A system hierarchy for brain-inspired computing

https://doi.org/10.1038/s41586-020-2782-y

Received: 24 January 2020

Accepted: 10 August 2020

Published online: 14 October 2020

Check for updates

Youhui Zhang[1,2,3,8]✉, Peng Qu[1,2,3,8], Yu Ji[1,2,3,8], Weihao Zhang[2,4,8], Guangrong Gao[5], Guanrui Wang[2,4], Sen Song[2,6], Guoqi Li[2,4], Wenguang Chen[1,3], Weimin Zheng[1,3], Feng Chen[2,7], Jing Pei[2,4], Rong Zhao[2], Mingguo Zhao[2,7] & Luping Shi[2,4]✉

Neuromorphic computing draws inspiration from the brain to provide computing technology and architecture with the potential to drive the next wave of computer engineering[1–13]. Such brain-inspired computing also provides a promising platform for the development of artificial general intelligence[14,15]. However, unlike conventional computing systems, which have a well established computer hierarchy built around the concept of Turing completeness and the von Neumann architecture[16–18], there is currently no generalized system hierarchy or understanding of completeness for brain-inspired computing. This affects the compatibility between software and hardware, impairing the programming flexibility and development productivity of brain-inspired computing. Here we propose 'neuromorphic completeness', which relaxes the requirement for hardware completeness, and a corresponding system hierarchy, which consists of a Turing-complete software-abstraction model and a versatile abstract neuromorphic architecture. Using this hierarchy, various programs can be described as uniform representations and transformed into the equivalent executable on any neuromorphic complete hardware—that is, it ensures programming-language portability, hardware completeness and compilation feasibility. We implement toolchain software to support the execution of different types of program on various typical hardware platforms, demonstrating the advantage of our system hierarchy, including a new system-design dimension introduced by the neuromorphic completeness. We expect that our study will enable efficient and compatible progress in all aspects of brain-inspired computing systems, facilitating the development of various applications, including artificial general intelligence.

Brain-inspired computing is a computing model and architecture that has the potential to break the von Neumann bottleneck[1] and drive the next wave of computer engineering[2]. Brain-inspired computing systems have been used for artificial intelligence[3–14], and may provide a route towards artificial general intelligence[15]. The application of brain-inspired computing to more general algorithms, other than artificial intelligence, has also been explored[19–21]. All these applications present challenges for the performance, programmability and productivity of brain-inspired computing systems.

Various algorithms, computational models and software designs for brain-inspired computing are emerging. Although numerous neuromorphic chips have been proposed[5–14], they usually require specific software toolchains[22–25]. As a result, multiple layers of the brain-inspired computing system—including the application model, system software and neuromorphic device—are bound together, impairing the programming flexibility and development productivity. Some studies have tried to bridge the various software and hardware through domain-specific languages[26] or development frameworks[19,20,27], but these studies usually either do not consider completeness or implicitly rely on Turing completeness. Little work has been done to address more fundamental issues, such as hardware completeness, programming-language completeness and the generalized system hierarchy of brain-inspired computing.

Existing computer hierarchy, such as of the Turing machine[16] and the von Neumann architecture[17,18], provides insight into the importance of these issues for computing systems (Supplementary Information section 1). Nearly all existing programming languages are Turing-complete (that is, they have the same capability as a universal Turning machine) and the von Neumann abstract architecture supports a Turing machine through a Turing-complete interface (that is, a general-purpose instruction set). Through the introduction of Turing completeness and a hierarchy based on Turing completeness and the von Neumann architecture, tight coupling between software and hardware is avoided in current computing systems, enabling efficient, compatible and independent

[1]Department of Computer Science and Technology, Tsinghua University, Beijing, China. [2]Center for Brain-Inspired Computing Research (CBICR), Tsinghua University, Beijing, China. [3]Beijing National Research Center for Information Science and Technology, Beijing, China. [4]Department of Precision Instruments, Tsinghua University, Beijing, China. [5]Department of Electrical and Computer Engineering, University of Delaware, Newark, DE, USA. [6]Department of Biomedical Engineering, Tsinghua University, Beijing, China. [7]Department of Automation, Tsinghua University, Beijing, China. [8]These authors contributed equally: Youhui Zhang, Peng Qu, Yu Ji, Weihao Zhang. ✉e-mail: zyh02@tsinghua.edu.cn; lpshi@tsinghua.edu.cn

progress. By setting the minimum requirements for hardware (Turing completeness), it became feasible to transform any program in a high-level language into an equivalent instruction sequence on any von Neumann processor (compilation).

By contrast, brain-inspired computing currently lacks a simple but sound system hierarchy to support overall development[28]. As a result, there are no clear and complete interfaces between neuromorphic software and hardware and the interactions between the different research aspects are complex[29]. Further, because many brain-inspired chips are not designed for general-purpose computing and few of them provide traditional instruction sets, it is unclear whether they are Turing-complete, or even whether Turing completeness is necessary.

Turing completeness is the feasibility foundation of traditional compilation, requiring equivalence in program expression and transformation. By contrast, brain-inspired systems possess distinctive attributes, such as approximation (brain-inspired systems usually follow the computing model of neural networks to mimic the behaviours or characteristics of biological neural networks)[30–32]. For example, for many brain-inspired chips, including early neural-inspired chips[33] and contemporary brain-inspired systems[5–14], approximation is a way to achieve low power and high performance, implemented using either low-precision digital calculations[5,6,8–10,14] or analogue circuits[7,11–13,33–38]. We therefore propose neuromorphic completeness, a more adaptive and broader definition of completeness for brain-inspired computing. It relaxes the completeness requirement for neuromorphic hardware, which could improve the compatibility between different hardware and software designs, and enlarge the design space by introducing a new dimension, the approximation granularity.

Neuromorphic computing is also distinct from traditional computing[2] in that it uses colocated computing and storage, uses event-driven computation based on spikes[39] (the characteristic of spiking neural networks) and has greater potential for high parallelism, among other things. These differences make it difficult for traditional computer hierarchies to describe brain-inspired applications intuitively and to execute them efficiently. We therefore further propose a system hierarchy for brain-inspired computing with high versatility and universality. This hierarchy has three levels: software, hardware and compilation.

Software
Software refers to programming languages or frameworks and the algorithms or models built on them. At this level, we propose a uniform and general software-abstraction model—the programming operator graph (POG)—to accommodate the various brain-inspired algorithms and model designs. The POG is composed of a unified description method and an event-driven, parallel program-execution model that integrates storage and processing. It describes what a brain-inspired program is and defines how it is executed. Because the POG is Turing complete, it support the various applications, programming languages and frameworks to the greatest extent.

Hardware
Hardware includes all the brain-inspired chips and architecture models. We design the abstract neuromorphic architecture (ANA) as the hardware abstraction. It includes an execution primitive graph (EPG) as the interface to the upper layer to describe the program it can execute. The EPG has a hybrid control-flow–dataflow representation, which maximizes its adaptability for different hardware and is consistent with a promising hardware trend, the hybrid architecture[3,4].

Compilation
It is the middle layer that transforms a program into an equivalent form that hardware supports. For feasibility, we present a basic set of hardware execution primitives that is widely supported by mainstream brain-inspired chips, and prove that hardware equipped with this set is neuromorphic-complete. We also implement a toolchain software as an instance of the compilation layer to demonstrate the feasibility, rationality and advantages of the hierarchy.

With this hierarchy, we avoid tight coupling between hardware and software, ensuring that any brain-inspired program can be represented by the Turing-complete POG and then compiled into an equivalent and executable EPG on any neuromorphic complete hardware (Fig. 1). We also ensure the programming portability, hardware completeness and compilation feasibility of brain-inspired computing systems. We present experiments that demonstrate the optimization effect of the system design dimension that is introduced by neuromorphic completeness. Moreover, we argue that our hierarchy facilitates software–hardware codesign.

Neuromorphic completeness
For any given error gap $\varepsilon \geq 0$ and any Turing-computable function $f(x)$, a computational system is called neuromorphic complete if it can achieve a function $F(x)$ such that $\|F(x) - f(x)\| \leq \varepsilon$ for any valid input x (Supplementary Information section 2).

Neuromorphic completeness is used to measure the compatibility of neuromorphic computing systems. It relaxes the requirement for completeness from exactly computing a function with an algorithm to approximating it. An algorithm in the terminology of computer science is a computational procedure defined by a Turing machine. Thus, computing a function with an algorithm means that the system simulates a Turing machine and then uses the algorithm to achieve the function. By contrast, achieving a function by approximation does not require such a computation procedure.

The approximation capability of neural networks is defined by the universal approximation theorem[40]. A multilayer perceptron with only one hidden layer can approximate any function arbitrarily well. It approximates a function by memorizing the mapping of the function. On the other hand, simulating a Turing machine requires mechanisms such as recursion and control flow to achieve any number of state transitions. A multilayer perceptron with one hidden layer has only two transitions between the input and output; thus, it is not Turing-complete. However, multilayer perceptrons and Turing-complete systems are both neuromorphic-complete.

In essence, neuromorphic completeness connects universal approximation with universal computability. It lays the theoretical foundation for the feasibility of converting a Turing-complete program into an equivalent program on a neuromorphic-complete system, which broadens the scope of complete hardware. Further, because neuromorphic completeness is compatible with approximate and exact computation, it expands the design space of brain-inspired systems.

System hierarchy
In the POG (Fig. 2a, Supplementary Information section 3), a program is defined as a directed graph in which each node is an operator, with the edges describing the precedence relationship of different operators. The operator carries out the actual computation and is triggered for execution when it receives all the input events; that is, the POG is event-driven. Moreover, an operator contains only the operations that deal with the local storage and external inputs. Thus, these operations are inherently suitable for the processing mode that integrates memory and computation.

