Title: NeuroGPU, software for NEURON modeling in GPU-based hardware

Authors: Roy Ben-Shalom[1,2], Nikhil S. Artherya[3], Christopher Cross[1?], Hersh Sanghevi[3], Alon Korngreen[4,5], Kevin J. Bender[1,2].

Affiliations:

1. Weill Institute for Neurosciences, University of California, San Francisco, San Francisco CA, USA
2. Department of Neurology, University of California, San Francisco, San Francisco, CA, USA
3. Department of Electrical Engineering and Computer Science, University of California, Berkeley, Berkeley CA, USA
4. The Leslie and Susan Gonda Multidisciplinary Brain Research Center, Bar-Ilan University, Ramat-Gan Israel
5. The Mina and Everard Goodman Faculty of Life Sciences, Bar-Ilan University, Ramat-Gan Israel

Lead contact: roy.benshalom@ucsf.edu

Abstract:

Generating biologically detailed models of neurons is an important goal for modern neuroscience. Unfortunately, constraining parameters within biologically detailed models can be difficult, leading to poor model predictions, especially if such models are extended beyond the specific problems for which they were designed. This major obstacle can be partially overcome by numerical optimization and detailed exploration of parameter space. These processes, which currently rely on central processing unit (CPU) computation, are computationally demanding, often with exponential increases in computing time and cost for marginal improvements in model behavior. As a result, models are often compromised in scale given available CPU-based resources. Here, we present a simulation environment, NeuroGPU, that takes advantage of the inherent parallelized structure of graphics processing unit (GPU) to accelerate neuronal simulation. NeuroGPU can simulate most of biologically detailed models from commonly used databases 1-2 orders of magnitude faster than traditional single core CPU processors, even when implemented on relatively inexpensive GPU systems. Thus, NeuroGPU offers the ability to apply compartmental, biologically detailed, modeling approaches with supercomputer-level speed at substantially reduced cost.

Introduction:

Detailed numerical models, faithfully capturing neuronal complexity, are invaluable for simulating the behavior of realistic neural networks (Einevoll et al., 2019). Generating models that accurately recapitulate neuronal activity often requires one to tune individual model parameters. This process can be aided by iterative rounds of parameter exploration and optimization that aim to minimize the differences between empirical data targets and their associated models. These processes can be computationally demanding. Indeed, each linear improvement in model accuracy requires an exponential increase in computational resources (Nocedal and Wright, 2006; Gurkiewicz and Korngreen, 2007) Thus, model optimization is often done on supercomputers that parallelize these computations across central processing unit (CPU) clusters. Unfortunately, due to the cost of constructing and running such supercomputing clusters, such efforts are typically restricted to large consortia, such as the Blue Brain Project (BBP) (Markram et al., 2015) and the Allen Institute (Gouwens et al., 2018). For more restricted budgets, simulations must typically be compromised in scale or complexity to produce results within budget and within a reasonable time frame.

In the past 10 years, graphical processing units (GPUs) have emerged as an alternative to CPU-based clusters that may offer comparable levels of performance at substantially reduced cost. GPUs utilize streaming multiprocessors with multiple simple cores that allow for distributed, parallelized computing. With software optimized for distributed computing, GPU-based applications can often outperform CPU-based applications in processing speed and cost (Payne et al., 2010). Today, GPUs are widely used in scientific fields like molecular dynamics (Go et al., 2012; Salomon-Ferrer et al., 2013) and climate modeling (Prein et al., 2015), and are the computational engine for most modern artificial intelligence applications (Schmidhuber, 2015). In neuroscience, GPUs are currently being used to accelerate complex imaging dataset processing (Eklund et al., 2013), spiking neural network analysis (Fidjeland and Shanahan, 2010), and clustering of activity from *in vivo* extracellular electrophysiological experiments (Pachitariu et al., 2016). Despite these advances until recently that two frameworks for simulating biophysical neuronal networks (Akar et al., 2019; Kumbhar et al., 2019) were published, relatively little effort has been made to leverage GPU-based architecture for biophysically detailed neuronal simulation.

Here, we describe NeuroGPU, a computational platform optimized to exploit GPU architecture to dramatically accelerate simulation of neuronal compartmental models. To do so, we

developed new approaches to parallelize compartmental models, utilizing the GPU-based programming language CUDA to optimize memory handling on GPUs. This resulted in simulation speedups of up to 200-fold on a single GPU and up to 800-fold using a set of 4 GPUs. Building on our previous efforts (Ben-Shalom et al., 2013), we developed an intuitive user interface that can import most compartmental models in ModelDB or the BBP portal. Further, we provide methods to explore model parameter space and to optimize models with evolutionary algorithms (DEAP). NeuroGPU therefore provides an open-source platform useful for neuronal simulation with increased speed and reduced cost.

Results:

General overview

Our primary goal for NeuroGPU was to improve compartmental modeling speed with relatively low-cost hardware and provide an interface to port NEURON models from public databases such as ModelDB and the BBP portal. Toward that end, we utilized the same basic structure as NEURON, including the use of hoc and mod files that define all aspects of the compartmental model. To increase simulation speed, we focused primarily on parallelizing the most computationally intensive aspect of NEURON simulations in GPU architecture. NEURON calculates the voltages of each segment of the model by solving a system of differential equations that describes current flow in each compartment. Within NEURON, this differential equation system is represented within a tri-diagonal matrix (Hines, 1984). Typically, matrix elements for neighboring compartments are solved in serial, as current flow in one compartment is interdependent on flow in neighboring compartments. We and others have previously developed methods to solve this tri-diagonal matrix in parallel across GPUs, despite the interdependence of current flow across compartments (Hines et al., 2008, n.d.; Ben-Shalom et al., 2013). At that time, the method was implemented only for Hodgkin-Huxley models (Ben-Shalom et al., 2013). Here, we extended this method to support a wider range of models, including most models available in ModelDB and the Blue Brain Project (BBP) repository. This is implemented in Python, with an iPython Graphical User Interface (GUI).

Porting NEURON simulations to NeuroGPU

NeuroGPU simulations begin by importing NEURON's mod and hoc files via a GUI developed in iPython notebook (Figure 1). The user must input a file containing model stimulation, which includes temporal aspects of the model and command currents delivered at a prescribed location. Furthermore, all free parameters, such as channel properties, must be described (Fig. 1B). These import components are translated into CUDA code, termed kernels, that can run on the GPU via the python script "extractmodel.py" (Fig. 1C). This script first takes runModel.hoc and loads it into NEURON, not to run simulations, but rather to query NEURON for model properties needed for subsequent porting to NeuroGPU, including compartment names and the tri-diagonal matrix (F-Matrix). Then, the script iterates over the .mod files in the directory, parses them and creates relevant kernels for each mechanism described. Mechanism kernels are written to the AllModels.cu in similar structure as described previously (Hines and Carnevale,

2000; Carnevale and Hines, 2006), iterating over all compartments defined in the model. A new hoc file is created to register mechanism values, which are stored in AllParams.csv and inserted in each compartment. Finally, the script writes code translated to CUDA in NeuroGPU.cu and packages the application to run on either Windows or Unix. After compiling the code, an executable is created that reads the AllParams.csv and the stimulation and runs the model on the GPU.

NeuroGPU implementation

We used CUDA to implement NEURON-based modeling using GPUs. CUDA is an extension of the C programming language that enables computation on the GPU (Nvidia, 2018). CUDA kernels which are procedures running on the GPU can be invoked from either the GPU or CPU. To invoke a kernel from the CPU, one must specify the number of parallel threads used. Threads, which allow for parallelization on the GPU, are organized into blocks, with each thread occupying a specific address within that block (idx.x, idx.y). GPUs are structured to operate well when computing 32 parallel threads, a computing structure termed a warp (Nvidia, 2018). Therefore, we structured NeuroGPU to utilize 32 threads in the x dimension, corresponding to individual morphological segments within the model. For a given model with more than 32 segments, individual threads are responsible for calculating every 32nd segment. For example, thread #1 would calculate segments 1, 33, 65, … 31N+1.

