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

2 ABSTRACT

More than half of the Top 10 supercomputing sites worldwide use GPU accelerators and they are becoming ubiquitous in workstations and edge computing devices. GeNN is a C++ library for generating efficient spiking neural network simulation code for GPUs. However, until now, the full flexibility of GeNN could only be harnessed by writing model descriptions and simulation code in C++. Here we present PyGeNN, a Python package which exposes all of GeNN's functionality to Python with minimal overhead. This provides an alternative, arguably more user-friendly, way of using GeNN and allows modellers to use GeNN within the growing Python-based machine learning and computational neuroscience ecosystems. In addition, we demonstrate that, in both Python and C++ GeNN simulations, the overheads of recording spiking data can strongly affect runtimes and show how a new spike recording system can reduce these overheads by up to $10\times$. Using the new recording system, we demonstrate that by using PyGeNN on a modern GPU, we can simulate a full-scale model of a cortical column faster even than real-time neuromorphic systems. Finally, we show that long simulations of a smaller model with complex stimuli and a custom three-factor learning rule defined in PyGeNN can be simulated up to $72\times$ faster than real-time.

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

30 to another language is convenient. However, both choices act as a barrier to potential users. Therefore, with
31 both the computational neuroscience and machine learning communities gradually coalescing towards a
32 Python-based ecosystem with a wealth of mature libraries for scientific computing (Hunter, 2007; Van Der
33 Walt et al., 2011; Millman and Aivazis, 2011), exposing spiking neural network simulators to Python seems
34 a pragmatic choice. NEST (Eppler et al., 2009), NEURON (Hines et al., 2009) and CARLsim (Balaji et al.,
35 2020) have all taken this route and now all offer Python interfaces. Furthermore, newer simulators such as
36 Arbor and Brian2 (Stimberg et al., 2019) have been designed from the ground up with a Python interface.

37 Our GeNN simulator can already be used as a backend for the Python-based Brian2 simulator (Stimberg
38 et al., 2019) using the Brian2GeNN interface (Stimberg et al., 2020) which modifies the C++ backend
39 "cpp_standalone" of Brian 2 to generate C++ input files for GeNN. As for cpp_standalone, initialisation of
40 simulations is mostly done in C++ on the CPU and recording data is saved into binary files and re-imported
41 into Python using Brian 2's native methods. While we have recently demonstrated some very competitive
42 performance results (Knight and Nowotny, 2018, 2020) using GeNN in C++, and through the Brian2GeNN
43 interface (Stimberg et al., 2020), GeNN could so far not be used directly from Python and it is not possible
44 to expose all of GeNN's unique features through the Brian2 API. Specifically, GeNN not only allows users
45 to easily define their own neuron and synapse models but, also 'snippets' for offloading the potentially
46 costly initialisation of model parameters and connectivity onto the GPU. Additionally, GeNN provides a
47 lot of freedom for users to integrate their own code into the simulation loop. In this paper we describe the
48 implementation of PyGeNN – a Python package which aims to expose the full range of GeNN functionality
49 with minimal performance overheads. Unlike in the majority of other SNN simulators PyGeNN allows
50 defining bespoke neuron and synapse models directly from Python without requiring users to extend the
51 underling C++ code. Below, we demonstrate the flexibility and performance of PyGeNN in two scenarios
52 where minimising performance overheads is particularly critical.

53 • In a simulation of a large, highly-connected model of a cortical microcircuit (Potjans and Diesmann,
54   2014) with small simulation timesteps. Here the cost of copying spike data off the GPU from a large
55   number of neurons every timestep can become a bottleneck.
56 • In a simulation of a much smaller model of Pavlovian conditioning (Izhikevich, 2007) where learning
57   occurs over $1\,\mathrm{h}$ of biological time and stimuli are delivered – following a complex scheme – throughout
58   the simulation. Here any overheads are multiplied by a large number of timesteps and copying stimuli
59   to the GPU can become a bottleneck.

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

2 MATERIALS AND METHODS

2.1 GeNN

63 GeNN (Yavuz et al., 2016) is a library for generating CUDA (NVIDIA et al., 2020) code for the simulation
64 of spiking neural network models. GeNN handles much of the complexity of using CUDA directly and
65 automatically performs device-specific optimizations so as to to maximize performance. GeNN consists
66 of a main library – implementing the API used to define models as well as the generic parts of the code
67 generator – and an additional library for each backend (currently there is a reference C++ backend for
68 generating CPU code and a CUDA backend. An OpenCL backend is under development). Users describe
69 their model by implementing a `modelDefinition` function within a C++ file. For example, a model

70 consisting of 4 Izhikevich neurons with heterogeneous parameters, driven by a constant input current might
71 be defined as follows:

```
72  void modelDefinition(ModelSpec &model)
73  {
74      model.setDT(0.1);
75      model.setName("izhikevich");
76
77      NeuronModels::IzhikevichVariable::VarValues popInit(
78          -65.0, -20.0, uninitialisedVar(), uninitialisedVar(),
79          uninitialisedVar(), uninitialisedVar());
80
81      model.addNeuronPopulation<NeuronModels::IzhikevichVariable>(
82          "Pop", 4, {}, popInit);
83
84      model.addCurrentSource<CurrentSourceModels::DC>(
85          "CS", "Pop", {10.0}, {});
86  }
```

