A GPU-based computational framework that bridges Neuron simulation and Artificial Intelligence

Yichen Zhang[1,5], Gan He[1,5], Xiaofei Liu[1], J.J.Johannes Hjorth[3,4], Alexander Kozlov[3,4], Yutao He[1], Shenjian Zhang[1], Lei Ma[1], Jeanette Hellgren Kotaleski[3,4], Yonghong Tian[1], Sten Grillner[4], Kai Du[2], Tiejun Huang[1,2]

[1]National Engineering Research Center of Visual Technology, School of Computer Science, Peking University, Beijing 100871, China
[2]Institute for Artificial Intelligence, Peking University, Beijing 100871, China
[3] Science for Life Laboratory, School of Electrical Engineering and Computer Science, Royal Institute of Technology, Stockholm SE-10044, Sweden
[4]Department of Neuroscience, Karolinska Institute, Stockholm SE-17165, Sweden
[5] Co-first author

Key words: Neuron simulation, computational neuroscience, dendritic computation, artificial intelligence, synaptic learning, artificial neural networks

**Correspondence should be addressed to:

Dr. Kai Du kai.du@pku.edu.cn

Abstract

Biophysically detailed multi-compartment models are powerful tools to explore computational principles of the brain and also serve as a theoretical framework to generate algorithms for artificial intelligence (AI) systems. However, the expensive computational cost severely limits the applications in both the neuroscience and AI fields. The major bottleneck during simulating detailed compartment models is the ability of a simulator to solve large systems of linear equations. Here, we present a novel **D**endritic **H**ierarchical **S**cheduling (DHS) method to markedly accelerate such process. We theoretically prove that the DHS implementation is computationally optimal and accurate. This GPU-based method performs at 2-3 orders of magnitude higher speed than that of the classic serial Hines method in the conventional CPU platform. We build a DeepDendrite framework, which integrates the DHS method and the GPU computing engine of the NEURON simulator and demonstrate applications of DeepDendrite in neuroscience and AI tasks. We investigated how spatial patterns of spine inputs affects neuronal excitability in a detailed human pyramidal neuron model with 25,000 spines; and examined how dendrites protect morphologically detailed neural networks against adversarial attacks in typical image classification tasks.

Introduction

Deciphering the coding and computational principles of neurons is essential to neuroscience. Mammalian brains are composed of more than thousands of different types of neurons with unique morphological and biophysical properties. Even though it is no longer conceptually true, the "point-neuron" doctrine[1], in which neurons were regarded as simple summing units, is still widely applied in neural computation, especially in neural network analysis. In recent years, modern artificial intelligence (AI) has utilized this principle and developed powerful tools, such as the artificial neural network (ANN)[2]. However, in addition to comprehensive computations at the single neuron level, subcellular compartments, such as neuronal dendrites, can also carry out nonlinear operations as independent computational units[3-7]. Furthermore, dendritic spines, small protrusions that densely cover dendrites in spiny neurons, can compartmentalize synaptic signals, allowing them to be separated from their parent dendrites *ex vivo* and *in vivo*[8-11].

Simulations using biologically detailed neurons provide a theoretical framework for linking biological details to computational principles. The core of the biophysically detailed multi-compartment model framework[12, 13] allows us to model neurons with realistic dendritic morphologies, intrinsic ionic conductance, and extrinsic synaptic inputs. The backbone of the detailed multi-compartment model, i.e., dendrites, is built upon the classical Cable theory[12], which models the biophysical membrane properties of dendrites as passive cables, providing a mathematical description of how electronic signals invade and propagate throughout complex neuronal processes. By incorporating Cable theory with active biophysical mechanisms such as ion channels, excitation, inhibition, etc., a detailed multi-compartment model can achieve cellular and subcellular neuronal computations beyond experimental limitations[4, 7].

In addition to its profound impact on neuroscience, biologically detailed neuron models recently were utilized to bridge between neuronal structural and biophysical details and AI. The prevailing technique in the modern AI field is ANNs consisting of point-neurons, an analog to biological neural network. Although ANNs with "backpropagation-of-error" (backprop) algorithm achieve remarkable performance in specialized applications, even beating top human professional players in the games of Go and chess[14, 15], the human brain still outperforms ANNs in other domains, for example, general intelligence. Recent theoretical studies suggest that dendritic integration is crucial in generating efficient learning algorithms that potentially exceed backprop in parallel information processing[16-18]. Furthermore, a single detailed multi-compartment model can learn network-level nonlinear computations for point-neurons by adjusting only the synaptic strength[19, 20], demonstrating the full potential of the detailed models in building more powerful brain-like AI systems. Therefore, it is of high

priority to expand paradigms in brain-like AI from single detailed neuron models to large-scale biologically detailed networks.

One long-standing challenge of the detailed simulation approach lies in its exceedingly high computational cost, which has severely limited its application to neuroscience and AI. The major bottleneck of the simulation is to solve linear equations based on the foundational theories of detailed modeling[12, 21, 22]. To improve the efficiency, the classic Hines method reduces the time complexity for solving equations from $O(n^3)$ to $O(n)$, which has been widely applied as the core algorithm in popular simulators such as NEURON[23] and GENESIS[24]. However, this method uses a serial approach to process each compartment sequentially. When a simulation involves multiple biophysically detailed dendrites with dendritic spines, the linear equation matrix ('Hines Matrix') scales accordingly with an increasing number of dendrites or spines (Fig. 1e), making Hines method no longer practical, since it poses a very heavy burden on the entire simulation.

During past decades, tremendous progress has been achieved to speed up the Hines method by using parallel methods at a cellular level, which enables to parallelize the computation of different parts in each cell[25-30]. However, current cellular level parallel methods often lack an efficient parallelization strategy or lack sufficient numerical accuracy as compared to the original Hines method.

Here, we develop a fully automatic, numerically accurate, and optimized simulation tool that can significantly accelerate computation efficiency and reduce computational cost. In addition, this simulation tool can be seamlessly adopted for establishing and testing neural networks with biological details for machine learning and AI applications. Critically, we formulate the parallel computation of the Hines method as a mathematical scheduling problem and generate a Dendritic Hierarchical Scheduling (DHS) method based on combinatorial-optimization[31] and parallel-computing theory[32]. We demonstrate that our algorithm provides an optimal scheduling without any loss of precision. Furthermore, we have optimized DHS for the currently most advanced GPU chip by leveraging the GPU memory hierarchy and memory accessing mechanisms. Together, DHS can speed up computation 60-1,500 times (Table S1) compared to the classic simulator NEURON[23] while maintaining identical accuracy.

To enable detailed dendritic simulations for use in AI, we next establish the DeepDendrite framework by integrating the DHS-embedded CoreNEURON (an optimized compute engine for NEURON) platform[33] as the simulation engine and two auxiliary modules (I/O module and learning module) supporting dendritic learning algorithms during simulations. DeepDendrite runs on the GPU hardware platform, supporting both regular simulation tasks in neuroscience and learning tasks in AI.

Last but not least, we also present several applications using DeepDendrite, targeting a few critical challenges in neuroscience and AI: (1) We demonstrate how

spatial patterns of dendritic spine inputs affect neuronal activities with neurons containing spines throughout the dendritic trees (full-spine models). DeepDendrite enables us to explore neuronal computation in a simulated human pyramidal neuron model with ~25,000 dendritic spines. (2) We establish an artificial neural network with morphologically detailed human pyramidal neurons and train this network to perform typical image classification tasks. DeepDendrite reduces the training time from days to minutes, which enables detailed network models to perform AI tasks. We further explore previously proposed hypothesis about the roles of the dendrites in defending against adversarial attacks[34], an intentionally designed perturbation to deceive ANNs but nearly unperceivable to human vision. Models with morphologically detailed dendrites show superior performance with strong robustness in defending against adversarial attacks.

All source code for DeepDendrite, the full-spine models and the detailed dendritic network model are publicly available online (see Method). Our open-source learning framework can be readily integrated with other dendritic learning rules, such as learning rules for nonlinear (full-active) dendrites[19], burst-dependent synaptic plasticity [18], and learning with spike prediction[35]. Overall, our study provides a complete set of tools that have the potential to change the current computational neuroscience community ecosystem. By leveraging the power of GPU computing, we envision that these tools will facilitate system-level explorations of computational principles of the brain's fine structures, as well as promote the interaction between neuroscience and modern AI.

Results

Computing ionic currents and solving linear equations are two critical phases when simulating biophysically detailed neurons, and which are time-consuming and poise severe computational burden. Fortunately, computing ionic currents of each compartment is a fully independent process so that it can be naturally parallelized on devices with massive parallel-computing units like GPUs[36]. As a consequence, solving linear equations becomes the remaining bottleneck for the parallelization process (Fig. 1a-f).

To tackle this bottleneck, cellular level parallel methods have been developed, which accelerate single-cell computation by “splitting” a single cell into several compartments that can be computed in parallel[25, 26, 37]. However, such methods rely heavily on prior knowledge in order to generate practical strategies on how to split a single neuron into compartments (Fig. 1g-i, S1). Hence, it becomes less efficient for neurons with asymmetrical morphologies, e.g., pyramidal neurons and Purkinje neurons.

Dendritic Hierarchical Scheduling (DHS) Method

Our aim is to develop a more efficient and precise parallel method for simulation of biologically detailed neural networks. First, we establish the criteria for the accuracy of a cellular level parallel method. Based on the theories in parallel computing[32], we propose three conditions to make sure a parallel method will yield identical solutions as the serial computing Hines method according to the data dependency in Hines method (see Methods). Then to theoretically evaluate the run time, i.e. efficiency, of the serial and parallel computing methods, we introduce and formulate the concept of computational cost as the number of steps a method takes in solving equations (see Methods).

Based on the simulation accuracy and computational cost, we formulate the parallelization problem as a mathematical scheduling problem (details are in Methods). In simple terms, we view a single neuron as a tree with many nodes (compartments). For k parallel threads, we can compute at most k nodes at each step, but we need to ensure a node is computed *only if* all its children nodes have been processed; our goal is to find a strategy with the minimum number of steps for the entire procedure.

To generate an optimal partition, we propose a method called Dendritic Hierarchical Scheduling (DHS) (theoretical proof is presented in the Methods). The key idea of DHS is to prioritize deep nodes (Fig 2a), which results in a hierarchical schedule order. The DHS method includes two steps: analyzing dendritic topology and finding the best partition: (1) Given a detailed model, we first obtain its corresponding dependency tree and calculate the depth of each node (the depth of a node is the number of its ancestor nodes) on the tree (Fig 2 b,c). (2) After topology analysis, we search the candidates and pick at most *k* deepest candidate nodes (a node is a candidate only if all its children nodes have been processed). This procedure repeats until all nodes are processed (Fig. 2d).

Take a simplified model with 15 compartments as an example, using the serial computing Hines method, it takes 14 steps to process all nodes, while using DHS with four parallel units can partition its nodes into five subsets (Fig. 2d): {{9,10,12,14}, {1,7,11,13}, {2,3,4,8}, {6}, {5}}. Because nodes in the same subset can be processed in parallel, it takes only five steps to process all nodes using DHS (Fig. 2e).

Next, we apply the DHS method on six representative detailed neuron models (selected from ModelDB, https://senselab.med.yale.edu/ModelDB) with different numbers of threads (Fig. 2f):, including cortical and hippocampal pyramidal neurons[39-41], cerebellar Purkinje neurons[42], striatal spiny-projection neurons (SPN[43]), and olfactory bulb mitral cells[44], covering the major principle neurons in sensory, cortical and subcortical areas. We then measured the computational cost. The relative computational cost here is defined by the proportion of the computational cost of DHS to that of the serial Hines method. The computational cost, i.e. the number of steps

taken in solving equations, drops dramatically with increasing thread numbers. For example, with 16 threads, the computational cost of DHS is 7%-10% as compared to the serial Hines method. Intriguingly, the DHS method reaches the lower bounds of their computational cost for presented neurons when given 16 or even 8 parallel threads (Fig. 2f), suggesting adding more threads does not improve performance further because of the dependencies between compartments.

Together, we generate a DHS method that enables automated analysis of the dendritic topology and optimal partition for parallel computing. It is worth noting that DHS finds the optimal partition before simulation starts, and no extra computation is needed to solve equations.

Speeding Up DHS by GPU Memory Boosting

DHS computes each neuron with multiple threads, which consumes a vast amount of threads when running neural network simulations. Graphics Processing Units (GPU) consists of massive processing units (streaming processor, SP, Fig. 3a,b) for parallel computing[45]. In theory, many SPs on the GPU should support efficient simulation for large-scale neural networks (Fig. 3c). However, we consistently observed that the efficiency of DHS significantly decreased when the network size grew, which might result from scattered data storage or extra memory access caused by loading and writing intermediate results (Fig. 3d, left).

We solve this problem by GPU memory-boosting, a method to increase memory throughput by leveraging GPU's memory hierarchy and access mechanism. Based on the memory loading mechanism of GPU, successive threads loading aligned and successively-stored data lead to a high memory throughput compared to accessing scatter-stored data, which reduces memory throughput[45, 46]. To achieve the high throughput, we first align computing orders of nodes and rearrange threads according to the number of nodes on them. Then we permute data storage in global memory, consistent with computing orders, i.e. nodes that are processed at the same step are stored successively in global memory. Moreover, we use GPU registers to store intermediate results, further strengthening memory throughput. The example shows that memory-boosting takes only two memory transactions to load eight request data (Fig. 3d, right). Furthermore, experiments on multiple numbers of pyramidal neurons with spines (Fig. 3e,f, S2) show that memory-boosting achieves a two fold speedup as compared to the naïve DHS.

To comprehensively test the performance of DHS with GPU memory-boosting, we select six typical neuron models and evaluate the run time of solving cable equations on massive numbers of each model (Fig. 4). We examined DHS with four threads (DHS-4) and sixteen threads (DHS-16) for each neuron, respectively. Compared to the GPU method in CoreNEURON[33], DHS-4 and DHS-16 can speed up about 5 and 15

times, respectively (Fig. 4a). Moreover, compared to the conventional serial Hines method in NEURON running with a single-thread of CPU, DHS speeds up the simulation by 2-3 orders of magnitude (Fig. S3), while retaining the identical numerical accuracy.

DHS Creates Cell-type-specific Optimal Partitioning

To gain insights into the working mechanism of the DHS method, we visualized the partitioning process by mapping compartments to each thread (every color presents a single thread in Fig. 4b,c). The visualization shows that a single thread frequently switches among different branches (Fig. 4b,c). Interestingly, DHS generates aligned partitions in morphologically symmetric neurons such as the striatal projection neuron (SPN) and the Mitral cell (Fig. 4b,c). By contrast, it generates fragmented partitions of morphologically asymmetric neurons like the pyramidal neurons and Purkinje cell (Fig. 4b,c), indicating that DHS splits the neural tree at individual compartment scale (i.e. tree node) rather than branch scale. This cell-type-specific fine-grained partition enables DHS to fully exploit all available threads.

In summary, DHS and memory boosting generate a theoretically proven optimal solution for solving linear equations in parallel with unprecedented efficiency. Using this principle, we built the open access DeepDendrite platform, which can be utilized by neuroscientists to implement models without any specific GPU programming knowledge.

Below, we demonstrate how we can utilize DHS and DeepDendrite in neuroscience- and AI-related tasks.

DHS Enables Spine Level Modelling

As dendritic spines receive most of the excitatory input to cortical and hippocampal pyramidal neurons, striatal projection neurons, etc, their morphologies and plasticity are crucial for regulating neuronal excitability[10, 47-50]. However, spines are too small (~1 μm length) to be directly measured experimentally with regard to voltage dependent processes. Thus, theoretical work is critical for the full understanding of the spine computations.