Complex neuronal models, including many described in the BBP (Hay et al., 2013; Ramaswamy et al., 2015), are memory intensive. GPUs have several forms of memory that have tradeoffs in terms of their size and relative speed that make them ideal for certain aspects of model processing and impractical for others. GLOBAL memory is the largest physical memory space available on the GPU but is relatively slow. Here, we use GLOBAL memory to store the largest data structures associated with a given model, in part because they simply cannot be held by other memory structures. SHARED memory is far faster, shared among the whole GPU block, but limited to 48 kilobytes. This makes it ideal for storing the tridiagonal matrix, as this matrix is the most accessed data structure within NeuroGPU. CONSTANT memory, which is a 64 kilobyte block of fast, read-only memory, is used to hold constant data structures, including the order in which the tri-diagonal matrix is solved in parallel (Ben-Shalom et al., 2013). Lastly, REGISTER\LOCAL memory is the fastest memory available on the GPU but is limited to maximum of 63 registers per thread and a total of 16 kilobytes of memory shared across the entire block. It is used to store local variables necessary for the course of the simulation.

To determine how best to utilize GPU parallel processing, we examined two ways in which to simulate compartmental models on the GPU. In both cases, the GPU is responsible for updating ionic currents from established mechanisms, solving the tridiagonal matrix, and updating model states and voltages at each time step. In the first configuration, termed SingleKernel, we computed all the time steps of each simulation in one kernel on the GPU, largely because this would limit the amount of time performing the relatively slow step of transferring memory between the GPU and CPU. In this case, the transfer is done only once and during the simulation the GPU communicates with the CPU only to transfer voltages reported at the recording electrode site. Alternatively, we also created a SplitKernel condition, in which the simulation is split into many small kernels that are invoked every single time step. Data are then registered back to the CPU and the next time step is run in serial. This approach may be advantageous if memory transfer between the GPU and CPU is not the rate-limiting step. Furthermore, in this case the GPU can also optimize computing timing by queueing certain steps for execution while other memory is being transferred. Both the SingleKernel and SplitKernal configuration were assessed in all cases reported below.

Benchmarking

To determine how NeuroGPU performs relative to NEURON, we benchmarked it for relative speed and accuracy across different conditions: CUDA implementation, hardware configurations and across a range of models. We first compared NeuroGPU performance with a single GPU to NEURON implemented on a single CPU core.

We began with a simple model of a soma and single dendritic branch that has 64 segments in total (Figure 1A), each containing a single external mechanism pas.mod that describes passive current flow. This model was stimulated with a simple current step (Fig. 1B). Voltage discrepancies that never exceeded 0.4 µV were observed between NeuroGPU and NEURON when simulation voltage changed rapidly. These discrepancies were due to small differences in timing that likely arise from how numbers are rounded in GPUs vs CPUs (Whitehead, 2011).

To benchmark relative speed, we evaluated computing time for multiple instances of the same model. NEURON computation speed scales linearly with the number of simulations, and, for low numbers of models (< 8), outperforms NeuroGPU. By contrast, models implemented on GPUs scale linearly only after saturating all streaming multiprocessors. With NeuroGPU, processing times are quite similar for any simulation incorporating fewer than 128 models, and begin to

outpace NEURON simulations when >32 simulations are run simultaneously. Relative gains in processing time were noted when 32 to 16,384 models were run simultaneously. These gains were dependent on hardware. For example, implementing NeuroGPU on an NVIDIA TitanXP GPU resulted in 25.2-fold improvements in processing speed, while the same models run on an NVIDIA Tesla V100 were 95.8-fold faster (both implemented in the "SingleKernel" configuration). It is worth noting that TitanXP hardware is relatively low cost (<$1099) and very similar card (NVIDIA GTX-1660) can currently be purchased for less than $300, suggesting that significant improvements in processing speed can be obtained even with modestly priced hardware.

More complex neuronal morphology could affect NeuroGPU processing speed. Therefore, we implemented the same passive mechanism on the more complex structure of a neocortical pyramidal neuron. While voltage discrepancies were similarly small in this instance (< $4x10^{-6}$ mV), the relative speedup was lower than with simpler morphology (TitanXP: 15.2x; Tesla V100: 58.1x). Thus, while morphology does affect relative speed, NeuroGPU still outperforms CPU-based modeling.

In addition to complex morphology, compartmental models typically contain an array of mechanisms that simulate voltage-gated channels or ligand-gated receptors. To assess NeuroGPU performance with such models, we began with a pyramidal model neuron first described by Mainen and Sejnowski (1996). This model has 7 different mechanisms, including voltage-gated sodium, potassium, and calcium channels, and a calcium-dependent potassium channel. As with the passive model described above, we implemented these mechanisms in both simple and complex morphologies (e.g., soma and primary dendrite alone, or complete pyramidal cell morphology). In models with simple morphology, NeuroGPU was 30.3x (TitanXP) or 153.1x (Tesla V100) faster than NEURON, with minimal voltage error (< 4 µV). In pyramidal cell morphology models, NeuroGPU was 45.3x (TitanXP) or 114.2x (Tesla V100) faster than NEURON. In this instance, we observed a relatively large voltage discrepancy of 6.6 mV. This discrepancy occurred during the last AP within a burst and was due largely to a shift in the timing of this AP (Fig. 4G). Indeed, we were able to reduce this error ~6x by interpolating the data and shifting the timing of this AP by ¼ of a timestep.

While the Mainen and Sejnowski model can generate physiologically-realistic spiking activity, these APs occur over a relatively narrow range of stimulus intensities. Outside this range the model is either subthreshold or enters depolarization block. As a result, we found this model to

be impractical for benchmarking NeuroGPU across a range of stimuli. Therefore, we tested NeuroGPU on more recently developed models from the Blue Brain Project portal. Here, we used two models: one of a layer 5 pyramidal neuron (BBP_PC, see Methods for specific model) and one of a layer 5 chandelier interneuron (BBP_CC). Models were interrogated with a range of stimulus intensities to determine relative differences between NeuroGPU and NEURON (Fig. 5). Similar to Mainen and Sejnowski, voltage differences were small (maximum differences: <0.2 mV) and were most commonly observed when voltage was changing markedly between time steps (Fig. 5C, G).

As with other models (Fig. 3, 4), implementing NeuroGPU on faster GPUs decreased processing time (Fig. 5D, H). Interestingly, CUDA has been recently updated to allow for memory sharing across GPUs, which could be leveraged to decrease processing time further. To test this, we connected up to 4 Tesla V100 GPUs together and measured speedup on both BBP models displayed in Figure 5. As expected, adding more GPUs increased the overall processing capacity, and we noted shifts in the number of models that could be handled simultaneously before reaching maximum GPU utilization (Fig. 6). Furthermore, speedup was almost 2 orders of magnitude faster relative to NEURON.

Profiling

To better understand why NeuroGPU accelerated some models more than others, we used the NVIDIA profiler to monitor GPU utilization. Further, we tested two different memory handling configurations—SingleKernel and SplitKernel—to determine how best to utilize GPU parallel processing. In both cases, the GPU is responsible for updating ionic currents from given mechanisms, solving the tridiagonal matrix, and updating model states and voltages at each time step.

We found that configuring NeuroGPU in SingleKernel mode produced the fastest runtimes in all models tested (Table 1), and had higher GPU utilization levels. This indicates that, for most models, memory transfer between GPU and CPU is rate-limiting, and models run most efficiently when the majority of calculations are isolated on the GPU. Nevertheless, the highest utilization values were ~10% in the SingleKernel configuration (3.8% in SplitKernel), suggesting that additional memory optimizations could be leveraged in future iterations of NeuroGPU.

Benefits of using NeuroGPU for parameter space exploration and genetic optimization

Neuronal simulations are often tested over a range of parameter values to both explore the range of output generated and to optimize models to best fit empirical data (Druckmann et al., 2007; Van Geit et al., 2008; Keren et al., 2009; Gouwens et al., 2018). These simulations essentially run the same model repeatedly with small differences in underlying parameters, making them ideal for parallelization with NeuroGPU. Indeed, relative speedups would be identical to situations considered above (Fig. 3-6) and depend simply on the number of parameter sets used. Based on this, we developed a GUI that streamlines parameter space exploration in NeuroGPU.

To provide an example of parameter space exploration, we examined neuronal output in the BBP_PC model when co-varying the density of the axonal fast inactivating sodium channel and axonal slow-inactivating potassium channel over a range of 0 to 10 and 0 to 20 S/cm^2, respectively. Total spike output and select single traces are shown in Figure 7. As expected, increasing sodium conductance allowed models to generate more APs until sodium conductance was so high that models entered depolarization block. Similarly, reducing potassium conductance produced comparable results. Interestingly, certain combinations of sodium and potassium conductance concentrations produced bursting phenotypes characterized by high-frequency APs riding atop long-duration depolarizations. These presumably reflect parameter ranges that then interact with other ion channels in the model (e.g., Ca_V3 channels) that promote such burst dynamics.

