a single simulation timestep are stored in a $N\mathrm{bit}$ bitfield where a '1' represents
227 a spike and a '0' the absence of one. Spiking data over multiple timesteps is then represented by bitfields
228 stored in a circular buffer. Using this approach, even the spiking output of relatively large models, running
229 for many timesteps can be stored in a small amount of memory. For example, the spiking output of a
230 model with 100×10^3 neurons running for 10×10^3 simulation timesteps, required less than $120\,\mathrm{MB}$ – a
231 small fraction of the memory on a modern GPU. While efficiently handling spikes stored in a bitfield is a
232 little trickier than working with a list of neuron indices, GeNN provides an efficient C++ helper function
233 for saving the spikes stored in a bitfield to a text file and a numpy-based method for decoding them in
234 PyGeNN.

235 ## 2.5 Cortical microcircuit model

Potjans and Diesmann (2014) developed a cortical microcircuit model of $1\,\mathrm{mm}^3$ of early-sensory cortex. The model consists of 77 169 LIF neurons, divided into separate populations representing the excitatory and inhibitory population in each of 4 cortical layers (2/3, 4, 5 and 6) as illustrated by figure 2. The membrane voltage V_i of each neuron i is modelled as

$$\tau_{\mathrm{m}}\frac{dV_i}{dt} = (V_{\mathrm{rest}} - V_i) + R_{\mathrm{m}}(I_{\mathrm{syn}_i} + I_{\mathrm{ext}_i}), \tag{1}$$

where $\tau_{\mathrm{m}} = 10\,\mathrm{ms}$ and $R_{\mathrm{m}} = 40\,\mathrm{M\Omega}$ represent the time constant and resistance of the neuron's cell membrane, $V_{\mathrm{rest}} = -65\,\mathrm{mV}$ defines the resting potential, I_{syn_i} represents the synaptic input current and I_{ext_i} represents an external input current. When the membrane voltage crosses a threshold $V_{\mathrm{th}} = -50\,\mathrm{mV}$ a spike is emitted, the membrane voltage is reset to V_{rest} and updating of V is suspended for a refractory period $\tau_{\mathrm{ref}} = 2\,\mathrm{ms}$. Neurons in each population are connected randomly with numbers of synapses derived from an extensive review of the anatomical literature. These synapses are current-based, i.e. presynaptic spikes lead to exponentially-decaying input currents I_{syn_i}

$$\tau_{\mathrm{syn}}\frac{dI_{\mathrm{syn}_i}}{dt} = -I_{\mathrm{syn}_i} + \sum_{i=0}^{n} w_{ij} \sum_{t_j} \delta(t - t_j), \tag{2}$$

where $\tau_{\mathrm{syn}} = 0.5\,\mathrm{ms}$ represents the synaptic time constant and t_j are the arrival times of incoming spikes from n presynaptic neurons. Within each synaptic projection, all synaptic strengths and transmission delays are normally distributed using the parameters presented in Potjans and Diesmann (2014, table 5) and, in total, the model has approximately 0.3×10^9 synapses. As well as receiving synaptic input, each neuron in the network also receives an independent Poisson input current, representing input from neighbouring not
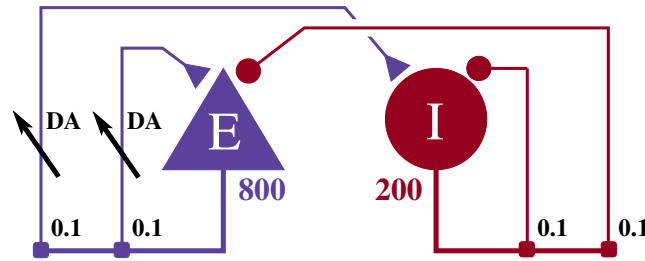
Figure 2. Illustration of the balanced random network model. The blue triangle represents the excitatory population, the red circle represents the inhibitory population, and the numbers beneath each symbol show the number of neurons in each population. Connection probabilities are shown in small bold numbers at the appropriate point in the connection matrix. All excitatory synaptic weights are plastic and initialised to 1 and all inhibitory synaptic weights are initialised to -1.

explicitly modelled cortical regions. The Poisson input is delivered to each neuron via I_{ext_i} with

$$\tau_{\text{syn}}\frac{dI_{\text{ext}_i}}{dt} = -I_{\text{ext}_i} + J\text{Poisson}(\nu_{\text{ext}}\Delta t), \tag{3}$$

236   where $\tau_{\text{syn}} = 0.5\,\text{ms}$, ν_{ext} represents the mean input rate and J represents the weight. The ordinary
237   differential equations 1, 2 and 3 are solved with an exponential Euler algorithm. For a full description of the
238   model parameters, please refer to Potjans and Diesmann (2014, tables 4 and 5) and for a description of the
239   strategies used by GeNN to parallelise the initialisation and subsequent simulation of this network, please
240   refer to Knight and Nowotny (2018, section 2.3). This model requires simulation using a relatively small
241   timestep of $0.1\,\text{ms}$, making the overheads of copying spikes from the GPU every timestep particularly
242   problematic.

243   ## 2.6   Pavlovian conditioning model

244   The cortical microcircuit model described in the previous section is ideal for exploring the performance
245   of short simulations of relatively large models. However, the performance of longer simulations
246   of smaller models is equally vital.**(TODO: DETERMINE E.G. PERCENTAGE MODELS E.G. ON**
247   **OPENSOURCEBRAIN WHICH ARE SMALL)**. Such models can be particularly troublesome for GPU
248   simulation as, not only might they not offer enough parallelism to fully occupy the device but, each
249   timestep can be simulated so quickly that the overheads of launching kernels etc can dominate. Additional
250   overheads can be incurred when models require injecting external stimuli throughout the simulation. Longer
251   simulations are particularly useful when exploring synaptic plasticity so, to explore the performance
252   of PyGeNN in this scenario, we simulate a model of Pavlovian conditioning using a three-factor
253   Spike-Timing-Dependent Plasticity (STDP) learning rule (Izhikevich, 2007).

