Large-scale brain simulations on the desktop using procedural connectivity

James C Knight[a,1] **and Thomas Nowotny**[a]

[a]Centre for Computational Neuroscience and Robotics, School of Engineering and Informatics, University of Sussex, Brighton, United Kingdom

This manuscript was compiled on March 27, 2020

Large-scale simulations of spiking neural networks are important for improving our understanding of the dynamics and ultimately function of brains. However, even small mammals such as mice have approximately 1×10^{12} synaptic connections which are typically charaterized by at least one floating point value per synapse. This amounts to several terabytes of connection data – an unrealistic memory requirement for a single desktop machine. Simulations of large spiking neural networks are therefore typically executed on large distributed supercomputers. This is costly and limits large-scale modelling to a select few research groups with the appropriate resources. In this work, we describe extensions to GeNN – our GPU-based spiking neural network simulator – that enable it to 'procedurally' generate connectivity and synaptic weights 'on the go' as spikes are triggered, instead of storing and retrieving them from memory. We find that GPUs are well-suited to this approach because of their raw computational power, which due to memory bandwidth limitations is often under-utilised when simulating spiking neural networks. We demonstrate the value of our approach with a recent model of the Macaque visual cortex consisting of 4.13×10^6 neurons and 24.2×10^9 synapses. Using our new method, this model can be simulated on a single GPU. Our results match those obtained on a supercomputer and the simulation runs 35 % faster on a single high-end GPU than a previous simulation executed on over 1000 supercomputer nodes.

spiking neural networks | GPU | high-performance computing | brain simulation

The brain of a mouse has around 70×10^6 neurons, but this number is dwarfed by the 1×10^{12} (?) synapses which connect them. In computer simulations of spiking neural networks, propagating spikes through synapses involves reading a 'row' of synapses connecting a spiking presynaptic neuron to its postsynaptic partners and adding the 'weight' of each synapse in the row to a 'bin' containing the postsynatic neuron's input for the next simulation timestep. Typically, the information describing which neurons are connected by a synapse and with what conductance, is generated before a simulation is run and stored in large matrices in random access memory (RAM). This creates high memory requirements for large-scale brain models, so that they can typically only be simulated on large distributed computer systems using software such as NEST (?) or NEURON (?). By careful design, these simulators can keep the memory requirements for each node constant, even when a simulation is distributed across thousands of nodes (?). However, high performance computer systems are bulky, expensive and consume large amounts of power, meaning that they are typically shared resources that are only accessible to a limited number of researchers and for strongly time-limited investigations.

Neuromorphic systems (? ? ? ? ? ?) take inspiration from the brain and have been developed specifically for simulating large spiking neural networks. One particular relevant feature of the brain is that its memory elements – the synapses – are co-located with the computing elements – the neurons – throughout the entire system. In neuromorphic systems, this often translates to dedicating a large proportion of each chip to memory. However, while such on-chip memory is fast, it can only be fabricated at relatively low density meaning that many of these systems economize – either by reducing the maximum number of synapses per neuron to as few as 256 or by reducing the precision of the synaptic weights to 6 (?), 4 (?) or even 1 bit (? ?). Such strategies allow some classes of spiking neural networks to be simulated very efficiently, but reducing the degree of connectivity in large-scale brain simulations to fit within the constraints of current neuromorphic systems inevitably changes their dynamics (?). Unlike the majority of other neuromorphic systems, the SpiNNaker (?) neuromorphic super-computer is entirely programmable and combines a large amount of on-chip memory with external memories, distributed across the system for the storage of synaptic connectivity. SpiNNaker's external memory bandwidth, on-chip memory capacity and the computational power of each core are all tailored to large-scale brain simulation meaning that the output bins of the synapse processing algorithm can fit in on-chip memory and there is enough external memory bandwidth to fetch synaptic rows fast enough for real-time simulation of large-scale models (?). This is a promising approach for future research but, because of its prototype nature, the availability of SpiNNaker hardware is limited and a physically large system is still required for even moderately-sized simulations (9 boards for a simulation with around 10×10^3 neurons and 300×10^6 synapses (?)).