To implement genetic optimization within NeuroGPU, we integrated the DEAP (Distributed Evolutionary Algorithms in Python) package (Gagn, 2012). Genetic algorithm success lies in the balance between exploration of the whole parameter space and the exploitation of specific areas that seem promising. For this, large sample populations are ideal, as this allows for effective and broad parameter space exploration. NeuroGPU is more efficient when many instances are running in parallel, allowing for more effective application of genetic algorithms.

Genetic optimization was tested here by fitting model-generated voltages to a single voltage epoch containing APs that was generated by the default values present in the BBP_PC model. We then determined how close different optimization sets could come to identifying these original parameter values. Optimization began with different population sizes comprised of 100 to 10,000 individual parameter sets with random initial values (Fig. 8A). These populations were

run in four independent trials, each for 50 generations, and the difference between the naïve model and ground-truth model was compressed to a single score value (see Methods). For these scores, lower values indicate less difference between the two cases.

Scores improved for each of these populations, but the variance across trials and the overall score were markedly affected by the population size, with score decreasing in a near-linear fashion with each doubling of population size (Fig 8C). These score improvements were paralleled by a decrease in total processing time. For example, optimization with 10,000 individual parameter sets ran 7.7x faster on NeuroGPU than NEURON (Fig. 8D; 10 vs 77 hours, respectively). While these are significant improvements in simulation speed, they are relatively modest compared to those observed in other conditions (Fig. 5), likely because current versions of NeuroGPU require NEURON to load the simulation and generate parameter values. This step is currently done using the CPU. Whether it is possible to parallelize this step will be explored in future versions of NeuroGPU.

Discussion:

In this work, we implemented a simulation environment to run single neuron compartmental models on GPUs. Based on our previous efforts (Ben-Shalom et al., 2013), we designed a user-friendly environment that enables one to port multi-compartmental models for implementation with CUDA. NeuroGPU was developed to be interoperable with NEURON (Cannon et al., 2007), thereby allowing anyone with expertise in the NEURON environment access to GPU-based acceleration. Towards this goal, we developed a platform to easily port NEURON models from either ModelDB or the BBP portal (Ramaswamy et al., 2015; McDougal et al., 2017) using a iPython notebook-based graphical user interface (GUI). We further developed GUIs for creating stimulation protocols, parameter exploration, and genetic optimization. By taking advantage of parallel processing inherent to GPUs, we were able to accelerate simulations dramatically, in some cases by almost two orders of magnitude.

NeuroGPU accelerates compartmental modeling largely through parallelization of matrix calculations. Solving the tridiagonal matrix is the most computationally demanding aspect of compartmental model simulations (Hines, 1984; Hines et al., 2008; Ben-Shalom et al., 2013). Therefore, we took advantage of fast, on-GPU memory and controlled the timing of calculations and memory transfers to optimize the use of computational resources (Volkov and Demmel, 2008; Ben-Shalom et al., 2013; Nvidia, 2018). Resulting speedups depended primarily on neuronal morphology, and in general we found the NeuroGPU performed best when processing anatomically complex cases. Even in these cases, overall GPU utilization was limited by execution dependencies, where one aspect of GPU processing could not proceed until another aspect either transferred or processed its own memory. In the future, these dependencies may be further reduced through either dynamic parallelization (Zhang et al., 2015) or by increasing instruction level parallelism (ILP) (Volkov and Demmel, 2008). Nevertheless, the current version of NeuroGPU can still accelerate single neuron compartmental simulations by several orders of magnitude.

NeuroGPU addresses a major gap in currently implemented GPU-based simulation environments. In addition to NeuroGPU, two other neuronal simulations environments for multi-compartmental models have been implemented using GPUs, CoreNeuron (Hines et al., n.d.) and Arbor (Akar et al., 2019). Both of these environments are designed primarily to accelerate large scale network simulations. NeuroGPU, by contrast, is focused more on exploring the parameter space of single models and optimizing such models to best fit empirical data. As such, NeuroGPU has expanded GUIs for parameter exploration, which allows for quick

assessment of how changes in ion channel density across compartments affects neuronal excitability (Fig. 7). This approach may be particularly useful to generate testable hypotheses regarding channel distribution with pharmacological manipulations (Keren et al., 2009; Almog and Korngreen, 2014; Mäki-Marttunen et al., 2018), modulation of ion channels (Byczkowicz et al., n.d.), or in disease states where ion channel density is thought to be affected (Migliore and Migliore, 2012; Miceli et al., 2013; Ben-Shalom et al., 2017; Spratt et al., 2019). Furthermore, one could also generate a range of cells with variable channel densities and confirm that their activity is physiologically realistic (e.g., Fig. 7, all cases before generating depolarization block). These conditions could then be used as building blocks for variable activity within neuronal networks (Prinz et al., 2003, 2004; Alonso and Marder, 2019).

In addition to parameter exploration, NeuroGPU is designed for extensive model optimization using DEAP. Fitting computational models to empirical data is computationally taxing, and fits typically improve two-fold with each doubling of computational resources. Here, we found that NeuroGPU can accelerate DEAP processing times 8x (Fig. 8). Of note, these speedups compare single GPUs and CPUs. Leveraging multiple GPUs should accelerate this process further.

Future iterations of NeuroGPU may expand on the strengths and address limitations in using GPUs for compartmental modeling. Ion channels are modeled typically with Markov-based kinetics, or a simpler Markov approximation based on Hodgkin-Huxley type equations. NeuroGPU currently supports Hodgkin-Huxley-based mechanisms only, as we found that implementation of full Markov-based mechanisms on GPUs requires too much shared memory and reduces performance drastically (Ben-Shalom et al., 2012). As with total GPU utilization, improvements in memory handling may improve these cases. Furthermore, GPUs work best when the same instructions are occurring simultaneously on multiple memory addresses. This makes them ideal for iterating through models with identical morphologies and different channel distributions, but less ideal for network models containing a diversity of neuron types. As an intermediate, one could address this limitation by modeling networks containing discrete sets of neurons. For example, a network could contain several compartmental morphology models that each support multiple instances with different channel parameters, similar to the Ring model applied by Arbor (Akar et al., 2019; Kumbhar et al., 2019).

In its current state of development, NeuroGPU may help democratize compartmental modeling. While NeuroGPU can support simulations in large clusters using UNIX-based mutli-GPU architectures, it also is ideal for individual laboratories running simulations on Windows-based

workstations. Indeed, a workstation with total costs <$3000, when kitted with appropriate GPUs, can out-perform large CPU-based clusters. This could help broaden the use and utility of computational modeling by bringing supercomputer-level processing power to a large range of academic settings.