254   ### 2.6.1   Neuron model

   This model consists of an 800 neuron excitatory population and a 200 neuron inhibitory population, within
which, each neuron i is modelled using the Izhikevich model (Izhikevich, 2003) whose dimensionless

membrane voltage V_i and adaption variables U_i evolve such that:

$$\frac{dV_i}{dt} = 0.04V_i^2 + 5V_i + 140 - U_i + I_{\text{syn}_i} + I_{\text{ext}_i} \tag{4}$$

$$\frac{dU_i}{dt} = a(bV_i - U_i) \tag{5}$$

When the membrane voltage rises above 30, a spike is emitted and V_i is reset to c and d is added to U_i. Excitatory neurons use the regular-spiking parameters (Izhikevich, 2003) where $a = 0.02$, $b = 0.2$, $c = -65.0$, $d = 8.0$ and inhibitory neurons use the fast-spiking parameters (Izhikevich, 2003) where $a = 0.1$, $b = 0.2$, $c = -65.0$, $d = 2.0$. Again, I_{syn_i} represents the synaptic input current and I_{ext_i} represents an external input current. While there are numerous ways to solve equations 4 and 5 (Humphries and Gurney, 2007; Hopkins and Furber, 2015; Pauli et al., 2018), we chose to use the forward Euler integration scheme employed by Izhikevich (2003). Under this scheme, equation 4 is first integrated for two $0.5\,\text{ms}$ timesteps and then, based on the updated value of V_i, equation 5 is integrated for a single $1\,\text{ms}$ timestep.

2.6.2 Synapse models

The excitatory and inhibitory neural populations are connected recurrently, as shown in figure 2, with instantaneous current-based synapses:

$$I_{\text{syn}_i}(t) = \sum_{i=0}^{n} w_{ij} \sum_{t_j} \delta(t - t_j), \tag{6}$$

where t_j are the arrival times of incoming spikes from n presynaptic neurons. Inhibitory synapses are static with $w_{ij} = -1.0$ and excitatory synapses are plastic. Each plastic synapse has an eligibility trace C_{ij} as well as a synaptic weight w_{ij} and these evolve according to a three-factor STDP learning rule (Izhikevich, 2007):

$$\frac{dC_{ij}}{dt} = -\frac{C_{ij}}{\tau_c} + \text{STDP}(\Delta t)\delta(t - t_{\text{pre/post}}) \tag{7}$$

$$\frac{dw_{ij}}{dt} = -C_{ij}D_j \tag{8}$$

where $\tau_c = 1000\,\text{ms}$ represents the decay time constant of the eligibility trace and $STDP(\Delta t)$ describes the magnitude of changes made to the eligibility trace based on the relative timing of a pair of pre and postsynaptic spikes with temporal difference $\Delta t = t_{post} - t_{pre}$. These changes are only applied to the trace at the times of pre and postsynaptic spikes as indicated by the Dirac delta function $\delta(t - t_{\text{pre/post}})$. Here, a double exponential STDP kernel is employed such that:

$$\text{STDP}(\Delta t) = \begin{cases} A_+ \exp\left(-\frac{\Delta t}{\tau_+}\right) & \text{if } \Delta t > 0 \\ A_- \exp\left(\frac{\Delta t}{\tau_-}\right) & \text{if } \Delta t < 0 \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

where the time constant of the STDP window $\tau_+ = \tau_- = 20\,\text{ms}$ and the strength of potentiation and depression are $A_+ = 0.1$ and $A_- = 0.15$ respectively. Finally, each excitatory neuron has an additional

variable D_j which describes extracellular dopamine concentration:

$$\frac{D_j}{t} = -\frac{D_j}{\tau_d} + \text{DA}(t) \tag{10}$$

264 where $\tau_d = 200\,\text{ms}$ represents the time constant of dopamine uptake and $\text{DA}(t)$ the dopamine input over
265 time.

2.6.3   PyGeNN implementation of three-factor STDP

267   The first step in implementing this learning rule in PyGeNN is to implement the STDP updates and decay
268 of C_{ij}. First, we create a new 'weight update model' with the learning rules parameters and the w_{ij} and
269 C_{ij} state variables:

```
izhikevich_stdp_model = create_custom_weight_update_class(
    "izhikevich_stdp",

    param_names=["tauPlus",  "tauMinus",
                 "tauC", "aPlus", "aMinus"],
    var_name_types=[("w", "scalar"), ("c", "scalar")],
```

276 We then instruct GeNN to record the times of current and previous pre and postsynaptic spikes. **(TODO:**
277 **IMPROVE SENTENCE)** The current spike time will equal the current time if a spike of this sort is being
278 processed in the current timestep whereas the previous spike time only tracks spikes which have occur
279 *before* the current timestep:

```
    is_pre_spike_time_required=True,
    is_post_spike_time_required=True,

    is_prev_pre_spike_time_required=True,
    is_prev_post_spike_time_required=True,
```

285 Next we define the 'sim code' which is called whenever presynaptic spikes arrive at the synapse. This code
286 first implements equation 6 – adding the synaptic weight (w_{ij}) to the postsynaptic neuron's input (I_{syn_i})
287 using the `$(addToInSyn,x)` function.

```
    sim_code=
        """
        $( addToInSyn ,  $(w));
```