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

```
95  #include "sharedLibraryModel.h"
96
97  int main()
98  {
99      SharedLibraryModel<float> model("./", "izhikevich");
100     model.allocateMem();
101     model.initialize();
102     float *aPop = model.getScalar<float>("aPop");
103     float *bPop = model.getScalar<float>("bPop");
104     float *cPop = model.getScalar<float>("cPop");
105     float *dPop = model.getScalar<float>("dPop");
106     aPop[0] = 0.02; bPop[0] = 0.2; cPop[0] = -65.0; dPop[0] = 8.0;
107     aPop[1] = 0.1;  bPop[1] = 0.2; cPop[1] = -65.0; dPop[1] = 2.0;
108     aPop[2] = 0.02; bPop[2] = 0.2; cPop[2] = -50.0; dPop[2] = 2.0;
109     aPop[3] = 0.02; bPop[3] = 0.2; cPop[3] = -55.0; dPop[3] = 4.0;
110     model.initializeSparse();
111
112     float *vPop = model.getScalar<float>("VPop");
113     while(model.getTime() < 200.0f) {
114         model.stepTime();
115         model.pullVarFromDevice("Pop", "V");
116         printf("%f, %f, %f, %f, %f\n",
```

```
117                  t, VPop[0], VPop[1], VPop[2], VPop[3]);
118      }
119      return EXIT_SUCCESS;
120  }
```

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, wrapper functions would need to be manually implemented for each GeNN function to be exposed which would result in a lot of maintenance overhead. Instead, we chose to use SWIG (Beazley, 1996) to automatically generate wrapper functions and classes. SWIG generates Python modules based on special interface files which can directly include C++ code as well as special 'directives' which control SWIG. For example. the following SWIG interface file would wrap the C++ code in test.h in a Python module called `test_module` within a Python packages called `test_package`:

```
%module(package="test_package") test_module
%include "test.h"
```

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

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

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

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

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

2.3 PyGeNN

While GeNN *could* be used from Python via the wrapper generated using SWIG, the resultant code would be unpleasant to use directly. For example, rather than being able to specify neuron parameters using a native Python types such as lists or dictionaries, one would have to use a wrapped type such as `DoubleVector([0.25, 10.0, 0.0, 0.0, 20.0, 2.0, 0.5])`. Therefore, in order to provide a more user-friendly and pythonic interface, we have built PyGeNN on top of the wrapper generated by SWIG. PyGeNN

combines the separate model building and simulation stages of building a GeNN model in C++ into a single API, likely to be more familiar to users of existing Python-based model description languages such as PyNEST (Eppler et al., 2009) or PyNN (Davison et al., 2008). By combining the two stages together, PyGeNN can provide a unified dictionary-based API for initialising homogeneous and heterogeneous parameters as shown in this re-implementation of the previous example:

```python
from pygenn import genn_wrapper, genn_model

model = genn_model.GeNNModel("float", "izhikevich")
model.dT = 0.1

izk_init = {"V": -65.0,
            "U": -20.0,
            "a": [0.02, 0.1, 0.02, 0.02],
            "b": [0.2, 0.2, 0.2, 0.2],
            "c": [-65.0, -65.0, -50.0, -55.0],
            "d": [8.0, 2.0, 2.0, 4.0]}

pop = model.add_neuron_population("Pop", 4, "IzhikevichVariable",
                                  {}, izk_init)
model.add_current_source("CS", "DC", "Pop",
                         {"amp": 10.0}, {})

model.build()
model.load()

v = pop.vars["V"].view
while model.t < 200.0:
    model.step_time()
    model.pull_state_from_device("Pop")
    print("%f, %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 initialisation kernels generated by GeNN and initialisation of variables with heterogeneous values – such as the a, b and c parameters – are copied to the GPU by PyGeNN after 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 directly inheriting from the SWIG director class:

```python
izk_model = genn_model.create_custom_neuron_class(
    "izk",
```

```
202        param_names=["a", "b", "c", "d"],
203        var_name_types=[("V", "scalar"), ("U", "scalar")],
204        sim_code=
205            """
206            $(V)+=0.5*(0.04*$(V)*$(V)+5.0*$(V)+140.0-$(U)+$(Isyn))*DT;
207            $(V)+=0.5*(0.04*$(V)*$(V)+5.0*$(V)+140.0-$(U)+$(Isyn))*DT;
208            $(U)+=$(a)*($(b)*$(V)-$(U))*DT;
209            """,
210        threshold_condition_code="$(V) >= 30.0",
211        reset_code=
212            """
213            $(V)=$(c);
214            $(U)+=$(d);
215            """)
```

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

2.4 Spike recording system

Internally, GeNN stores the spikes emitted by a neuron population during one simulation timestep in an array containing the indices of the neurons that spiked alongside a counter of how many spikes have been emitted. Previously, recording spikes in GeNN was very similar to the recording of voltages shown in the previous example code – the array of neuron indices was simply copied from the GPU to the CPU every timestep. However, especially when simulating models with a small simulation timestep, such frequent synchronization between the CPU and GPU is costly – especially if a slower, interpreted language such as Python is involved. Furthermore, biological neurons typically spike at a low rate (in the cortex, the average firing rate is only around $3\,\text{Hz}$ (Buzsáki and Mizuseki, 2014)) meaning that the amount of spike data transferred every timestep is typically very small. To address both of these sources of inefficiency, we have added a new data structure to GeNN which stores spike data for many timesteps on the GPU. To reduce the memory required for this data structure and to make its size independent of neural activity, the spikes emitted by a population of N neurons in a single simulation timestep are stored in a Nbit bitfield where a '1' represents a spike and a '0' the absence of one. Spiking data over multiple timesteps is then represented by a circular buffer of these bitfields. Using this approach, even the spiking output of relatively large models, running for many timesteps can be stored in a small amount of memory. For example, the spiking output of a model with 100×10^3 neurons running for 10×10^3 simulation timesteps, required less than $120\,\text{MB}$ – a small fraction of the memory on a modern GPU. While efficiently handling spikes stored in a bitfield is a little trickier than working with a list of neuron indices, GeNN provides an efficient
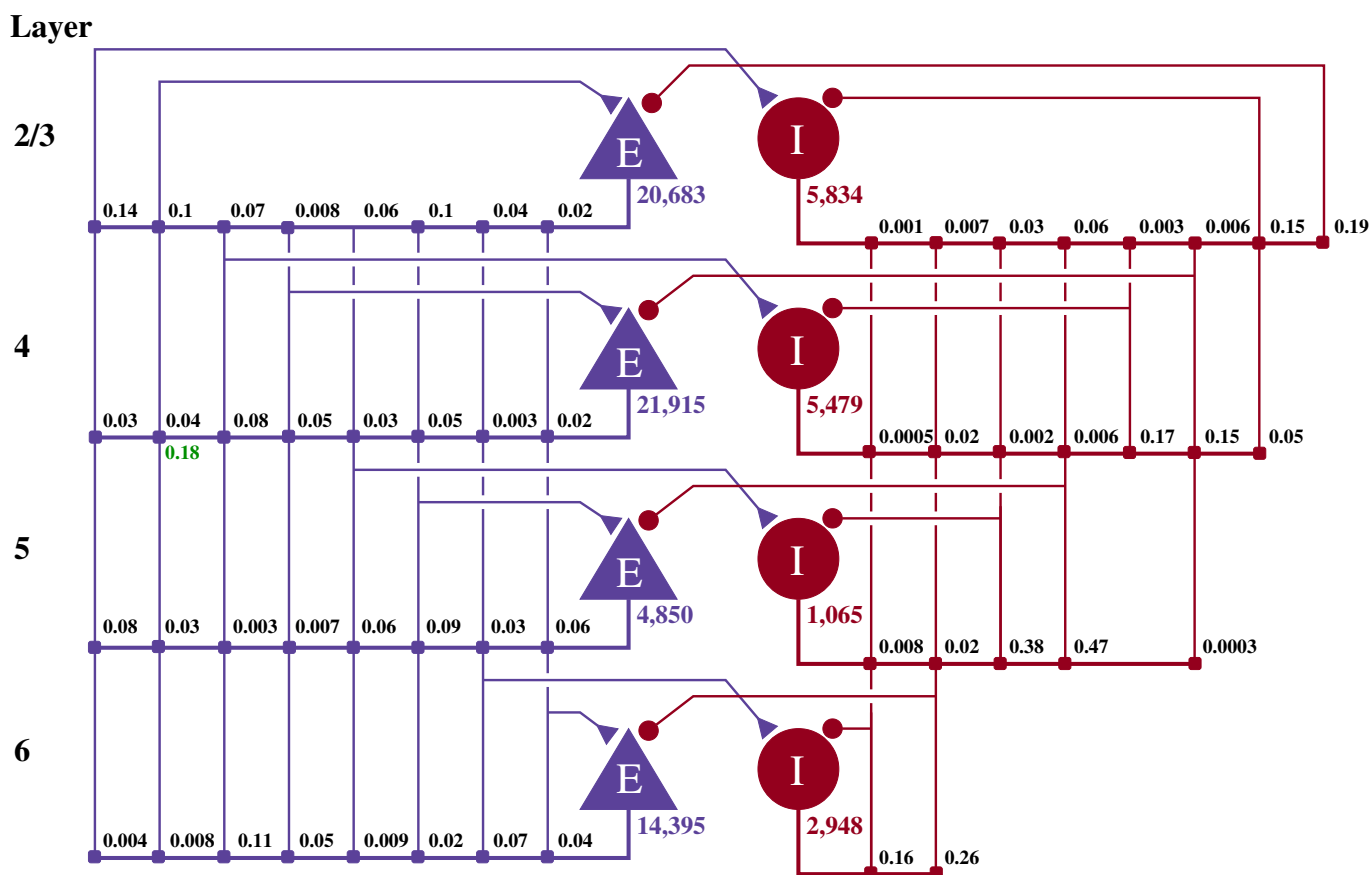
Figure 1. Illustration of the microcircuit model. Blue triangles represent excitatory populations, red circles represent inhibitory populations and the numbers beneath each symbol shows the number of neurons in each population. Connection probabilities are shown in small bold numbers at the appropriate point in the connection matrix. All excitatory synaptic weights are normally distributed with a mean of $0.0878\,\mathrm{nA}$ (unless otherwise indicated in green) and a standard deviation of $0.008\,78\,\mathrm{nA}$. All inhibitory synaptic weights are normally distributed with a mean of $0.3512\,\mathrm{nA}$ and a standard deviation of $0.035\,12\,\mathrm{nA}$.

246 C++ helper function for saving the spikes stored in a bitfield to a text file and a numpy-based method for
247 decoding them in PyGeNN.

248 ## 2.5 Cortical microcircuit model

Potjans and Diesmann (2014) developed the cortical microcircuit model of $1\,\mathrm{mm}^3$ of early-sensory cortex illustrated in figure 1. 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). 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
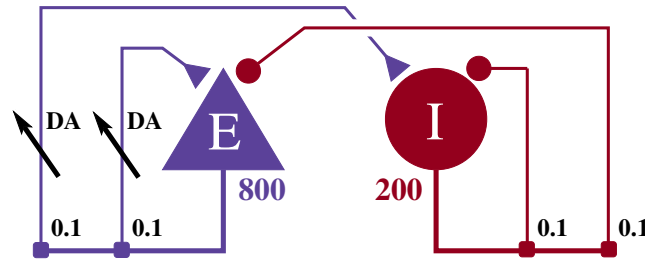
Figure 2. Illustration of the balanced random network model. The blue triangle represents the excitatory population, the red circle represents the inhibitory population, and the numbers beneath each symbol show the number of neurons in each population. Connection probabilities are shown in small bold 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.

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}

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

where $\tau_{\mathrm{syn}} = 0.5\,\mathrm{ms}$ represents the synaptic time constant, w_{ij} represents the synaptic weight 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 explicitly modelled cortical regions. The Poisson input is delivered to each neuron via I_{ext_i} with

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

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

2.6 Pavlovian conditioning model

The cortical microcircuit model described in the previous section is ideal for exploring the performance of short simulations of relatively large models. However, the performance of longer simulations of smaller models is equally vital. Such models can be particularly troublesome for GPU simulation as, not only might they not offer enough parallelism to fully occupy the device but, each timestep can be simulated so quickly that the overheads of launching kernels etc can dominate. Additional overheads can be incurred when models require injecting external stimuli throughout the simulation. Longer simulations are particularly useful when exploring synaptic plasticity so, to explore the performance of PyGeNN in this scenario, we

8

264 simulate a model of Pavlovian conditioning using a three-factor Spike-Timing-Dependent Plasticity (STDP)
265 learning rule (Izhikevich, 2007).

2.6.1 Neuron model

The model illustrated in figure 2 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:

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

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

267 When the membrane voltage rises above 30, a spike is emitted and V_i is reset to c and d is added to
268 U_i. Excitatory neurons use the regular-spiking parameters (Izhikevich, 2003) where $a = 0.02$, $b = 0.2$,
269 $c = -65.0$, $d = 8.0$ and inhibitory neurons use the fast-spiking parameters (Izhikevich, 2003) where
270 $a = 0.1$, $b = 0.2$, $c = -65.0$, $d = 2.0$. Again, I_{syn_i} represents the synaptic input current and I_{ext_i} represents
271 an external input current. While there are numerous ways to solve equations 4 and 5 (Humphries and
272 Gurney, 2007; Hopkins and Furber, 2015; Pauli et al., 2018), we chose to use the idiomatic forward Euler
273 integration scheme employed by Izhikevich (2003) in the original work (Izhikevich, 2007). Under this
274 scheme, equation 4 is first integrated for two $0.5\,\text{ms}$ timesteps and then, based on the updated value of V_i,
275 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 1, with instantaneous current-based synapses:

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

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 in response to 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:

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

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

2.6.3 PyGeNN implementation of three-factor STDP

The first step in implementing this learning rule in PyGeNN is to implement the STDP updates and decay of C_{ij} using GeNN's event-driven plasticity system, the implementation of which was described in our previous work (Knight and Nowotny, 2018). Using a similar syntax to that described in section 2.3, we first create a new 'weight update model' with the learning rule parameters and the w_{ij} and 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")],
```