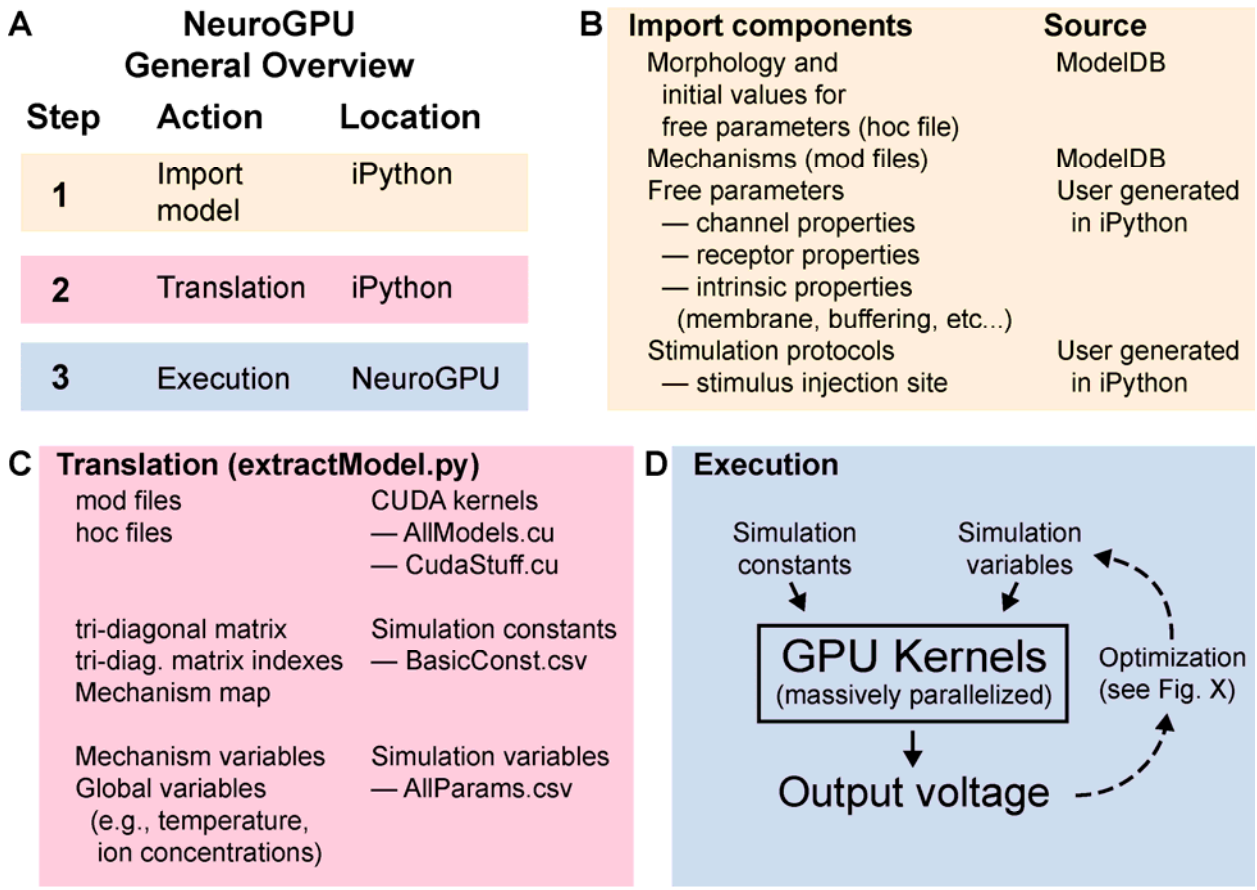

Figure 1: ***NeuroGPU overview and flowchart***

A: Overview of the general workflow in NeuroGPU: The user ports a model via the iPython GUI and customizes the simulation (panel B). NeuroGPU translates the model to CUDA code that can run on the GPU and compiles executable code.

B: Sources for model components: The morphology and model's properties are described in the hoc file. Additional mechanisms such as ion-channels are described in .mod files. The stimulation protocols can be either imported or can be generated with our provided GUI

C: Import to NeuroGPU is done by the extractModel.py script. It translates mod files to GPU kernels (see methods), which are written to AllModels.cu, and updates the course of the simulation at CudaStuff.cu. extractModel.py writes to the BasicConst.csv the tri-diagonal matrix and mechanism map, which indicates the mechanisms for each compartment. Finally, extractModel.py writes all the mechanism parameters to AllParams.csv.

D: After extractModel.py terminates, it creates NeuroGPU.exe. When NeuroGPU is invoked it reads the input files and runs the simulations for the different instances of the model and writes their voltages output to a file. When NeuroGPU is used for optimization, new instances of the models are created each iteration, and only AllParams.csv is updated via a python script.

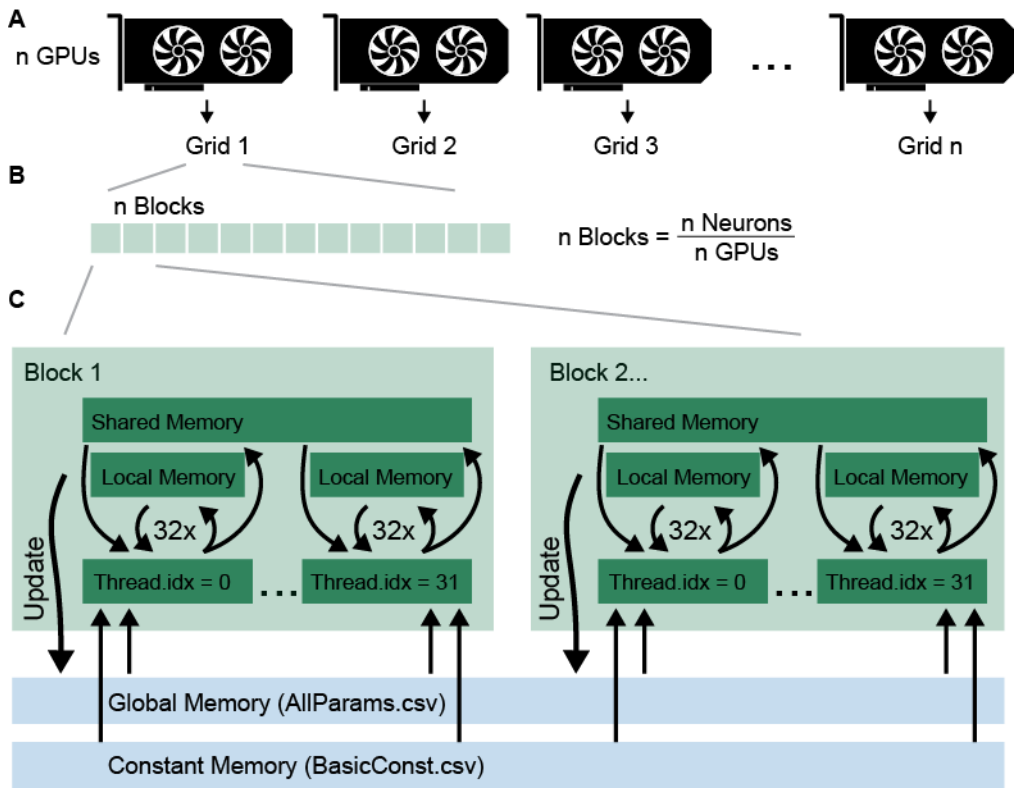

Figure 2: ***NeuroGPU CUDA implementation***

A: NeuroGPU can be run on multiple GPUs; each GPU will run a separate grid of block/neurons (Nvidia, 2018).

B: Grids are distributed in blocks, with each block representing an instance of a model. The number of blocks in a grid is set by the number of model instances that will be simulated on an individual GPU.

C: A block is the basic simulation unit upon which 32 threads each update the memory in an ILP manner (see Methods). Global memory, which can be accessed by all blocks, stores mechanism parameters for every compartment. Constant memory, which is limited in size, stores the simulation constants such as the tri-diagonal matrix and the mechanism map.