The POG greatly extends the pure dataflow activity model[41,42] (Supplementary Information section 3.6), while inheriting its support for fine-grained parallelism. The POG is Turing-complete (Supplementary Information section 3.5, Supplementary Fig. 1). Therefore, it can also be regarded as a base programming language for various brain-inspired applications and is compatible with existing brain-inspired frameworks[19,20,26,27]. The POG provides dedicated operators (Extended Data Fig. 1, Supplementary Information section 3.3) to help users describe

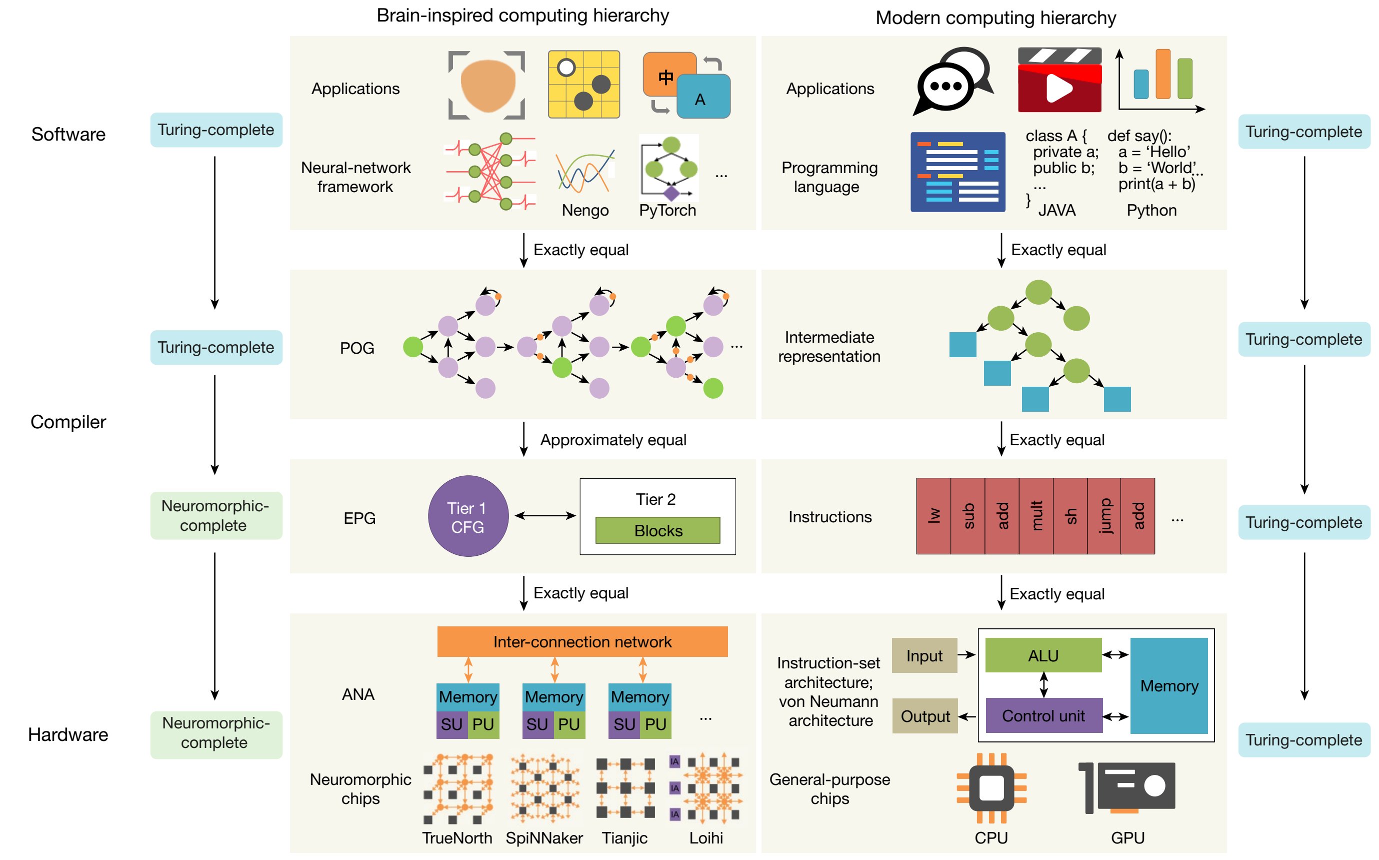


Fig. 1 | Hierarchies of the brain-inspired computing system and traditional computing systems. Inspired by traditional computing system hierarchy (right), we propose a brain-inspired computing system hierarchy (left), which also has three levels: software (top), compiler (middle) and hardware (bottom). In the traditional computing system hierarchy, the software layer refers to various applications and the Turing-complete programming languages (such as JAVA and Python). During the compilation procedure, intermediate representations of software (such as the abstract syntax tree) will be converted to intermediate representations of hardware (such as instructions). In the hardware layer, the instructions are run on central processing units (CPUs) or graphics processing units (GPUs) that follow the von Neumann architecture. The von Neumann architecture includes an arithmetic and logic unit (ALU), control unit, memory, input and output. The precise equivalence between different layers is assured by Turing completeness. For the brain-inspired computing system hierarchy, the software layer refers to the neuromorphic applications and developing frameworks (such as Nengo and PyTorch). Correspondingly, we propose the POG as the intermediate representations of software and the EPG as intermediate representations of hardware (CFG, control-flow graph). The compilation tools are introduced to transform the POG into the EPG. For the hardware layer, we propose ANA, which includes schedule units (SUs), processing units (PUs), memory and an inter-connection network as the abstraction of the neuromorphic hardware (TrueNorth, SpiNNaker, Tianjic and Loihi). Considering the approximation property of brain-inspired computing, we further propose the notion of neuromorphic completeness, which introduces approximation equivalence in addition to precise equivalence.

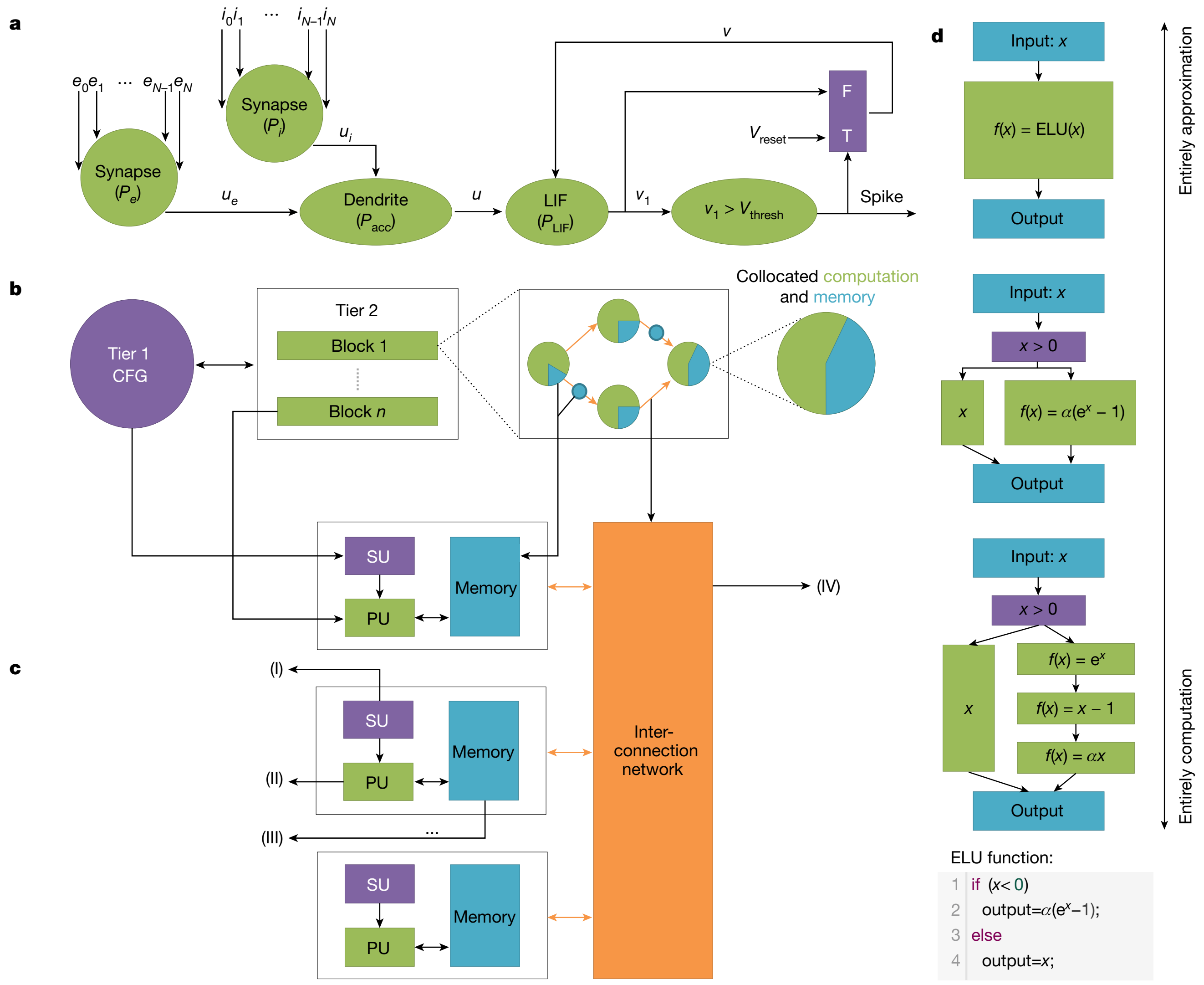
brain-inspired computing operations more easily and provide more performance hints. Another feature of the POG is composability (Supplementary Information section 3.4). We can define any part of a POG as a new operator, while keeping the rest unchanged, as long as it is supported by the underlying hardware. This enables the description of complicated models and is conducive to software–hardware codesign (Supplementary Information section 10.2).

The ANA (Supplementary Information section 6) contains massive processing units, each of which is colocated with a private memory and scheduling unit(s). The processing units provide hardware execution primitives, which perform the major computation in parallel, and are scheduled by scheduling units. All these units communicate through an interconnected network. The ANA is a logical design that is flexible to different hardware implementations (Fig. 2c). For example, the processing unit can be implemented by memristor crossbars[34–37] or general-purpose processors. The ANA is therefore capable of being instantiated into several well known neuromorphic chips (Supplementary Information section 6.1, Supplementary Fig. 3).

The interface of the ANA (the EPG) is a hybrid control-flow–dataflow two-tiered graph (Fig. 2b, Supplementary Information section 4, Supplementary Fig. 2). Tier one is a control-flow graph in which each node is a basic block containing one or more execution primitives and the directed edges represent jumps from one basic block to another. Tier two is a dataflow graph that is formed by execution primitives according to the data dependency inside each basic block.