We can model a single spine with two compartments: the spine-head where synapses are located and the spine-neck that links the spine-head to dendrites[51]. The theory predicts that the very thin spine-neck (0.1-0.5 um in diameter), which electronically isolates the spine-head from its parent dendrite, thus compartmentalizing the signals generated at the spine-head[52]. However, the detailed model with fully distributed spines on dendrites ('full-spine model') is computationally very expensive. A common compromising solution is to modify the capacitance and resistance of the membrane by a F_{spines} factor[53], instead of modeling all spines explicitly. Here, the F_{spines}

factor aims at approximating the spine effect on the biophysical properties of the cell membrane[53].

Inspired by the previous work of ref.[50], we investigated how different spatial patterns of excitatory inputs formed on dendritic spines shape neuronal activities in a human pyramidal neuron model with explicitly modeled spines (Fig. 5a). Noticeably, Eyal et al. employed the F_{spines} factor to incorporate spines into dendrites while only a few activated spines were explicitly attached to dendrites ('few-spine model' in Fig. 5a). The value of F_{spines} in their model was computed from the dendritic area and spine area in the reconstructed data. Accordingly, we calculated the spine density from their reconstructed data to make our full-spine model more consistent with Eyal's few-spine model. With the spine density set to 1.3/μm, the pyramidal neuron model contained about 25,000 spines without altering the model's original morphological and biophysical properties. Further, we repeated the previous experiment protocols with both full-spine and few-spine models. We use the same synaptic input as in Eyal's work but attach extra background noise to each sample. By comparing the somatic traces (Fig. 5b,c) and spike probability (Fig. 5d) in full-spine and few-spine models, we found that the full-spine model is much leakier than the few-spine model. In addition, the spike probability triggered by the activation of clustered spines appeared to be more nonlinear in the full-spine model (the solid blue line in Fig. 5d) than in the few-spine model (the dashed blue line in Fig. 5d). These results indicate that the conventional F-factor method may underestimate the impact of dense spine on the computations of dendritic excitability and nonlinearity.

In the DeepDendrite platform, both full-spine and few-spine models achieved 8 times speedup compared to CoreNEURON in the GPU platform and 100 times speedup compared to serial NEURON in the CPU platform (Fig. 5e, Table S1). Therefore, DHS method enables explorations of the dendritic excitability under more realistic anatomic conditions.

DHS-based Learning on Detailed Dendritic Neural Networks

Deep artificial neural networks (DNNs), also called "deep learning", have made remarkable breakthroughs in AI and exceed human performance in games and image classification tasks[2]. Because biological neural networks initially inspired DNNs, some fascinating questions remain to be answered: how does our brain realize efficient learning? Is there any learning algorithm similar to the "backprop" in DNNs? Although these questions remain open, recent work suggests that dendrites are crucial for the credit assignment in our brain, because of their enriched biophysical properties and anatomically segregated axonal projections to the dendrite from feedforward and feedback pathways. However, the earlier foundational work primarily focused on either coarse-grained dendritic network models or a single detailed compartment model.

There is a lack of studies at the neural network scale with biophysically detailed models. The principal technological barriers are: 1) the computational cost of biophysically detailed networks is prohibitive; 2) the traditional simulators (NEURON, GENESIS, etc.) are not compatible with performing deep learning tasks (for details, please see Methods).

First, DHS solves the problem of the computational cost. The I/O and the learning module enable DeepDendrite to cope with the technical obstacles in learning. Thus, DeepDendrite can fully exploit detailed neural networks to explore learning algorithms and perform learning tasks (Fig. S4).

Next, we demonstrate how to implement a detailed dendritic network for a typical image classification task and utilize DHS to speed up its training procedure. We want to emphasize that here our purpose is not to build a state-of-the-art model but to explore the feasibility of training biophysically detailed networks for specific tasks. Inspired by learning rules on simplified dendritic models with only several passive compartments[17, 35], we set up a 3-layer hybrid neural network (784x64x10), HPC-net (Human Pyramidal Cell based network). Precisely, the middle layer of HPC-net consists of 64 morphologically detailed passive human pyramidal neuron models[50], taken from the ModelDB, access No.238347. In the HPC-net, we follow the segregation of feedforward and feedback pathways[17, 35], while error signals are propagated from soma to dendrites for each detailed neuron to compute gradients for all the dendritic synapses (Fig. 6a, details in Method). Critically, DeepDendrite supports training with mini-batch, allowing to train dozens of copies of the network and thousands of pyramidal neuron models simultaneously in a single GPU, making the training phase converge significantly faster (Fig. 6b). With DHS, training with mini-batch becomes highly efficient, achieving a 30-fold speedup on a single GPU (NVIDIA Tesla A100) compared to the same training procedure in a 40-process-parallel NEURON on a single CPU (Intel Xeon E5-2698 v4) (Fig. 6f).

To test the performance of our 3-layer HPC-Net, we next train it on typical datasets MNIST[54] and fashion-MNIST[55]. Before training, hidden neurons are insensitive to input stimuli (Fig. 6c, upper left) as all synaptic weights are randomly distributed in -0.1 to 0.1 (Fig. 6c, upper right). After training, the distribution of synaptic weights was significantly changed (Fig. 6d, bottom right), and the detailed neurons in the hidden layer presented strong and diverse responses to input stimuli (Fig. 6d, bottom left). Furthermore, HPC-Net can achieve nearly identical prediction accuracy to a corresponding ANN (HPC-Net, 96.46% in MINIST and 86.04% in f-MNIST; ANN, 97.58% in MINIST and 88.13% in f-MNIST, Fig. 6f). For fair comparisons, the hyper-parameters of HPC-Net and the ANN, e.g. number of hidden neurons, activation function, training batch size, etc. are set the same. These data demonstrate that DeepDendrite can effectively train detailed neural networks on the image classification task.

Finally, we ask what could be the strength of detailed neural networks on image classification tasks besides that it is more biologically plausible? In deep learning, it is well known that DNNs are fragile to the intentionally designed perturbation, i.e. adversarial attacks, which the human observers can hardly perceive[56]. One intriguing hypothesis is that the dendrites and synapses protect our visual system from adversarial attacks[57]. Here we test this hypothesis with the detailed HPC-Net.

The adversarial sample is generated from intrinsic gradients of a trained network[56]. Considering the HPC-net and the ANN are two distinct networks, we adopt a popular attack technique named transferred attack to ensure that the attack is identical for both networks. We generated the adversarial samples from a third-party 20-layer ResNet[58] rather than directly on ANN. These test adversarial samples are fed to the HPC-Net and the ANN, which are both trained with original clean images (Fig. 6d). Our results show that with the increase of attack strength, the prediction accuracy of the ANN dramatically drops to only 70.30% on MNIST and 56.47% on Fashion MNIST. In stark contrast, HPC-Net has a strong capability in defending adversarial attacks, achieving prediction accuracy of 89.30% and 73.19% under large attack strength, which is 19% and 16.72% higher than that of the ANN on MNIST and Fashion MNIST respectively (Fig. 6e). Our results are the first to show that dendrites might be crucial for biological vision robustness.

In summary, the DHS-based DeepDendrite framework enables performing learning tasks with detailed neural network models. Further, the intrinsic biophysical properties of dendrites strengthen the robustness of deep learning neural networks. Although here we only present a purely passive morphologically detailed network, such biologically detailed neural networks can be extended to train fully active detailed models with DHS. Therefore, the deep learning network will benefit from enriched biophysical properties of detailed neuron and neural network models.

Discussion

In this work, we propose the DHS method to parallelize the computation of Hines method[38] and we mathematically demonstrate that the DHS provides an optimal solution without any loss of precision. Next, we implement DHS on the GPU hardware platform and use GPU memory-boosting techniques to refine the DHS (Fig. 3). When simulating a large number of neurons with complex morphologies, DHS with memory-boosting achieves a 15-folds speedup (Table S1), as compared to the GPU method used in CoreNEURON[33] and up to 1,500-folds speedup compared to serial Hines method in the CPU platform (Fig. 4, Table S1). Furthermore, we develop the DeepDendrite framework on the GPU by integrating the DHS-embedded CoreNEURON and two additional modules for learning. Thus, the DeepDendrite

supports general simulations for neuroscience and for performing AI-related tasks. Finally, as demonstrations of the capacity of DeepDendrite, we present two representative applications: the first one explores the spine computations in a detailed pyramidal neuron model with 25,000 spines, while the second trains a morphologically detailed neural network for image classification tasks and tests the hypothesis of dendrites against adversarial attacks[34]. DeepDendrite can implement these tasks with unprecedented speed and therefore, significantly promotes detailed neuroscience simulations and AI explorations.

Comparisons with previous methods

Decades of efforts have been invested in speeding up the Hines method with parallel methods. Early work mainly focuses on network-level parallelization. In network simulations, each cell independently solves its corresponding linear equations with the Hines method. Network-level parallel methods distribute a network on multiple threads and parallelize the computation of each cell groups with each thread[59, 60]. With network-level methods, we can simulate detailed networks on clusters or supercomputers[61]. In recent years, GPU has been used for detailed network simulation. Because the GPU contains massive computing units, one thread is usually assigned one cell rather than a cell group[33, 62, 63]. With further optimization, GPU-based methods achieve much higher efficiency in network simulation. However, the computation inside the cells is still serial in network-level methods, so they still cannot deal with the problem when the 'Hines matrix' of each cell scales large.

Cell-level parallel methods further parallelize the computation inside each cell. The main idea of cell-level parallel methods is to split each cell into several sub-blocks and parallelize the computation of those sub-blocks[25, 26]. However, typical cell-level methods (e.g. the "multi-split" method[26]) pay fewer attention to the parallelization strategy. The lack of fine parallelization strategy results in an unsatisfactory performance. To achieve higher efficiency, some studies try to obtain finer-grained parallelization by introducing extra computation operations[27, 37, 64] or making approximation on some crucial compartments, while solving linear equations[65, 66]. These finer-grained parallelization strategies can get higher efficiency but lack sufficient numerical accuracy as in the original Hines method.

Unlike previous methods, DHS adopts the finest-grained parallelization strategy, i.e. compartment level parallelization. By modeling the problem of "how to parallelize" as a combinatorial optimization problem, DHS provides an optimal compartment-level parallelization strategy. Moreover, DHS does not introduce any extra operation or value approximation, so it achieves the lowest computational cost and retains sufficient numerical accuracy as in the original Hines' method at the same time.

Promoting explorations on dendritic spines and structural plasticity

Dendritic spines are the most abundant microstructures in the brain for projection neurons in cortex, hippocampus, cerebellum and the basal ganglia. As spines receive most of the excitatory inputs in the central nervous system, electrical signals generated by spines are the main driving force for large-scale neuronal activities in the forebrain and cerebellum[10, 11]. The structure of the spine, with an enlarged spine head and a very thin spine-neck—leads to surprisingly high input impedance at the spine-head, which could be up to 500 MΩ, combining experimental data and the detailed compartment modeling approach[47, 67]. Due to such high input impedance, a single synaptic input can evoke a "gigantic" EPSP (~20 mV) at the spine-head level[47, 68], thereby boosting NMDA currents and ion channel currents in the spine[11]. However, in the classic single detailed compartment models, all spines are replaced by the *F* coefficient modifying the dendritic cable geometries[53]. This approach may compensate for the leak currents and capacitance currents for spines. Still, it cannot reproduce the high input impedance at the spine-head, which may weaken excitatory synaptic inputs, particularly NMDA currents, thereby reducing the nonlinearity in the neuron's input-output curve. Our modeling results are in line with this interpretation.

On the other hand, the spine's electrical compartmentalization is always accompanied by the biochemical compartmentalization[8, 51, 69], resulting in a drastic increase internal $[Ca]^{2+}$,within the spine and a cascade of molecular processes involving synaptic plasticity of importance for learning and memory. Intriguingly, the biochemical process triggered by learning, in turn, remodels the spine's morphology, enlarging (or shrinking) the spine-head, or elongating (or shortening) the spine-neck, which significantly alters the spine's electrical capacity[69-72]. Such experience-dependent changes in spine morphology, also referred to as "structural plasticity", have been widely observed in the visual cortex[73, 74], somatosensory cortex[75, 76], motor cortex [77], hippocampus[9] and the basal ganglia[78] *in vivo*. They play a critical role in motor and spatial learning, and memory formation. However, due to the computational costs, nearly all detailed network models exploit the "*F-factor*" approach to replace actual spines, and are thus unable to explore the spine functions at the system level. By taking advantage of our framework and the GPU platform, we can run a few thousands of detailed neurons models, each with tens of thousands of spines in a single GPU, while maintaining ~100 times faster than the traditional serial method in a single CPU (Fig. 5e). Therefore, it enables us to explore of structural plasticity in large-scale circuit models across diverse brain regions.

Promoting explorations of dendritic functions at the system level

Another critical issue is how to link dendrites to brain functions at the systems/network level. It has been well established that dendrites can perform

comprehensive computations on synaptic inputs due to enriched ion channels and local biophysical membrane properties[5-7]. For example, cortical pyramidal neurons can carry out sublinear synaptic integration at the proximal dendrite but progressively shift to supralinear integration at the distal dendrite[79]. Moreover, distal dendrites can produce regenerative events such as dendritic sodium spikes, calcium spikes, and NMDA spikes/plateau potentials[6, 80]. Such dendritic events are widely observed in mice[6] or even human cortical neurons[81] *in vitro*, which may offer various logical operations[6, 81] or gating functions[82, 83]. Recently, *in vivo* recordings in awake or behaving mice provide strong evidence that dendritic spikes/plateau potentials are crucial for orientation selectivity in the visual cortex[84], sensory-motor integration in the whisker system[85, 86], and spatial navigation in the hippocampal CA1 region[87].

To establish the causal link between dendrites and animal (including human) patterns of behavior, large-scale biophysically detailed neural circuit models are a powerful computational tool to realize this mission. However, running a large-scale detailed circuit model of 10,000 – 100,000 neurons generally requires the computing power of supercomputers. It is even more challenging to optimize such models for *in vivo* data, as it needs iterative simulations of the models. The DeepDendrite framework can directly support many state-of-the-art large-scale circuit models[88-90], which were initially developed based on NEURON. Moreover, using our framework, a single GPU card such as Tesla A100 could easily support the operation of detailed circuit models of up to 10,000 neurons, thereby providing carbon-efficient and affordable plans for ordinary labs to develop and optimize their own large-scale detailed models.

Promoting dendritic learning systems with biophysically detailed models

Recent works on unraveling the dendritic roles in task-specific learning have achieved remarkable results in two directions, i.e., solving challenging tasks such as image classification dataset ImageNet with simplified dendritic networks[18], and exploring full learning potentials on more realistic neuron[19, 20]. However, as there lies a trade-off between model size and biological detail, the increase in network scale is often sacrificed for neuron-level complexity[17, 18, 91]. Likewise, more detailed neuron models are less mathematically tractable and computationally expensive[19].

With the help of DeepDendrite, we are able to establish our HPC-Net along with the routines for training and evaluation and reduce the total run time to an affordable amount. Although we focus only on the feasibility of training biophysically detailed networks for specific tasks, more details and insights can be brought into our HPC-net. First, as we only consider passive dendritic properties here, introducing active dendritic properties is of great importance. The work of ref.[19] provides a fundamental framework for that and can naturally fit into our HPC-Net, since they are both based on multi-compartment modeling and computation of Hines' method. Second, recent studies

have shown that inhibitory interneurons can play an essential role in biologically plausible credit assignment. For example, local interneuron circuitry can be used along with dendritic segregation to achieve learning without assuming forward and backward phases[17, 91]. Also, disynaptic inhibition combined with short-term plasticity can decode multiplexed information streams, enabling parallel bottom-up and top-down signals[16, 18]. These features can be incorporated into our future development of deep dendritic neural networks. Overall, DeepDendrite provides first-hand feasibility for investigating the learning capability of detailed dendritic networks on the challenging tasks that require deep network architectures. It will benefit future research that seeks to bridge the field of neuroscience and modern artificial intelligence to solve credit assignments in the brain.