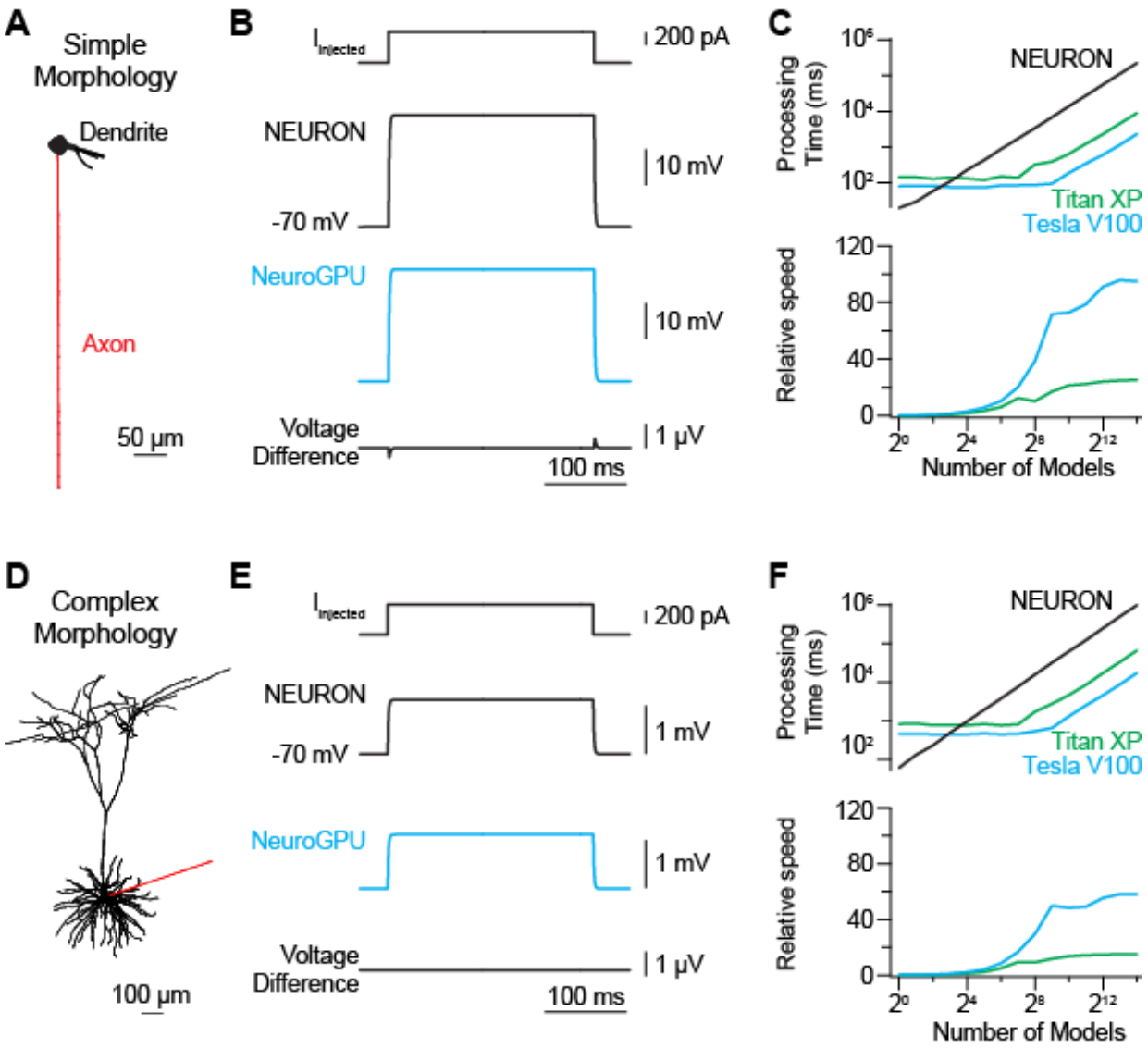

Figure 3: ***Passive model simulations***

A: Simple morphology with artificial axon. This model contains passive channels (pas.mod) in all compartments.

B: Top: injected current at the soma Middle: NEURON voltage response as recorded at the soma. Blue: NeuroGPU response as recorded at the soma. Bottom: difference in voltage between NEURON and NeuroGPU.

C: Top: Runtimes for the model using the different architectures: black – NEURON, green – NeuroGPU on TitanXP, blue – NeuroGPU on TeslaV100. X-axis in log2 scale, Y-axis in log10 scale. Bottom: Speedup compared to NEURON.

D-F: Same as A-C, but for complex morphology from (Mainen and Sejnowski, 1996).

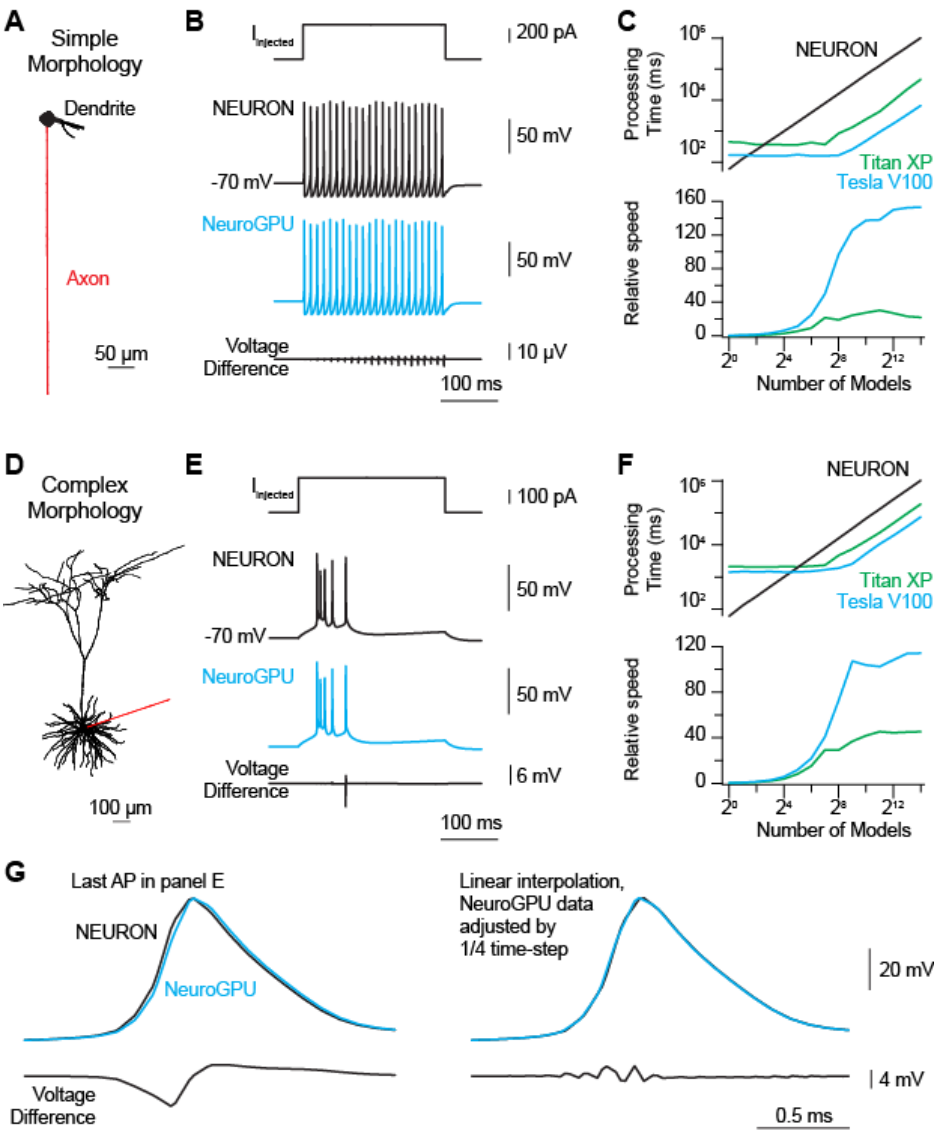

Figure 4: ***Mainen and Sejnowski model neuron simulations***

A: Simple morphology with artificial axon and active and passive components distributed as in (Mainen and Sejnowski, 1996)

B: Top: injected current at the soma. Middle: NEURON voltage response as recorded at the soma. Cyan: NeuroGPU response as recorded at the soma. Bottom: difference in voltage between NEURON and NeuroGPU.

C: Top: Runtimes for the model using the different architectures: black – NEURON, green – NeuroGPU on TitanXP, blue – NeuroGPU on TeslaV100. X-axis in log2 scale, Y-axis in log10 scale. Bottom: Speedup compared to NEURON.

D-F: Same as A-C, but for neocortical layer 5 pyramidal cell morphology, as in (Mainen and Sejnowski, 1996).

G: Last AP in panel E, with expanded timebase, highlighting differences in voltage during the rising phase of the AP. Voltage differences are minimized by linearly interpolating the data 4-fold and advancing NeuroGPU simulation by ¼ time-step.