Basic execution primitives

The basic set of execution primitives (Supplementary Information section 4.1) contains two types of computation primitive, as a multilayer perceptron does: the weighted-sum operation and the element-wise rectified linear unit operation. These primitives are generally applicable to mainstream brain-inspired chips; for example, chips that support the leaky integrate-and-fire model can also be considered to provide two primitives (Supplementary Information section 4.2). We provide a constructive proof that the EPG, with the basic execution primitives, is neuromorphic-complete (Supplementary Information section 4.3). This proof also provides direction for building the corresponding compiler that can transform any Turing-complete POG into an equivalent EPG.



```
if (x< 0)
  output=α(e^x–1);
else
  output=x;
```

Fig. 2 | POG, EPG and ANA. a, A POG of the leaky integrate-and-fire (LIF) model—a complicated model that involves normal operators, control-flow operators, a parameter updater, and so on. *e*, *i*, excitatory and inhibitory synaptic inputs, $P_{e,i}$, corresponding synaptic weights; *v*, membrane potential; V_{reset}, reset potential, V_{thresh} firing threshold; $P_{acc,LIF}$, model-related parameters; F, false; T, true. The computation of the operators are defined as follows: synapse, $u_e = \sum_k e_k P_{e_k}$ and $u_i = \sum_k i_k P_{i_k}$; dendrite, $u = P_{acc1}u_e + P_{acc2}u_i$; LIF, $v_1 = P_{LIF1}v + P_{LIF2} + u$. **b**, EPG. The tier-one control-flow graph (CFG) is executed by scheduling units (SUs) to determine which basic block is ready. If one is ready, the corresponding scheduling unit will assign all enabled primitives to some processing units (PUs). If an assigned processing unit is free, it will load all necessary data from memory, carry out the computation and deliver the output to those processing units for the subsequent primitives. **c**, The correspondence between the elements of the EPG, the units of the ANA, and possible implementations: (I) the scheduling units can be implemented centralized or distributed in the form of a soft core, look-up table, configurable logic block, and so on; (II) the processing units can be implemented as a general-purpose core or dedicated functional unit; (III) the memory options include isolated memory, near or colocated processing memory, and so on; (IV) the inter-connection network can be a bus network, network-on-chip, and so on. **d**, Programs of an activation function of the exponential linear unit (ELU) with different approximate granularities; *x*, input value; *f*(*x*), functions with different approximate granularities; *α*, a constant.

Moreover, the EPG, with this basic set, is an ideal example to show that the neuromorphic completeness connects universal approximation with universal computability. On the one hand, the universal approximator can be expressed as an extreme instance of the EPG: the tier-one control-flow graph degrades to only one block, which contains a multilayer perceptron that approximates the entire program. On the other hand, if we use basic execution primitives to achieve some basic operations precisely (for example, Boolean functions), and use these basic operations to compose more complicated computations and control-flow schemas, then the EPG constructed in this way is Turing-complete (modern digital computers are based on deterministic Boolean circuits). Between these two extremes, an EPG can be expressed in various forms with different approximation granularities (Fig. 2d), with different trade-offs between performance and resource consumption.

Toolchain, applications and experiments

We build a framework for the toolchain software according to the hierarchy, which consists of two parts: the compiler and the mapper. The compiler (Supplementary Information section 5.1) transforms the POG into an equivalent EPG. As shown in Fig. 3b, the compiler first splits or merges the operators in the POG to the proper granularity of approximation. Then, all operators that execution primitives cannot precisely implement (determined using 'template matching') are approximated using a method that follows the above constructive proof of the neuromorphic completeness of the EPG. There are several optimization techniques to reduce the resource consumption of the EPG that is generated (Supplementary Information sections 5.2–5.5). The mapper (Supplementary Information section 7, Fig. 3c,

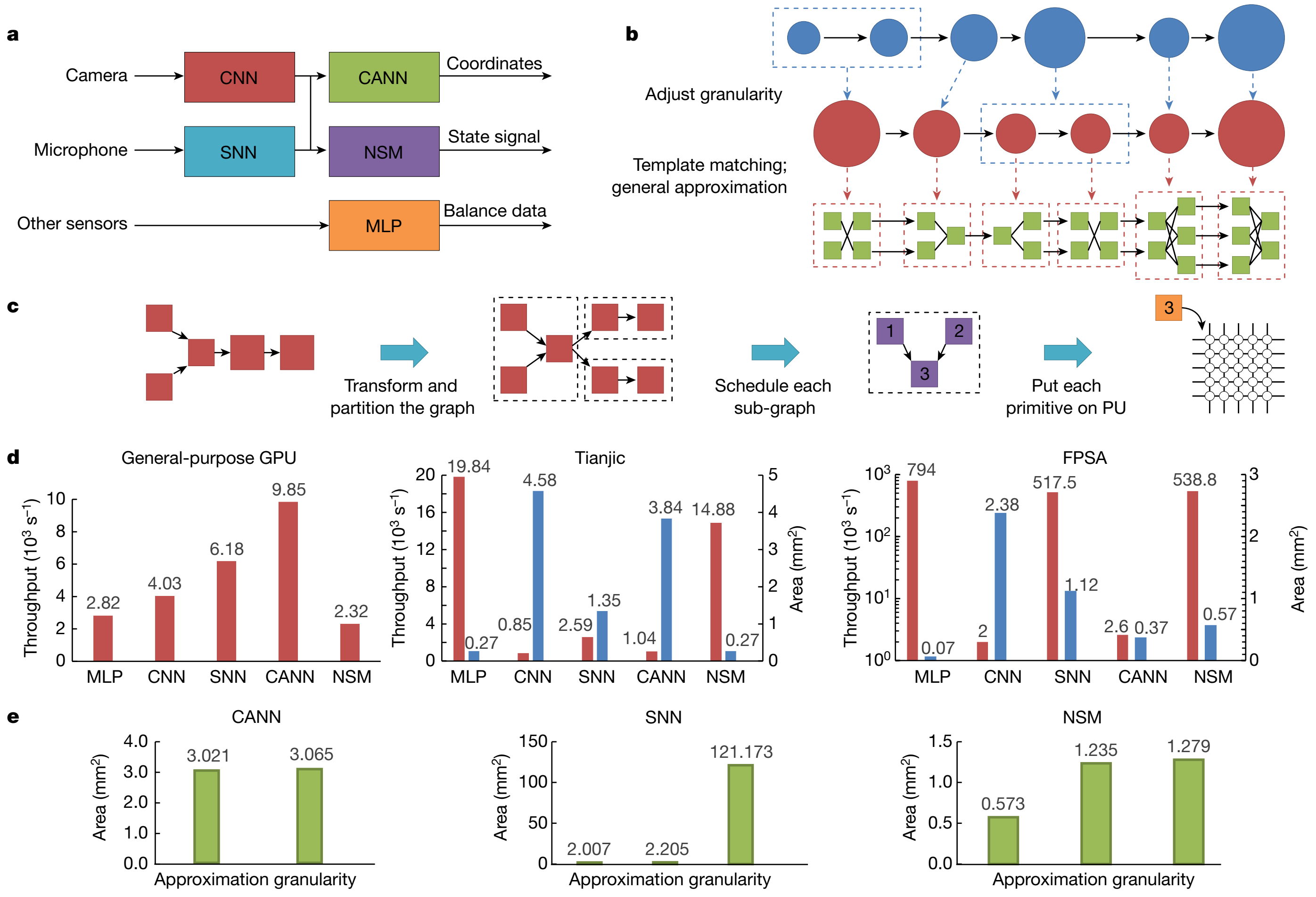


Fig. 3 | Toolchain and bicycle driving and tracking experiment. a, A convolutional neural network (CNN) for image processing and object detection, a spiking neural network (SNN) for speech recognition, a continuous attractor neural network (CANN) for object tracking and a multilayer perceptron (MLP) for sensory and control tasks; an SNN-based neural state machine (NSM) integrates them for decision-making. **b**, The compilation workflow. We first adjust the POG to an appropriate granularity and then convert it to an EPG through template-matching and/or general approximation. The details are provided in Supplementary Information section 5.1. **c**, The mapping workflow. The mapper maps the EPG to the specific hardware. It contains three steps: Partition the graph into sub-graphs, schedule each sub-graph, and map each operator to a specific component (Supplementary Information section 7). **d**, The performance (throughput; red, left axis) and hardware overheads (area; blue, right axis) of the neural networks on the three platforms. **e**, Resource consumption (area) versus approximation granularity (three neural networks on FPSA). The abscissa indicates the gradual decrease in approximation granularity (left to right). As the granularity grows, the cost decreases gradually. If we further increase the granularity, the hardware consumption increases exponentially and so cannot be illustrated in this figure.

Supplementary Fig. 4) deploys the EPG that is generated to the hardware as efficiently as possible, while satisfying the hardware constraints. We implement a toolchain instance (Methods, Supplementary Information section 8) that can convert various applications into uniform and hardware-independent intermediate representations (POGs), and compile each POG to the EPG of execution primitives specific to the target before mapping.

Currently, three hardware platforms are supported, all of which are typical neuromorphic-complete systems: (1) the general-purpose graphics processing unit (GPU), a brain-inspired chip; (2) Tianjic[14]; and (3) a memristor-based deep neural network accelerator, FPSA[36]. The general-purpose GPU is a traditional Turing-complete system, which is completely dependent on precise computing. FPSA provides efficient and high-density basic execution primitives, realizing different functions mainly through approximation. Tianjic supports both precise computing and approximation.

We carried out experiments for three applications to demonstrate the feasibility and versatility of the hierarchy, and the design tradeoff introduced by neuromorphic completeness (Methods). The first application is a hybrid spiking–artificial neural network model for bicycle driving and tracking[14]. It contains five neural networks, each a different type (Fig. 3a, Supplementary Information section 9.1). The POG of each neural network is the same across different hardware platforms before compilation. The approximation error is set to zero; that is, all three platforms behave the same in this experiment. The performance and area consumption for the three platforms are shown in Fig. 3d. Because FPSA realizes functions through approximation, the choice of approximation granularity has a large effect on the hardware cost (Fig. 3e).