Methods

Code availability

The source code of DeepDendrite and the models in this work are available at the following link:

https://disk.pku.edu.cn:443/link/AAC281DE41777C2BBB622DB2E7EF8051

Simulation with DHS

CoreNEURON simulator (https://github.com/BlueBrain/CoreNeuron) uses the NEURON architecture and is optimized for both memory usage and computational speed. We implement our Dendritic Hierarchical Scheduling (DHS) method in CoreNEURON environment by modifying its source code. All models that can be simulated on GPU with CoreNEURON can also be simulated with DHS by executing the following command:

./coreneuron_exec -d */path/to/models* -e *time* –cell-permute *3* –cell-nthread *16* --gpu

The usage options are:

-d	Path containing the model data
-e	Simulation time (ms)
--cell-permute	Strategy for optimizing simulation: 1 and 2 for original strategies in CoreNEURON, **3 for DHS method**
--cell-nthread	Number of threads used for each cell
--gpu	Simulate on GPU

Problem formulation

Accuracy of the Simulation Using Cellular Level Parallel Computation

To ensure the accuracy of the simulation, we first need to define the correctness of a cellular level parallel algorithm to judge whether it will generate identical solutions compared with the proven correct serial methods, like the Hines method used in the NEURON simulation platform. Based on the theories in parallel computing[32], a parallel algorithm will yield an identical result as its corresponding serial algorithm, *if* and *only if* the data process order in the parallel algorithm is consistent with data dependency in the serial method. The Hines method has two symmetrical phases: triangularization and back-substitution. By analyzing the serial computing Hines method[38], we find that its data dependency can be formulated as a tree structure, where the nodes on the tree represent the compartments of the detailed neuron model. In the triangularization process, the value of each node depends on its children nodes. In contrast, during the back-substitution process, the value of each node is dependent on its parent node (Fig. 1d). Thus, we can compute nodes on different branches in parallel as their values are not dependent.

Based on the data dependency of the serial computing Hines method, we propose

three conditions to make sure a parallel method will yield identical solutions as the serial computing Hines method: 1) The tree morphology and initial values of all nodes are identical to those in the serial computing Hines method; 2) In the triangularization phase, a node can be processed if and only if all its children nodes are already processed; 3) In the back-substitution phase, a node can be processed only if its parent node is already processed. Once a parallel computing method satisfies these three conditions, it will produce identical solutions as the serial computing method.

Computational Cost of Cellular level Parallel Computing Method

To theoretically evaluate the run time, i.e. efficiency, of the serial and parallel computing methods, we introduce and formulate the concept of computational cost as follows: given a tree T and k threads (basic computational units) to perform triangularization, parallel triangularization equals to divide the node set V of T into n subsets, i.e. $V = \{V_1, V_2, \ldots V_n\}$ where the size of each subset $|V_i| \leqslant k$, i.e. at most k nodes can be processed each step since there are only k threads. The process of the triangularization phase follows the order: $V_1 \rightarrow V_2 \rightarrow \ldots\ldots \rightarrow V_n$, and nodes in the same subset V_i can be processed in parallel. So, we define $|V|$ (the size of set V, i.e. n here) as the computational cost of the parallel computing method. In short, we define the computational cost of a parallel method as the number of steps it takes in the triangularization phase. Because the back-substitution is symmetrical with triangularization, the total cost of the entire solving equation phase is twice as that of the triangularization phase.

Mathematical Scheduling Problem

Based on the simulation accuracy and computational cost, we formulate the parallelization problem as a mathematical scheduling problem:

Given a tree $T=\{V, E\}$ and a positive integer k, where V is the node-set and E is the edge set. Define partition $P(V) = \{V_1, V_2, \ldots V_n\}$, $|V_i| \leqslant k$, $1 \leqslant i \leqslant n$, where $|V_i|$ indicates the cardinal number of subset V_i, i.e., the number of nodes in V_i, and for each node $v \in V_i$, all its children nodes $\{c \mid c \in children(v)\}$ must in a previous subset V_j, where $1 \leqslant j < i$. Our goal is to find an optimal partition $P^*(V)$ whose computational cost $|P^*(V)|$ is minimal.

Here subset V_i consists of all nodes that will be computed at i^{th} step (Fig 2e), so $|V_i| \leq k$ indicates that we can compute k nodes each step at most because the number of available threads is k. The restriction "*for each node $v \in V_i$, all its children nodes $\{c \mid c \in children(v)\}$ must in a previous subset Vj, where $1 \leqslant j < i$*" indicates that that node v can be processed only if all its child nodes are processed.

DHS implementation

We aim to find an optimal way to parallelize the computation of solving linear equations for each neuron model by solving the mathematical scheduling problem above. To get the optimal partition, DHS first analyzes the topology and calculates the depth $d(v)$ for all nodes $v \in V$. Then, the following two steps will be executed iteratively until every node $v \in V$ is assigned to a subset: 1) find all candidate nodes and put these nodes into candidate set Q. A node is a candidate only if all its child nodes have been processed or it does not have any child nodes. 2) if $|Q| \leq k$, i.e., the number of candidate nodes is smaller or equivalent to the number of available threads, remove all nodes in Q and put them into V_i; otherwise, remove k deepest nodes from Q and add them to subset V_i. Label these nodes as processed nodes (Fig 2d). After filling subset V_i, go to step 1) to fill in the next subset V_{i+1}.

Proof of correctness and optimality for DHS

After applying DHS to a neural tree $T = \{V, E\}$, we get a partition $P(V) = \{V_1, V_2, \ldots V_n\}$, $|V_i| \leq k$, $1 \leq i \leq n$. Nodes in the same subset V_i will be computed in parallel, taking n steps to perform triangularization and back-substitution, respectively. We then demonstrate that the reordering of the computation in DHS will result in a result identical to the serial Hines method and that its computation cost (n) is minimum.

Correctness proof

The partition $P(V)$ obtained from DHS decides the computation order of all nodes in a neural tree. Below we demonstrate that the computation order determined by $P(V)$ satisfies the correctness conditions. $P(V)$ is obtained from the given neural tree T. Operations in DHS do not modify the tree topology and values of tree nodes (corresponding values in the linear equations), so the tree morphology and initial values of all nodes are not changed, which satisfies condition 1: the tree morphology and initial values of all nodes are identical to those in serial Hines method. In triangularization, nodes are processed from subset V_1 to V_n. As shown in the implementation of DHS, all nodes in subset V_i are selected from the candidate set Q, and a node can be put into Q only if all its child nodes have been processed. Thus the child nodes of all nodes in V_i are in $\{V_1, V_2, \ldots V_{i-1}\}$, meaning that a node is only computed after all its children have been processed, which satisfies condition 2: in triangularization, a node can be processed if and only if all its child nodes are already processed. In back-substitution, the computation order is the opposite of that in triangularization, i.e., from V_n to V_1. As shown before, the child nodes of all nodes in V_i are in $\{V_1, V_2, \ldots V_{i-1}\}$, so parent nodes of nodes in V_i are in $\{V_{i+1}, V_{i+2}, \ldots V_n\}$, which satisfies condition 3: in back-substitution, a node can be processed only if its parent

node is already processed.

Optimality proof

The idea of the proof is that if there is another optimal solution, it can be transformed into our DHS solution without increasing the number of steps the algorithm requires, thus indicating that the DHS solution is optimal.

For each subset V_i in $P(V)$, DHS moves k (thread number) deepest nodes from the corresponding candidate set Q_i to V_i. If the number of nodes in Q_i is smaller than k, move all nodes from Q_i to V_i. To simplify, we introduce D_i, indicating the depth sum of k deepest nodes in Q_i. All subsets in $P(V)$ satisfy the max-depth criteria: $\sum_{v_i \in V_i} d(v_i) = D_i$. We then prove that selecting the deepest nodes in each iteration makes $P(V)$ an optimal partition. If there exists an optimal partition $P^*(V) = \{V_1^*, V_2^*, \ldots V_s^*\}$ containing subsets that do not satisfy the max-depth criteria, we can modify the subsets in $P^*(V)$ so that all subsets consist of the deepest nodes from Q and the number of subsets ($|P^*(V)|$) remain the same after modification.

Without any loss of generalization, we start from the first subset V_i^* not satisfying the criteria, i.e., $\sum_{v_i^* \in V_i^*} d(v_i^*) < D_i$. There are two possible cases that will make V_i^* not satisfy the max-depth criteria: 1) $|V_i^*| < k$ and there exists some valid nodes in Q_i that are not put to V_i^*; 2) $|V_i^*| = k$ but nodes in V_i^* are not the k deepest nodes in Q_i.

For case 1), because some candidate nodes are not put to V_i^*, these nodes must be in the subsequent subsets. As $|V_i^*| < k$, we can move the corresponding nodes from subsequent subsets to V_i^*, which will not increase the number of subsets and make V_i^* satisfy the criteria (Fig. S6b, top). For case 2), $|V_i^*| = k$, these deeper nodes that are not moved from the candidate set Q_i into V_i^* must be added to subsequent subsets (Fig. S6b, bottom). These deeper nodes can be moved from subsequent subsets to V_i^* through the following method. Assume that after filling V_i^*, v is picked and one of the kth deepest nodes v' is still in Q_i, thus v' will be put into a subsequent subset V_j^* ($j > i$). We first move v from V_i^* to V_{i+1}^*, then modify subset V_{i+1}^* as follows: if $|V_{i+1}^*| \leq k$ and none of the nodes in V_{i+1}^* is the parent of node v, stop modifying the latter subsets. Otherwise, modify V_{i+1}^* as follows: if the parent node of v is in V_{i+1}^*, move this parent node to V_{i+2}^*; else move the node with minimum depth from V_{i+1}^* to V_{i+2}^*. After adjusting V_i^*, modify subsequent subsets $V_{i+1}^*, V_{i+2}^*, \ldots V_{j-1}^*$ with the same strategy. Finally, move v' from V_j^* to V_i^*.

With the modification strategy described above, we can replace all shallower nodes in V_i^* with the kth deepest node in Q_i and keep the number of subsets, i.e. $|P^*(V)|$ the same after modification. We can modify the nodes with the same strategy for all subsets in $P^*(V)$ that do not contain the deepest nodes. Finally, all subsets $V_i^* \in P^*(V)$ can satisfy the max-depth criteria, and $|P^*(V)|$ does not change after modifying.

In conclusion, DHS generates a partition *P(V),* and all subsets $V_i \in P(V)$ satisfy the max-depth condition: $\sum_{v_i \in V_i} d(v_i) = D_i$. For any other optimal partition $P^*(V)$ we can modify its subsets to make its structure the same as *P(V),* i.e., each subset consists of the deepest nodes in the candidate set, and keep $|P^*(V)|$ the same after modification. So, the partition *P(V)* obtained from DHS is one of the optimal partitions.

GPU implementation and memory boosting

To achieve high memory throughput, GPU utilizes the memory hierarchy of 1) global memory, 2) cache, 3) register, where global memory has large capacity but low throughput, while registers have low capacity but high throughput. We aim to boost memory throughput by leveraging the memory hierarchy of GPU.

GPU employs SIMT (Single-Instruction, Multiple-Thread) architecture. Warps are the basic scheduling units on GPU (a warp is a group of 32 parallel threads). A warp executes the same instruction with different data for different threads[45]. Correctly ordering the nodes is essential for this batching of computation in warps, to make sure DHS obtains identical results as the serial Hines method. When implementing DHS on GPU, we first group all cells into multiple warps based on their morphologies. Cells with similar morphologies are grouped in the same warp. We then apply DHS on all neurons, assigning the compartments of each neuron to multiple threads. Because neurons are grouped into warps, the threads for the same neuron are in the same warp. Therefore, the intrinsic synchronization in warps keeps the computation order consistent with the data dependency of the serial Hines method. Finally, threads in each warp are aligned and rearranged according to the number of compartments.

When a warp loads pre-aligned and successively-stored data from global memory, it can make full use of the cache, which leads to high memory throughput, while accessing scatter-stored data would reduce memory throughput. After compartments assignment and threads rearrangement, we permute data in global memory to make it consistent with computing orders so that warps can load successively-stored data when executing the program. Moreover, we put those necessary temporary variables into registers rather than global memory. Registers have the highest memory throughput, so the use of registers further accelerates DHS.

Explore neuronal excitability with full spine models

Biophysical model

We used the published human pyramidal neuron[50]. The membrane capacitance c_m = 0.44 μF/cm^2, membrane resistance r_m = 48300 Ω cm^2 and axial resistivity r_a = 261.97 Ω cm. In this model, all dendrites were modeled as passive cables while somas were active. The leak reversal potential E_l = -83.1 mV. Ion channels such as Na^+ and K^+

were inserted on soma and axon, and their reversal potentials were E_{Na} = 67.6 mV, E_K = -102 mV respectively. All these specific parameters were set the same as in the model of ref.[50], for more details please refer to the published model (ModelDB, access No.238347).

In the few-spine model, the membrane capacitance and maximum leak conductance of the dendritic cables 60 μm away from soma were multiplied by a F_{spine} factor to approximate dendritic spines. In this model, F_{spine} was set to 1.9. Only the spines that receive synaptic inputs were explicitly attached to dendrites.

In the full-spine model, all spines were explicitly attached to dendrites. We calculated the spine density with the reconstructed neuron in ref[50]. The spine density was set to 1.3/μm, and each cell contained 24994 spines on dendrites 60 μm away from the soma.

The morphologies and biophysical mechanisms of spines were the same in few-spine and full-spine models. The length of the spine neck L_{neck} = 1.35 μm and the diameter D_{neck} = 0.25 μm, whereas the length and diameter of the spine head were 0.944 μm, i.e., the spine head area was set to 2.8 μm^2. Both spine neck and spine head were modeled as passive cables, with the reversal potential E_l = -86 mV. The specific membrane capacitance, membrane resistance, and axial resistivity were the same as those for dendrites.

Synaptic inputs

We investigated neuronal excitability for both distributed and clustered synaptic inputs. All activated synapses were attached to the terminal of the spine head. For distributed inputs, all activated synapses were randomly distributed on all dendrites. For clustered inputs, each cluster consisted of 20 activated synapses that were uniformly distributed on a single randomly-selected compartment. All synapses were activated simultaneously during the simulation.

AMPA-based and NMDA-based synaptic currents were simulated as in Eyal et al.'s work. AMPA conductance was modeled as a double-exponential function and NMDA conduction as a voltage-dependent double-exponential function. For AMPA model, the specific τ_{rise} and τ_{decay} were set to 0.3 and 1.8 ms. For NMDA model, τ_{rise} and τ_{decay} were set to 8.019 and 34.9884 ms, respectively. The maximum conductance of AMPA and NMDA were 0.73 nS and 1.31 nS.

Background Noise

We attached background noise to each cell to simulate a more realistic environment. Noise patterns were implemented as Poisson spike trains with a constant rate of 1.0 Hz. Each pattern started at t_{start} =10ms and lasted until the end of the simulation. We generated 400 noise spike trains for each cell and attached them to randomly-selected synapses. The model and specific parameters of synaptic currents were the same as those in ***Synaptic Inputs***, except that the maximum conductance of NMDA was

uniformly distributed from 1.57 to 3.275, resulting in a higher AMPA to NMDA ratio.