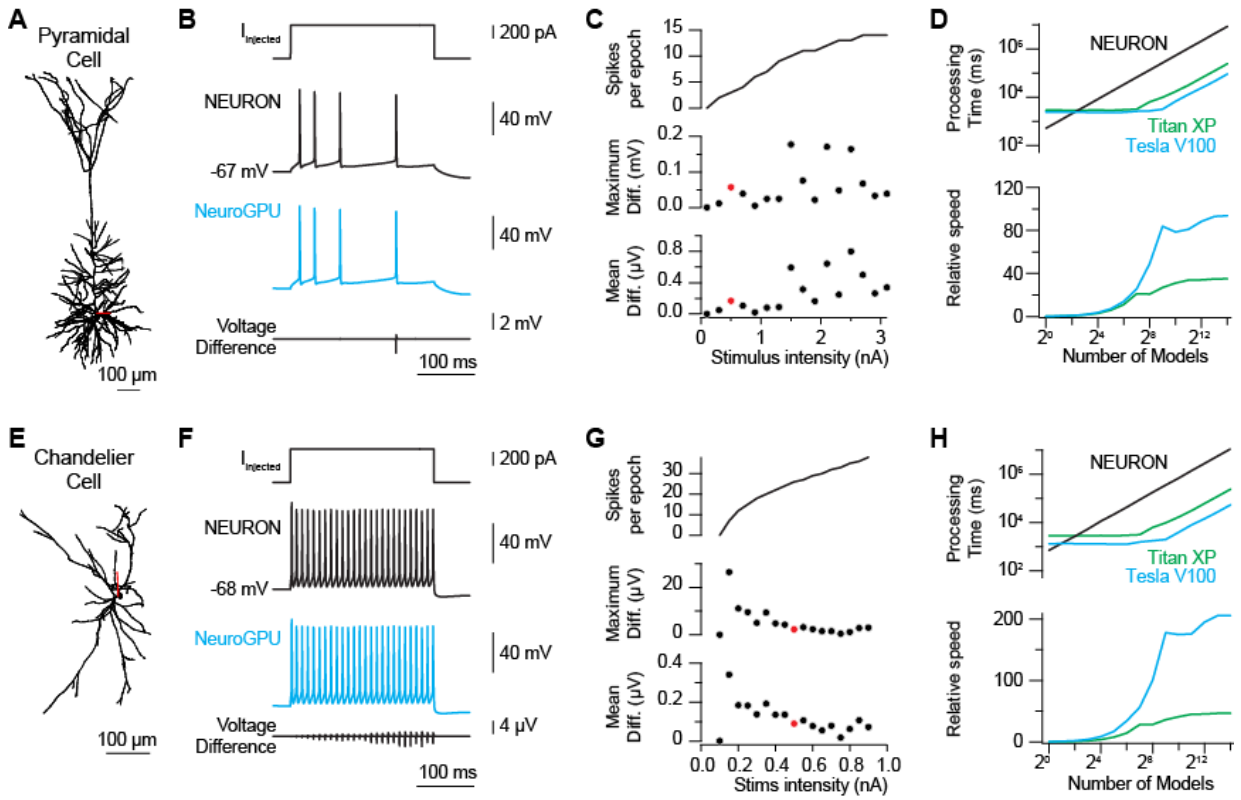

Figure 5: ***BBP portal model simulations***

A: Morphology of a BBP portal layer 5 neocortical pyramidal cell (Ramaswamy et al., 2015). Dendrite in black, axon in red.

B: Top: injected current at the soma. Middle: NEURON voltage response as recorded at the soma. Cyan: NeuroGPU response as recorded at the soma. Bottom: difference in voltage between NEURON and NeuroGPU.

C: Top: APs generated per current injection intensity in the soma. Middle, bottom: Peak and average voltage difference between the voltage response in NEURON and NeuroGPU. Red circles denote examples in B.

D: Top: Runtimes for the model using the different architectures: black – NEURON, green – NeuroGPU on TitanXP, blue – NeuroGPU on TeslaV100. X-axis in log2 scale, Y-axis in log10 scale. Bottom: Speedup compared to NEURON.

E-H: Same as A-D, but for a model chandelier cell.

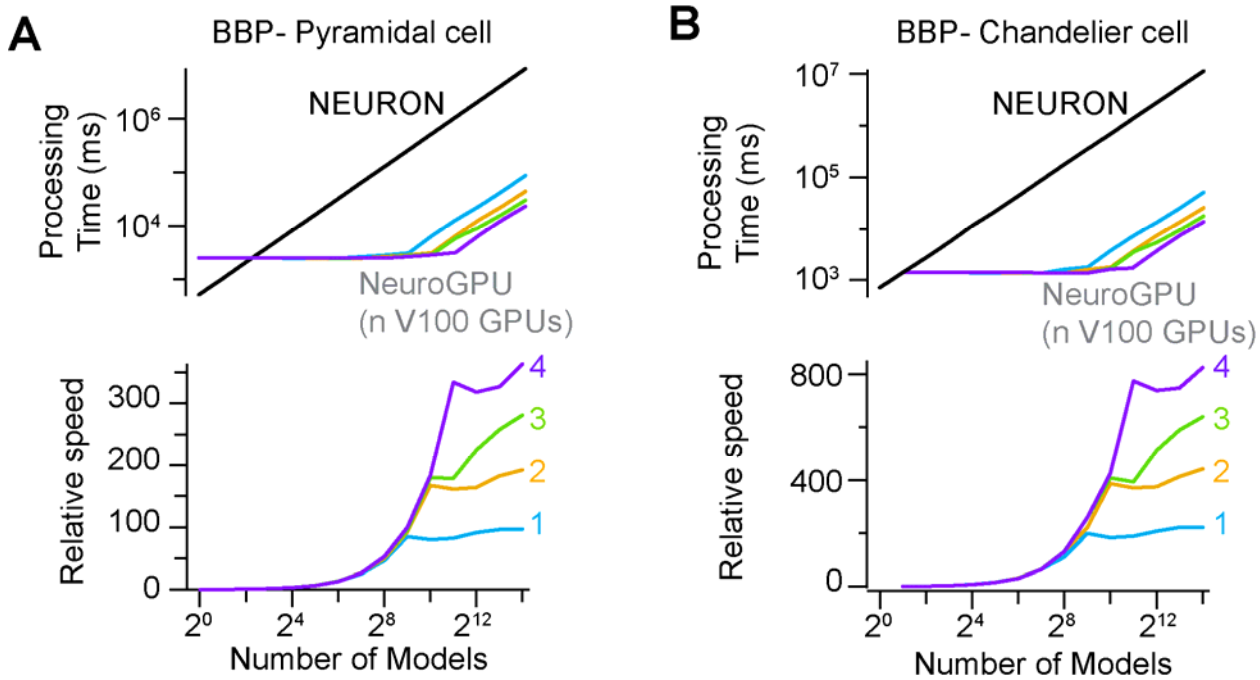

Figure 6: ***NeuroGPU simulation on multiple GPUs***

A: Top: Runtimes for pyramidal cell model using a different numbers of V100 GPUs (cyan – 1 orange – 2 green -3 purple – 4). X-axis is in log2 scale and Y-axis is in log10 scale. Bottom: Speedup compared to NEURON.

B: Same as A, but for chandelier cell model.

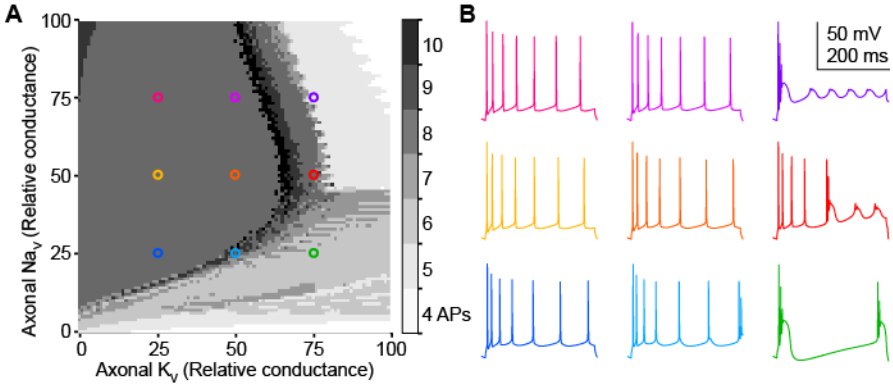

Figure 7: ***Parameter space exploration in the BBP pyramidal model***

A: Each point in the grid represents the number of APs in the relevant model. Points on the axis represent the varied conductances of Na_v and K_v at the axon in the range of [0,10] and [0,20] S/cm^2, respectively.

B: Example voltage responses for chosen models from A. Colors matched to the corresponding model location in A.

.