Modern GPUs have relatively small amounts of on-chip memory and, instead, dedicate the majority of their silicon area to arithmetic logic units (ALUs). GPUs use dedictated

Significance Statement

Authors must submit a 120-word maximum statement about the significance of their research paper written at a level understandable to an undergraduate educated scientist outside their field of speciality. The primary goal of the Significance Statement is to explain the relevance of the work in broad context to a broad readership. The Significance Statement appears in the paper itself and is required for all research papers.

J.K. and T.N. wrote the paper. T.N. is the original developer of GeNN. J.K. is currently the primary GeNN developer and was responsible for extending the code generation approach to the procedural simulation of synaptic connectivity. J.K. performed the experiments and the analysis of the results that are presented in this work.

The authors declare no conflict of interest.

[1]To whom correspondence should be addressed. E-mail: J.C.Knightsussex.ac.uk

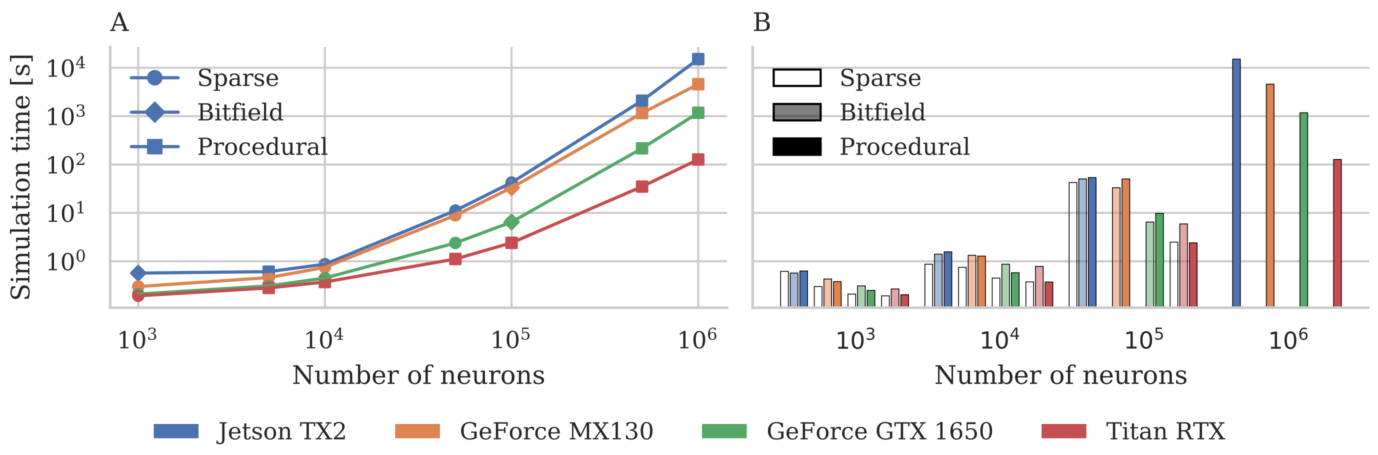

Fig. 1. Simulation time performance scaling on a range of modern GPUs (colors). **A** The best performing approach at each scale on each GPU (indicated by the symbols). For the largest models, the procedural method is always best. **B** Raw performance of each approach on each GPU. Missing bars indicate insufficient memory to simulate.

hardware to rapidly switch between tasks so that the latency of accessing external memory can be 'hidden' behind computation, as long as there is sufficient computation to be performed. For example, the memory latency of a typical modern GPU can be completely hidden if each CUDA core performs approximately 10 arithmetic operations per byte of data accessed from memory. Unfortunately, processing a synapse in a spiking neural network simulation is likely to require accessing approximately 8 B of memory and performing many fewer than the required 80 instructions. This makes synaptic updates highly memory bound. Nonetheless, we have shown in previous work (?) that, as GPUs have significantly higher total memory bandwidth than even the most expensive CPU, moderately sized models of around 10×10^3 neurons and 1×10^9 synapses can be simulated on a single GPU with competitive speed and energy requirements. However, individual GPUs do not have enough memory to simulate truly large-scale brain models and, although small numbers of GPUs can be connected together using the high-speed NVLink **(TODO: cite)** interconnect, beyond such small GPU clusters, scaling will be dictated by the same communication overheads as for other MPI-based distributed systems.