Explore neuronal excitability

We investigated the spike probability when multiple numbers of synapses were activated simultaneously. For distributed inputs, we tested 14 cases, from 0 to 240 activated synapses. For clustered inputs, we tested 9 cases in total, activating from 0 to 12 clusters respectively. Each cluster consisted of 20 synapses. For each case in both distributed and clustered inputs, we calculated the spike probability with 50 random samples. Spike probability was defined as the ratio of fired neuron number to the total sample number. All 1150 samples were simulated simultaneously on our DeepDendrite platform, reducing the simulation time from days to minutes.

Perform AI task with DeepDendrite platform

Conventional simulators lack two functionalities important to modern AI tasks: 1) processing multiple stimuli samples in a batch-like manner and 2) updating synaptic weights during simulation without manual intervention. Here we present the DeepDendrite platform, which supports both biophysical simulating and performing deep learning tasks with detailed dendritic models.

DeepDendrite consists of three modules: 1) I/O module; 2) DHS-based simulating module; 3) learning module. When training a biophysically detailed model to perform learning tasks, users first define the learning rule, then feed all training samples to the detailed model for learning. In each step during training, the I/O module picks a specific stimulus and its corresponding teacher signal (if necessary) from all training samples and attaches the stimulus to the network model. Then, DHS-based simulating module initializes the model and starts the simulation. After simulation, the learning module updates all synaptic weights according to the difference between model responses and teacher signals. After training, the learned model can achieve performance comparable with ANN. The testing phase is similar to training, except that all synaptic weights are fixed.

Image classification with the biophysically detailed network model

Image classification is a typical task in the artificial intelligence (AI) field. In this task, a model should learn to recognize the content in a given image and output the corresponding label. Here we present the HPC-Net, a network consisting of detailed human pyramidal neuron models that can learn to perform image classification tasks by utilizing the DeepDendrite platform.

HPC-Net model

HPC-Net has three layers, i.e., an input layer, a hidden layer, and an output layer.

The neurons in the input layer receive spike trains converted from images as their input. Hidden layer neurons receive the output of input layer neurons and deliver responses to neurons in the output layer. The responses of the output layer neurons are taken as the final output of HPC-Net. Neurons between adjacent layers are fully connected.

For each image stimulus, we first convert each normalized pixel to a homogeneous spike train. The interval *int(x, y)* between two spikes is constant in each spike train and is determined by the pixel value *p(x, y)* as shown in Equation 1.

$$int(x,y) = \frac{5.0}{p(x,y) + 0.01} \qquad \text{(Equation 1)}$$

In our experiment, the simulation for each stimulus lasted 50 ms. All spike trains started at 9 ms and lasted until the end of the simulation. Then we attached all spike trains to the input layer neurons in a one-to-one manner. The synaptic current triggered by the spike arriving at time t_0 is given by

$$i_{syn} = g_{syn}(v - e_{syn}) \qquad \text{(Equation 2)}$$

$$g_{syn} = g_{max} e^{-(t-t_0)/\tau} \qquad \text{(Equation 3)}$$

where v is the voltage of post synapse, the reversal potential e_{syn} = 1mV, the maximum synaptic conductance g_{max}=0.05uS, and the time constant τ = 0.5ms.

Neurons in the input layer were modeled with a passive single-compartment model. The specific parameters were set as follows: membrane capacitance c_m = 1.0 μF/cm^2, membrane resistance r_m = 1e4 Ω cm^2, axial resistivity r_a = 100 Ω cm, reversal potential of passive compartment E_l = 0mV.

The hidden layer contains a group of human pyramidal neuron models, receiving the somatic voltages of input layer neurons. The morphology was from ref.[50], and all neurons were modeled with passive cables. The specific membrane capacitance c_m = 1.5 μF/cm^2, membrane resistance r_m = 48300 Ω cm^2, axial resistivity r_a = 261.97 Ω cm, and the reversal potential of all passive cables E_l = -86 mV. Input neurons made multiple connections to randomly-selected locations on the dendrites of hidden neurons. The synaptic current activated by the ith input neuron on dendrites is defined as in Equation 4, where g_h is a constant value for scaling, w_h^i is the synaptic weight and v_{in}^i is the somatic voltage of the ith input neuron.

$$i_{dend} = \begin{cases} g_h * w_h^i * v_{in}^i, & v_{in}^i > 0 \\ 0, & v_{in}^i \leq 0 \end{cases} \qquad \text{(Equation 4)}$$

Neurons in the output layer were also modeled with a passive single-compartment model, and each neuron took the somatic voltages of hidden neurons as input. All specific parameters were set the same as those of the input neurons. Synaptic currents activated by hidden neurons are also in the form of Equation 4.

Image classification with HPC-Net

For each input image stimulus, we first normalized all pixel values to 0.0-1.0. Then

we converted normalized pixels to spike trains and attached them to input neurons. Somatic voltages of the output neurons are used to compute the predicted probability of each class, as shown in Equation 5, where p_i is the probability of ith class predicted by the HPC-Net, $\overline{v_i}$ is the mean somatic voltage during 20ms to 50ms of the ith output neuron, and C indicates the number of classes, which equals the number of output neurons. The class with maximum predicted probability is the final classification result. In this paper, we built the HPC-Net with 784 input neurons, 64 hidden neurons and 10 output neurons.

$$p_i = \frac{\exp(\overline{v_i})}{\sum_{c=0}^{C-1} \exp(\overline{v_c})} \qquad \text{(Equation 5)}$$

Learning on HPC-Net

Inspired by previous work[35], we use a gradient-based learning rule to train our HPC-Net to perform the image classification task. The loss function we use here is cross-entropy, given in Equation 6, where p_i is the predicted probability for class i, y_i indicates the actual class the stimulus image belongs to, y_i=1 if input image belongs to class i, and y_i=0 if not.

$$Loss = -\sum_{i=0}^{C-1} y_i log p_i \qquad \text{(Equation 6)}$$

When training HPC-Net, we compute the update for weight W_{ijk} (the synaptic weight of the kth synapse connecting neuron i to neuron j) at each timestep. After the simulation of each image stimulus, W_{ijk} is updated as shown in Equation 7:

$$\Delta W_{ijk} = \eta \frac{dt}{t_e - t_s} \sum_{t=t_s}^{t_e} \Delta W_{ijk}^t \qquad \text{(Equation 7)}$$

$$\Delta W_{ijk}^t = \frac{\partial Loss}{\partial v_j} \frac{\partial v_j}{\partial W_{ijk}} = \frac{\partial Loss}{\partial v_j} \frac{\partial v_j}{\partial I_{ijk}} \frac{\partial I_{ijk}}{\partial W_{ijk}} \approx \frac{\partial Loss}{\partial v_j} * r_{ijk} * g_{ijk} * v_i^t \qquad \text{(Equation 8)}$$

Here η is the learning rate, ΔW_{ijk}^t is the update value at time t, v_j, v_i are somatic voltages of neuron i and j respectively, I_{ijk} is the kth synaptic current activated by neuron i on neuron j, g_{ijk} its synaptic conductance, r_{ijk} is the transfer resistance between the kth connected compartment of neuron i on neuron j's dendrite to neuron j's soma, t_s = 30 ms, t_e = 50 ms are start time and end time for learning respectively. For output neurons, the error term $\frac{\partial Loss}{\partial v_j}$ can be computed as shown in Equation 9. For hidden neurons, the error terms are calculated from the error terms in the output layer, given in Equation 10.

$$\frac{\partial Loss}{\partial v_j^{out}} = y_j - p_j \qquad \text{(Equation 9)}$$

$$\frac{\partial Loss}{\partial v_j^h} = \sum_{k=0}^{C-1} \frac{\partial Loss}{\partial v_k^{out}} W_{jk} * r_{jk} * g_{jk} \qquad \text{(Equation 10)}$$

Since all output neurons are single-compartment, r_{jk} equals to the input resistance of the corresponding compartment, r_{kk}.

Training with mini-batch is a typical method in deep learning for achieving higher prediction accuracy and accelerating convergence in training. DeepDendrite also supports mini-batch training. When training HPC-Net with mini-batch size N_{batch}, we make N_{batch} copies of HPC-Net. During training, each copy is fed with a different training sample from the batch. DeepDendrite first computes the weight update for each copy separately. After all copies in the current training batch are done, the average weight update is calculated and weights in all copies are updated by this same amount.

Defense adversarial attack with HPC-Net

To demonstrate the robustness of HPC-Net, we tested its prediction accuracy on adversarial samples and compared it with an analogous ANN (one with the same 784-64-10 structure and ReLU activation). We first trained HPC-Net and ANN with the original training set (original clean images). Then we added adversarial noise to the test set and measured their prediction accuracy on the noisy test set. We used the foolbox (https://github.com/bethgelab/foolbox) to generate adversarial noise with FGSM method[56]. ANN was trained with Tensorflow (https://tensorflow.google.cn), and HPC-Net was trained with our DeepDendrite. For fairness, we generated adversarial noise on a significantly different network model, a 20-layer ResNet[58]. The noise level ranged from 0.02 to 0.2. We experimented on two typical datasets, MNIST[54] and Fashion MNIST[55]. Results show that the prediction accuracy of HPC-Net is 19% and 16.72% higher than that of the analogous ANN, respectively.

Acknowledgments

The authors thank Rita Zhang, Daochen Shi and members at NVIDIA for the technical help. This work is supported by National Key R&D Program of China (No. 2020AAA0130400).

References

1. McCulloch, W.S. & Pitts, W. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics* **5**, 115-133 (1943).
2. LeCun, Y., Bengio, Y. & Hinton, G. Deep learning. *Nature* **521**, 436-444 (2015).
3. Poirazi, P., Brannon, T. & Mel, B.W. Arithmetic of subthreshold synaptic summation in a model ca1 pyramidal cell. *Neuron* **37**, 977-987 (2003).
4. London, M. & Häusser, M. Dendritic computation. *Annu. Rev. Neurosci.* **28**, 503-532 (2005).
5. Branco, T. & Häusser, M. The single dendritic branch as a fundamental functional unit in the nervous system. *Current opinion in neurobiology* **20**, 494-502 (2010).

6. Stuart, G.J. & Spruston, N. Dendritic integration: 60 years of progress. *Nature neuroscience* **18**, 1713-1721 (2015).
7. Poirazi, P. & Papoutsi, A. Illuminating dendritic function with computational models. *Nature Reviews Neuroscience* **21**, 303-321 (2020).
8. Yuste, R. & Denk, W. Dendritic spines as basic functional units of neuronal integration. *Nature* **375**, 682-684 (1995).
9. Engert, F. & Bonhoeffer, T. Dendritic spine changes associated with hippocampal long-term synaptic plasticity. *Nature* **399**, 66-70 (1999).
10. Yuste, R. Dendritic spines and distributed circuits. *Neuron* **71**, 772-781 (2011).
11. Yuste, R. Electrical compartmentalization in dendritic spines. *Annual Review of Neuroscience* **36**, 429-449 (2013).
12. Rall, W. Branching dendritic trees and motoneuron membrane resistivity. *Experimental Neurology* **1**, 491-527 (1959).
13. Segev, I. & Rall, W. Computational study of an excitable dendritic spine. *journal of neurophysiology* **60**, 499-523 (1988).
14. Silver, D.*, et al.* Mastering the game of go with deep neural networks and tree search. *Nature* **529**, 484-489 (2016).
15. Silver, D.*, et al.* A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science* **362**, 1140-1144 (2018).
16. Naud, R. & Sprekeler, H. Sparse bursts optimize information transmission in a multiplexed neural code. *Proceedings of the National Academy of Sciences* **115**, E6329-E6338 (2018).
17. Sacramento, J., Costa, R.P., Bengio, Y. & Senn, W. Dendritic cortical microcircuits approximate the backpropagation algorithm. in *Proceedings of the 32nd International Conference on Neural Information Processing Systems* 8735–8746 (Curran Associates Inc., Montréal, Canada, 2018).
18. Payeur, A., Guerguiev, J., Zenke, F., Richards, B.A. & Naud, R. Burst-dependent synaptic plasticity can coordinate learning in hierarchical circuits. *Nature Neuroscience* **24**, 1010-1019 (2021).
19. Bicknell, B.A. & Häusser, M. A synaptic learning rule for exploiting nonlinear dendritic computation. *Neuron* **109**, 4001-4017.e4010 (2021).
20. Moldwin, T., Kalmenson, M. & Segev, I. The gradient clusteron: A model neuron that learns to solve classification tasks via dendritic nonlinearities, structural plasticity, and gradient descent. *PLoS computational biology* **17**, e1009015-e1009015 (2021).
21. Hodgkin, A.L. & Huxley, A.F. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology* **117**, 500-544 (1952).
22. Rall, W. Theory of physiological properties of dendrites. *Annals of the New York Academy of Sciences* **96**, 1071-1092 (1962).
23. Hines, M.L. & Carnevale, N.T. The neuron simulation environment. *Neural Computation* **9**, 1179-1209 (1997).
24. Bower, J.M. & Beeman, D. Neural modeling with genesis. in *The book of genesis: Exploring realistic neural models with the general neural simulation system* (ed. J.M. Bower & D. Beeman) 17-27 (Springer New York, New York, NY, 1998).
25. Hines, M.L., Eichner, H. & Schürmann, F. Neuron splitting in compute-bound parallel

network simulations enables runtime scaling with twice as many processors. *Journal of Computational Neuroscience* **25**, 203-210 (2008).
26. Hines, M.L., Markram, H. & Schürmann, F. Fully implicit parallel simulation of single neurons. *Journal of Computational Neuroscience* **25**, 439-448 (2008).
27. Ben-Shalom, R., Liberman, G. & Korngreen, A. Accelerating compartmental modeling on a graphical processing unit. *Frontiers in Neuroinformatics* **7**, 4 (2013).
28. Tsuyuki, T., Yamamoto, Y. & Yamazaki, T. Efficient numerical simulation of neuron models with spatial structure on graphics processing units. in *International Conference on Neural Information Processing* (ed. A. Hirose*, et al.*) 279-285 (Springer International Publishing, Cham, 2016).
29. Vooturi, D.T., Kothapalli, K. & Bhalla, U.S. Parallelizing hines matrix solver in neuron simulations on gpu. in *IEEE 24th International Conference on High Performance Computing (HiPC)* 388-397 (2017).
30. Huber, F. Efficient tree solver for hines matrices on the gpu. *arXiv e-prints*, arXiv:1810.12742 (2018).
31. Korte, B. & Vygen, J. *Combinatorial optimization theory and algorithms* (Springer, Berlin, Heidelberg, 2018).
32. Gebali, F. *Algorithms and parallel computing* (Wiley, 2011).
33. Kumbhar, P.*, et al.* Coreneuron : An optimized compute engine for the neuron simulator. *Frontiers in Neuroinformatics* **13**, 63 (2019).
34. Nayebi, A. & Ganguli, S. Biologically inspired protection of deep networks from adversarial attacks. *arXiv preprint arXiv:1703.09202* (2017).
35. Urbanczik, R. & Senn, W. Learning by the dendritic prediction of somatic spiking. *Neuron* **81**, 521-528 (2014).
36. Ben-Shalom, R., Aviv, A., Razon, B. & Korngreen, A. Optimizing ion channel models using a parallel genetic algorithm on graphical processors. *Journal of Neuroscience Methods* **206**, 183-194 (2012).
37. Mascagni, M. A parallelizing algorithm for computing solutions to arbitrarily branched cable neuron models. *Journal of Neuroscience Methods* **36**, 105-114 (1991).
38. Hines, M. Efficient computation of branched nerve equations. *International Journal of Bio-Medical Computing* **15**, 69-76 (1984).
39. Migliore, M., Messineo, L. & Ferrante, M. Dendritic ih selectively blocks temporal summation of unsynchronized distal inputs in ca1 pyramidal neurons. *Journal of Computational Neuroscience* **16**, 5-13 (2004).
40. Hemond, P.*, et al.* Distinct classes of pyramidal cells exhibit mutually exclusive firing patterns in hippocampal area ca3b. *Hippocampus* **18**, 411-424 (2008).
41. Hay, E., Hill, S., Schürmann, F., Markram, H. & Segev, I. Models of neocortical layer 5b pyramidal cells capturing a wide range of dendritic and perisomatic active properties. *PLoS computational biology* **7**, e1002107 (2011).
42. Masoli, S., Solinas, S. & D'Angelo, E. Action potential processing in a detailed purkinje cell model reveals a critical role for axonal compartmentalization. *Frontiers in Cellular Neuroscience* **9** (2015).
43. Lindroos, R.*, et al.* Basal ganglia neuromodulation over multiple temporal and structural scales—simulations of direct pathway msns investigate the fast onset of dopaminergic effects and predict the role of kv4.2. *Frontiers in Neural Circuits* **12** (2018).