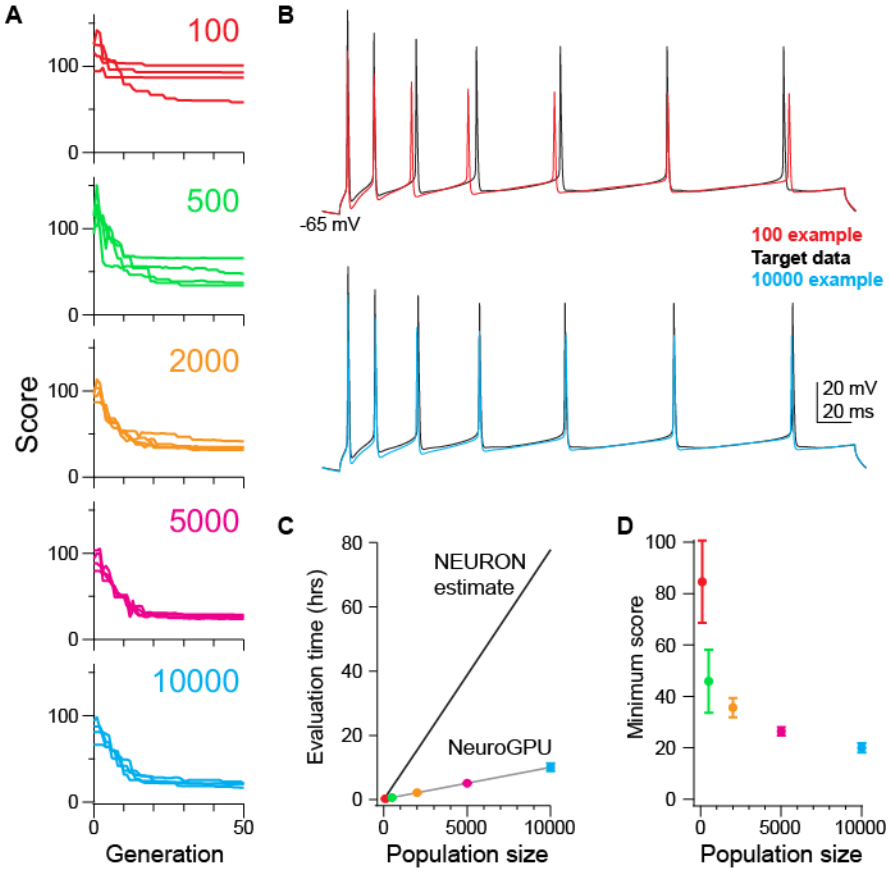

Figure 8: ***Evolutionary optimization with NeuroGPU***

A: Optimizations examples using DEAP with different sizes of populations. Four Optimizations with different random starting population over 50 generations. Y axis is the error from the target voltage as described in the methods section. Lower values denote less error from target data.

B: Voltage traces obtained from optimization (worst case from population of 100: red; best case from population of 10,000: cyan) compared to ground truth (black).

C: Comparing runtimes for optimizations using NeuroGPU and NEURON (linearly extrapolated from 5 generations). Circles are color coded for population size as in A.

D: Best score in each optimization in A. Circles and error bars are mean ± SEM.

Table 1:

Model	Morphology	SingleKernel		SplitKernel	
		Acceleration	Utilization	Acceleration	Utilization
Passive membrane	Soma & apical dendrite	95.8	0.30%	78.8	0.41%
	Pyramidal Cell	58.1	10%	55.2	3.62%
Mainen and Sejnowski	Soma & apical dendrite	153.1	10%	99.5	1%
	Pyramidal Cell	114.2	10%	111.8	3.80%
Blue Brain Project	Pyramidal Cell	93	10%	107	3.70%
	Chandelier Cell	205.6	10%	197.5	1.60%

Acceleration: fold increase in processing speed relative to single core CPU (MODEL)

Utilization: Percent of time the GPU is being used

Table 2:

Parameter Name	Base value	Lower Bound	Upper Bound
gNaTa_tbar_NaTa_t	3.137968	0.3137968	31.37968
gNaTs2_tbar_NaTs2_t	0.983955	0.0983955	9.83955
gK_Tstbar_K_Tst	0.089259	0.0089259	0.89259
glhbar_Ih	0.00008	0.000008	0.0008
glmbar_Im	0.000143	0.0000143	0.00143
gSKv3_1bar_SKv3_1	0.303472	0.0303472	3.03472

Methods:

Hardware — NEURON and TitanXP-based simulations were run on a PC with Intel Core I7-7700K 4.2GHz with 16GB of RAM. Tesla V100-based simulations were run using the NVIDIA PSG cluster. Here, each simulation was run on a single node with Haswell or Skylake CPU cores. For multi-GPU simulations, we used cluster nodes with NVLINK (Li et al., 2019) between the GPUs to enable memory peer-access.

Software — Simulations were performed in NEURON 7.6 and CUDA 9.1. All scripts were written in Python 3.7. All software is available at https://github.com/roybens/NeuroGPU.

Importing NEURON models — The python script extractmodel.py (Fig 1) exports NEURON models to NeuroGPU. This script reads all simulation details from runModel.hoc, which is populated using the GUI. NEURON models are described using either hoc or python scripts. The scripts include a morphology that can either be called as a separate file or constructed within the script.

Translating mechanisms to CUDA — Mechanisms in NEURON are described by NMODL (.mod) files (Hines and Carnevale, 2000), that update the mechanism states every simulation time step. This is done using three different procedures within NEURON that initialize mechanisms (nrn_init), update currents that mechanisms affect (nrn_cur), and then update mechanism states (nrn_state) (Carnevale and Hines, 2006). In NeuroGPU, CUDA kernels are written for each of these procedures using .mod and .c files that are generated by NEURON when running nrnivmodl. Kernels are saved and editable in AllModels.cu and AllModels.h.

Extracting simulation properties from NEURON — NeuroGPU utilizes NEURON for simulation pre-processing, including generating the mechanism map for mechanism distribution across compartments and exporting the tri-diagonal matrix using the fmatrix(). These are stored in BasicConstSegP.csv. NEURON extracts all parameters for cable equations and mechanism values within each compartment to AllParams.csv. External stimulation delivery location, intensity, and timecourse are written in stim.csv. Resting membrane potential and number of time steps in the simulation are written in sim.csv.

Solving the tridiagonal matrix — Matrix solutions were performed here using the branch-based parallelism approach as described in (Ben-Shalom et al., 2013), with morphology analysis guiding iterative matrix computations. This analysis is done in extractmodel.py and the data structures to solve the tri-diagonal in parallel is stored in BasicConstSegP.csv.

Benchmarking — All benchmarking was done compared to NEURON 7.6 running in a single thread. The morphology was adjusted to have one segment per compartment in both NEURON and NeuroGPU comparison. Simulation runtimes were compared without hard drive read/write file steps, as these aspects depend more on hard drive properties than CPU/GPU comparisons.

Multi-compartmental models — NeuroGPU performance was tested with 4 different models: 1) A passive model, utilizing passive channels described in NEURON distribution pas.mod file. These channels were distributed on both simple and complex morphologies (see Fig. 3A, D) (Mainen and Sejnowski, 1996). The simple morphology was based on the simple morphology described in Mainen and Sejnowski, with compartments reduced to 32, as this is the minimum number of compartments required for NeuroGPU-based simulations.

2) The Mainen and Sejnowski (1996) model, with channels distributed on the same complex and simple morphologies (Fig 4). Channels are distributed as in (Mainen and Sejnowski, 1996)

3) A pyramidal cell model from the Blue Brain Project portal (Ramaswamy et al., 2015) (Fig 5). BBP_PC refers to the model named L5_TTPC1_cADpyr232_1.

4) A chandelier cell model, termed BBP_CC, referring to L5_ChC_dNAC222_1. For this model, the Kdshu2007.mod files were altered to run on NeuroGPU. Specifically, global variables were removed from the neuron block and instead placed in the assigned block (Carnevale and Hines, 2006).

Optimization algorithm — The *eaMuPlusLambda* algorithm from the DEAP package was implemented by modifying the varOR procedure to call NeuroGPU (Rainville et al., 2012). Optimization was performed on the BBP_PC model. For each iteration, the algorithm began with a new population of parameters with values randomly chosen with the range specified in Table 2. The model was modified to accept new values from the optimization algorithm (similar changes were necessary to run the parameter space exploration for Figure 7). Target data were generated using the original parameters values described in Table 2. Optimization was targeted to reduce error between target data and test data using both the interspike interval (ISI) and the root mean square (RMS) of the voltage as the error function. Error was reduced to a single variable by weighting these two variables as: 10*ISI + RMS.

Acknowledgments

We are grateful to Dr. Gilad Liberman who helped conceptualize this project. To the support and advice of NVIDIA developers – Dr. Jonathan Lefman, Dr. Jonathan Bentz, Dr. Xuemeng Zhang and Angela Chen in optimizing the CUDA code. To NVIDIA Corporation for donating the GPUs used in this study. To all the members of the Bender Lab for critically assessing this work. This research was supported by NIH Grants F32 NS095580 (RBS), MH112729 (KJB), and DA035913 (KJB).

References

Akar NA, Cumming B, Karakasis V, Küsters A, Klijn W, Peyser A, Yates S (2019) Arbor - A Morphologically-Detailed Neural Network Simulation Library for Contemporary High-Performance Computing Architectures. In: Proceedings - 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2019, pp 274–282.

Almog M, Korngreen A (2014) A quantitative description of dendritic conductances and its application to dendritic excitation in layer 5 pyramidal neurons. J Neurosci 34:182–196.

Alonso LM, Marder E (2019) Visualization of currents in neural models with similar behavior and different conductance densities. Elife 8.

Ben-Shalom R, Aviv A, Razon B, Korngreen A (2012) Optimizing ion channel models using a parallel genetic algorithm on graphical processors. J Neurosci Methods 206:183–194.