In this work we present a novel approach which converts large-scale brain simulation from a problem which is memory-bound on a GPU to one where the large amount of computational power available on a GPU can be used to reduce both memory and memory bandwidth requirements and enable truly large-scale brain simulations on a single GPU workstation.

Results

In the following subsections, we first present two recent innovations in our GeNN simulator (?) which allow to use it for simulating very large models on a single GPU. We then demonstrate the power of these new features by simulating a recent model of the Macaque visual cortex (?) consisting of 4.13×10^6 neurons and 24.2×10^9 synapses on a single GPU. We find that we not only obtain the same results as in a previous simulation on a high-performance supercomputer, but that our simulation also runs faster.

Procedural connectivity. The first crucial innovation to enable large scale simulations on a GPU is what we call 'procedural connectivity'. Our GeNN simulator (?) uses code generation to convert neuron and synapse models – described using 'snippets' of C-like code – into CUDA code for GPU simulation. We previously extended GeNN to allow the same approach to be used for generating efficient, parallel model initialisation code from code snippets describing state variable and synaptic connectivity initialisation algorithms (?). Offloading initialisation to the GPU in this way sped up model initialisation by around 20× on a desktop PC (?), demonstrating that initialisation algorithms are well-suited to GPU acceleration. **(TODO: something explaining why this is for synapses ... or not? (Thomas))** In fact, if each synapse can be re-initialised in less than the 80 operations required to hide the latency for fetching its parameter values from memory, it will be faster and vastly more memory efficient to regenerate synapses on demand and never store them in memory. This is the concept of procedural connectivity. Although a similar approach was used by Eugene Izhikevich for simulating an extremely large thalamo-cortical model with 1×10^{11} neurons and 1×10^{15} synapses on a modest PC cluster in 2005 **(TODO: cite)** – an incredible achievement – it has not been used on modern hardware since.

We implemented procedural connectivity as an option in GeNN by repurposing our previously developed parallel initialisation methods. Instead of being run once for all synapses at the beginning of the simulation, the methods are rerun during the simulation for each pre-synaptic spike until all postsynaptic targets have been identified and the post-synaptic code can be run. For instance, when connecting two populations of neurons with a fixed connections probability P_{conn}, the postsynaptic targets of a presynaptic neuron can be modelled as a Bernoulli process with success probability P_{conn}. An instance of the Bernoulli process can be generated by repeatedly drawing from the uniform distribution $\mathrm{Unif}[0,1]$ and comparing each sample to P_{conn}, but this is innefficient for sparse connectivity. Instead we can sample from the geometric distribution $\mathrm{Geom}[P_{\mathrm{conn}}]$ which is the distribution of the number of Bernoulli trials required to get a success (i.e. a synapse). The geometric distribution can be sampled in constant time by inverting the cumulative density function (CDF) of the equivalent continuous distribution (the exponential distribution) to obtain $\frac{log(\mathrm{Unif}[0,1])}{log(1-P_{\mathrm{conn}})}$ (? , p499). Therefore, as long as one has