The second application is the boids model[43] for bird-flock simulation. It is a non-neural-network application that requires many nonlinear tensor computations (Fig. 4a, Supplementary Information section 9.2). The toolchain can support it on the three platforms; the running performance and cost are shown in Fig. 4b. Figure 4c illustrates the behaviour of this application with different approximation errors. The greater the error (which generally means the smaller the hardware overhead), the greater the difference from the behaviour of the exact calculation. Because of the chaotic aspect of this model, the attributes of the flock movement are maintained as the approximation error is

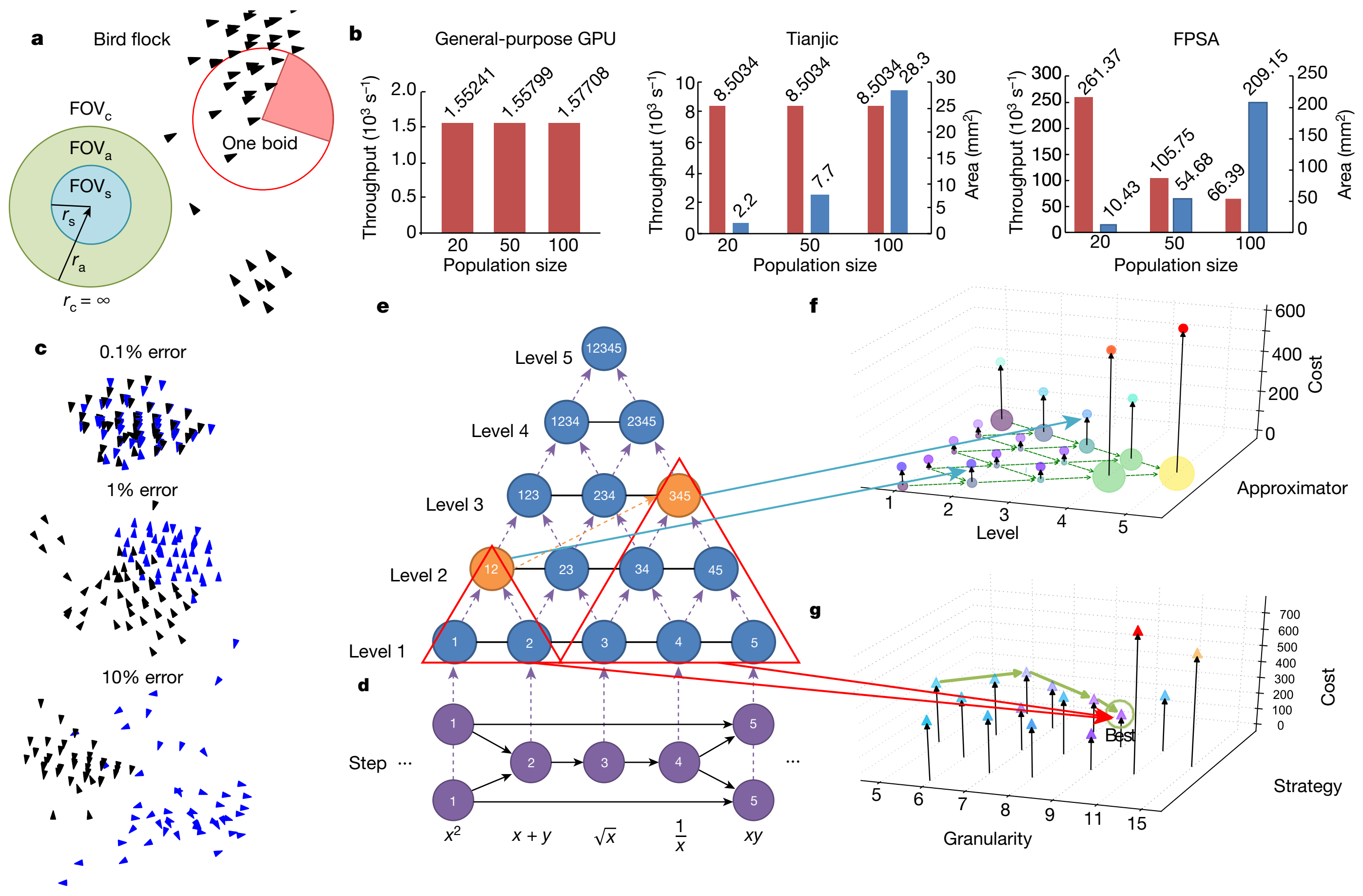


Fig. 4 | Experimental results. a, Boids model. The formal definition is provided in Methods section 'Boids model for bird flock simulation'. Each bird (or boid; black triangles) follows three rules to determine their behaviour: a separation rule, an alignment rule and a coherent rule. Each rule has an associated field of perception (defined by the relevant perception distance; eg, red circle) and field of view (FOV; eg, red shaded region). For simplicity, we adopt the configuration on the left (green and blue). The perception distances for the alignment rule (r_a) and the separation rule (r_s) are limited (with $r_a > r_s$); the corresponding fields of view are the entire green (FOV$_a$) and blue (FOV$_s$) circles. The perception distance for the cohesion rule (r_c) is unlimited; the corresponding field of view (FOV$_c$) is the whole simulation space. **b**, Performance (throughput; red, left axis) and hardware consumption (area; blue, right axis) of the boids model. **c**, Boids model at different error rates. All images are captured at frame 500, in which every triangle represents a bird: blue is the result of approximation; black is the result of exact calculation, for comparison. **d**, The partial calculation steps for QR decomposition. Each node is a basic step, and the numbers represent the calculation function (indicated below). **e**, The fusion space network enumerates all the possible approximators in **d**. Each node identifies a unique approximator and the numbers represent the successive steps it approximates. The red triangles indicate the coverage of a given approximator (orange nodes). The two red triangles shown (for two approximators, '12' and '345') form an approximation strategy of the entire QR decomposition. **f**, Cost of all approximators. Each corresponds to a node in the fusion space network in **e**; the values of each point are provided in Extended Data Table 1. The colour of each circle identifies the fusion level; the size indicates the cost. Moreover, the redder the point, the higher the cost. **g**, Cost of all approximation strategies and their approximate granularity (values of each point are provided in Extended Data Table 2). The strategy of approximators '12' and '345' is optimal (red lines; 'Best'); the green lines are the search path from the heuristic algorithm.

limited. This experiment demonstrates that the application scope of our proposal extends to non-neural networks, by taking advantage of relevant approximations.

The third application is QR decomposition, a common mathematical algorithm (Supplementary Information section 9.3). It requires various nonlinear calculations, which makes it a challenge for some brain-inspired platforms. We use the universal approximator to realize all the calculations shown in Fig. 4d (others are linear, which can be calculated exactly). An approximator can cover one or more successive steps (different approximate granularity), which leads to multiple approximation strategies. We therefore use a fusion space network (Fig. 4e) to visually represent the strategy space, and a heuristic searching method as an optimization strategy (Fig. 4f, g). This experiment further demonstrates that our proposal supports arbitrary applications. It also shows that the tradeoff between approximation granularity and performance introduced by neuromorphic completeness is beneficial to reducing hardware cost, provided that some error limit is met.

Conclusion

We have proposed definition of completeness for brain-inspired systems, which broadens the scope of the complete hardware and introduces a new dimension of system design, the approximation granularity. Combined with the proposed system hierarchy, which includes the software- and hardware-abstraction models, the extended definition of completeness enables the equivalent conversion between Turing-complete software and neuromorphic-complete hardware; that is, it decouples the software and hardware. Our design philosophy makes clearer the interfaces and divisions between the different aspect of the system, which may help multi-disciplinary studies. We hope that further effort will be devoted to this fundamental hierarchy to improve the productivity of brain-inspired computing development[44], including the development of artificial general intelligence (Supplementary Information section 10).

Online content

Any methods, additional references, Nature Research reporting summaries, source data, extended data, supplementary information, acknowledgements, peer review information; details of author contributions and competing interests; and statements of data and code availability are available at https://doi.org/10.1038/s41586-020-2782-y.