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

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

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

305 Within the sim code we also need to calculate the time that has elapsed since the last update of C_{ij} using
306 the spike times we previously requested that GeNN record. Within a timestep, GeNN processes presynaptic
307 spikes before postsynaptic spikes so the time of the last update to C_{ij} will be the latest time either type of
308 spike was processed in previous timesteps:

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

311 Using this time, we can now calculate how much to decay C_{ij} using the closed-form solution to equation 7:

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

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

```
317        const scalar dt = $(t) − $(sT_post);
318        if (dt > 0) {
319            newTag −= ($(aMinus) * exp(−dt / $(tauMinus)));
320        }
321        $(c) = newTag;
322        """,
```

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

```
327    learn_post_code=
328        """
329        const scalar tc = fmax($(sT_pre),
330                               $(prev_sT_post));
331
332        const scalar tagDecay = exp(−($(t) − tc) / $(tauC));
333        scalar newTag = $(c) * tagDecay;
334
335        const scalar dt = $(t) − $(sT_pre);
336        if (dt > 0) {
337            newTag += ($(aPlus) * exp(−dt / $(tauPlus)));
338        }
339        $(c) = newTag;
340        """)
```

341 Adding the synaptic weight w_{ij} update described by equation 8 requires two further additions to the model.
342 As well as the pre and postsynaptic spikes, the weight update model needs to receive events whenever
343 dopamine is injected via DA. GeNN supports such events via the 'spike-like event' system which allows
344 events to be triggered based on an expression evaluated on the presynaptic neuron. In this case, this
345 expression simply tests an `injectDopamine` flag which gets set by the dopamine injection logic in our
346 presynaptic neuron model:

347   `event_threshold_condition_code="injectDopamine",`

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

350   `is_pre_spike_event_time_required=True,`
351   `is_prev_pre_spike_event_time_required=True,`

352 The spike-like events can now be handled using a final 'event code' string:

```
353    event_code=
354        """
355        const scalar tc = fmax($(sT_pre), fmax($(prev_sT_post), $(prev_seT_pre)));
356        const scalar tagDecay = exp(-($(t) - tc) / $(tauC));
357        $(c) *= tagDecay;
358        """,
```

After updating the previously defined calculations of `tc` in the sim code and learn post code in the same way 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 solved 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:

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

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

2.6.4 PyGeNN implementation of Pavlovian conditioning experiment

362   To perform the Pavlovian conditioning experiment described by Izhikevich (2007) using this model, we
363 chose 100 random groups of 50 neurons (each representing stimuli $S_1...S_{100}$) from amongst the two neural
364 populations. Stimuli are presented to the network in a random order, separated by intervals sampled from
365 $U(100, 300)$ms. The neurons associated with an active stimulus are stimulated for a single 1 ms simulation
366 timestep with a current of 40.0 nA, in addition to the random background current of $U(-6.5, 6.5)$nA,
367 delivered to each neuron via I_{ext_i} throughout the simulation. S_1 is arbitrarily chosen as the Conditional
368 Stimuli (CS) and, whenever this stimuli is presented, a reward in the form of an increase in dopamine
369 is delivered by setting $\text{DA}(t) = 0.5$ after a delay sampled from $U(0, 1000)$ms. This delay period is large
370 enough to allow a few irrelevant stimuli to be presented which act as distractors. The simplest way to
371 implement this stimulation regime is to add a current source to the excitatory and inhibitory neuron

372 populations which adds the uniformly-distributed input current to an externally-controllable per-neuron
373 current. In PyGeNN, the following model can be defined to do just that:

```
374 stim_noise_model = create_custom_current_source_class(
375     "stim_noise",
376     param_names=["n"],
377     var_name_types=[("iExt", "scalar", VarAccess_READ_ONLY)],
378     injection_code=
379         """
380         $(injectCurrent, $(iExt) + ($(gennrand_uniform) * $(n) * 2.0) − $(n));
381         """)
```

382 where the `n` parameter sets the magnitude of the background noise, the `$(injectCurrent, I)` function
383 injects a current of InA into the neuron and `$(gennrand_uniform)` samples from $U(0, 1)$ using the
384 'XORWOW' pseudo-random number generator provided by cuRAND (NVIDIA Corporation, 2019). Once
385 a current source population using this model has been instantiated and a memory view to `iExt` obtained
386 in the manner described in section 2.3, in timesteps when stimulus injection is required, current can be
387 injected into the list of neurons contained in `stimuli_input_set` with:

```
388 curr_ext_view[stimuli_input_set] = 40.0
389 curr_pop.push_var_to_device("iExt")
```

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

```
393 stim_noise_model = create_custom_current_source_class(
394     "stim_noise",
395     param_names=["n", "stimMagnitude"],
396     var_name_types=[("startStim", "unsigned int"),
397                     ("endStim", "unsigned int", VarAccess_READ_ONLY)],
398     extra_global_params=[("stimTimes", "scalar*")],
399     injection_code=
400         """
401         scalar current = ($(gennrand_uniform) * $(n) * 2.0) − $(n);
402         if($(startStim) != $(endStim) && $(t) >= $(stimTimes)[$(startStim)]) {
403             current += $(stimMagnitude);
404             $(startStim)++;
405         }
406         $(injectCurrent, current);
407         """)
```

408 This model retains the same logic for generating background noise but adds a `stimTimes` array which
409 contains the times at which each neuron should have current injected. This array is an 'extra global
410 parameter' – a read-only memory area that can be allocated and populated from PyGeNN, in this case by
411 'stacking' together a list of lists of spike times:

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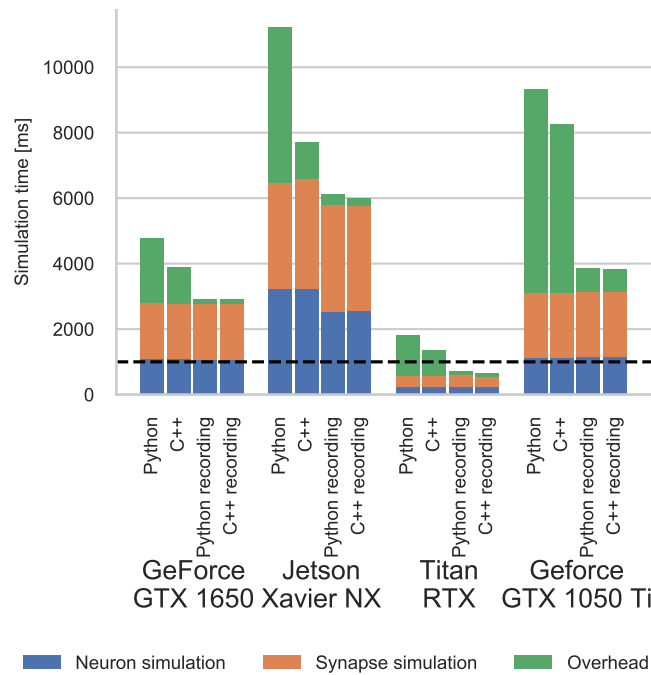
Figure 3. Simulation times of the microcircuit model running on various GPU hardware for $1\,\mathrm{s}$ of biological time. 'Overhead' refers to time spent in simulation loop but not within CUDA kernels. The dashed horizontal line indicates realtime performance

The `startStim` and `endStim` variables are then used to point to the subset of the `stimTimes` array corresponding to each neuron. Once the simulation time (`$(t)`) passes the time at `stimTimes[startStim]`, current is injected and `startStim` is advanced.