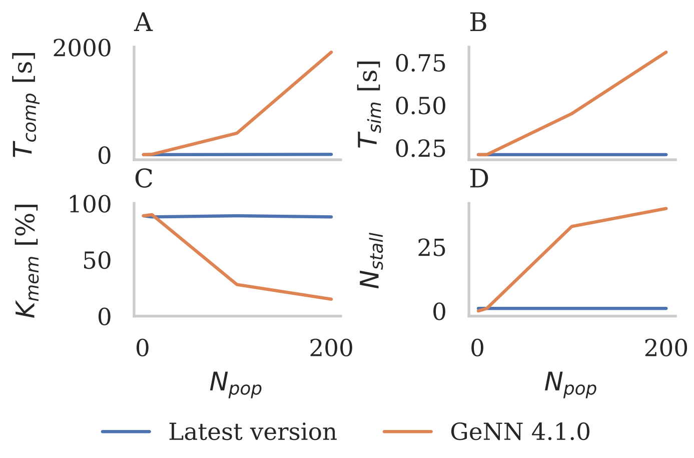

Fig. 2. Performance of a simulation of 1 000 000 LIF neurons driven by a gaussian input current, partitioned into varying numbers (N_{pop}) of populations and running on a workstation equipped with a Titan RTX GPU. **A** Compilation time (T_{comp}) using GCC 7.5.0. **B** Simulation time (T_{sim}) for an 1 s simulation. **C** Memory throughput (K_{mem}) reported by NVIDIA Nsight compute profiler 'Speed of light' metric. **D** Number of 'No instruction' stalls reported by NVIDIA Nsight compute profiler (N_{stall}).

the ability to generate a unique but repeatable stream of random numbers for each presynaptic neuron, the postsynaptic targets of a presynaptic neuron can be 'procedurally' generated in parallel. **(TODO: This is confusing to me: Do you mean different pre-synaptic neurons can dice out their connections in parallel but each presyn neuron must do it in a loop, right? Am I missing something?)** While suitable random number streams *could* be provided by a 'convential' random number generator (RNG), each presynaptic neurons would need to maintain its own RNG state which would have a significant memory overhead. Instead, we use a 'counter-based' Philox4×32-10 RNG (?). Counter-based RNGs are designed for parallel applications and essentially consist of a pseudo-random bijective function which takes a counter as an input (in this case a 128 bit number) and outputs random numbers. In constrast to convential RNGs, this means that generating the n^{th} random number in a stream has exactly the same cost as generating the 'next' random number, allowing us to trivially divide up the random number stream between multiple parallel processes (in this case presynaptic neurons). **(TODO: do we need some more explanation of how you get from this to a network simulation?)**

For an initial demonstration of the performance and scalability of procedural connectivity, we used a network that was initially designed to investigate signal propagation through cortical networks (?), but subsequently has been widely used as a scalable benchmark (?). The network consists of N integrate-and-fire neurons, partitioned into $\frac{4N}{5}$ excitatory and $\frac{N}{5}$ inhibitory neurons. The two populations of neurons are connected to each other and with themselves with a fixed $P_{\text{conn}} = 10\,\%$ connection probability.

We ran simulations of this network at scales ranging from 1×10^3 to 1×10^6 neurons (corresponding to 100×10^3 and 100×10^9 synapses respectively) on a selection of modern NVIDIA GPU hardware:

Jetson TX2 a low-power embedded system designed for robotic applications with 8 GB of shared memory
Geforce MX130 a laptop GPU with 2 GB of dedicated memory
Geforce GTX 1650 a low-end desktop GPU with 4 GB of dedicated memory
Titan RTX a high-end workstation GPU with 24 GB of dedicated memory

In Fig. 1 we compare the duration of these simulations using our new procedural approach against the standard approach of storing synaptic connections in memory using two different data structures. Both data structures are described in more detail in our previous work (?) but briefly, in the 'sparse' data structure, a presynaptic neuron's postsynaptic targets are represented as a sorted array of indices whereas, in the 'bitfield' data structure, they are represented as a $N_{\text{pre}} \times N_{\text{post}}$ array of bits where '1' indicates the existence of a synapse and '0' its absence. None of the devices have enough memory to store the 100×10^9 synapses required for the largest scale using either data structure but, at the 100×10^3 neuron scale, the bitfield data structure allows the model to fit into the memory of several devices it otherwise would not. However, not only is the new procedural approach the *only* way of simulating models at the largest scales but, as Fig. 1 illustrates, even at smaller scales its performance is competitive with and sometime better than the standard approach. **(TODO: note that this is using synapses with fixed synaptic weight)**

Kernel merging. The second innovation we named "kernel merging" and relates to the way code is organised into CUDA kernel functions. While the procedural connectivity approach presented in the previous section allows us to simulate models which would otherwise not fit within the memory of a single GPU, there are additional problems when using code generation to generate simulation code for models with a large number of neuron and synapse populations.