291 Now we need to calculate the time that has elapsed since the last update of C_{ij} using the spike times we
292 previously requested that GeNN record. Within a timestep, GeNN processes presynaptic spikes before
293 postsynaptic spikes so the time of the last update to C_{ij} will be the latest time either type of spike was
294 processed in previous timesteps:

```
        const scalar tc = fmax($(prev_sT_pre),
                               $(prev_sT_post));
```

297 Using this time, we can now calculate how much to decay C_{ij} following equation 7:

```
        const scalar tagDecay = exp(-($(t) - tc) / $(tauC));
        scalar newTag = $(c) * tagDecay;
```

300 To complete the 'sim code' we calculate the depression case of equation 9 (here we use the *current*
301 postsynaptic spike time as, if a postsynaptic and presynaptic spike occur in the same timestep, there should
302 be no update).

```
303        const scalar dt = $(t) − $(sT_post);
304        if (dt > 0) {
305            newTag −= ($(aMinus) * exp(−dt / $(tauMinus)));
306        }
307        $(c) = newTag;
308        """,
```

309 Finally we define the 'learn post code' which is called whenever a postsynaptic spike arrives at the synapse.
310 Other than implementing the potentiation case of equation 9 and using the *current* presynaptic spike time
311 when calculating the time since the last update of C_{ij} – in order to correctly handle presynaptic updates
312 made in the same timestep – this code is very similar to the sim code:

```
313    learn_post_code=
314        """
315        const scalar tc = fmax($(sT_pre),
316                               $(prev_sT_post));
317
318        const scalar tagDecay = exp(−($(t) − tc) / $(tauC));
319        scalar newTag = $(c) * tagDecay;
320
321        const scalar dt = $(t) − $(sT_pre);
322        if (dt > 0) {
323            newTag += ($(aPlus) * exp(−dt / $(tauPlus)));
324        }
325        $(c) = newTag;
326        """)
```

327 Adding the synaptic weight w_{ij} update described by equation 8 requires two components. In addition to pre
328 and postsynaptic spikes, the weight update model needs to receive events whenever dopamine is injected
329 via DA. **(TODO: IMPROVE SENTANCE)** GeNN supports such events via the 'spike-like event' system
330 which allows events to be triggered based on a condition applied to the presynaptic neuron. In this case,
331 this condition is simply used to check an `injectDopamine` flag set by the dopamine injection logic in our
332 presynaptic neuron model:

```
333    event_threshold_condition_code="injectDopamine",
```

334 In order to extend our event-driven update of C_{ij} to include these events we need to instruct GeNN to
335 record the times at which they occur:

```
336    is_pre_spike_event_time_required=True,
337    is_prev_pre_spike_event_time_required=True,
```

338 The spike-like events can now be handled using an 'event code' string:

```
339    event_code=
340        """
341        const scalar tc = fmax($(sT_pre), fmax($(prev_sT_post), $(prev_seT_pre)));
```

```
342          const scalar tagDecay = exp(-($(t) - tc) / $(tauC));
343          $(c) *= tagDecay;
344          """,
```

After updating the previously defined calculations of `tc` in the sim code and learn post code to also include the times of spike-like events, all that remains is to update w_{ij}. Mikaitis et al. (2018) showed how equation 8 could be integrated algebraically, allowing w_{ij} to be updated in an event-driven manner with:

$$\Delta w_{ij} = \frac{C(t_c^{last})D(t_d^{last})}{-\left(\frac{1}{\tau_c} + \frac{1}{\tau_d}\right)} \left(e^{-\frac{t-t_c^{last}}{\tau_c}} e^{-\frac{t-t_d^{last}}{\tau_d}} - e^{-\frac{t_w^{last}-t_c^{last}}{\tau_c}} e^{-\frac{t_w^{last}-t_d^{last}}{\tau_d}} \right) \tag{11}$$

where t_c^{last}, t_w^{last} and t_d^{last} represent the last times at which C_{ij}, W_{ij} and D_j respectively were updated. Because we will always update w_{ij} and C_{ij} together when presynaptic, postsynaptic and spike-like events occur, $t_c^{last} = t_w^{last}$ and equation 12 can be simplified to:

$$\Delta w_{ij} = \frac{C(t_c^{last})D(t_d^{last})}{-\left(\frac{1}{\tau_c} + \frac{1}{\tau_d}\right)} \left(e^{-\frac{t-t_c^{last}}{\tau_c}} e^{-\frac{t-t_d^{last}}{\tau_d}} - e^{-\frac{t_c^{last}-t_d^{last}}{\tau_d}} \right) \tag{12}$$

and this update can now be added to each of our three event handling code strings to complete the implementation of the learning rule.