44. Migliore, M.*, et al.* Synaptic clusters function as odor operators in the olfactory bulb. *Proceedings of the National Academy of Sciences of the United States of America* **112** (2015).
45. NVIDIA. Cuda c++ programming guide. (2021).
46. NVIDIA. Cuda c++ best practices guide. (2021).
47. Harnett, M.T., Makara, J.K., Spruston, N., Kath, W.L. & Magee, J.C. Synaptic amplification by dendritic spines enhances input cooperativity. *Nature* **491**, 599-602 (2012).
48. Chiu, C.Q.*, et al.* Compartmentalization of gabaergic inhibition by dendritic spines. *Science* **340**, 759-762 (2013).
49. Tønnesen, J., Katona, G., Rózsa, B. & Nägerl, U.V. Spine neck plasticity regulates compartmentalization of synapses. *Nature Neuroscience* **17**, 678-685 (2014).
50. Eyal, G.*, et al.* Human cortical pyramidal neurons: From spines to spikes via models. *Frontiers in Cellular Neuroscience* **12** (2018).
51. Koch, C. & Zador, A. The function of dendritic spines: Devices subserving biochemical rather than electrical compartmentalization. *Journal of Neuroscience* **13**, 413-422 (1993).
52. Koch, C. Dendritic spines. in *Biophysics of computation* (Oxford University Press, 1999).
53. Rapp, M., Yarom, Y. & Segev, I. The impact of parallel fiber background activity on the cable properties of cerebellar purkinje cells. *Neural Computation* **4**, 518-533 (1992).
54. Lecun, Y., Bottou, L., Bengio, Y. & Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* **86**, 2278-2324 (1998).
55. Xiao, H., Rasul, K. & Vollgraf, R. Fashion-mnist: A novel image dataset for benchmarking machine learning algorithms. (2017).
56. Goodfellow, I.J., Shlens, J. & Szegedy, C. Explaining and harnessing adversarial examples. in *International Conference on Learning Representations* (San Diego, CA, USA, 2015).
57. Aran, N. & Ganguli, S. Biologically inspired protection of deep networks from adversarial attacks. (arXiv, 2017).
58. He, K., Zhang, X., Ren, S. & Sun, J. Deep residual learning for image recognition. in *IEEE Conference on Computer Vision and Pattern Recognition* 770-778 (Las Vegas, NV, USA, 2016).
59. Goddard, N.H. & Hood, G. Large-scale simulation using parallel genesis. in *The book of genesis: Exploring realistic neural models with the general neural simulation system* (ed. J.M. Bower & D. Beeman) 349-379 (Springer New York, New York, NY, 1998).
60. Migliore, M., Cannia, C., Lytton, W.W., Markram, H. & Hines, M.L. Parallel network simulations with neuron. *Journal of Computational Neuroscience* **21**, 119 (2006).
61. Lytton, W.W.*, et al.* Simulation neurotechnologies for advancing brain research: Parallelizing large networks in neuron. *Neural Computation* **28**, 2063-2090 (2016).
62. Valero-Lara, P.*, et al.* Cuhinesbatch: Solving multiple hines systems on gpus human brain project. in *International Conference on Computational Science, ICCS 2017, 12-14 June 2017* 566-575 (Zurich, Switzerland, 2017).
63. Akar, N.A.*, et al.* Arbor — a morphologically-detailed neural network simulation library for contemporary high-performance computing architectures. in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)* 274-282 (2019).

64. Ben-Shalom, R.*, et al.* Neurogpu: Accelerating multi-compartment, biophysically detailed neuron simulations on gpus. *Journal of Neuroscience Methods* **366**, 109400 (2022).
65. Rempe, M.J. & Chopp, D.L. A predictor-corrector algorithm for reaction-diffusion equations associated with neural activity on branched structures. *siam journal on scientific computing* **28**, 2139-2161 (2006).
66. Kozloski, J. & Wagner, J. An ultrascalable solution to large-scale neural tissue simulation. *Frontiers in Neuroinformatics* **5**, 15 (2011).
67. Jayant, K.*, et al.* Targeted intracellular voltage recordings from dendritic spines using quantum-dot-coated nanopipettes. *Nature nanotechnology* **12**, 335-342 (2017).
68. Palmer, L.M. & Stuart, G.J. Membrane potential changes in dendritic spines during action potentials and synaptic input. *Journal of Neuroscience* **29**, 6897-6903 (2009).
69. Nishiyama, J. & Yasuda, R. Biochemical computation for spine structural plasticity. *Neuron* **87**, 63-75 (2015).
70. Yuste, R. & Bonhoeffer, T. Morphological changes in dendritic spines associated with long-term synaptic plasticity. *Annual review of neuroscience* **24**, 1071-1089 (2001).
71. Holtmaat, A. & Svoboda, K. Experience-dependent structural synaptic plasticity in the mammalian brain. *Nature Reviews Neuroscience* **10**, 647-658 (2009).
72. Caroni, P., Donato, F. & Muller, D. Structural plasticity upon learning: Regulation and functions. *Nature Reviews Neuroscience* **13**, 478-490 (2012).
73. Keck, T.*, et al.* Massive restructuring of neuronal circuits during functional reorganization of adult visual cortex. *Nature neuroscience* **11**, 1162 (2008).
74. Hofer, S.B., Mrsic-Flogel, T.D., Bonhoeffer, T. & Hübener, M. Experience leaves a lasting structural trace in cortical circuits. *Nature* **457**, 313-317 (2009).
75. Trachtenberg, J.T.*, et al.* Long-term in vivo imaging of experience-dependent synaptic plasticity in adult cortex. *Nature* **420**, 788-794 (2002).
76. Marik, S.A., Yamahachi, H., McManus, J.N., Szabo, G. & Gilbert, C.D. Axonal dynamics of excitatory and inhibitory neurons in somatosensory cortex. *PLoS biology* **8**, e1000395 (2010).
77. Xu, T.*, et al.* Rapid formation and selective stabilization of synapses for enduring motor memories. *Nature* **462**, 915-919 (2009).
78. Albarran, E., Raissi, A., Jáidar, O., Shatz, C.J. & Ding, J.B. Enhancing motor learning by increasing the stability of newly formed dendritic spines in the motor cortex. *Neuron* **109**, 3298-3311. e3294 (2021).
79. Branco, T. & Häusser, M. Synaptic integration gradients in single cortical pyramidal cell dendrites. *Neuron* **69**, 885-892 (2011).
80. Major, G., Larkum, M.E. & Schiller, J. Active properties of neocortical pyramidal neuron dendrites. *Annual review of neuroscience* **36**, 1-24 (2013).
81. Gidon, A.*, et al.* Dendritic action potentials and computation in human layer 2/3 cortical neurons. *Science* **367**, 83-87 (2020).
82. Doron, M., Chindemi, G., Muller, E., Markram, H. & Segev, I. Timed synaptic inhibition shapes nmda spikes, influencing local dendritic processing and global i/o properties of cortical neurons. *Cell reports* **21**, 1550-1561 (2017).
83. Du, K.*, et al.* Cell-type–specific inhibition of the dendritic plateau potential in striatal spiny projection neurons. *Proceedings of the National Academy of Sciences* **114**, E7612-

E7621 (2017).
84. Smith, S.L., Smith, I.T., Branco, T. & Häusser, M. Dendritic spikes enhance stimulus selectivity in cortical neurons in vivo. *Nature* **503**, 115-120 (2013).
85. Xu, N.-l.*, et al.* Nonlinear dendritic integration of sensory and motor input during an active sensing task. *Nature* **492**, 247-251 (2012).
86. Takahashi, N., Oertner, T.G., Hegemann, P. & Larkum, M.E. Active cortical dendrites modulate perception. *Science* **354**, 1587-1590 (2016).
87. Sheffield, M.E. & Dombeck, D.A. Calcium transient prevalence across the dendritic arbour predicts place field properties. *Nature* **517**, 200-204 (2015).
88. Markram, H.*, et al.* Reconstruction and simulation of neocortical microcircuitry. *Cell* **163**, 456-492 (2015).
89. Billeh, Y.N.*, et al.* Systematic integration of structural and functional data into multi-scale models of mouse primary visual cortex. *Neuron* **106**, 388-403.e318 (2020).
90. Hjorth, J.*, et al.* The microcircuits of striatum in silico. *Proceedings of the National Academy of Sciences* **117**, 202000671 (2020).
91. Guerguiev, J., Lillicrap, T.P. & Richards, B.A. Towards deep learning with segregated dendrites. *eLife* **6**, e22901 (2017).

Figure Legends

Figure 1 Accelerate simulation of biophysically detailed neuron models.

a, A reconstructed layer-5 pyramidal neuron model and the mathematical formula used with detailed neuron models.

b, Workflow when numerically simulating detailed neuron models. The equation solving phase is the bottleneck in the simulation.

c, An example of linear equations in the simulation.

d, Data dependency of the Hines method when solving linear equations in **c**.

e, The size of Hines matrix scales with model complexity. The number of linear equations system to be solved undergoes a significant increase when models are growing more detailed.

f, Computational cost of the serial Hines method on different types of neuron models.

g, Illustration of different solving methods. Different parts of a neuron are assigned to multiple processing units in parallel methods (mid, right), shown with different colors. In serial method (left), all compartments are computed with one unit.

h, Computational cost of three methods in **g** when solving equations of a pyramidal model with spines.

i, Run time of different methods on solving equations for 500 pyramidal models with spines. The run time indicates the time consumption of 1s simulation (solving the equation 40000 times during the simulation). p-Hines: parallel method in CoreNEURON (on GPU); Branch based: branch based parallel method (on GPU); DHS: Dendritic hierarchical scheduling method (on GPU).

Figure 2 Dendritic Hierarchical Scheduling (DHS) method significantly reduce the computational cost, i.e. computational steps in solving equations.

a, DHS work flow. DHS processes k deepest candidate nodes each iteration.

b, Illustration of calculating node depth of a compartmental model. The model is first converted to a tree structure then the depth of each node is computed. Colors indicate different depth values.

c, Topology analysis on different neuron models. Six neurons with distinct morphologies are shown here. For each model, soma is selected as root of the tree so the depth of the node increases from the soma (0) to the distal dendrites.

d, Illustration of performing DHS on the model in **b** with four threads. Candidates: nodes that can be processed. Selected candidates: nodes that are picked by DHS, i.e. the k deepest candidates. Processed nodes: nodes that have been processed before.

e, Parallelization strategy obtained by DHS after the process in **d**. Each node is assigned to one of the four parallel threads. DHS reduces the steps of serial node processing from 14 to 5 by distributing nodes to multiple threads.

f, Computational cost (steps taken in the equation solving phase) when applying DHS with different thread numbers on different types of models. Relative cost, i.e. the ratio of DHS cost to NEURON's serial Hines method cost, is used here.

Figure 3 GPU memory boosting further accelerates DHS.

a, GPU architecture and its memory hierarchy. Each GPU contains massive processing units (stream processors). Different types of memory have different throughput.

b, Architecture of Streaming Multiprocessors (SMs). Each SM contains multiple streaming processors, registers and L1 cache.

c, Apply DHS on two neurons, each with four threads. During simulation, each thread executes on one stream processor.

d, Memory optimization strategy on GPU. Top pannel, thread assignment and data storage of DHS, before (left) and after (right) memory boosting. Bottom, an example of single step in triangularization when simulating two neurons in **d**. Processors send a data request to load data for each thread from global memory. Without memory boosting (left), it takes seven transactions to load all request data and some extra transactions for intermediate results. With memory boosting (right), it takes only two transactions to load all request data, registers are used for intermediate results, which further improve memory throughput.

e, Run time of DHS (32 threads each cell) with and without memory boosting on multiple layer 5 pyramidal models with spines.

f, Speed up of memory boosting on multiple layer 5 pyramidal models with spines. Memory boosting brings 1.6-2 folds speedup.

Figure 4 DHS enables cell-type-specific finest partition.

a, Run time of solving equations for a 1s-simulation on GPU (dt = 0.025ms, 40000 iterations in total). CoreNEURON: parallel method used in CoreNEURON; DHS-4: DHS with four threads each neuron; DHS-16: DHS with 16 threads each neuron

b, c, Visualization of the partition by DHS-4 and DHS-16, each color indicates a single thread. During computation, each thread switches among different branches.

Figure 5 DHS enables spine level modelling

a, Experiment setup. We examine two major types of models: few spine model and full spine model. Few spine models (two in left) are the models that incorporated spine area globally into dendrites and only attach individual spines together with activated synapses. In full spine models (two in right), all spines are explicitly attached over whole dendrites. We explore the effects of clustered and randomly distributed synaptic inputs on the few spine model and the full spine model, respectively.

b, Somatic voltages recorded for cases in **a**.Colors of the voltage curves e correspond to **a**, scale bar: 20ms, 20mV.

c, Color-coded voltages during the simulation in **b** at specific time. Colors indicate the magnitude of voltage.

d, Somatic spike probability as a function of the number of simultaneously activated synapses (as in Eyal et al.'s work) for four cases in **a**. Background noise are attached.

e, Run time of experiments in **d** with different simulation methods. NEURON: conventional NEURON simulator run on single CPU core. CoreNEURON: CoreNEURON simulator on GPU. DeepDendrite: DeepDendrite on GPU.

Figure 6 DeepDendrite enables learning with detailed neural networks.

a, The illustration of a Human Pyramidal Cell network (HPC-Net) for image classification. Images are transformed to spike trains and fed into the network model. Learning are triggered by error signals propagated from soma to dendrites.

b, Training with mini-batch. Multiple networks are simulated simultaneously with different images as inputs. The weights update amounts ΔW are computed as the average of ΔW_i from each network.

c, Comparison of HPC-Net before and after training. Left, the visualization of hidden neuron responses to a specific input before (top) and after (bottom) training. Right, hidden layer weights (from input to hidden layer) distribution before (top) and after (bottom) training.

d, Workflow of the defending adversarial attack experiment. We first generate adversarial samples of the test set on a 20-layer ResNet. Then use these adversarial samples (noisy images) to test the classification accuracy of models trained with clean images.

e, Prediction accuracy of each model after training 30 epochs on adversarial samples of MNIST (left) and fashion MNIST (right) datasets.

f, Run time of training and testing of HPC-Net, we set batch size to 16. Left, run time of training one epoch. Right, run time of testing. Parallel NEURON + Python: training and testing on multiple CPU cores, use 40-processes parallel NEURON to simulate HPC-Net, and implement I/O module and learning module with extra Python code. DeepDendrite: use DeepDendrite to train or test HPC-Net, the whole procedure of training and testing are on GPU.