Ben-Shalom R, Keeshen CM, Berrios KN, An JY, Sanders SJ, Bender KJ (2017) Opposing Effects on NaV1.2 Function Underlie Differences Between SCN2A Variants Observed in Individuals With Autism Spectrum Disorder or Infantile Seizures. Biol Psychiatry 82:224–232.

Ben-Shalom R, Liberman G, Korngreen A (2013) Accelerating compartmental modeling on a graphical processing unit. Front Neuroinform 7:4.

Byczkowicz N, Eshra A, Montanaro J, Trevisiol A, Hirrlinger J, P Kole MH, Shigemoto R (n.d.) HCN channel-mediated neuromodulation can control action 1 potential velocity and fidelity in central axons.

Cannon RC, Gewaltig M-O, Gleeson P, Bhalla US, Cornelis H, Hines ML, Howell FW, Muller E, Stiles JR, Wils S, De Schutter E (2007) Interoperability of Neuroscience Modeling Software: Current Status and Future Directions. Neuroinformatics 5:127–138.

Carnevale NT, Hines ML (2006) The NEURON Book. Cambridge University Press.

Druckmann S, Banitt Y, Gidon A, Schürmann F, Markram H, Segev I (2007) A novel multiple objective optimization framework for constraining conductance-based neuron models by experimental data. Front Neurosci 1:7.

Einevoll GT, Destexhe A, Diesmann M, Grün S, Jirsa V, de Kamps M, Migliore M, Ness T V., Plesser HE, Schürmann F (2019) The Scientific Case for Brain Simulations. Neuron 102:735–744.

Eklund A, Dufort P, Forsberg D, LaConte SM (2013) Medical image processing on the GPU - past, present and future. Med Image Anal 17:1073–1094.

Fidjeland AK, Shanahan MP (2010) Accelerated Simulation of Spiking Neural Networks Using GPUs. Ijcnn'10:1–8.

Gagn C (2012) DEAP□: Evolutionary Algorithms Made Easy. J Mach Learn Res 13:2171–2175.

Go AW, Williamson MJ, Xu D, Poole D, Grand S Le, Walker RC, Götz AW, Williamson MJ, Xu D, Poole D, Le Grand S, Walker RC (2012) Routine microsecond molecular dynamics simulations with AMBER on GPUs. 1. generalized born. J Chem Theory Comput 8:1542–1555.

Gouwens NW, Berg J, Feng D, Sorensen SA, Zeng H, Hawrylycz MJ, Koch C, Arkhipov A

(2018) Systematic generation of biophysically detailed models for diverse cortical neuron types. Nat Commun 9.

Gurkiewicz M, Korngreen A (2007) A Numerical Approach to Ion Channel Modelling Using Whole-Cell Voltage-Clamp Recordings and a Genetic Algorithm. PLoS Comput Biol 3:e169.

Hay E, Schurmann F, Markram H, Segev I, Schürmann F, Markram H, Segev I (2013) Preserving axosomatic spiking features despite diverse dendritic morphology. J Neurophysiol 109:2972–2981.

Hines M (1984) Efficient computation of branched nerve equations. Int J Biomed Comput 15:69–76.

Hines ML, Carnevale NT (2000) Expanding NEURON's Repertoire of Mechanisms with NMODL. Neural Comput 12:995–1007.

Hines ML, Eichner H, Schürmann F (2008) Neuron splitting in compute-bound parallel network simulations enables runtime scaling with twice as many processors. J Comput Neurosci 25:203–210.

Hines P, Fouriaux M, Jan NC (n.d.) An Optimized Compute Engine for the NEURON Simulator.

Keren N, Bar-Yehuda D, Korngreen A (2009) Experimentally guided modelling of dendritic excitability in rat neocortical pyramidal neurones. J Physiol 587:1413–1437.

Kumbhar P, Hines M, Fouriaux J, Ovcharenko A, King J, Delalondre F, Schürmann F (2019) CoreNEURON□: An Optimized Compute Engine for the NEURON Simulator.

Li A, Song SL, Chen J, Li J, Liu X, Tallent N, Barker K (2019) Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect.

Mainen ZF, Sejnowski TJ (1996) Influence of dendritic structure on firing pattern in model neocortical neurons. Nature 382:363–366.

Mäki-Marttunen T, Halnes G, Devor A, Metzner C, Dale AM, Andreassen OA, Einevoll GT (2018) A stepwise neuron model fitting procedure designed for recordings with high spatial resolution: Application to layer 5 pyramidal cells. J Neurosci Methods 293:264–283.

Markram H et al. (2015) Reconstruction and Simulation of Neocortical Microcircuitry. Cell 163:456–492.

McDougal RA, Morse TM, Carnevale T, Marenco L, Wang R, Migliore M, Miller PL, Shepherd GM, Hines ML (2017) Twenty years of ModelDB and beyond: building essential modeling tools for the future of neuroscience. J Comput Neurosci 42:1–10.

Miceli F, Soldovieri MV, Ambrosino P, Barrese V, Migliore M, Cilio MR, Taglialatela M (2013) Genotype–phenotype correlations in neonatal epilepsies caused by mutations in the voltage sensor of K_v 7.2 potassium channel subunits. Proc Natl Acad Sci 110:4386–4391.

Migliore M, Migliore R (2012) Know Your Current Ih: Interaction with a Shunting Current Explains the Puzzling Effects of Its Pharmacological or Pathological Modulations Attali B, ed. PLoS One 7:e36867.

Nocedal J, Wright S (2006) Numerical optimization.

Nvidia C (2018) Cuda c programming guide, version 9.1. NVIDIA Corp.

Pachitariu M, Steinmetz N, Kadir S, Carandini M, D. HK (2016) Kilosort: realtime spike-sorting for extracellular electrophysiology with hundreds of channels. bioRxiv:061481.

Payne JL, Sinnott-Armstrong NA, Moore JH (2010) Exploiting Graphics Processing Units for Computational Biology and Bioinformatics. Interdiscip Sci 2:213–220.

Prein AF, Langhans W, Fosser G, Ferrone A, Ban N, Goergen K, Keller M, Tölle M, Gutjahr O, Feser F, Brisson E, Kollet S, Schmidli J, Van Lipzig NPM, Leung R (2015) A review on regional convection-permitting climate modeling: Demonstrations, prospects, and challenges. Rev Geophys 53:323–361.

Prinz AA, Billimoria CP, Marder E (2003) Alternative to Hand-Tuning Conductance-Based Models: Construction and Analysis of Databases of Model Neurons. J Neurophysiol 90:3998–4015.

Prinz AA, Bucher D, Marder E (2004) Similar network activity from disparate circuit parameters. Nat Neurosci 7:1345–1352.

Rainville F De, Fortin F, Gardner M, Parizeau M, Gagné C (2012) DEAP□: A Python Framework for Evolutionary Algorithms. Companion proc Genet Evol Comput Conf:85–92.

Ramaswamy S et al. (2015) The neocortical microcircuit collaboration portal: a resource for rat somatosensory cortex. Front Neural Circuits 9:44.

Salomon-Ferrer R, Götz AW, Poole D, Le Grand S, Walker RC, Go AW, Poole D, Grand S Le, Walker RC (2013) Routine microsecond molecular dynamics simulations with AMBER on GPUs. 2. Explicit solvent particle mesh ewald. J Chem Theory Comput 9:3878–3888.

Schmidhuber J (2015) Deep Learning in neural networks: An overview. Neural Networks 61:85–117.

Spratt PWE, Ben-Shalom R, Keeshen CM, Burke KJ, Clarkson RL, Sanders SJ, Bender KJ (2019) The Autism-Associated Gene Scn2a Contributes to Dendritic Excitability and Synaptic Function in the Prefrontal Cortex. Neuron. epub ahead of print

Van Geit W, De Schutter E, Achard P (2008) Automated neuron model optimization techniques: a review. Biol Cybern 99:241–251.

Volkov V, Demmel JW (2008) Benchmarking GPUs to tune dense linear algebra. In: 2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis, pp 1–11. IEEE.

Whitehead N (2011) Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs.

Zhang P, Holk E, Matty J, Misurda S, Zalewski M, Chu J, McMillan S, Lumsdaine A (2015) Dynamic parallelism for simple and efficient GPU graph algorithms. In: Proceedings of the 5th Workshop on Irregular Applications Architectures and Algorithms - IA3 ’15, pp 1–4. New York, New York, USA: ACM Press.