2.6.4 PyGeNN implementation of Pavlovian conditioning experiment

To perform the Pavlovian conditioning experiment using this model, we chose 100 random groups of 50 neurons (each representing stimuli $S_1...S_{100}$) are chosen from amongst the two neural populations. Stimuli are presented to the network in a random order, separated by intervals sampled from $U(100, 300)$ms. The neurons associated with an active stimulus are stimulated for a single $1\,$ms simulation timestep with a current of $40.0\,$nA, in addition to the random background current of $U(-6.5, 6.5)$nA, delivered to each neuron via I_{ext_i} throughout the simulation. S_1 is arbitrarily chosen as the Conditional Stimuli (CS) and, whenever this stimuli is presented, a reward in the form of an increase in dopamine is delivered by setting $\text{DA}(t) = 0.5$ after a delay sampled from $U(0, 1000)$ms. This delay period is large enough to allow a few irrelevant stimuli to be presented which act as distractors. The simplest way to implement this stimulation regime is to add a current source to the excitatory and inhibitory neuron populations which adds the uniformly-distributed input current to an externally-controllable per-neuron current. In PyGeNN, the following model can be defined to do just that:

```
stim_noise_model = create_custom_current_source_class(
    "stim_noise",
    param_names=["n"],
    var_name_types=[("iExt", "scalar", VarAccess_READ_ONLY)],
    injection_code=
        """
        $(injectCurrent, $(iExt) + ($(gennrand_uniform) * $(n) * 2.0) - $(n));
        """)
```

where the `n` parameter sets the magnitude of the background noise, the `$(injectCurrent, I)` function injects a current of $I\,$nA into the neuron and `$(gennrand_uniform)` uses the 'XORWOW' pseudo-random

370 number generator provided by cuRAND **(TODO: CITE)** to sample from $U(0,1)$. Once a current source
371 population using this model has been instantiated and a memory view to `iExt` obtained in the manner
372 described in section 2.3, in timesteps when stimulus injection is required, current can be injected into the
373 list of neurons contained in `stimuli_input_set` with:

```
374  curr_ext_view[stimuli_input_set] = 40.0
375  curr_pop.push_var_to_device("iExt")
```

376 The same approach can then be used to zero the current afterwards. However, as almost $20\,000$ stimuli will
377 be injected over the course of a $1\,\mathrm{h}$ simulation, in order to reduce potential overheads, we can offload the
378 stimulus delivery entirely to the GPU using the following slightly more complex model:

```
379  stim_noise_model = create_custom_current_source_class(
380      "stim_noise",
381      param_names=["n", "stimMagnitude"],
382      var_name_types=[("startStim", "unsigned int"),
383                      ("endStim", "unsigned int", VarAccess_READ_ONLY)],
384      extra_global_params=[("stimTimes", "scalar*")],
385      injection_code=
386          """
387          scalar current = ($(gennrand_uniform) * $(n) * 2.0) - $(n);
388          if($(startStim) != $(endStim) && $(t) >= $(stimTimes)[$(startStim)]) {
389              current += $(stimMagnitude);
390              $(startStim)++;
391          }
392          $(injectCurrent, current);
393          """)
```

394 This model retains the same logic for generating background noise but, additionally, uses a simple sparse
395 matrix data structure to store the times at which each neuron should have current injected. **(TODO:**
396 **FIGURE)** The `startStim` and `endStim` variables point to the subset of the `stimTimes` array used by each
397 neuron's current source and, once the simulation time (t) passes the time pointed to by `startStim`,
398 current is injected and `startStim` is advanced. This array is stored in a 'extra global parameter' which
399 is a read-only memory area that can be allocated and populated from PyGeNN, in this case by 'stacking'
400 together a list of lists of spike times:

```
401  curr_pop.set_extra_global_param("stimTimes", np.hstack(neuron_stimuli_times))
```

3 RESULTS

402 In the following subsections we will analyse the performance of the models introduced in
403 sections 2.5 and 2.6 on a representative selection of NVIDIA GPU hardware:

404 • Jetson Xavier NX – a low-power embedded system with a GPU based on the Volta architecture with
405 $8\,\mathrm{GB}$ of shared memory.

406 • GeForce GTX 1050Ti – a low-end desktop GPU based on the Pascal architecture with $4\,\mathrm{GB}$ of
407 dedicated memory.

408 • GeForce GTX 1650 – a low-end desktop GPU based on the Turing architecture with $4\,\mathrm{GB}$ of dedicated
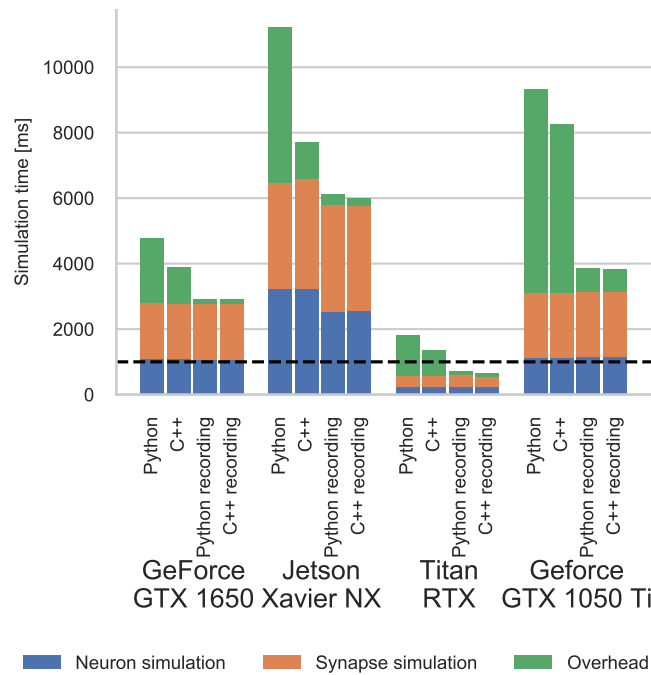409 memory.

Figure 3. Simulation times of the microcircuit model running on various GPU hardware for $1\,\mathrm{s}$ of biological time. 'Overhead' refers to time spent in simulation loop but not within CUDA kernels. The dashed horizontal line indicates realtime performance

- Titan RTX – a high-end workstation GPU based on the Turing architecture with $24\,\mathrm{GB}$ of dedicated memory.