1. Waldrop, M. The chips are down for Moore's law. *Nature* **530**, 144–147 (2016).
2. Kendall, J. D. & Kumar, S. The building blocks of a brain-inspired computer. *Appl. Phys. Rev*. **7**, 011305 (2020).
3. Zhang, B., Shi, L. P. & Song, S. Creating more intelligent robots through brain-inspired computing. *Science* **354** (Spons. Suppl.), 4–9 (2016).
4. Roy, K., Jaiswal, A. & Panda, P. Towards spike-based machine intelligence with neuromorphic computing. *Nature* **575**, 607–617 (2019).
5. Chen, Y. et al. DianNao family: energy-efficient hardware accelerators for machine learning. *Commun. ACM* **59**, 105–112 (2016).
6. Jouppi, N. P. et al. In-datacenter performance analysis of a tensor processing unit. In *Proc. 44th Annu. Int. Symp. Computer Architecture* 1–12 (IEEE, 2017).
7. Schemmel, J. et al. A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *Proc. 2010 IEEE Int. Symp. Circuits and Systems* 1947–1950 (IEEE, 2010).
8. Merolla, P. A. et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science* **345**, 668–673 (2014).
9. Furber, S. B. et al. The spinnaker project. *Proc. IEEE* **102**, 652–665 (2014).
10. Davies, M. et al. Loihi: a neuromorphic manycore processor with on-chip learning. *IEEE Micro* **38**, 82–99 (2018).
11. Benjamin, B. V. et al. Neurogrid: a mixed-analog-digital multichip system for large-scale neural simulations. *Proc. IEEE* **102**, 699–716 (2014).
12. Friedmann, S. et al. Demonstrating hybrid learning in a flexible neuromorphic hardware system. *IEEE Trans. Biomed. Circuits Syst*. **11**, 128–142 (2017).
13. Neckar, A. et al. Braindrop: a mixed-signal neuromorphic architecture with a dynamical systems-based programming model. *Proc. IEEE* **107**, 144–164 (2019).
14. Pei, J. et al. Towards artificial general intelligence with hybrid Tianjic chip architecture. *Nature* **572**, 106–111 (2019).
15. Goertzel, B. Artificial general intelligence: concept, state of the art, and future prospects. *J. Artif. Gen. Intell*. **5**, 1–48 (2014).
16. Turing, A. M. On computable numbers, with an application to the Entscheidungsproblem. *Proc. Lond. Math. Soc*. **2**, 230–265 (1937).
17. Eckert, J. P. Jr & Mauchly, J. W. *Automatic High-speed Computing: A Progress Report on the EDVAC*. Report No. W-670-ORD-4926 (Univ. Pennsylvania, 1945).
18. von Neumann, J. First draft of a report on the EDVAC. *IEEE Ann. Hist. Comput*. **15**, 27–75 (1993).
19. Aimone, J. B., Severa, W. & Vineyard, C. M. Composing neural algorithms with Fugu. In *Proc. Int. Conf. Neuromorphic Systems* 1–8 (ACM, 2019).
20. Lagorce, X. & Benosman, R. Stick: spike time interval computational kernel, a framework for general purpose computation using neurons, precise timing, delays, and synchrony. *Neural Comput*. **27**, 2261–2317 (2015).
21. Aimone, J. B. et al. Non-neural network applications for spiking neuromorphic hardware. *Proc. 3rd Int. Worksh. Post Moores Era Supercomputing* 24–26 (IEEE–TCHPC, 2018).
22. Sawada, J. et al. Truenorth ecosystem for brain-inspired computing: scalable systems, software, and applications. In *Proc. Int. Conf. High Performance Computing, Networking, Storage and Analysis* 130–141 (IEEE, 2016).
23. Rowley, A. G. D. et al. SpiNNTools: the execution engine for the SpiNNaker platform. *Front. Neurosci*. **13**, 231 (2019).
24. Rhodes, O. et al. sPyNNaker: a software package for running PyNN simulations on SpiNNaker. *Front. Neurosci*. **12**, 816 (2018).
25. Lin, C. K. et al. Programming spiking neural networks on Intel's Loihi. *Computer* **51**, 52–61 (2018).
26. Davison, A. P. et al. PyNN: a common interface for neuronal network simulators. *Front. Neuroinform*. **2**, 11 (2009).
27. Bekolay, T. et al. Nengo: a Python tool for building large-scale functional brain models. *Front. Neuroinform*. **7**, 48 (2014).
28. Hashmi, A., Nere, A., Thomas, J. J. and Lipasti, M. A case for neuromorphic ISAs. In *ACM SIGARCH Computer Architecture News* Vol. 39, 145–158 (ACM, 2011).
29. Schuman, C. D. et al. A survey of neuromorphic computing and neural networks in hardware. Preprint at https://arxiv.org/abs/1705.06963 (2017).
30. LeCun, Y., Bengio, Y. & Hinton, G. Deep learning. *Nature* **521**, 436–444 (2015).
31. Poggio, T. & Girosi, F. Networks for approximation and learning. *Proc. IEEE* **78**, 1481–1497 (1990).
32. Esmaeilzadeh, H., Sampson, A., Ceze, L. & Burger, D. Neural acceleration for general-purpose approximate programs. *IEEE Micro* **33**, 16–27 (2013).
33. Mead, C. & Ismail, M. *Analog VLSI Implementation of Neural Systems* Ch. 5–6 (Springer, 1989).
34. Strukov, D. B., Snider, G. S., Stewart, D. R. & Williams, R. S. The missing memristor found. *Nature* **453**, 80–83 (2008).
35. Prezioso, M. et al. Training and operation of an integrated neuromorphic network based on metal-oxide memristors. *Nature* **521**, 61–64 (2015).
36. Ji, Y. et al. FPSA: a full system stack solution for reconfigurable ReRAM-based NN accelerator architecture. In *Proc. 24th Int. Conf. Architectural Support for Programming Languages and Operating Systems* 733–747 (ACM, 2019).
37. Tuma, T., Pantazi, A., Le Gallo, M., Sebastian, A. & Eleftheriou, E. Stochastic phase-change neurons. *Nat. Nanotechnol*. **11**, 693–699 (2016).
38. Negrov, D. et al. An approximate backpropagation learning rule for memristor based neural networks using synaptic plasticity. *Neurocomputing* **237**, 193–199 (2016).
39. Maass, W. Networks of spiking neurons: the third generation of neural network models. *Neural Netw*. **10**, 1659–1671 (1997).
40. Leshno, M. et al. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Netw*. **6**, 861–867 (1993).
41. Dennis, J. B., Fosseen, J. B. & Linderman, J. P. Data flow schemas. In *Int. Symp. Theoretical Programming* 187–216 (Springer, 1974).
42. Jagannathan, R. *Coarse-grain dataflow programming of* conventional parallel computers. In *Advanced Topics in Dataflow Computing and Multithreading* 113–129 (IEEE, 1995).
43. Zhang, W. & Yang, Y. A survey of mathematical modeling based on flocking system. *Vibroengineering PROCEDIA* **13**, 243–248 (2017).
44. Hennessy, J. & Patterson, D. A new golden age for computer architecture. *Commun. ACM* **62**, 48–60 (2019).

Methods

We carry out three experiments to show the decoupling feature and the new optimization space introduced by neuromorphic completeness. The first two experiments, bicycle driving and tracking and the boids model for bird flock simulation, are deployed on three target hardware by the toolchain: general-purpose GPU, Tianjic chip[14] and FPSA[36]. The last one, QR decomposition by Givens rotations is a theoretical analysis experiment.

Hardware platforms

General-purpose GPU. It is Turing-complete and provides rich application development interfaces (such as CUDA and cuBLAS). The GPU server we used has an Intel Xeon E5-2680 v4 CPU, NVIDIA Tesla P100 with 3,584 CUDA cores and 512 GB memory.

Tianjic. Tianjic[14] is a many-core neuromorphic chip that supports the massive parallel execution of ANNs, SNNs and ANN–SNN hybrids. Its scheduling components support the general control-flow logic of the ANN/SNN. Moreover, Tianjic adopts the near-memory computing mode, and the memory in a Tianjic core can be shared by many primitives in the same core or used as a buffer for intermediate data (Supplementary Fig. 5).

FPSA. The architecture of FPSA[36] includes massive compact and efficient memristor-based processing elements (Supplementary Fig. 6), which support only the ReLU and in situ weighted-sum operations. Its communication subsystem is an FPGA-like reconfigurable routeing architecture with massive wiring resources. Moreover, it provides spiking memory blocks as on-chip buffers for caching intermediate data and configurable logic blocks to support arbitrary control logic.

Toolchain

The toolchain includes compilation and mapping. For compilation, one key technique is template matching (Supplementary Fig. 7). It is an equivalent conversion that uses one or more execution primitives to match specific operator graph(s) in the POG.

The other technique is to construct the universal approximator for any given function (Supplementary Fig. 7). It is based on the aforementioned constructive proof and requires the points to be sorted to satisfy the induction condition and determine the hypersurface for each point. Directly picking the points according to the definition of the induction condition is time-consuming. By contrast, we pick them in a reverse order, from $X_{(m)}$ to $X_{(1)}$. We first construct a convex hull with all m points, and then randomly pick one vertex of the convex hull as $X_{(m)}$ and remove it from the points. The rest of the points form a new convex hull. The facets facing $X_{(m)}$ can be used as a hypersurface to separate $X_{(m)}$ from other points. We pick the one with the largest distance from $X_{(m)}$. Then, we pick a vertex from the new convex hull as $X_{(m-1)}$, and repeat the process until only $n+1$ points remain. The last points satisfy the induction condition in any order. We randomly pick them, and get the best separation hypersurface for each of them. The best one is the one with the largest distance. Thus, if we move the origin to the picked points, the normal vector of the hypersurface should fall in the linear subspace spanned by the rest of the points and the hypersurface should pass through all the rest of the points. Suppose the chosen point is $\mathbf{X}_0$, the rest of the points are $X = (\mathbf{X}_1, \ldots, \mathbf{X}_k)$ and the normal vector is $\mathbf{N}$. Then, $\mathbf{N} = \boldsymbol{\alpha}(X - \mathrm{expand}(\mathbf{X}_0))$, and $\mathbf{N}(X - \mathrm{expand}(\mathbf{X}_0)) + \mathbf{b} = \mathbf{0}$. Here, $\boldsymbol{\alpha}$ is a coefficient vector, $\mathrm{expand}(\mathbf{X}_0) = (\mathbf{X}_0, \mathbf{X}_0, \ldots, \mathbf{X}_0)$ which has k elements and $\mathbf{b} = (b, b, \ldots, b)$; because we care only about the normal vector $\mathbf{N}$, we can set b to any non-zero value. Thus, we solve $(X - \mathrm{expand}(\mathbf{X}_0))(X - \mathrm{expand}(\mathbf{X}_0))^{\mathrm{T}}\boldsymbol{\alpha} + \mathbf{b} = \mathbf{0}$ to get $\boldsymbol{\alpha}$ and then $\mathbf{N}$.

With the sorted points and the corresponding hypersurface, we use the aforementioned constructive proof to construct universal approximators. The cost depends on the number of points. To reduce the cost, we decrease the number of points and fine-tune the universal approximator using backpropagation with Adam optimization. Usually, we set a condition to stop the fine-tune iteration, such as the error being smaller than a certain threshold.

The mapping is hardware dependent. For the GPU, the primitives in the EPG are the same as those in the POG, and the control flow is expressed as the control flow of CUDA.

For the FPSA, the primitives are supported by the processing element directly. The control flow is synthesized to configurable logic blocks and the buffers required are synthesized to spiking memory blocks. The processing elements, configurable logic blocks and spiking memory blocks form a netlist to achieve the functionality of the EPG. Then, the placement and routeing tools of the FPSA generate the chip configuration from the netlist. Details are provided in Supplementary Information sections 8.2.5 and 8.2.6.

For Tianjic, the EPG should be divided onto many cores and the task of each core should be satisfied with the resource restriction (that is, storage restriction and computation restriction). The corresponding control, memory and routeing information will also be configured[45].

Bicycle driving and tracking

The bicycle driving and tracking experiment is a hybrid ANN–SNN system[14], constructed from five different neural networks: a CNN, an SNN, a CANN, an MLP and a NSM.

CNN. It is for image processing and object detection, which has three convolutional layers, two max-pooling layers and two fully connected layers. It takes 70 × 70 greyscale images as the input and outputs the coordinates of the human and obstacles.

SNN. It processes voice signal from the microphone and outputs the corresponding control commands. It is a 510-256-7 fully connected network. Each neuron is an LIF model. A detailed definition is provided in Supplementary Information section 9.2.

CANN. It is designed for object tracking. It is a one-layer fully connected recurrent neural network, which contains 20 × 24 neurons. It receives the images clipped by the initial human coordinates from the CNN and outputs the coordinates of the tracked target.

MLP. It takes in the motion information from the sensors and some related state signals from NSM, and outputs information about the balance state of the bike. It is a three-layered (30-256-32-1) network.