3 RESULTS

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

- Jetson Xavier NX – a low-power embedded system with a GPU based on the Volta architecture with $8\,\mathrm{GB}$ of shared memory.
- GeForce GTX 1050Ti – a low-end desktop GPU based on the Pascal architecture with $4\,\mathrm{GB}$ of dedicated memory.
- GeForce GTX 1650 – a low-end desktop GPU based on the Turing architecture with $4\,\mathrm{GB}$ of dedicated memory.
- 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. We measured the total simulation time by querying the `std::chrono::high_resolution_clock` in C++ and the
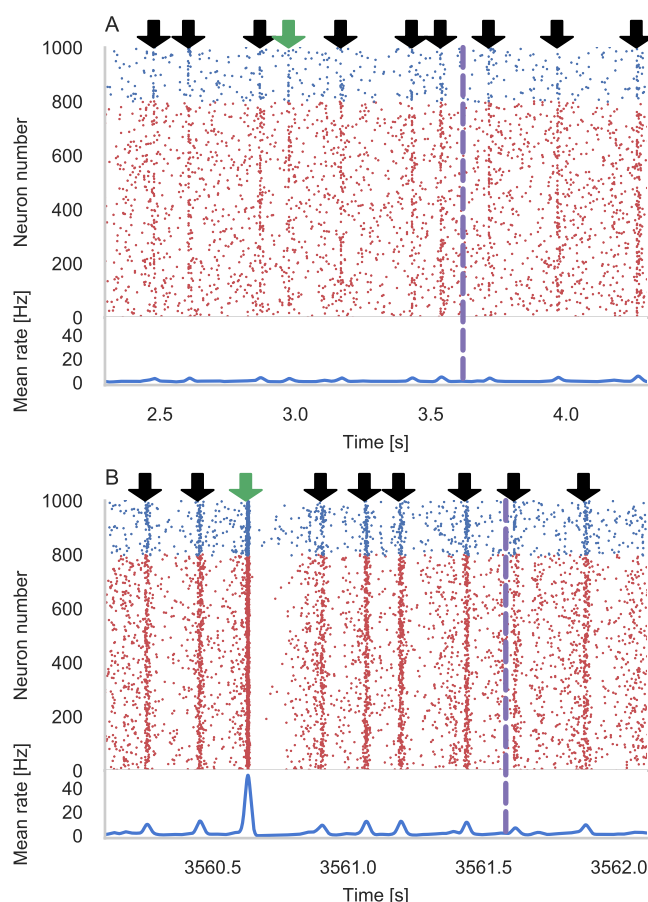
Figure 4. Results of Pavlovian conditioning experiment. Raster and spike density plots showing activity centred around first delivery of Conditional Stimulus (CS) during initial (A) and final (B) $50\,\mathrm{s}$ of simulation. Downward green arrows indicate times at which CS is delivered and downward black arrows indicate times when other, un-rewarded stimuli are delivered. Vertical dashed lines indicate times at which dopamine is delivered. Mean rates are calculated by convolving the spike train with a spike density function with $\sigma = 10\,\mathrm{ms}$.**(TODO: THOMAS:IS THERE A STANDARD CITATION FOR THIS TECHNIQUE/IS THIS A REASONABLE DESCRIPTION?)**

431 `time.perf_counter` in Python before and after the simulation loop; and used CUDA's own event timing
432 system (NVIDIA Corporation, 2021, Section 3.2.5.6.2) to record the time taken by the neuron and synapse
433 kernels. As one might predict, the Jetson Xavier NX is slower than the three desktop GPUs but, considering
434 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
435 RTX respectively, it still performs impressively. The time taken to actually simulate the models ('Neuron
436 simulation' and 'Synapse simulation') are the same when using Python and C++ as all GeNN optimisation
437 options are exposed to PyGeNN. Interestingly, when simulating *this* model, the larger L1 cache and
438 architectural improvements present in the Turing-based GTX 1650 do not result in significantly improved
439 performance over the Pascal-based GTX 1050Ti. Instead, the slightly improved performance of the GTX
440 1650 can probably be explained by its additional 128 CUDA cores.

441 Without the recording system described in section 2.4, the CPU and GPU need to to synchronised
442 after every timestep to allow spike data to be copied off the GPU and stored in a suitable data structure.
443 The 'overheads' shown in figure 3 indicate the time taken by these processes as well as the unavoidable
444 overheads of launching CUDA kernels etc. Because Python is an interpreted language, updating the spike
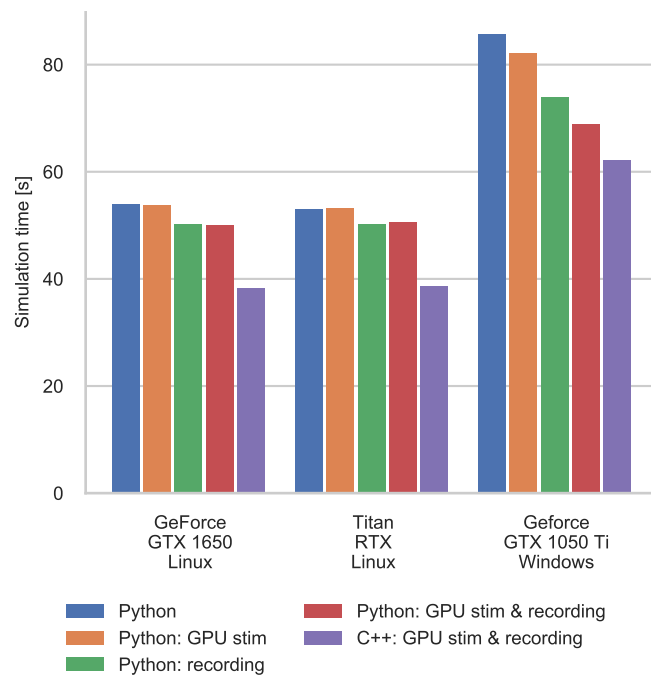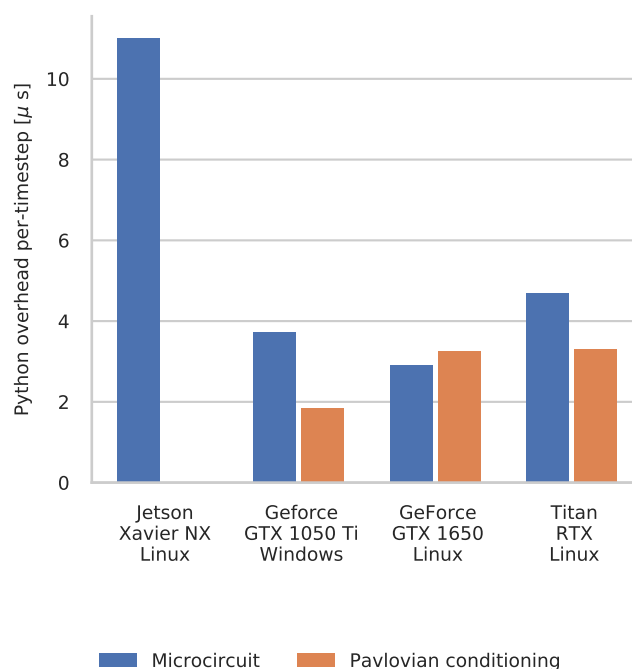
Figure 5. Simulation times of the Pavlovian Conditioning model running on various GPU hardware for $1\,\mathrm{h}$ of biological time. 'GPU stim' indicates simulations using the GPU stimuli delivery model and 'recording' indicates simulations where the new recording system is employed.