All of these systems run Ubuntu 18 apart from the system with the GeForce 1050 Ti which runs Windows 10.

3.1 Cortical microcircuit model performance

Figure 3 shows the simulation times for the full-scale microcircuit model and, as one might predict, the Jetson Xavier NX is slower than the three desktop GPUs. However, considering that it only consumes a maximum of $15\,\mathrm{W}$ compared to $75\,\mathrm{W}$ or $320\,\mathrm{W}$ for the GeForce cards and Titan RTX respectively, it still performs impressively. The time taken to actually simulate the models ('Neuron simulation' and 'Synapse simulation') are the same when using Python and C++ as all GeNN optimisation options are exposed to PyGeNN. Interestingly, when simulating *this* model, the larger L1 cache and architectural improvements present in the Turing-based GTX 1650 do not result in significantly improved performance over the Pascal-based GTX 1050Ti. Instead, the slightly improved performance of the GTX 1650 can probably be explained by its additional 128 CUDA cores.

Without the recording system described in section 2.4, the CPU and GPU need to to synchronised after every timestep to allow spike data to be copied off the GPU and stored in a suitable data structure. The 'overheads' shown in figure 3 indicate the time taken by these processes as well as the unavoidable overheads of launching CUDA kernels etc. Because Python is an interpreted language, updating the spike data structures is somewhat slower and this is particularly noticeable on devices with a slower CPU such as the Jetson Xavier NX. However, unlike the desktop GPUs, the Jetson Xavier NX's $8\,\mathrm{GB}$ of memory is shared between the GPU and the CPU meaning that data doesn't have to be copied between their memories
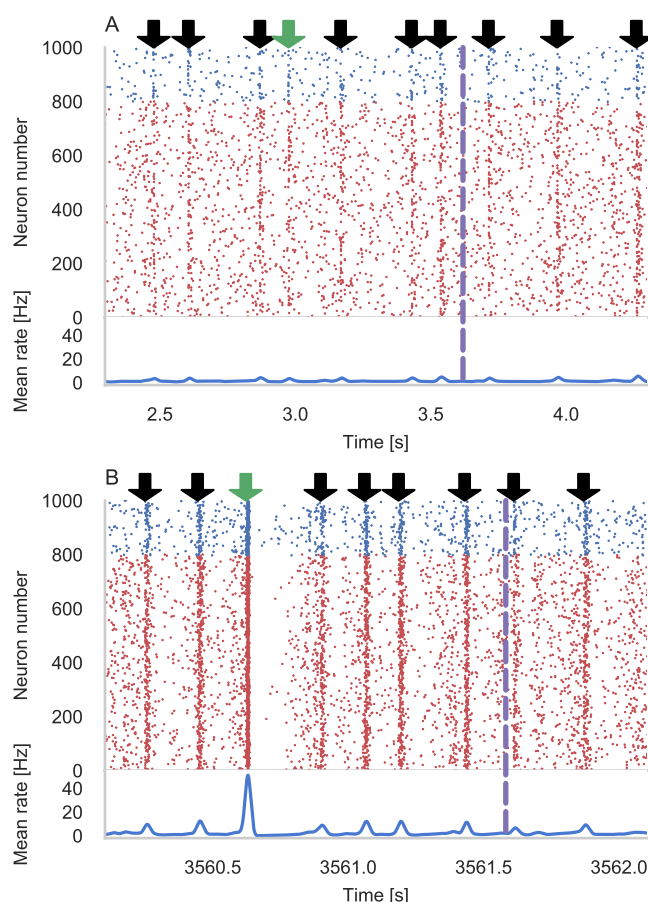
Figure 4. Results of Pavlovian conditioning experiment. Raster and spike density plots showing activity centred around first delivery of Conditional Stimulus (CS) during initial (A) and final (B) $50\,\text{s}$ of simulation. Downward green arrows indicate times at which CS is delivered and downward black arrows indicate times when other, un-rewarded stimuli are delivered. Vertical dashed lines indicate times at which dopamine is delivered

431 and can instead by accessed by both. While, using this shared memory for recording spikes reduces the
432 overhead of copying data off the device, because the GPU and CPU caches are not coherent, caching
433 must be disabled on this memory which reduces the performance of the neuron kernel. Although the
434 Windows machine has a relatively powerful CPU, the overheads measured in both the Python and C++
435 simulations run on this system are extremely large due to additional queuing between the application and
436 the GPU driver caused by the Windows Display Driver Model (WDDM). When small – in this case $0.1\,\text{ms}$
437 – simulation timesteps are used, this makes per-timestep synchronisation disproportionately expensive.