NSM. It controls all the above networks. It performs as a finite state machine with six states and nine transition conditions. The inputs are the signals from the CNN and the SNN, and the signals of internal states. The state transition and decision-making are achieved by a series of linear operations and LIF neurons, which are the same as the SNN.

The POGs of these cases and their connection relationship are shown in Extended Data Fig. 2.

We deploy these networks on three target hardware: general-purpose GPU, Tianjic chip[14] and FPSA[36]. For Tianjic, vector–matrix accumulation ($\mathbf{y} = W\mathbf{s}$, where W is the input matrix, $\mathbf{s}$ is the input vector and $\mathbf{y}$ is the resultant vector) replaces the weighted-sum operations in the convolution and fully connected layers, and the vector–matrix multiplications in the CANN and NSM. The element-wise operations are replaced with vector–vector accumulation and vector–vector multiplication. The pooling primitives are used as the pooling layer. The LIF neuron is replaced by the vector–matrix accumulation primitive and the LIF primitive.

The compiler also approximates other operators, for example, using the look-up table to support the division operation in the CANN. Because the look-up table supports the mapping of only 8-bit input–output, we scale the operands beforehand. The Tianjic computation primitives used in this case are listed in Extended Data Table 3. During mapping, all the primitives of the EPG are distributed to each core, as load-balanced as possible, and meet the hardware constraints.

For FPSA, because the complexity of the universal approximator depends on the number of possible valid inputs, approximating the entire model is not practical. Therefore, we partition each network into parts, divided by continuous weighted-sum operations. Each weighted-sum operation is supported directly. Most of the remaining operations (between the weighted-sum operations) are element-wise operations (such as multiplication, addition and computation in an LIF model) and the inputs are either 1-bit spikes or 8-bit numbers. Accordingly, we generate approximators for these unary and binary operations with zero error; that is, the transformation is exactly equivalent. Besides, some operations have a large number of inputs, such as normalization, which would lead to an impractical cost if we approximate them directly. We therefore split them into many addition and division operations according to the mathematic definition, and then approximate these operations instead. With this partition strategy, the cost for all models is acceptable. Moreover, adjacent operations can be fused and approximated as a whole to further reduce the cost.

The compilation results are evaluated on the FPSA simulator. It is a cycle-accurate simulator based on the circuit-level parameters extracted from either a memristor circuit-level simulator (Nvsim[46]) or register transfer language synthesis. These parameters are listed in Extended Data Table 4. The communication subsystem in the FPSA is a memristor-based FPGA-like routeing architecture[47]. We developed a mapping tool to turn the EPG into a netlist composed of memristor-based processing elements, buffers and configurable logic blocks. We use the corresponding placement and routeing tool (mrVPR[47]), which is extended from a widely used FPGA placement and routeing tool[48], to map the generated netlist onto the FPSA and get the critical communication latency of the routeing.

The resources used to approximate different types of operation are presented in Extended Data Fig. 3. Our toolchain enables the flexible approximation of these models with an acceptable cost.

Boids model for bird flock simulation

The boids model (Supplementary Figs. 8, 9) is used to study the behaviour of biological flock, such as bird flock[49]. In this model, each bird is called a boid, and follows three rules to achieve natural reality: (1) the separation rule, whereby a bird tries to keep a certain distance from nearby birds (that is, not too close); (2) the cohesion rule, whereby a bird tries to fly towards the centre of all the other birds; and (3) the alignment rule, whereby a bird tries to maintain the same speed as the surrounding birds.

A formal definition of Boids model is as follows[50]. There are N boids in a Euclidean vector space (N is the population size) $V = \mathbb{R}^d$ (in general, $d = 2, 3$). Each boid has an internal state $q \in Q$:

$$Q = \{q | q = (\mathbf{p}, \mathbf{v}, r, \mathrm{FOV}, m, \upsilon_\mathrm{m}, f_\mathrm{m})\}$$

where $\mathbf{p}, \mathbf{v} \in V$ are position and velocity of the boid, respectively; $r = (r_\mathrm{s}, r_\mathrm{a}, r_\mathrm{c})$ represents the separation, alignment and cohesion perception distances; $\mathrm{FOV} = (\mathrm{FOV}_\mathrm{s}, \mathrm{FOV}_\mathrm{a}, \mathrm{FOV}_\mathrm{c})$ represents the fields of view of the three rules; m is the mass of the boid, which is often 1 and so ignored by the model; υ_m is the maximal speed; and f_m is the maximal available change in the speed. Here, we refer to an open-source implementation named XBoids[51]. We adopt the two-dimensional simulation. In our implementation, we choose $r = (25, 50, \infty)$, and the field of view is the whole circle. Detailed configurations are provided in Supplementary Information section 9.2.

We implement the boids model with the POG and deploy it on the three hardware platforms. All the platforms evaluate the boids model for population sizes of $N = 20$, $N = 50$ and $N = 100$ (Supplementary Tables 1–3).

The boids model contains several linear tensor operations, which can be converted to the EPG through template-matching; the nonlinear operations (for example, square, square-root and reciprocal) are supported through approximation. For Tianjic, we use the look-up table to approximate (Supplementary Fig. 10). For the FPSA, we use universal approximators instead. For the GPU, we can flexibly support these operations in its EPG.

We measure the performance and resource utilization for the boids model (throughput for all hardware; area for Tianjic and FPSA). We also study the effect of the degree of approximation. We first construct three square-root approximators, with relative errors of 0.1%, 1% and 10%. Then, all the square-root operations in the EPG are replaced by these approximators. The running results of the 500th frame are shown in Fig. 4c.

QR decomposition by Givens rotations

QR decomposition is a mathematical procedure that decomposes a matrix A into an orthogonal matrix Q and an upper-triangular matrix R. Here, we adopt the Givens rotation method[52], the pseudocode of which is shown in Extended Data Fig. 4a. We focus on the steps (1–5) in Extended Data Fig. 4a, which are also the steps shown in Fig. 4d.

In this experiment, we test the design-space exploration of the approximation granularity: from the most finely grained to coarser granularities. In the most finely grained case, each basic operation is approximated by one universal approximator. In other cases, several successive operations are approximated by an approximator as a compositive operator. We further propose a dedicated representation—a fusion space network—to illustrate different approximators (Fig. 4e, Supplementary Fig. 11). In the fusion space network, the red triangle (cover triangle) represents the cover scope of one approximator. A valid approximation strategy should ensure that the triangles can cover all basic operations and that there is no overlap. The approximators are then fine-tuned by backpropagation.

An approximator can be viewed as a three-layer MLP with an ReLU activation function. Assuming the number of nodes in the input, hidden and output layers are m, n and 1, respectively, the cost of this approximator is measured as $C(A) = (mn + n)t$. Here t is the number of times this approximator is used during the computation, which also means that t copies of it should be mapped on the hardware if there is no time-division multiplexing. The approximation granularity $G(\cdot)$ is defined as the number of nodes covered by the triangule; for example, $G(A_{123}) = 6$ and $G(A)_{45} = 3$. Accordingly, the granularity (cost) of a given strategy is defined as the sum of the granularity (cost) of its approximators.

Intuitively, the cost of an approximation is positively correlated with the approximation precision. To focus on the relation between granularity and cost, we set a fixed upper limit of the error for each approximator, $E_\mathrm{max} = 3\%$. Then, we use a binary search to find an approximator with minimal cost. The error metric we adopt is the mean absolute percentage error:

$$\mathrm{MAPE} = \frac{100\%}{n} \sum_{i=1}^{n} \left| \frac{y_\mathrm{precise} - y}{y_\mathrm{precise}} \right|$$

where n is the number of points sampled for error calculation. All points should be within the input domain. The domain we set for step 1 is [−8, 8]; domains for other steps are deduced from this. To avoid having a zero in the denominator, [−0.01, 0.01] is excluded.

We present the cost distributions on approximators and strategies in Fig. 4f, g. Specific information about approximators and strategies is listed in Extended Data Tables 1 and 2, respectively.

We also propose a simple heuristic search algorithm to find an optimized approximation strategy on fusion space network, which consists of three steps.

Step 1. The algorithm starts with the most finely grained strategy. The selected approximators in the strategy form trees. Two adjacent and selected approximators are the tree's leaves, and the lowest approximator in the fusion space network that covers them is the tree's root. For example, A_1 and A_2, with A_{12} as the root, forms tree $\{A_{12} : A_1, A_2\}$.

Step 2. We evaluate the cost of the root approximator in each tree and then determine the saved cost of each tree (the cost saved if we replace the leaf approximators with the corresponding root approximator). If the saved cost is positive, then we can reduce the total cost.

Step 3. We select the root approximator with the highest saved cost to replace the corresponding leaf approximators.

These steps are repeated until there is no positive saved cost. Extended Data Fig. 4b shows the heuristic search algorithm for QR decomposition. The search path is also marked by a green arrow in Fig. 4g.

We control the resulting error of the QR decomposition in a limited and acceptable range. We repeat the experiment 10 times. During the experiment, the mean square error of the Q matrix is less than 0.1, that of the R matrix is less than 0.5 and the input is a random 4×4 matrix with element values ranging from −8 to 8.

Data availability

The example applications that we used are publicly available, as described in the text and the relevant references. The experimental setups for demonstration and measurements are detailed in the text and the relevant references. Other data that support the findings of this study are available from the corresponding authors on reasonable request.

Code availability

The codes used for the software toolchain and the demonstration neural networks are available from the corresponding authors on reasonable request.