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

However, when the spike recording system described in section 2.4 is used, spike data is kept in GPU memory until the end of the simulation and overheads are reduced by up to $10\times$. Because synchronisation with the CPU is no longer required every timestep, simulations run approximately twice as fast on the Windows machine. Furthermore, on the high-end desktop GPU, the simulation now runs faster than real-time in both Python and native C++ versions – significantly faster than other recently published GPU simulators (Golosio et al., 2020) and even specialised neuromorphic systems (Rhodes et al., 2020).

3.2 Pavlovian conditioning performance

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

Figure 6. Comparison of the duration of individual timestep in Python and C++ simulation in microcircuit and Pavlovian conditioning experiments. Times are taken from the fastest version of each model i.e. the microcircuit using the recording system and the Pavlovian conditioning model using the recording system and the GPU stimuli delivery.

467　　In figure 5, we show the runtime performance of simulations of the Pavlovian conditioning model,
468　running on a selection of desktop GPUs using PyGeNN with and without the recording system described
469　in section 2.4 and the optimized stimuli-delivery described in section 2.6. These PyGeNN results are
470　compared to a C++ simulation using both optimizations. Because each simulation timestep only takes
471　a few μs, the overhead of using CUDA timing events significantly alters the performance so, for this
472　model, we only measure the duration of the simulation loop using the approaches described in the previous
473　section. Interestingly the Titan RTX and GTX 1650 perform identically in this benchmark with speedups
474　ranging from $62\times$ to $72\times$ real-time. This is because, as discussed previously, this model is simply not
475　large enough to fill the 4608 CUDA cores present on the Titan RTX. Therefore, as the two GPUs share
476　the same Turing architecture and have very similar clock speeds ($1350\,\mathrm{MHz}$–$1770\,\mathrm{MHz}$ for the Titan RTX
477　and $1485\,\mathrm{MHz}$–$1665\,\mathrm{MHz}$ for the GTX 1650), the two GPUs perform very similarly. Although we only
478　record the spiking activity during the first and last $50\,\mathrm{s}$, using the recording system on these two systems
479　still significantly improves the overall performance whereas, delivering stimuli on the GPU only provides
480　a minimal improvement. However, unlike in the simulations of the microcircuit model, here the GTX
481　1050 Ti performs rather differently. Although the clock speed of this device is approximately the same
482　as the other GPUs ($1290\,\mathrm{MHz}$–$1392\,\mathrm{MHz}$) and it has a similar number of CUDA cores to the GTX 1650,
483　its performance is significantly worse. The difference in performance across all configurations is likely
484　to be due to architectural differences between the older Pascal; and newer Volta and Turing architectures.
485　Specifically, Pascal GPUs have one type of Arithmetic Logic Unit (ALU) which handles both integer
486　and floating point arithmetic whereas, the newer Volta and Turing architectures have equal numbers of
487　dedicated integer and floating point ALUs as well as significantly larger L1 caches. As discussed in our
488　previous work (Knight and Nowotny, 2018), these architectural features are particularly beneficial for

SNN simulations with STDP where a large amount of floating point computation is required to update the synaptic state *and* additional integer arithmetic is required to calculate the indices into the sparse matrix data structures. Furthermore, due to the additional synchronisation overheads caused by the Windows Display Driver Model (WDDM) which we discussed in the previous section, offloading stimuli delivery to the GPU improves the performance significantly on the Windows machine.

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

4 DISCUSSION

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

Uniquely, the new interface provides access to all the features of GeNN, without leaving the comparative simplicity of Python and with, as we have shown, typically negligible overheads from the Python bindings. PyGeNN also allows bespoke neuron and synapse models to be defined from within Python, making PyGeNN much more flexible and broadly applicable than, for instance, the Python interface to NEST (Eppler et al., 2009) or the PyNN model description language used to expose CARLsim to Python (Balaji et al., 2020).