438 However, when the spike recording system described in section 2.4 is used, spike data is kept in GPU
439 memory until the end of the simulation and overheads are reduced by up to $10\times$. Because synchronisation
440 with the CPU is no longer required every timestep, simulations run approximately twice as fast on the
441 Windows machine. Furthermore, on the high-end desktop GPU, the simulation now runs faster than
442 real-time in both Python and native C++ versions – significantly faster than other recently published GPU
443 simulators (Golosio et al., 2020) and even specialised neuromorphic systems (Rhodes et al., 2020).
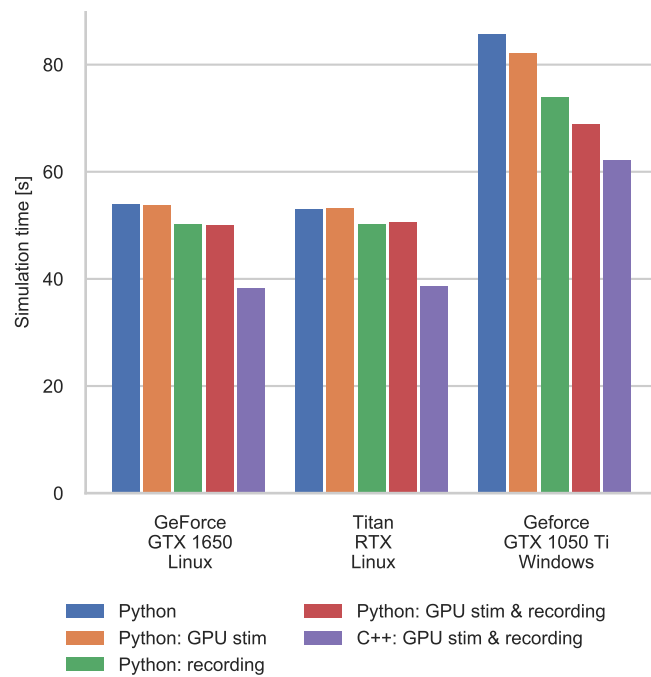
Figure 5. Simulation times of the Pavlovian Conditioning model running on various GPU hardware for 1 h of biological time. **(TODO: EXPLANATION OF BARS)**

3.2 Pavlovian conditioning performance

444

445   Figure 4 shows the results of an example simulation of the Pavlovian conditioning model. At the beginning
446   of each simulation (Figure 4A), the neurons representing every stimulus respond equally. However, after
447   1 h of simulation, the response to the CS becomes much stronger (Figure 4B) – showing that these neurons
448   have been selectively associated with the stimulus even in the presence of the distractors and the delayed
449   reward.

450   In figure 5, we show the runtime performance of simulations of the Pavlovian conditioning model,
451   running on a selection of desktop GPUs using PyGeNN with and without the recording system described
452   in section 2.4 and the optimized stimuli-delivery described in section 2.6. These PyGeNN results are
453   compared to a C++ simulation using both optimizations. Interestingly the Titan RTX and GTX 1650
454   perform identically in this benchmark with speedups ranging from $62\times$ to $72\times$ real-time. This is because, as
455   discussed previously, this model is simply not large enough to fill the 4608 CUDA cores present on the Titan
456   RTX. Therefore, as the two GPUs share the same Turing architecture and have very similar clock speeds
457   (1350 MHz–1770 MHz for the Titan RTX and 1485 MHz–1665 MHz for the GTX 1650), the two GPUs
458   perform very similarly. Furthermore, on these two systems, while using the recording system significantly
459   improves performance, the impact of delivering stimuli on the GPU is minimal. However, unlike in the
460   simulations of the microcircuit model, here the GTX 1050 Ti performs rather differently. Although the
461   clock speed of this device is approximately the same as the other GPUs (1290 MHz–1392 MHz) and it has
462   a similar number of CUDA cores to the GTX 1650, its performance is significantly worse. The difference
463   in performance across all configurations is likely to be due to architectural differences between the older
464   Pascal; and newer Volta and Turing architectures. Specifically, Pascal GPUs have one type of Arithmetic
465   Logic Unit (ALU) which handles both integer and floating point arithmetic whereas, the newer Volta
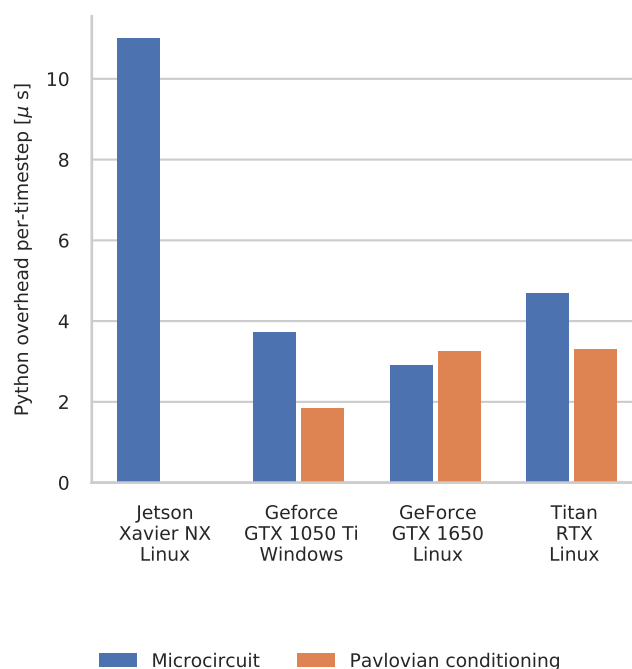466   and Turing architectures have equal numbers of dedicated integer and floating point ALUs as well as

Figure 6. Comparison of per-timestep overhead in microcircuit and Pavlovian conditioning experiments.

467 significantly larger L1 caches. As discussed in our previous work (Knight and Nowotny, 2018), these
468 architectural features are particularly beneficial for SNN simulations with STDP where a large amount
469 of floating point computation is required to update the synaptic state *and* additional integer arithmetic is
470 required to calculate the indices into the sparse matrix data structures. Furthermore, due to the additional
471 synchronisation overheads caused by the Windows Display Driver Model (WDDM) which we discussed in
472 the previous section, offloading stimuli delivery to the GPU improves the performance significantly on the
473 Windows machine.