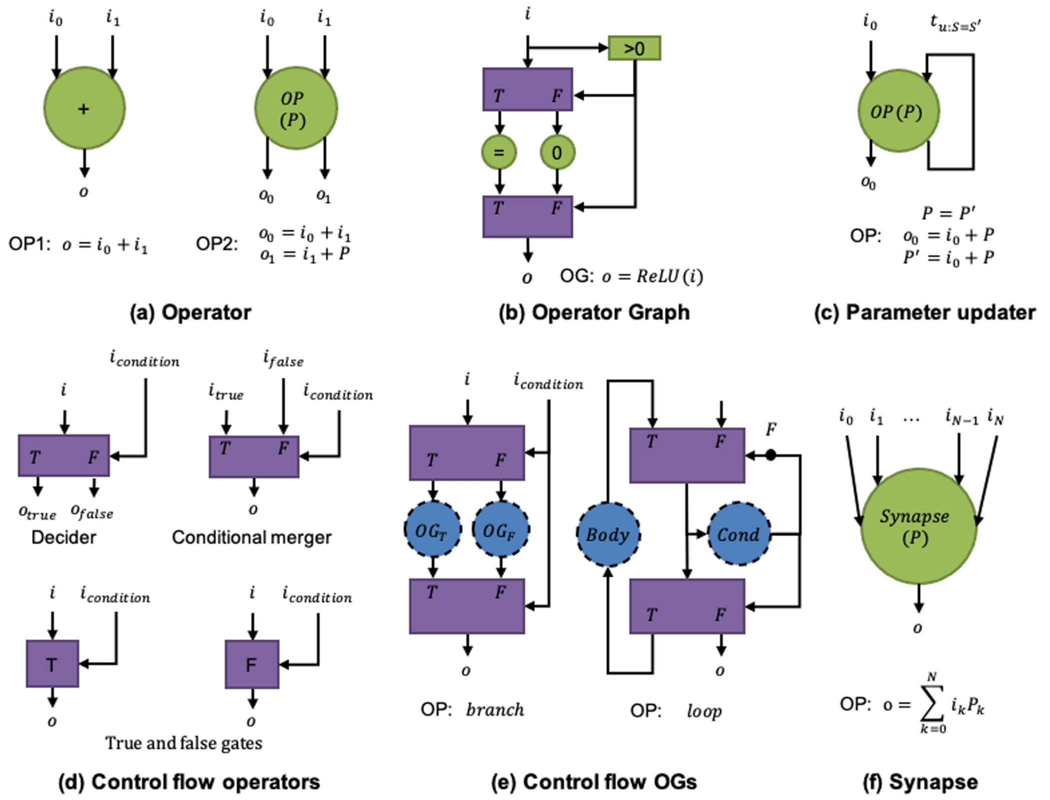
45. Deng, L. et al. Tianjic: a unified and scalable chip bridging spike-based and continuous neural computation. *IEEE J. Solid-State Circuits* **55**, 2228–2246 (2020).
46. Dong, X. et al. Nvsim: a circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE Trans. Comput. Aided Des. Integrated Circ. Syst.* **31**, 994–1007 (2012).
47. Cong, J. & Xiao, B. mrFPGA: a novel FPGA architecture with memristor-based reconfiguration. In *2011 IEEE/ACM Int. Symp. Nanoscale Architectures* 1–8 (IEEE, 2011).
48. Luu, J. et al. VTR 7.0: next generation architecture and CAD system for FPGAs. *ACM Trans. Reconfig. Technol. Syst.* **7**, 6 (2014).
49. Reynolds, C. W. Flocks, herds and schools: a distributed behavioral model. In *Proc. 14th Annu. Conf. Computer Graphics and Interactive Techniques* 25–34 (ACM, 1987).
50. Bajec, I. L., Zimic, N. & Mraz, M. The computational beauty of flocking: boids revisited. *Math. Comput. Model. Dyn. Syst.* **13**, 331–347 (2007).
51. Parker, C. XBoids, GPL version 2 licensed. http://www.vergenet.net/~conrad/boids/download (2002).
52. Aslan, S., Niu, S. & Saniie, J. FPGA implementation of fast QR decomposition based on givens rotation. *Proc. 55th Int. Midwest Symp. Circuits and Systems* 470–473 (IEEE, 2012).

Acknowledgements This work was partly supported by Beijing Academy of Artificial Intelligence (No. BAAI2019ZD0403), NSFC (No. 61836004), Brain-Science Special Program of Beijing under grant Z181100001518006, Beijing National Research Center for Information Science and Technology, Beijing Innovation Center for Future Chips, Tsinghua University and Tsinghua University-China Electronics Technology HIK Group Co. Joint Research Center for Brain-inspired Computing, JCBIC.

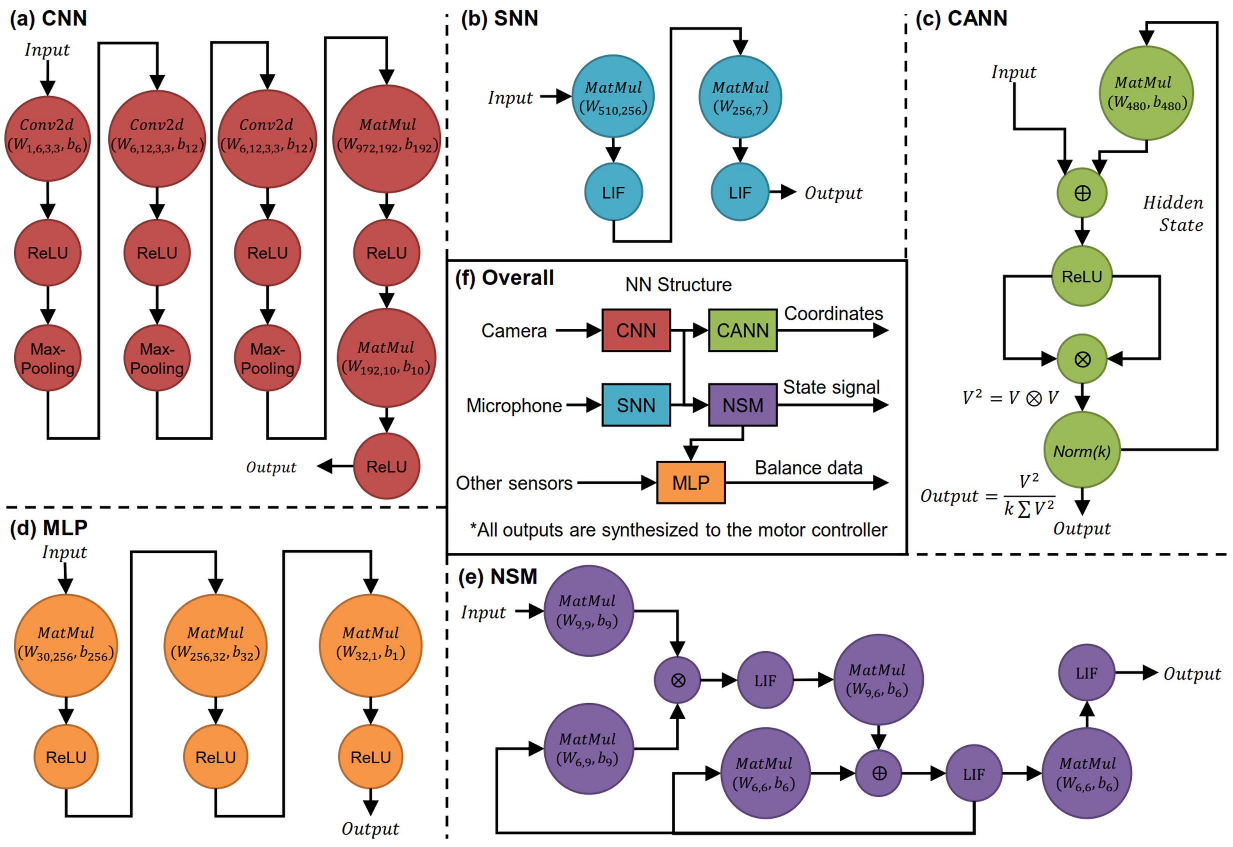
Author contributions Y.Z., P.Q., Y.J. and W. Zhang proposed the idea for the brain-inspired hierarchy. Y.Z. was in charge of the whole design. P.Q. proposed the ideas for the POG and the proof of its Turing completeness. Y.J. proposed the ideas for neuromorphic completeness, the EPG, the basic execution primitives and the constructive proof of its neuromorphic completeness. W. Zhang proposed the ideas for the ANA and the mapping from the EPG to it. P.Q. performed the experiment on the GPU. Y.J. performed the experiment on the FPSA. W. Zhang and G.W. performed the experiments on Tianjic. Y.J. and W. Zhang performed the experiments on the Boid model and QR decomposition for the revision. G.G. gave advice on the theory of architecture and hierarchy. L.S. was in charge of Tianjic work and proposed the idea to bridge dual-driven brain-inspired computing with artificial general intelligence. G.G., S.S., G.L., W.C. W. Zheng, F.C., J.P., R.Z., M.Z. and L.S. contributed to the analysis and interpretation of results. All authors contributed to the discussion of the design principle of the brain-inspired hierarchy. Y.Z., L.S., P.Q. and R.Z. revised the manuscript, with input from all authors. Y.Z. and L.S. supervised the project.

Competing interests The authors declare no competing interests.

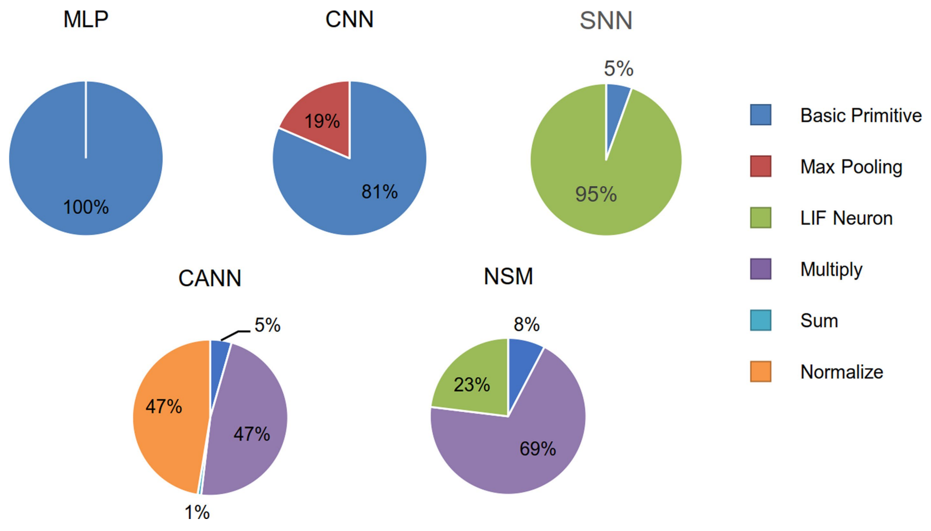
Additional information
Supplementary information is available for this paper at https://doi.org/10.1038/s41586-020-2782-y.
Correspondence and requests for materials should be addressed to Y.Z. or L.S.
Peer review information *Nature* thanks Oliver Rhodes and the other, anonymous, reviewer(s) for their contribution to the peer review of this work.