GeNN and – to the best of our knowledge (?) – all other SNN simulators which use code generation to generate all of their simulation code (as opposed to, for example NESTML (?), which uses code generation only to generate neuron simulation code) generate seperate pieces of code to simulate each population of neurons and synapses. This approach allows optimizations such as the hard-coding of constant parameters to be easily performed and, although generating code for models with many populations will result in large code size, C++ CPU code can be easily divided between multiple modules and compiled in parallel, minimizing the effect on build time. However, GPUs can only run a small number of kernels – which are equivalent to modules in this context – simultaneously (128 on the latest NVIDIA GPUs **(TODO: cite)**). Therefore, in GeNN, multiple neuron populations are simulated within each kernel, resulting in code of the form shown in the following pseudocode which illustrates how 3 populations of 1000 neurons could be simulated in a single kernel: **(TODO: very minimal sentance about SIMT here)**

```
void updateNeurons()
{
  if(thread < 1000) {
    // Update neuron population A
  }
  else if(thread >= 1000 && thread < 2000) {
    // Update neuron population B
  }
```

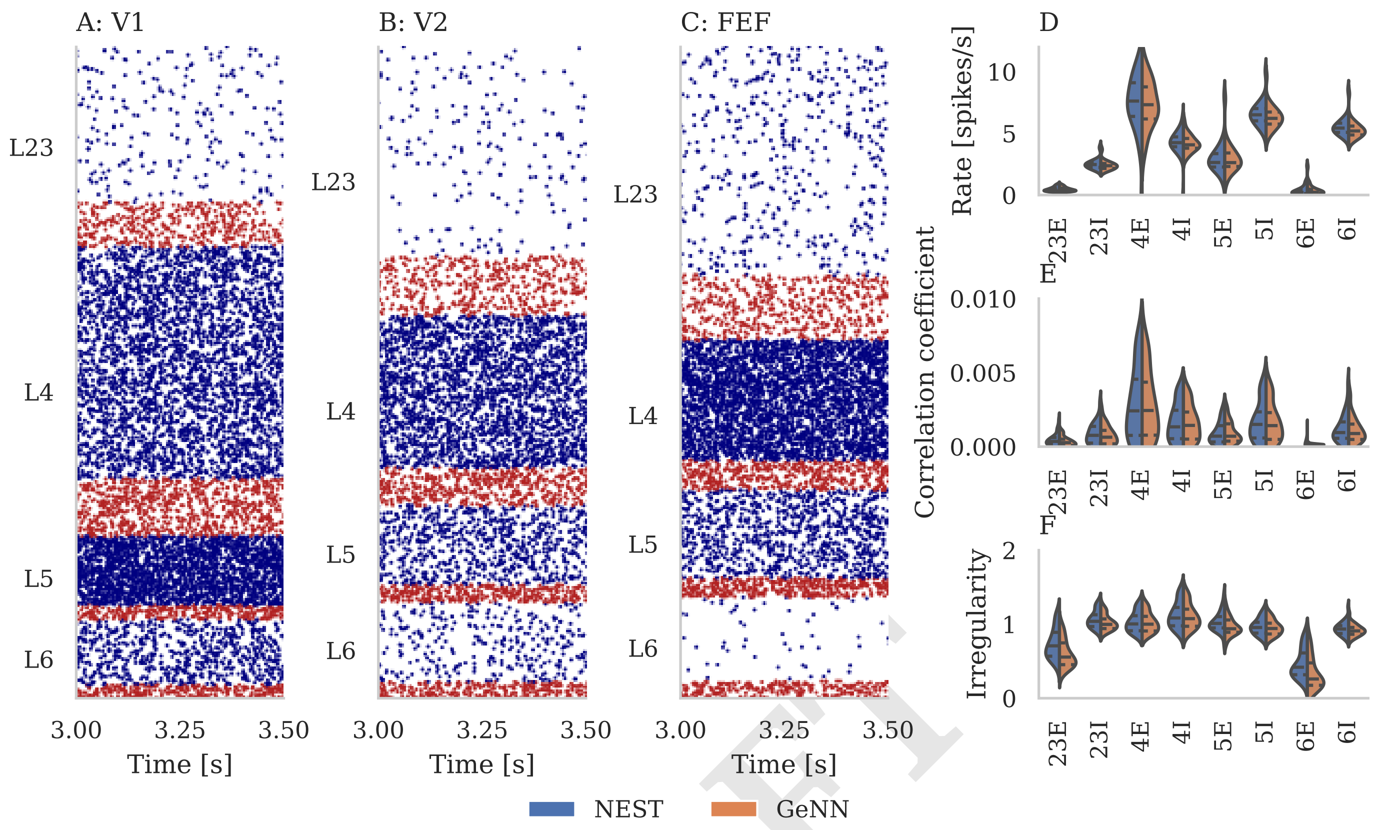

Fig. 3. Results of full-scale multi-area model simulation. **A-C** Raster plots of spiking activity of 3 % of the neurons in area V1 **A**, V2 **B**, and FEF **C**. Blue: excitatory neurons, red: inhibitory neurons. **D-F** Spiking statistics for each population across all 32 areas simulated using GeNN and NEST shown as split violin plots. Solid lines: medians, Dashed lines: Interquartile range (IQR). **D** Population-averaged firing rates. **E** Average pairwise correlation coefficients ofspiking activity. **F** Irregularity measured by revised local variation LvR (?) averaged across neurons.

```
  else if(thread >= 2000 && thread < 3000) {
    // Update neuron population C
  }
}
```

This approach works well for models with small numbers of populations but, as Fig. 2A illustrates, when we partition a model consisting of 1 000 000 LIF neurons into a large number of populations, which increases the size of the neuron kernel, compilation times increases super-linearly – quickly becoming impractical. Furthermore, Fig. 2B shows that when the model is partitioned into a large number of populations, the simulation also runs much more slowly. We would expect this model to be memory bound as each thread in the model reads 32 B of data and, as we discussed previously, hiding the latency of these memory accesses would require approximately 320 arithmetic