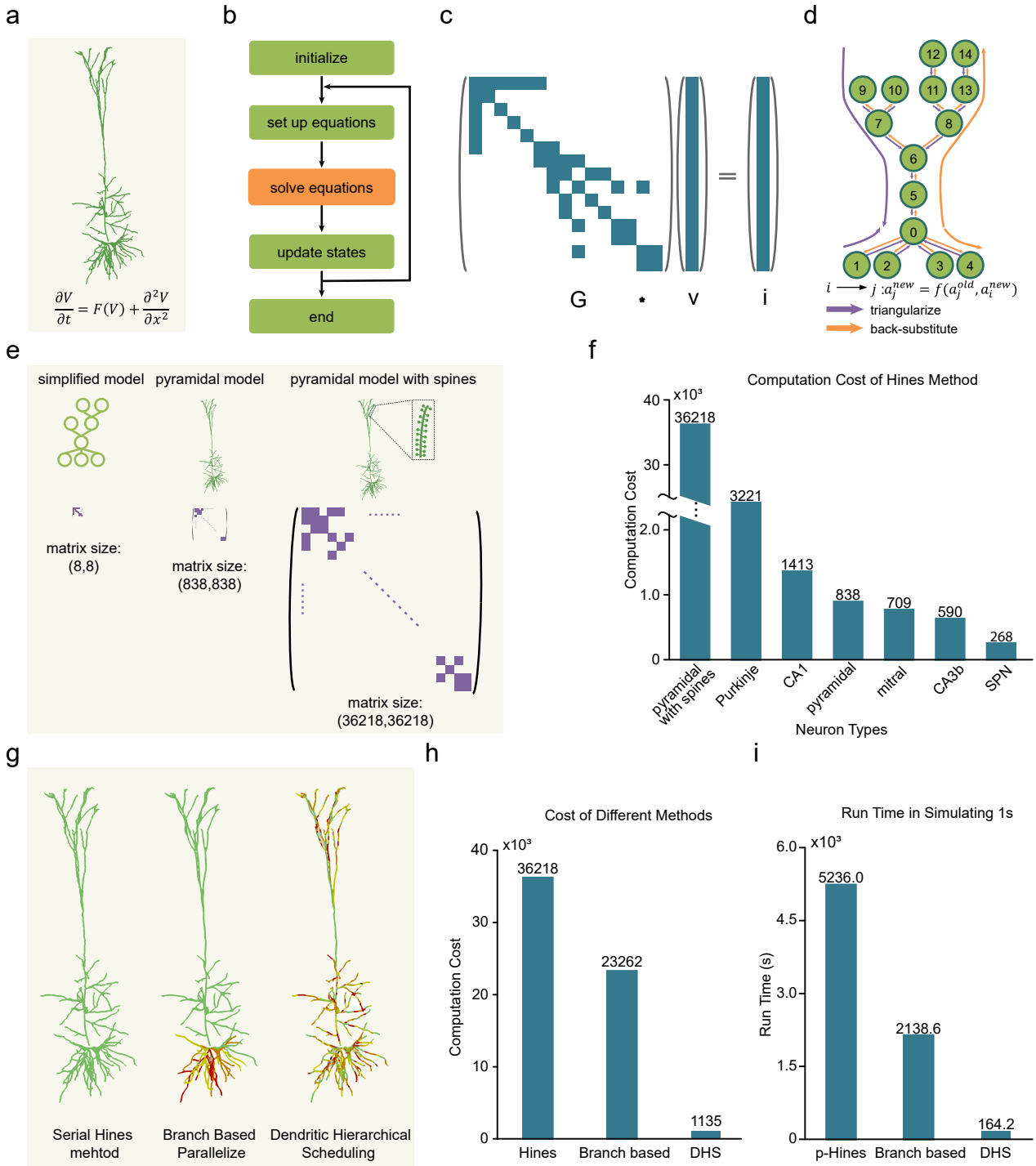

a
$\frac{\partial V}{\partial t} = F(V) + \frac{\partial^2 V}{\partial x^2}$
b
initialize
set up equations
solve equations
update states
end
c
G
*
v
=
i
d
$i \longrightarrow j : a_j^{new} = f(a_j^{old}, a_i^{new})$
triangularize
back-substitute
e
simplified model
pyramidal model
pyramidal model with spines
matrix size: (8,8)
matrix size: (838,838)
matrix size: (36218,36218)
f
Computation Cost of Hines Method
x10³
Computation Cost
36218
3221
1413
838
709
590
268
pyramidal with spines
Purkinje
CA1
pyramidal
mitral
CA3b
SPN
Neuron Types
g
Serial Hines mehtod
Branch Based Parallelize
Dendritic Hierarchical Scheduling
h
Cost of Different Methods
x10³
Computation Cost
36218
23262
1135
Hines
Branch based
DHS
i
Run Time in Simulating 1s
x10³
Run Time (s)
5236.0
2138.6
164.2
p-Hines
Branch based
DHS

Figure 1

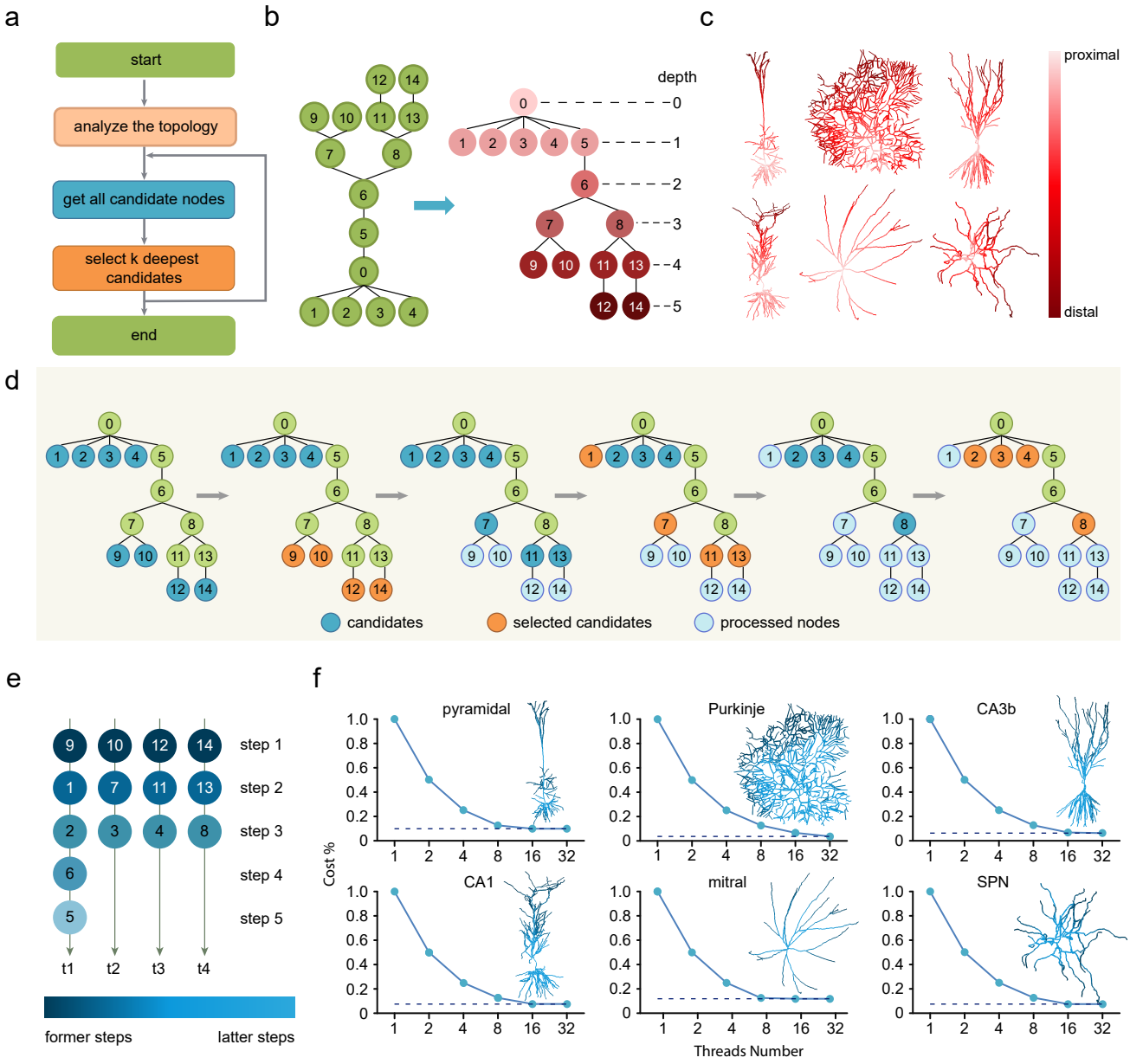

a
start
analyze the topology
get all candidate nodes
select k deepest candidates
end
b
depth
c
proximal
distal
d
candidates
selected candidates
processed nodes
e
step 1
step 2
step 3
step 4
step 5
t1
t2
t3
t4
former steps
latter steps
f
pyramidal
Purkinje
CA3b
CA1
mitral
SPN
Cost %
Threads Number

Figure 2

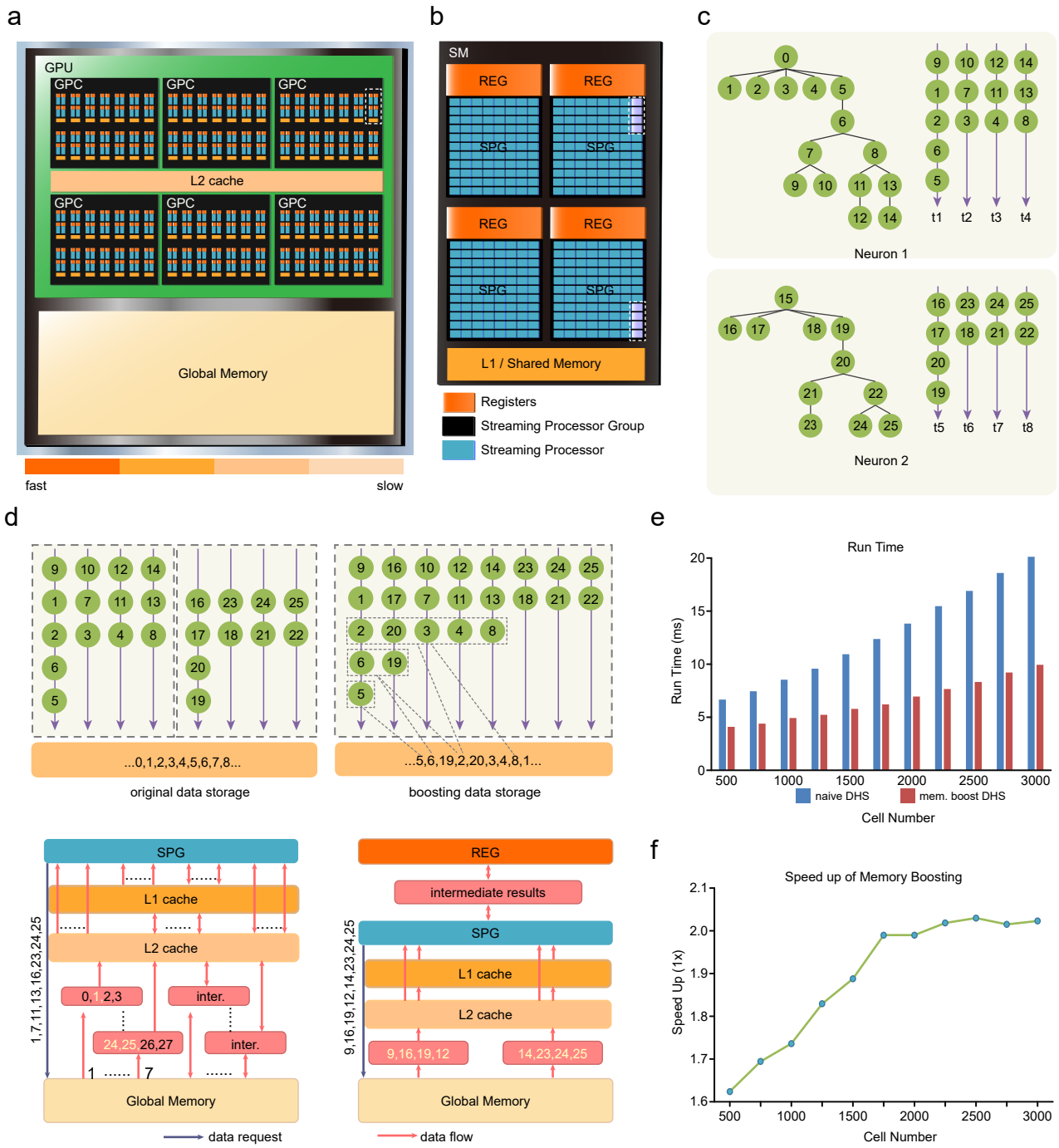

a
GPU
GPC
L2 cache
Global Memory
fast
slow
b
SM
REG
SPG
L1 / Shared Memory
Registers
Streaming Processor Group
Streaming Processor
c
Neuron 1
Neuron 2
t1
t2
t3
t4
t5
t6
t7
t8
d
...0,1,2,3,4,5,6,7,8...
original data storage
...5,6,19,2,20,3,4,8,1...
boosting data storage
SPG
L1 cache
L2 cache
inter.
Global Memory
data request
REG
intermediate results
9,16,19,12
14,23,24,25
data flow
e
Run Time
Run Time (ms)
naive DHS
mem. boost DHS
Cell Number
f
Speed up of Memory Boosting
Speed Up (1x)
Cell Number