Extended Data Fig. 1 | POG. a, Two sample operators: the left is a single mathematical operation and the right is a simple algorithm that consists of several computation operations. **b**, A sample operator graph for rectified linear units (ReLU). **c**, Parameter updater. More details are provided in Supplementary Information section 3.3.1. **d**, Main control-flow operators: conditional decider, conditional merger, true gate and false gate. **e**, Control-flow operator graphs of the branch and the loop. **f**, Synapse operator: it is enabled when any of the inputs arrive. *i*, input; *o*, output; T/F, true/false branch; *P*, parameter of the operator; *P′*, the new value of *P*; $OG_{T/F}$, operator graph of the true/false branch; Body, operator graph of the loop body; Cond, operator graph of the loop condition.



Extended Data Fig. 2 | Bicycle driving and tracking task. a–e, POGs of the five neural network examples. Conv2d, two-dimensional convolution operator; MatMul, matrix multiplication operator; LIF, operator of the LIF model; Norm, normalization operator; *W*, *b*, weight and bias parameters for the corresponding operator. **f**, The overall relationships between these networks.



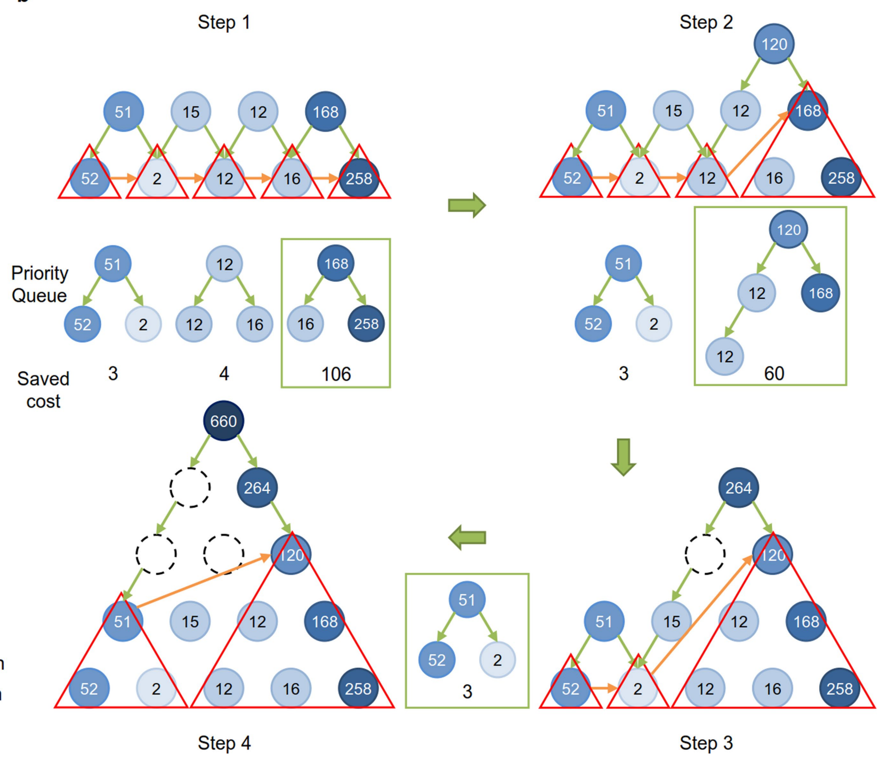
Extended Data Fig. 3 | Resource consumption of the FPSA for different operators.

a

```
Input: A square matrix A
Output: Q and R
n, n = A.shape
R = A
$Q^T$ = Identical Matrix(n, n)
for i = 0 to n - 1
 for j = 0 to n – 1
  $x_i$=R[i][i]
  $x_j$=R[j][i]
  $x_i^2, x_j^2 = x_i \times x_i, x_j \times x_j$                1
  $x_{ij}^2 = x_i^2 + x_j^2$                                   2
  $x_{ij}^2\_s = \sqrt{x_{ij}^2}$                              3
  $x_{ij}^2\_s\_1 = 1/x_{ij}^2\_sqrt$                         4
  $cos\_, sin\_ = x_i \times x_{ij}^2\_s\_1, x_j \times x_{ij}^2\_s\_1$   5

  Q_this = Identical Matrix(n, n)
  Q_this[i][i]= $cos\_$
  Q_this[i][j]= $sin\_$
  Q_this[j][i]= $-sin\_$
  Q_this[j][j]= $cos\_$
  R = Q_this × R            // Matrix Multiplication
  $Q^T$ = $Q^T$ × Q_this       // Matrix Multiplication
return $Q^T$.transpose, R
```

b



Extended Data Fig. 4 | The experiment using QR decomposition. a, Pseudocode for QR decomposition by Givens rotation. **b**, The heuristic search algorithm on QR decomposition. Each step of this figure includes approximators that have been evaluated and the current priority queue. The number on each node represents the cost of that approximator. The red triangle represents the current strategy. Each step, the first tree in the queue is selected (green rectangle) and the leaves are replaced by the root approximator. The dashed circles represent the approximators that do not need to be constructed and evaluated. The whole procedure explores only four (out of 16) strategies. In this case, the final strategy happens to be the best, but in general, this algorithm is not guaranteed to produce the optimal result.

Extended Data Table 1 | Universal approximators in the QR decomposition

Approximator	Function	Input	Domain	Granularity	Cost
A_1	$f(x) = x^2$	Unary	-8 ~ 8	1	52
A_2	$f(x,y) = x + y$	Binary	(0, 0) ~ (64, 64)	1	2
A_3	$f(x) = \sqrt{x}$	Unary	0 ~ 128	1	12
A_4	$f(x) = \frac{1}{x}$	Unary	0.01 ~ 12($8\sqrt{2}$)	1	16
A_5	$f(x,y) = xy$	Binary	(0, -8) ~ (100, 8)	1	258
A_{12}	$f(x,y) = x^2 + y^2$	Binary	(-8, -8) ~ (8, 8)	3	51
A_{23}	$f(x,y) = \sqrt{x+y}$	Binary	(0, 0) ~ (64, 64)	3	15
A_{34}	$f(x) = \frac{1}{\sqrt{x}}$	Unary	0.01 ~ 128	3	12
A_{45}	$f(x,y) = \frac{y}{x}$	Binary	(0.01, -8) ~ (12, 8)	3	168
A_{123}	$f(x,y) = \sqrt{x^2 + y^2}$	Binary	(-8, -8) ~ (8, 8)	6	21
A_{234}	$f(x,y) = \frac{1}{\sqrt{x+y}}$	Binary	(0.01, 0.01) ~ (64, 64)	6	15
A_{345}	$f(x,y) = \frac{y}{\sqrt{x}}$	Binary	(0.01, -8) ~ (128, 8)	6	120
A_{1234}	$f(x,y) = \frac{1}{\sqrt{x^2 + y^2}}$	Binary	(-8, -8)~(8, 8) / [-0.01,0.01]	10	573
A_{2345}	$f(x,y,z) = \frac{z}{\sqrt{x+y}}$	Ternary	(0.01, 0.01, -8) ~ (64, 64,8)	10	264
A_{12345}	$f(x,y) = \frac{y}{\sqrt{x^2 + y^2}}$	Binary	(-8, -8)~(8, 8) / [-0.01,0.01]	15	660

Extended Data Table 2 | Granularity and cost of different strategies

Strategy	Granularity	Cost
$S\{1,2,3,4,5\}$	5	340
$S\{12,3,4,5\}$	6	337
$S\{1,23,4,5\}$	6	341
$S\{1,2,34,5\}$	6	324
$S\{1,2,3,45\}$	6	234
$S\{12,34,5\}$	7	321
$S\{12,3,45\}$	7	231
$S\{1,23,45\}$	7	235
$S\{123,4,5\}$	8	295
$S\{1,234,5\}$	8	325
$S\{1,2,345\}$	8	174
$S\{123,45\}$	9	289
$S\{12,345\}$	9	171
$S\{1234,5\}$	11	831
$S\{1,2345\}$	11	316
$S\{12345\}$	15	660

Extended Data Table 3 | Part of the Tianjic primitives

Primitive	Full name	Definition	Used in
VMA	Vector Matrix Accumulation	$\boldsymbol{y} = \boldsymbol{W} \cdot \boldsymbol{s}$	SNN, NSM
VMM	Vector Matrix Multiplication	$\boldsymbol{y} = \boldsymbol{W} \cdot \boldsymbol{x}$	CNN, MLP, NSM, CANN
VVA	Vector Vector Accumulation	$\boldsymbol{y} = \sum_i \boldsymbol{x_i}, i = 1, \ldots, N$	NSM, CANN
VVM	Vector Vector Multiplication	$\boldsymbol{y} = \boldsymbol{x_1} \odot \boldsymbol{x_2}$	NSM, CANN
VS	Vector Scale	$\boldsymbol{y} = \alpha \boldsymbol{x}$	CANN
Pooling	-	$y_i = \frac{max}{avg(\{x_j \mid j \in pool_i\})}$	CANN
ReLU	Rectified Linear Unit	$y_i = x_i > 0 ? x_i : 0$	CNN, MLP
LUT	Look-up Table	$\boldsymbol{y} = \varphi(\boldsymbol{x})$	CANN
LIF	Leaky Integrate-and-Fire	LIF model: $v_1 = \alpha v + leacky + u$ $spike = v_1 > V_{thresh} ? 1 : 0$ $v = spike ? V_{reset} : v_1$	SNN, NSM

Extended Data Table 4 | FPSA function blocks at 400 MHz, 45 nm

Component	Specification	Power (mW)	Area (mm^2)	Latency (ns)
Configurable Logic Block	LUT: 6-in 1-out #LUT: 128	1.271	0.0060	0.229
Spiking Memory Block	Size: 16Kb	0.471	0.0054	0.578
	Processing element (PE) breakdown (latency for 256 cycles to support 8-bit I/O)			
Memristor Crossbar	Level/cell: 16	0.429	0.0085	0.0
Charging Unit	Number: 256	0.094	0.0006	18.1
Neuron Unit	Number: 512	8.130	0.0099	374.8
Spike Subtractor	Number: 256	3.661	0.0031	232.8
PE Total	Size: 256 × 256 W-8bit & I/O-8bit	11.909	0.0221	625.7
			9.501 $\mathrm{TOP/s/mm^2}$	