operations which is many more than are required to sample from the uniform distribution and update a LIF neuron. Fig. 2C – obtained using data from the NVIDIA Nsight compute profiler **(TODO: cite)** – shows that this to be true with the memory system being around 90 % utilised for small numbers of populations. However, when the model is partitioned into a large number of populations, the kernel stops being able to efficiently use the memory and becomes latency bound (neither memory *nor* compute are used efficiently). Investigating further using the profiler showed that this drop in performance was accompanied by an increasing number of “No instruction” stalls as shown in Fig. 2D. Stalls are events which prevent the GPU from doing any work during a clock cycle and the profiler documentation suggests that these particular events are likely to be caused by “Excessively jumping across large blocks of assembly code”**(TODO: cite)** – which makes sense when we are generating kernels with hundreds of thousands of lines of code. **(TODO: is more detail required here as to why?)**

To address these issues, we developed a new code generator for GeNN which first ‘merges’ the model description, grouping populations together which can be simulated using the same generated code. From this merged description, structures are generated to store the pointers to state variables and parameters which can differ between merged populations:

```
struct NeuronUpdateGroup
{
  unsigned int numNeurons;
  float* V;
};
```

An array of these structures is then declared for each merged population and each element is initialised with pointers to state variables and parameter values:

```
NeuronUpdateGroup neuronUpdateGroup[3];
neuronUpdateGroup[0] = {1000, VA};
neuronUpdateGroup[1] = {1000, VB};
neuronUpdateGroup[2] = {1000, VC};
```

In order for a thread to determine which neuron in which population it should be simulating, we generate an additional data structure – an array containing a cumulative sum of threads used for each population. Each thread performs a simple binary search within this to find the index of the neuron and population it should simulate:

```
unsigned int startThread[3] = {0, 1000, 2000};
void updateNeurons()
{
  if(thread < 3000) {
    // Binary search startThread to determing
    // which population thread should be
    // processed. Then update using variables
    // in neuronUpdateGroup
  }
}
```

As Fig. 2 shows, this approach solves the issues with compilation time and simulation performance caused by large numbers of populations. Therefore, we apply this approach to initialisation and simulation kernels for both neuron and synapse populations.