474   The difference between the speeds of the Python and C++ simulations of the Pavlovian conditioning
475 model (figure 5) *appear* much larger than those of the microcircuit model (figure 3). However, as figure 6
476 illustrates, the difference between the duration of individual timestep in Python and C++ simulations of
477 both models is approximately constant and consistent with the cost of a small number of Python to C++
478 function calls (Apache Crail, 2019). However, depending on the size and complexity of the model as well
479 as the hardware used, this overhead may still be significant.**(TODO: NOT REALLY SURE WHETHER**
480 **SIGNIFICANT IS WHAT WE WANT TO SAY HERE)** For example, when simulating the microcircuit model
481 for $1\,\mathrm{s}$ on the Titan RTX, the overhead of using Python is less than $0.2\,\%$ but, when simulating the Pavlovian
482 conditioning model on the same device, the overhead of using Python is almost $31\,\%$.

4 DISCUSSION

483 In this paper we have introduced PyGeNN, a Python interface to the C++ based GeNN library for GPU
484 accelerated spiking neural network simulations.

485   Uniquely, the new interface provides access to all the features of GeNN, without leaving the comparative
486 simplicity of Python and with, as we have shown, typically negligible overheads from the Python
487 bindings. PyGeNN also allows bespoke neuron and synapse models to be defined from within Python,

488 making PyGeNN much more flexible and broadly applicable than, for instance, the Python interface
489 to NEST (Eppler et al., 2009) or the PyNN model description language used to expose CARLsim to
490 Python (Balaji et al., 2020).

491 In many ways, the new interface resembles elements of the Python-based Brian 2 simulator (Stimberg
492 et al., 2019) (and it's Brian2GeNN backend (Stimberg et al., 2020)) with two key differences. Unlike in
493 Brian 2, bespoke models in PyGeNN are defined with 'C-like' code snippets. This has the advantage of
494 unparalleled flexibility for the expert user but, comes at the cost of more complexity as the code for a
495 timestep update needs to include a suitable solver as well as merely differential equations. The second
496 difference lies in how data structures are handled. Whereas simulations run using the C++ or Brian2GeNN
497 Brian 2 backends use files to exchange data with Python, the underlying GeNN data structures are directly
498 accessible from PyGeNN meaning that no disk access is involved.

499 As we have demonstrated, the PyGeNN wrapper, exactly like native GeNN, can be used on a variety
500 of hardware from data centre scale down to mobile devices such as the NVIDIA Jetson. This allows for
501 the same codes to be used in large-scale brain simulations and embedded and embodied spiking neural
502 network research. Supporting the popular Python language in this interface makes this ecosystem available
503 to a wider audience of researchers in both Computational Neuroscience, bio-mimetic machine learning and
504 autonomous robotics.

505 The new interface also opens up opportunities to support researchers that work with other Python based
506 systems. In the Computational Neuroscience and Neuromorphic computing communities, we can now build
507 a PyNN (Davison et al., 2008) interface on top of PyGeNN and, infact, a prototype of such an interface is
508 in development. Furthermore, for the burgeoning spike-based machine learning community, we can use
509 PyGeNN as the basis for a spike-based machine learning framework akin to TensorFlow or PyTorch for
510 rate-based models. A prototype interface of this sort called mlGeNN is in development and close to release.

511 Finally, in this work we have introduced a new spike recording system for GeNN and have shown
512 that, using this system, we can now simulate the Potjans microcircuit (Potjans and Diesmann, 2014)
513 model faster than real-time, which thus far was only possible on the large SpiNNaker neuromorphic
514 supercomputer (Rhodes et al., 2020).

515 • do we need to discuss the wide variety of uses, i.e. MC versus Pavlovian demonstrated in this paper?
516 • Turing architecture is great for GeNN! Presented results improve on state-of-the-art.
517 • PyGeNN as an intermediate layer - PyNN, ML
518 • Cost of C++ - Python calls in models
519 • something about neuromorphic systems often being real-time / BS accelerated time

CONFLICT OF INTEREST STATEMENT

520 The authors declare that the research was conducted in the absence of any commercial or financial
521 relationships that could be construed as a potential conflict of interest.

AUTHOR CONTRIBUTIONS

522 JK and TN wrote the paper. TN is the original developer of GeNN. AK was the original developer of
523 PyGeNN. JK is currently the primary developer of both GeNN and PyGeNN and was responsible for

524 implementing the spike recording system. JK performed the experiments and the analysis of the results that
525 are presented in this work.