Figure 3

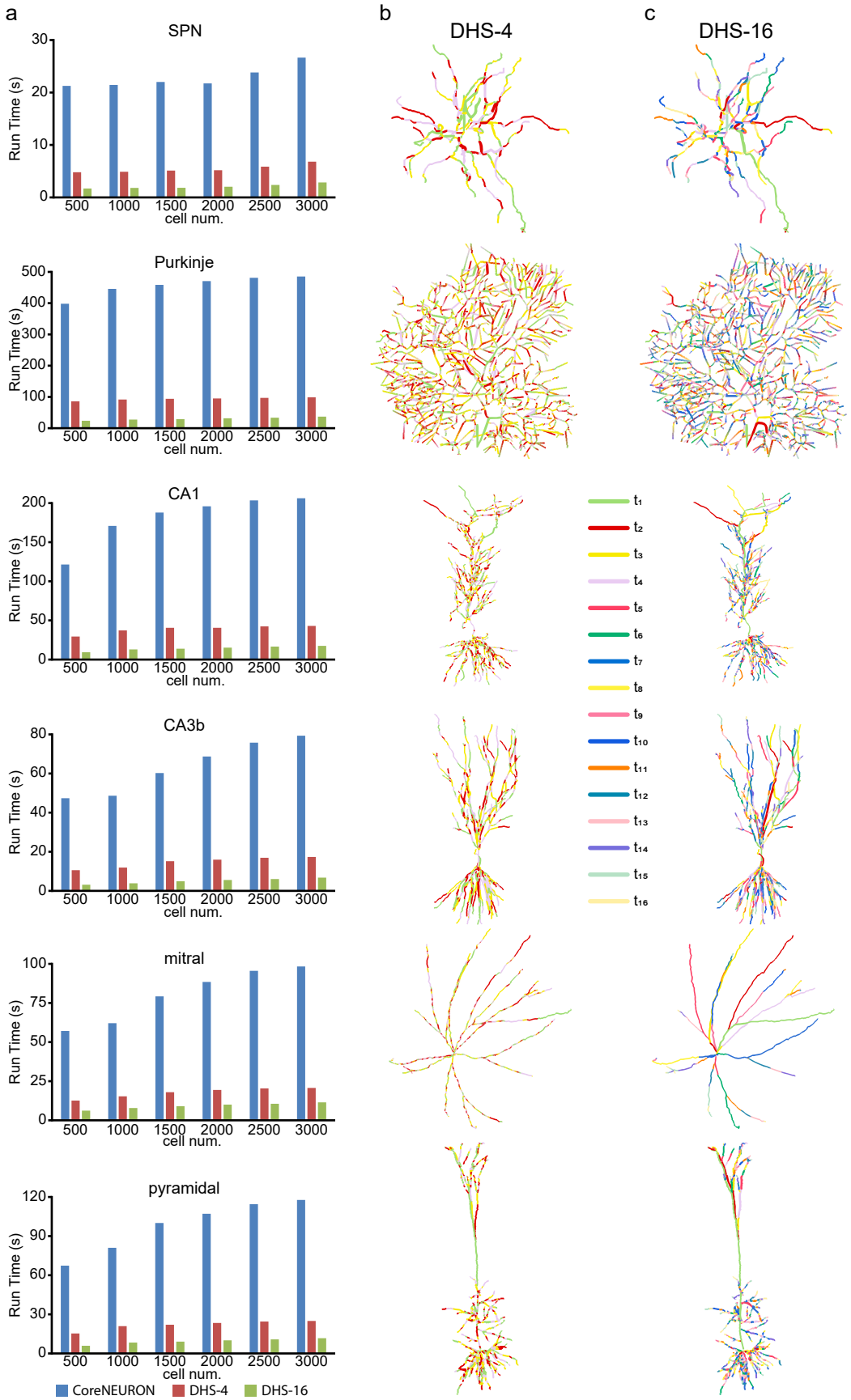

a
b
c
DHS-4
DHS-16
SPN
Purkinje
CA1
CA3b
mitral
pyramidal
Run Time (s)
cell num.
500
1000
1500
2000
2500
3000
CoreNEURON
DHS-4
DHS-16
t1
t2
t3
t4
t5
t6
t7
t8
t9
t10
t11
t12
t13
t14
t15
t16

Figure 4

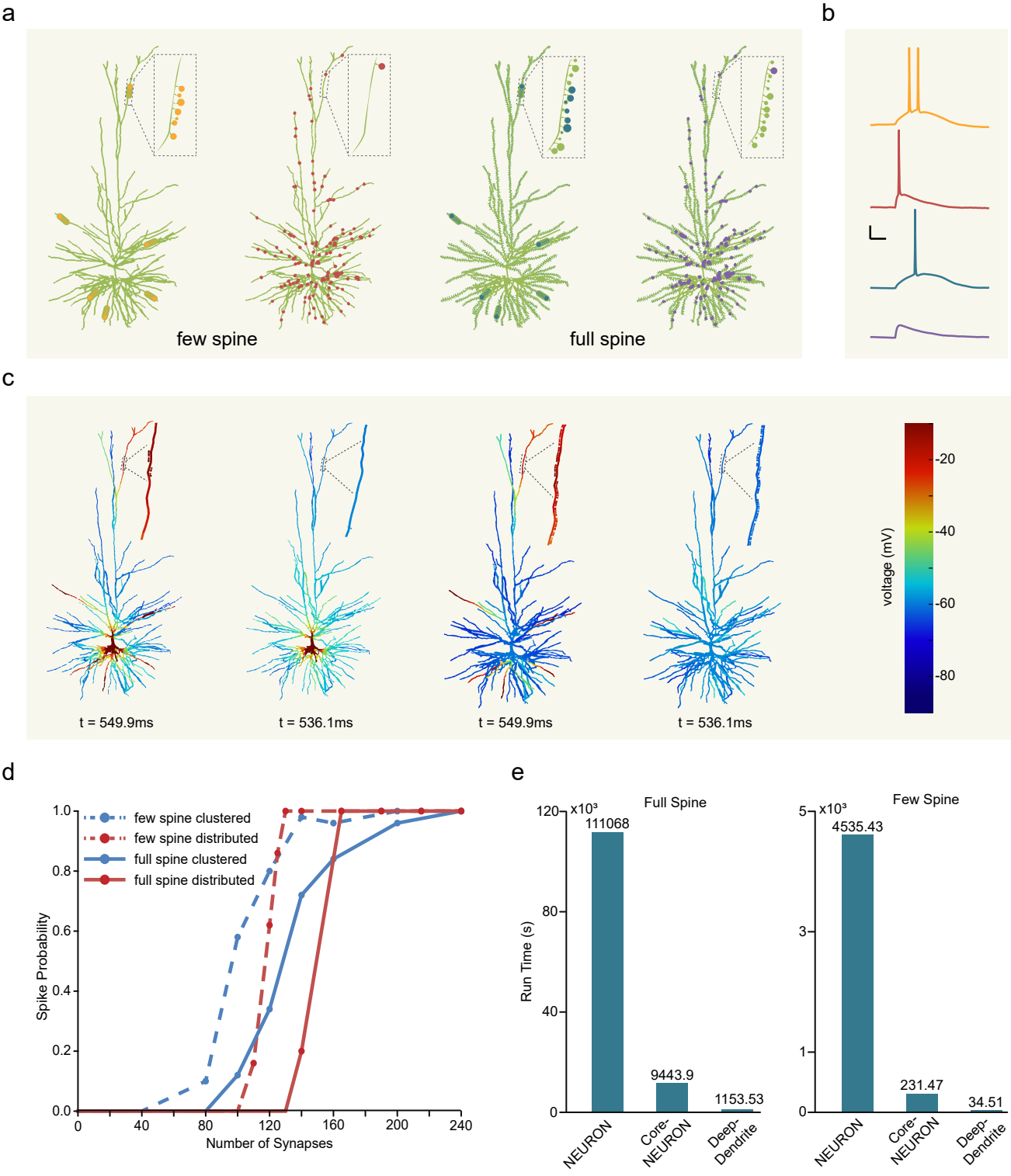

a
few spine
full spine
b
c
t = 549.9ms
t = 536.1ms
t = 549.9ms
t = 536.1ms
voltage (mV)
-20
-40
-60
-80
d
few spine clustered
few spine distributed
full spine clustered
full spine distributed
Spike Probability
Number of Synapses
e
Full Spine
Few Spine
Run Time (s)
x10³
111068
9443.9
1153.53
4535.43
231.47
34.51
NEURON
Core-NEURON
Deep-Dendrite

Figure 5

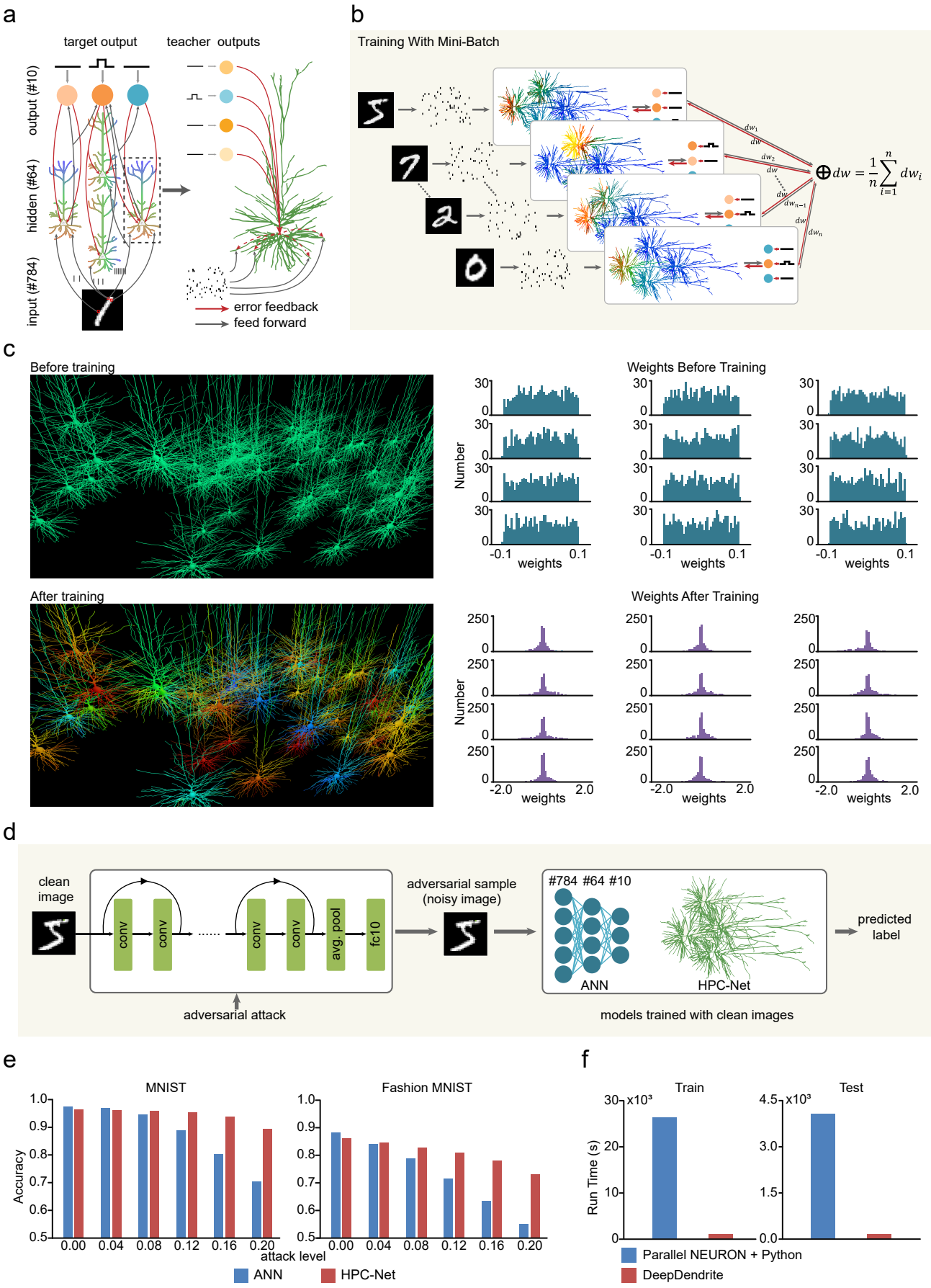

a
target output
teacher outputs
output (#10)
hidden (#64)
input (#784)
error feedback
feed forward
b
Training With Mini-Batch
$dw = \frac{1}{n}\sum_{i=1}^{n} dw_i$
c
Before training
Weights Before Training
Number
weights
After training
Weights After Training
d
clean image
conv
avg. pool
fc10
adversarial attack
adversarial sample (noisy image)
#784 #64 #10
ANN
HPC-Net
predicted label
models trained with clean images
e
MNIST
Fashion MNIST
Accuracy
attack level
ANN
HPC-Net
f
Train
Test
Run Time (s)
Parallel NEURON + Python
DeepDendrite

Figure 6

Supplemental information

A GPU-based computational framework that bridges Neuron simulation and Artificial Intelligence

Yichen Zhang, Gan He, Xiaofei Liu, J.J.Johannes Hjorth, Alexander Kozlov, Yutao He, Shenjian Zhang, Lei Ma, Jeanette Hellgren Kotaleski, Yonghong Tian, Sten Grillner, Kai Du, Tiejun Huang

Table S1 Summary of speedup referred in our experiments

Ours	Phase	Compared method	Model	SpeedUp
DHS	solving equations	CoreNEURON method (GPU)	pyramidal	10.08-11.35
			purkinje	13.11-17.00
			CA3b	11.76-15.00
			CA1	11.78-13.63
			mitral	7.85-9.14
			SPN	9.39-12.56
		serial Hines method (CPU)	pyramidal	174.11-953.65
			purkinje	285.67-1589.42
			CA3b	145.43-800.48
			CA1	238.41-1162.62
			mitral	127.85-713.91
			SPN	67.10-270.73
DeepDendrite	simulation	serial NEURON (CPU)	full spine model	96.37
		CoreNEURON (GPU)	full spine model	8.19
		serial NEURON (CPU)	few spine model	131.42
		CoreNEURON (GPU)	few spine model	6.71
	training	40-process parallel NEURON (CPU)	HPC-Net	25.39
	testing	40-process parallel NEURON (CPU)	HPC-Net	26.54

CoreNEURON version: 0.14, NEURON version: 7.6.2; CPU: Intel Xeon E5-2698 v4, GPU: NVIDIA Tesla A100

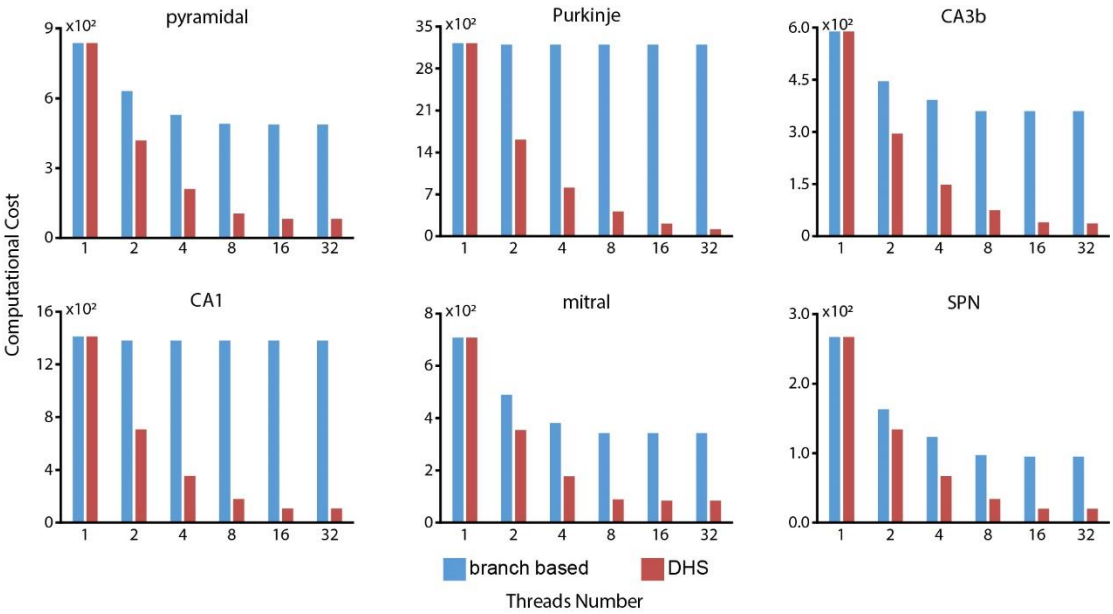


Figure S1 DHS outperforms branch based method in computational cost. DHS achieves low computational cost on various types of neurons while the costs of branch based method are much higher. The cost of branch based method remains high despite the growth of thread number, and in some cases (e.g. Purkinje cell and CA1 cell) only little gain on cost is achieved.