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

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

The new interface also opens up opportunities to support researchers that work with other Python based systems. In the Computational Neuroscience and Neuromorphic computing communities, we can now build a PyNN (Davison et al., 2008) interface on top of PyGeNN and, infact, a prototype of such an interface is in development. Furthermore, for the burgeoning spike-based machine learning community, we can use

529 PyGeNN as the basis for a spike-based machine learning framework akin to TensorFlow or PyTorch for
530 rate-based models. A prototype interface of this sort called mlGeNN is in development and close to release.

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

535 • do we need to discuss the wide variety of uses, i.e. MC versus Pavlovian demonstrated in this paper?

536 • Turing architecture is great for GeNN! Presented results improve on state-of-the-art.

537 • PyGeNN as an intermediate layer - PyNN, ML

538 • Cost of C++ - Python calls in models

539 • something about neuromorphic systems often being real-time / BS accelerated time

CONFLICT OF INTEREST STATEMENT

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

AUTHOR CONTRIBUTIONS

542 JK and TN wrote the paper. TN is the original developer of GeNN. AK was the original developer of
543 PyGeNN. JK is currently the primary developer of both GeNN and PyGeNN and was responsible for
544 implementing the spike recording system. JK performed the experiments and the analysis of the results that
545 are presented in this work.

FUNDING

ACKNOWLEDGMENTS

548 This is a short text to acknowledge the contributions of specific colleagues, institutions, or agencies that
549 aided the efforts of the authors.

DATA AVAILABILITY STATEMENT

550 All models, data and analysis scripts used for this study can be found in `https://github.com/`
551 `BrainsOnBoard/pygenn_paper`.

REFERENCES

552 Akar, N. A., Cumming, B., Karakasis, V., Kusters, A., Klijn, W., Peyser, A., et al. (2019). Arbor — A
553   Morphologically-Detailed Neural Network Simulation Library for Contemporary High-Performance
554   Computing Architectures. In *2019 27th Euromicro International Conference on Parallel, Distributed*
555   *and Network-Based Processing (PDP)* (IEEE), 274–282. doi:10.1109/EMPDP.2019.8671560

556   [Dataset] Apache Crail (2019). Crail Python API: Python -> C/C++ call overhead

557   Balaji, A., Adiraju, P., Kashyap, H. J., Das, A., Krichmar, J. L., Dutt, N. D., et al. (2020). PyCARL: A
558   PyNN Interface for Hardware-Software Co-Simulation of Spiking Neural Network

559   Beazley, D. M. (1996). Using SWIG to control, prototype, and debug C programs with Python. In *Proc.*
560   *4th Int. Python Conf*

561   Buzsáki, G. and Mizuseki, K. (2014). The log-dynamic brain: how skewed distributions affect network
562   operations. *Nature reviews. Neuroscience* 15, 264–78. doi:10.1038/nrn3687

563   Carnevale, N. T. and Hines, M. L. (2006). *The NEURON book* (Cambridge University Press)

564   Chou, T.-s., Kashyap, H. J., Xing, J., Listopad, S., Rounds, E. L., Beyeler, M., et al. (2018). CARLsim 4:
565   An Open Source Library for Large Scale, Biologically Detailed Spiking Neural Network Simulation
566   using Heterogeneous Clusters. In *2018 International Joint Conference on Neural Networks (IJCNN)*
567   (IEEE), 1–8. doi:10.1109/IJCNN.2018.8489326

568   Davison, A. P., Brüderle, D., Eppler, J., Kremkow, J., Muller, E., Pecevski, D., et al. (2008). PyNN: A
569   Common Interface for Neuronal Network Simulators. *Frontiers in neuroinformatics* 2, 11. doi:10.3389/
570   neuro.11.011.2008

571   Eppler, J. M., Helias, M., Muller, E., Diesmann, M., and Gewaltig, M. O. (2009). PyNEST: A convenient
572   interface to the NEST simulator. *Frontiers in Neuroinformatics* 2, 1–12. doi:10.3389/neuro.11.012.2008

573   Gewaltig, M.-O. and Diesmann, M. (2007). NEST (NEural Simulation Tool). *Scholarpedia* 2, 1430

574   Givon, L. E. and Lazar, A. A. (2016). Neurokernel: An open source platform for emulating the fruit fly
575   brain. *PLOS ONE* 11, 1–25. doi:10.1371/journal.pone.0146581

576   Golosio, B., Tiddia, G., De Luca, C., Pastorelli, E., Simula, F., and Paolucci, P. S. (2020). A new GPU
577   library for fast simulation of large-scale networks of spiking neurons , 1–27

578   Hines, M. L., Davison, A. P., and Muller, E. (2009). NEURON and Python. *Frontiers in Neuroinformatics*
579   3, 1–12. doi:10.3389/neuro.11.001.2009

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

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

584   Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering* 9,
585   90–95. doi:10.1109/MCSE.2007.55

586   Izhikevich, E. M. (2003). Simple model of spiking neurons. *IEEE Transactions on neural networks* 14,
587   1569–72. doi:10.1109/TNN.2003.820440

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

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

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

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

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

599   NVIDIA, Vingelmann, P., and Fitzek, F. H. (2020). *CUDA, developer.nvidia.com/cuda-toolkit*

600   NVIDIA Corporation (2019). *cuRAND Library, docs.nvidia.com/cuda/pdf/CURAND_Library.pdf*

601 NVIDIA Corporation (2021). *CUDA C Programming Guide,*
602 *docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf*

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

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

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

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

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

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

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

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