FUNDING

ACKNOWLEDGMENTS

DATA AVAILABILITY STATEMENT

529 The datasets [GENERATED/ANALYZED] for this study can be found in the [NAME OF REPOSITORY]
530 [LINK].

REFERENCES

531 Akar, N. A., Cumming, B., Karakasis, V., Kusters, A., Klijn, W., Peyser, A., et al. (2019). Arbor — A
532   Morphologically-Detailed Neural Network Simulation Library for Contemporary High-Performance
533   Computing Architectures. In *2019 27th Euromicro International Conference on Parallel, Distributed
534   and Network-Based Processing (PDP)* (IEEE), 274–282. doi:10.1109/EMPDP.2019.8671560
535 [Dataset] Apache Crail (2019). Crail Python API: Python -> C/C++ call overhead
536 Balaji, A., Adiraju, P., Kashyap, H. J., Das, A., Krichmar, J. L., Dutt, N. D., et al. (2020). PyCARL: A
537   PyNN Interface for Hardware-Software Co-Simulation of Spiking Neural Network
538 Beazley, D. M. (1996). Using SWIG to control, prototype, and debug C programs with Python. In *Proc.
539   4th Int. Python Conf*
540 Buzsáki, G. and Mizuseki, K. (2014). The log-dynamic brain: how skewed distributions affect network
541   operations. *Nature reviews. Neuroscience* 15, 264–78. doi:10.1038/nrn3687
542 Carnevale, N. T. and Hines, M. L. (2006). *The NEURON book* (Cambridge University Press)
543 Chou, T.-s., Kashyap, H. J., Xing, J., Listopad, S., Rounds, E. L., Beyeler, M., et al. (2018). CARLsim 4:
544   An Open Source Library for Large Scale, Biologically Detailed Spiking Neural Network Simulation
545   using Heterogeneous Clusters. In *2018 International Joint Conference on Neural Networks (IJCNN)*
546   (IEEE), 1–8. doi:10.1109/IJCNN.2018.8489326
547 Davison, A. P., Brüderle, D., Eppler, J., Kremkow, J., Muller, E., Pecevski, D., et al. (2008). PyNN: A
548   Common Interface for Neuronal Network Simulators. *Frontiers in neuroinformatics* 2, 11. doi:10.3389/
549   neuro.11.011.2008
550 Eppler, J. M., Helias, M., Muller, E., Diesmann, M., and Gewaltig, M. O. (2009). PyNEST: A convenient
551   interface to the NEST simulator. *Frontiers in Neuroinformatics* 2, 1–12. doi:10.3389/neuro.11.012.2008
552 Gewaltig, M.-O. and Diesmann, M. (2007). NEST (NEural Simulation Tool). *Scholarpedia* 2, 1430
553 Givon, L. E. and Lazar, A. A. (2016). Neurokernel: An open source platform for emulating the fruit fly
554   brain. *PLOS ONE* 11, 1–25. doi:10.1371/journal.pone.0146581
555 Golosio, B., Tiddia, G., De Luca, C., Pastorelli, E., Simula, F., and Paolucci, P. S. (2020). A new GPU
556   library for fast simulation of large-scale networks of spiking neurons , 1–27
557 Hines, M. L., Davison, A. P., and Muller, E. (2009). NEURON and Python. *Frontiers in Neuroinformatics*
558   3, 1–12. doi:10.3389/neuro.11.001.2009

Hopkins, M. and Furber, S. B. (2015). Accuracy and Efficiency in Fixed-Point Neural ODE Solvers. *Neural computation* 27, 2148–2182

Humphries, M. D. and Gurney, K. (2007). Solution Methods for a New Class of Simple Model Neurons M. *Neural Computation* 19, 3216–3225. doi:doi:10.1162/neco.2007.19.12.3216

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering* 9, 90–95. doi:10.1109/MCSE.2007.55

Izhikevich, E. M. (2003). Simple model of spiking neurons. *IEEE Transactions on neural networks* 14, 1569–72. doi:10.1109/TNN.2003.820440

Izhikevich, E. M. (2007). Solving the Distal Reward Problem through Linkage of STDP and Dopamine Signaling. *Cerebral Cortex* 17, 2443–2452. doi:10.1093/cercor/bhl152

Knight, J. C. and Nowotny, T. (2018). GPUs Outperform Current HPC and Neuromorphic Solutions in Terms of Speed and Energy When Simulating a Highly-Connected Cortical Model. *Frontiers in Neuroscience* 12, 1–19. doi:10.3389/fnins.2018.00941

Knight, J. C. and Nowotny, T. (2020). Larger GPU-accelerated brain simulations with procedural connectivity. *bioRxiv* doi:10.1101/2020.04.27.063693

Mikaitis, M., Pineda García, G., Knight, J. C., and Furber, S. B. (2018). Neuromodulated Synaptic Plasticity on the SpiNNaker Neuromorphic System 12, 1–13. doi:10.3389/fnins.2018.00105

Millman, K. J. and Aivazis, M. (2011). Python for scientists and engineers. *Computing in Science and Engineering* 13, 9–12. doi:10.1109/MCSE.2011.36

Pauli, R., Weidel, P., Kunkel, S., and Morrison, A. (2018). Reproducing Polychronization: A Guide to Maximizing the Reproducibility of Spiking Network Models. *Frontiers in Neuroinformatics* 12, 1–21. doi:10.3389/fninf.2018.00046

Potjans, T. C. and Diesmann, M. (2014). The Cell-Type Specific Cortical Microcircuit: Relating Structure and Activity in a Full-Scale Spiking Network Model. *Cerebral Cortex* 24, 785–806. doi:10.1093/cercor/bhs358

Rhodes, O., Peres, L., Rowley, A. G. D., Gait, A., Plana, L. A., Brenninkmeijer, C., et al. (2020). Real-time cortical simulation on neuromorphic hardware. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 378, 20190160. doi:10.1098/rsta.2019.0160

Stimberg, M., Brette, R., and Goodman, D. F. (2019). Brian 2, an intuitive and efficient neural simulator. *eLife* 8, 1–41. doi:10.7554/eLife.47314

Stimberg, M., Goodman, D. F., and Nowotny, T. (2020). Brian2GeNN: accelerating spiking neural network simulations with graphics hardware. *Scientific Reports* 10, 1–12. doi:10.1038/s41598-019-54957-7

Van Der Walt, S., Colbert, S. C., and Varoquaux, G. (2011). The NumPy array: A structure for efficient numerical computation. *Computing in Science and Engineering* 13, 22–30. doi:10.1109/MCSE.2011.37

Vitay, J., Dinkelbach, H., and Hamker, F. (2015). ANNarchy: a code generation approach to neural simulations on parallel hardware. *Frontiers in Neuroinformatics* 9, 19. doi:10.3389/fninf.2015.00019

Yavuz, E., Turner, J., and Nowotny, T. (2016). GeNN: a code generation framework for accelerated brain simulations. *Scientific reports* 6, 18854. doi:10.1038/srep18854