The multi-area model. Due to lack of computing power and sufficiently detailed connectivity data, previous models of the cortex have either focussed on modelling individual local microcircuits at the level of individual cells (? ?) or modelling multiple connected areas at a higher level of abstraction where entire ensembles of neurons are described by a small number of differential equations **(TODO: find citation)**. However, data from several species **(TODO: find citation)** has shown that cortical activity has distinct features at both the global and local levels which can only be captured by modelling interconnected microcircuits at the level of individual cells. The recent multi-area model (? ?) is an example of such multi-scale modeling – using scaled versions of a previous 4 layer microcircuit model (?) to implement $1\,\mathrm{mm}^2$ 'patches' for each of 32 areas of the macaque cortex involved in visual processing. The 32 areas are connected together with connectivity based on inter-area axon tracing data from the CoCoMac (?) database, further refined using additional anatomical data (?) and heuristics (?) to obtain estimates for the number of synapses between areas. These synapses are distributed between populations in the source and target area using layer-specific tracing data (?) and cell-type-specific dendritic densities (?). For a full description of the construction of the multi-area model please refer to the original works (? ?). This model was simulated using NEST (?) on a rack of an IBM Blue Gene/Q supercomputer (a 2 m high enclosure containing 1024 compute nodes and weighing over 2 t). On this system, initialization of the model took around 5 min and simulating 1 s of biological time took approximately 12 min (?).

The multi-area model consists of 4.13×10^6 neurons split into 254 populations and 24.2×10^9 synapses split into 64 516 populations meaning that, without the kernel merging approach presented earlier in this work, the model would be unlikely to compile or simulate at a reasonable speed using GeNN. Additionally, each synapse in this model has an independant weight and synaptic delay sampled from a normal distribution, meaning that the bitfield data structure cannot be used to represent the connectivity. Even if we assume that 16 bit floating point would provide sufficient weight precision, delays could be expressed as an 8 bit integer and that the neuron populations are all small enough to index using 16 bit indices, our sparse data structure would still require 5 B per synapse, meaning that this model's synaptic data would require over 100 GB of GPU memory. While a cluster of GPUs connected using NVLink could be built with this much memory, it is more than any single GPU has available.

In this model, the density of the synaptic connections between pairs of neuronal populations is described as a total number (N_{syn}) of random synapses. In order to use our procedural approach with this connectivity, the subset of the N_{syn} synapses which end up in each row must be calculated by sampling from the multinomial distribution $\text{Mult}[N_{\text{syn}}, \{P_{row}, P_{row}, \ldots, P_{row}\}]$ where $P_{row} = \frac{N_{\text{post}}}{N_{\text{pre}} N_{\text{post}}} = \frac{1}{N_{\text{pre}}}$. This operation cannot be efficiently parallelised so we perform it on the host, storing the results in a GPU memory array. However, once the length of each row is determined, the postsynaptic targets for a presynaptic neuron can be procedurally generated very efficiently by sampling from the discrete uniform distribution $\text{Unif}[0, N_{\text{post}}]$. The weights and delays for each synapse can then be sampled from the normal distribution using the same repeatable random number stream used to sample the postsynaptic indices.

In order to validate our GeNN simulations of the multi-area model, we ran a 10.5 s simulation of the model in the ground state on a workstation with a single Titan RTX GPU. Initialization of our model took 6 min – 3 min of which was spent generating and compiling code – and each biological second of simulation took 7.7 min – 35 % faster than the supercomputer simulation.**(TODO: is this the best way of expressing this?)** Fig. 3A-C shows some example spike rasters from three of the modelled areas, illustrating the asynchronous irregular nature of the model's ground state. Next, we calculated the per-layer distributions of rates, spike-train irregularity and cross-correlation coefficients across all areas (disregarding the first 500 ms of simulation) and compared them to the values of the exact same measures obtained from the supercomputer simulations. We calculated irregularity using the revised local variation LvR (?), averaged over a subsample of 2000 neurons and cross-correlation from spike histograms with 1 ms bins, calculated from a subset of 2000 non-silent neurons. The violin plot in Fig. 3D-F shows the comparison of the distributions of values obtained from the two simulations, which are essentially identical.