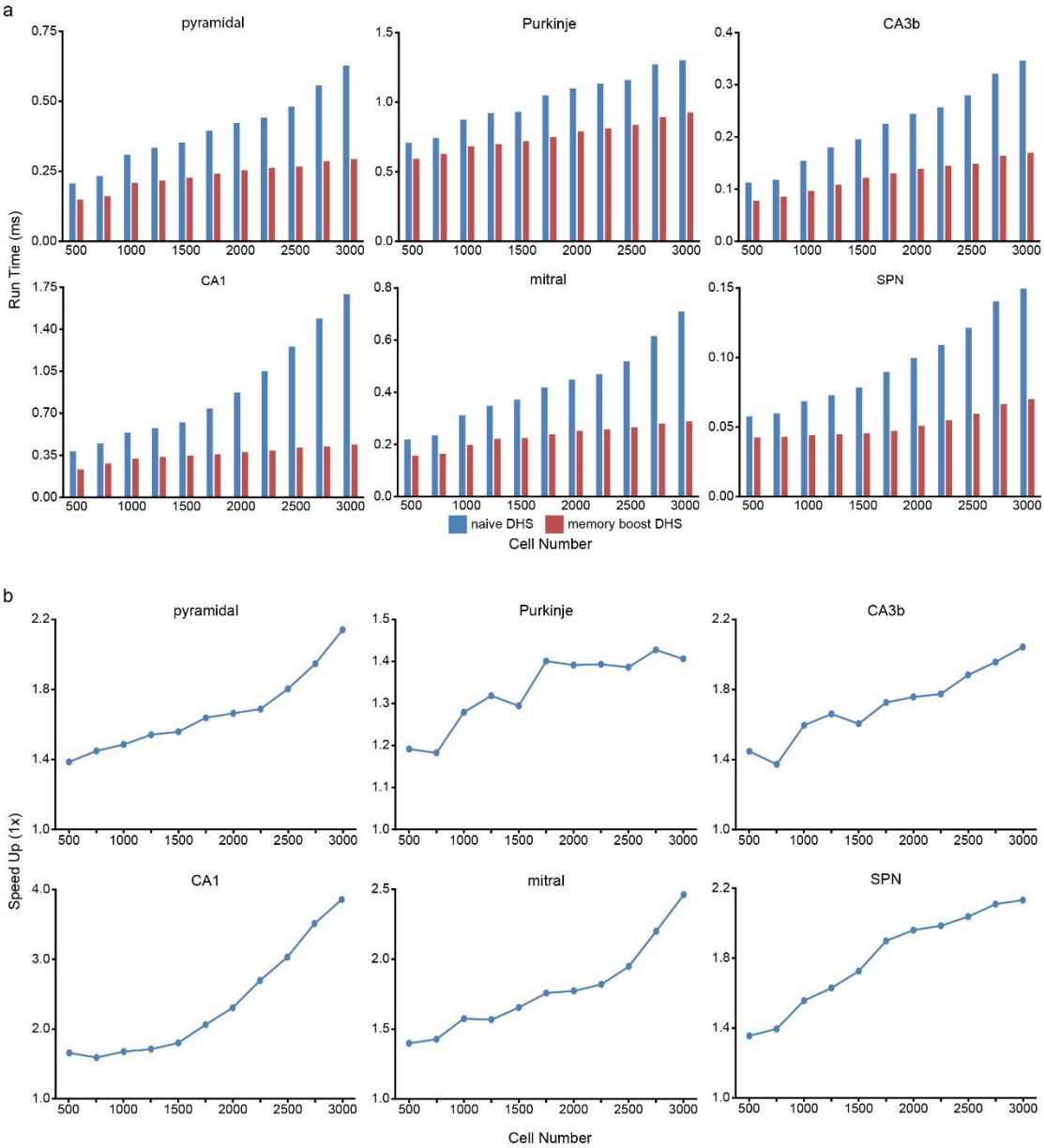


Figure S2 GPU memory boosting further speeds up DHS. a. GPU memory boosting reduce run time of DHS on various types of neurons. **b.** With GPU memory boosting, DHS becomes 1.2-3.8 times faster on different types of neurons

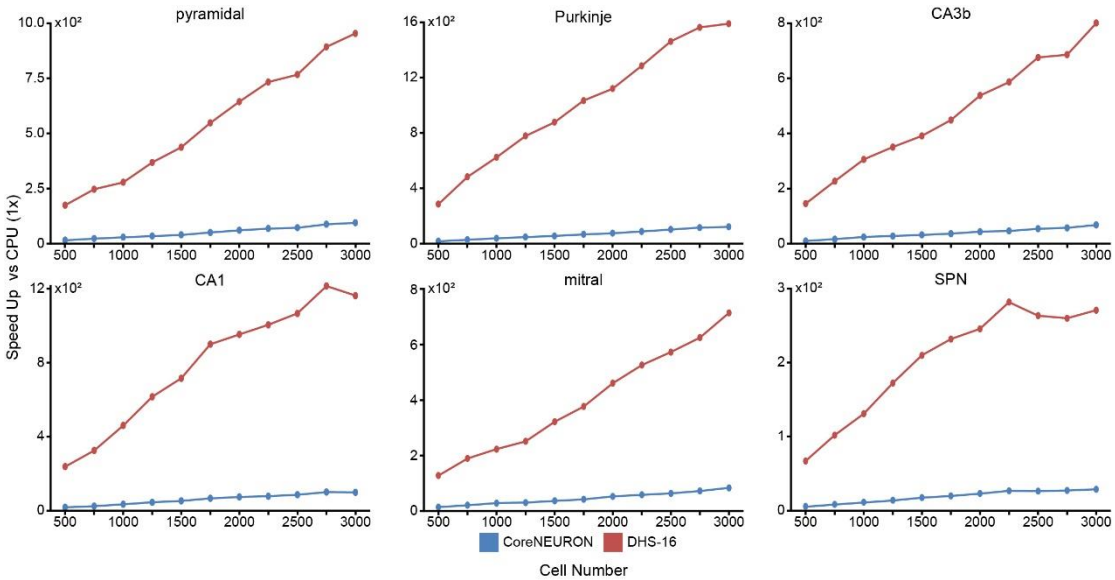
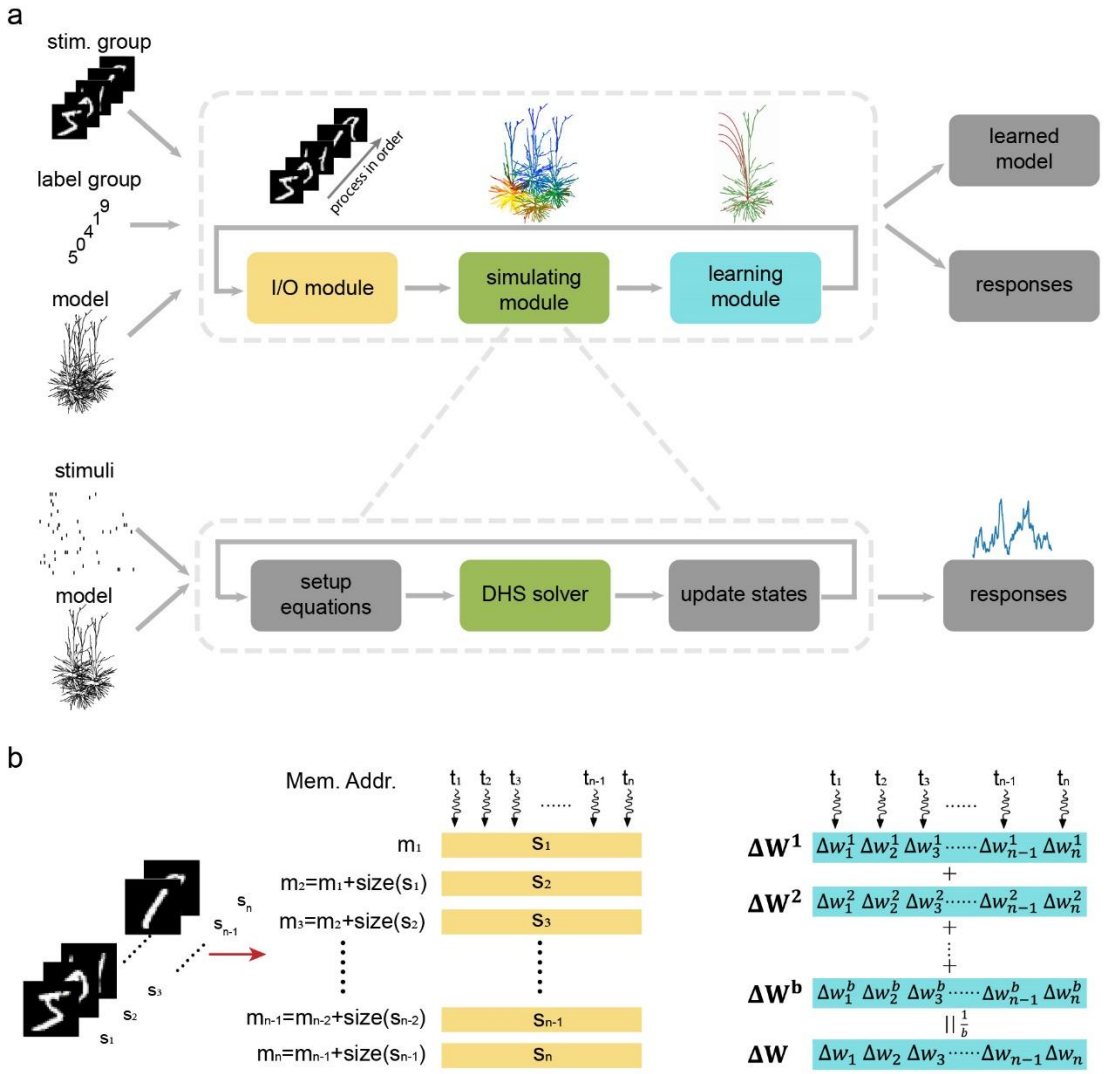


Figure S3 DHS achieves 2-3 orders of magnitude speedup compared to the serial Hines method on CPUs. On the test neurons used in this study, the GPU parallel method used in CoreNEURON achieves speed up of 2.5-120 folds compared to serial Hines method on CPU. DHS method is 7-17 times faster than the method in CoreNEURON, and achieves speed up of 60-1500 folds compared to serial Hines method on CPU.



c

Algorithm 1 Training on DeepDendrite

Input Model M, Training Set $TrainSet$, Training Set size N, batch size b, iteration number N_{iter}

Output Weights after training $\mathbf{W}$

```
function GPUTrain(M, TrainSet, N, b, N_iter)
    istim = 0
    iter = 0
    while iter < N_iter do
        s, l = TrainSet[istim : (istim + b)%N]
        Attach stimuli s and labels l to the model M
        Simulate to get the responses of all neurons
        Compute ΔW_1, ... ΔW_b
        Set ΔW = mean(ΔW_1, ... ΔW_b)
        Update all synaptic weights W = W + η ΔW
        istim = (istim + b)%N
        iter += 1
    end while
end function
```

Algorithm 2 Testing on DeepDendrite

Input Model M, Test Set $TestSet$, Set size N, batch size b

Output Responses of the model $\mathbf{Y}_{pred}$

```
function GPUTest(M, TestSet, N, b)
    istim = 0
    while istim < N do
        s, l = TestSet[istim : istim + b]
        Attach stimuli s to the model
        Simulate to get all neural responses y_1, y_2 ... y_b
        Set Y_pred[istim : istim + b] = {y_1, y_2, ..., y_b}
        istim = (istim + b)%N
    end while
end function
```

Figure S4 The framework and implementation of DeepDendrite. a. DeepDendrite framework**.** DeepDendrite consists of three modules: I/O module, simulating module and learning module. During learning, DeepDendrite takes the detailed model and all

training samples as input, and save the model after learning. During each iteration in training, I/O module picks specific stimuli from all training samples and attach them to the detailed model. Then simulating module starts simulation and gets the responses of the detailed model. After simulation, learning module updates synaptic weights according to the network responses and the target signal. **b.** GPU implementation and optimization of I/O module (left) and learning module (right). **c.** The procedure of training (left) and testing (right) when performing AI tasks with detailed network model on DeepDendrite.

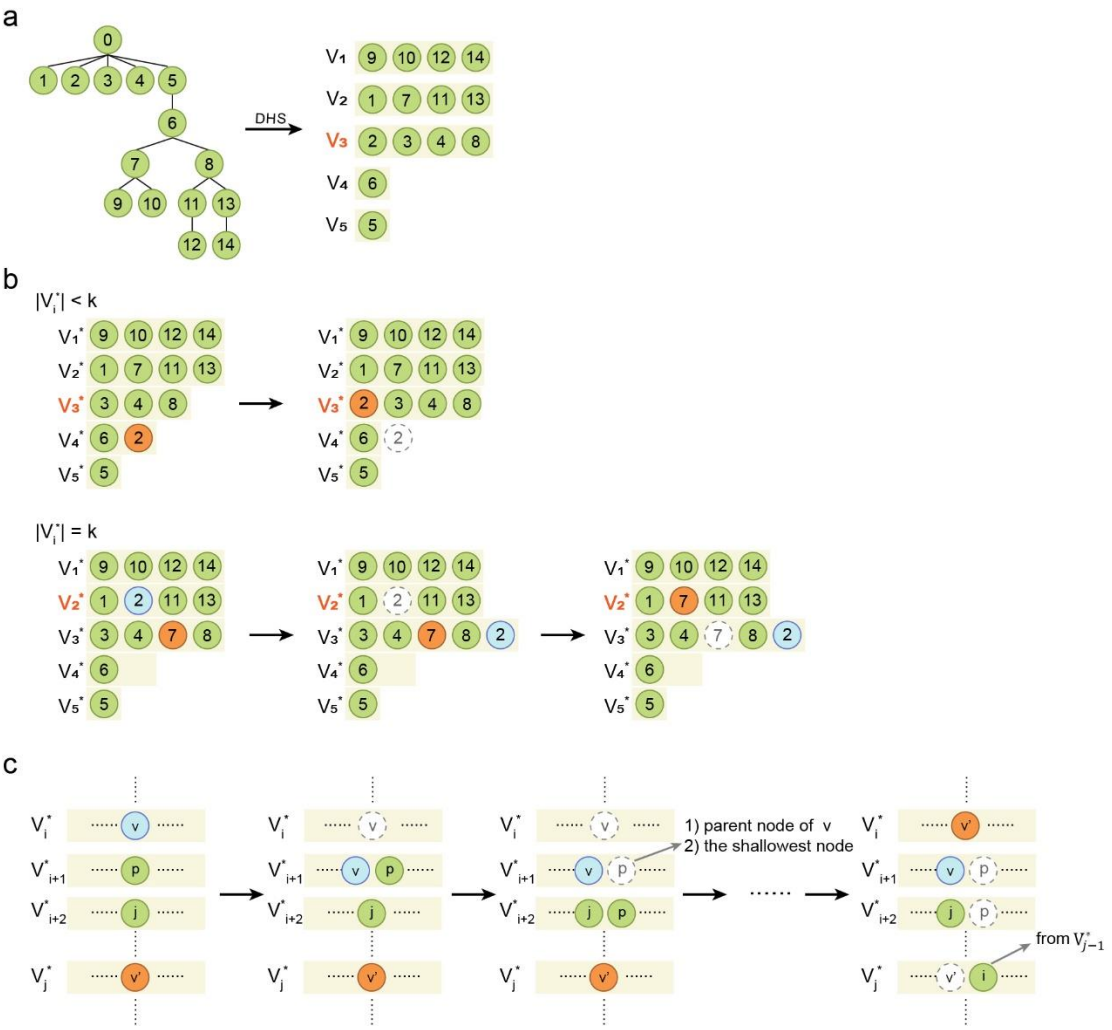


Figure S5 Strategy for modifying subset V_i^* to make it satisfy the max-depth criteria. a. partition generated by DHS. **b.** A simple example showing how to modify subset V_i^*. **c.** The general case when $|V_i^*| = k$