Discussion

In this work we have presented a novel approach for large-scale brain simulation on GPU devices which entirely removes the need to store connectivity data in memory. We have shown that this approach allows us to simulate a cortical model with 4.13×10^6 neurons and 24.2×10^9 synapses (? ?). While this represents a significant step forward in terms of making truly large-scale brain modelling tools accesible to individual researchers, this model still has around 20× fewer neurons and 40× fewer synapses than the brain of even a small mammal such as a mouse (?). Our implementation of the multi-area model requires a little over 12 GB of GPU memory in total, with the majority (8.5 GB) being used for the implementation of dendritic delay ring buffers (described in more detail in our previous work (?)). While these are a per-neuron (rather than per-synapse) data structure, because the inter-area connections in the model have delays up to around 50 ms, the delay buffers become extremely large. Additionally, as these buffers are accumulators for synaptic inputs, reducing the precision of thee **(TODO: something missing here)**

One important aspect of large-scale brain simulations that we have not addressed in this work is synaptic plasticity and its role in learning. As discussed in our previous work (**?**), GeNN has support for a wide variety of synaptic plasticity rules. While, in order to modify synaptic weights, they need to be in memory rather than generated procedurally, connectivity could still be generated procedurally, potentially halving the memory requirements of models with synaptic plasticity. However, while this would be sufficient for many synaptic plasticity rules that only require access to presynaptic spikes and postsynaptic neuron states such as membrane voltage (**? ?**), many Spike-Timing-Dependent Plasticity (STDP) rules also require access to *postsynaptic* spikes. GeNN supports such rules by automatically generating a suitable lookup table structure (see our previous work (**?**) for more details) and this process could be adapted to generate a lookup table from procedural connectivity but this would further erode the memory savings that procedural connectivity would result in.

In this work, we have discussed the idea of procedural connectivity purely in the context of GPU hardware. However, we believe that there is also the potential to develop a new type of neuromorphic hardware, built from the ground up for procedural connectivity with key components such as the counting random number generator implemented in hardware.

Materials and Methods

Neuron models. The membrane voltage (V_j) of each neuron is modelled as a leaky integrate-and-fire (LIF) unit:

$$\tau_m \frac{dV_j}{dt} = (V_j - V_{rest}) + R_m I_{in_j} \qquad [1]$$

where τ_m and R_m represent the time constant and resistance of the neuron's cell membrane, V_{rest} defines the membrane voltage the neuron returns to if it receives no synaptic input and I_{in_j} represents the input current to the neuron. When the membrane voltage crosses a threshold (V_{thresh}) a spike is emitted, the membrane voltage is reset back to V_{rest} and a countdown timer is started which, while running, disables the integration of further input thus providing a simulated refractory period. Incoming spikes induce an exponentially-shaped input current in I_{in_j}:

$$\tau_{syn} \frac{dI_{in_j}}{dt} = -I_{in_j} + I_{p_j} + \sum_{i=0}^{n} w_{ij} \sum_{t_i^f} \delta(t - t_i^f) \qquad [2]$$

where τ_{syn} represents the time constant with which any spikes (modelled as Dirac delta functions δ) from n presynaptic input neurons occuring at time t are integrated. In addition to its synaptic input, each neuron in the network also receives an independent Poisson input current I_{p_j} (also exponentially shaped by equation 2) which represents input from adjacent cortical regions.

ACKNOWLEDGMENTS. We like to thank Jari Pronold, Sacha van Albada and Maximilian Schmidt for their valuable assistance with using the data and analysis tools published alongside the multi-area model – without these contributions, our comparison with these results would not have been possible.