nature research

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

Supplementary information

A system hierarchy for brain-inspired computing

In the format provided by the authors and unedited

Supplementary Information:

A System Hierarchy for Brain-inspired Computing

1. Turing Machine and Von Neumann Architecture

In 1936, Alan Turing proposed an idealized computing model which consists of an infinitely long tape and a read/write head[1]. This simple, solid and easy-to-understand model is now known as the *Turing machine* and has become the major computation model in computer communities ever since.

In Turing machine, the infinite tape is viewed as contiguous cells each of which contains a symbol ("0","1", or "blank"). And the read/write head may move along the tape and change its target cell. It can either read out the symbol in the target cell or write a new symbol into the cell.

The "program" for a Turing machine is defined as a finite state machine (FSM) which guides the movement of the head and the modifications of the tape. The FSM is usually defined as a five-elements tuple $\psi = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is a finite, nonempty input alphabet (for Turing machine, the alphabet is from $\{0, 1, blank\}$), δ is a series of transition functions, q_0 is the starting state, and F is the set of accepting states. The transition function is usually defined as a mapping $f: q, c \to q, c, m$, where q is the current state of the FSM, c is the content of the cell under the read/write head, m is the corresponding movement of the head (move left, move right, or *no* move).

When executing an FSM, the Turing machine will start at the state q_0 which is specified by the FSM, and then move the read/write head and modify the content of the tape according to the transition function. When reaching an accepting state or no transition function is available, the computation is finished.

The Turing machine plays a fundamental role in the computing community, thus there are quite a few studies that try to enhance or generalize it. For example, the universal Turing machine (UTM) stores the FSM on the tape firstly and reads in the FSM before it carries out the dedicated computation. As the Turing machine is a sequential model, there are also several studies that try to extend it to support parallel computing, including extending to multiple head/tapes[2][3] and multiple dimensions[4][5][6].

Based on the Turing machine, an important classification for a system of data-manipulation rules is proposed, namely, Turing completeness. A system is said to be Turing complete if it can be used to simulate any Turing machine. A Turing complete system has at least the same computation capability as the Turing machine.

In 1945, John von Neumann proposed a stored-program computer which consists of a control unit, an arithmetic and logic unit, memory units, and inputs/outputs[7]. The computer stores the instructions and the data together in the memory and then fetches and executes the instructions serially. This model is now known as the *von Neumann architecture* and it builds the foundation of almost all the modern general-purpose computer architectures.

The Turing machine and the von Neumann architecture play a key role in the traditional computer hierarchy. Turing completeness ensures that any program described in a modern programming language is a process of executing the Turing machine (almost all the main stream programming language are Turing complete). The von Neumann abstract architecture model supports Turing machine through a Turing complete instruction set, which makes it feasible to compile any program in a high-level language into a strictly equivalent instruction sequence on a von Neumann processor.

That is, based on Turing completeness, the computer hierarchy provides compatibility and flexibility between software (i.e. programming languages and all applications on them) and hardware (i.e. various von Neumann processors): We continue to improve hardware efficiency while designing more sophisticated programming languages. The two are always compatible. This hierarchy also benefits the software-hardware co-design by defining/extending different instruction sets.

2. Neuromorphic Completeness

2.1. Universal Approximation Theorem

Approximation is an inherent property of neural networks. The approximation capability of neural networks has been studied since Kurt Hornik et al. [8] proposed the universal approximation theorem. It proved that there exists a three-layered feedforward network with bounded and non-constant semi-linear functions, e.g. sigmoidal function or threshold function, which can approximate any continuous function arbitrarily well. Blum et al.[9] and Kolmogorov et al. [10] also provide a constructive proof on the approximation property.

The way the universal approximator works is quite different from what a Turing machine computes a function. The universal approximation just memorizes the mapping of the function instead of computing it through an algorithm. More specifically, it achieves Turing-computable functions by memorizing the mapping directly, instead of by simulating the Turing machine to solve them. Thus, it is not Turing complete.

Accordingly, we introduce the concept of *neuromorphic computing capability* and *neuromorphic complete* to combine the approximation property with Turing completeness.

2.2. Neuromorphic Computing Capability

Given an error gap $\epsilon \geq 0$, for any function f_A that one computational system A can achieve, if there is a function f_B that can be achieved by system B and $\|f_A(X) - f_B(X)\| \leq \epsilon$ where X is any valid input, it is called that system B has the equal or stronger *neuromorphic computing capability* compared to system A.

We use this definition to measure the capabilities of brain-inspired computing systems.

2.3. Neuromorphic Complete

If a computational system has the equal *neuromorphic computing capability* compared to a Turing complete system, it is *neuromorphic complete*, i.e., a *neuromorphic complete* system can approximate any Turing-computable and terminable function with arbitrary precision.

The definition of *neuromorphic complete* relaxes the requirement of *Turing-complete* from precisely computing a function according to a composable algorithm to just approximating it.

The universal approximators and *Turing-complete* systems are both *neuromorphic complete*. They are two different types of typical *neuromorphic complete* systems.

- **Universal approximators (e.g. multilayer perceptron):** A universal approximator can approach any functions in arbitrary precision. Thus, they are *neuromorphic complete*. They mainly leverage the *approximation* property of neural networks.
- ***Turing-complete* systems (e.g. universal Turing machine):** A *Turing-complete* system can approach any functions precisely. Thus, they are also *neuromorphic complete*. They mainly leverage composable algorithms (i.e. a finite, executable sequence of steps) to precisely compute a function.

Universal approximators are not composable: A multilayer perceptron that approximates a complicated function is not composed of multiple MLPs that approximate simpler functions. In contrast, we may define some procedures on Turing machines and more complex procedures can be constructed based on them; the computation is accurate.

The composability and the approximation property are both important for brain-inspired computing. The proposed *execution primitive graph* can combine them together well.

3. Programming Operator Graph

We propose Programming Operator Graph (POG) as a unified description method and program execution model for brain-

inspired computing. In POG, a program is described as an operator graph and then executed according to the program execution model (PXM) of POG.

3.1. Operator Graph

3.1.1. Operator

An *Operator* is the unit of computation and scheduling, i.e. the basic description and execution unit of programs in the POG (Extended Data Fig. 1-a). An *operator* can be either one or more instructions or a complicated function, which has three arguments: input(s), output(s) and parameter(s). Inputs are the input data that comes from other *operators* or the environment, outputs are the results that this *operator* produces, and the parameters are the internal data that can only be accessed by the *operator* itself. An *operator* that has received all the input events could be triggered for execution, i.e. *enabled*. Execution of an enabled *operator* will consume all the ready input events, carry out the computation and generate corresponding output events that will be delivered to the destination *operator(s).*

3.1.2. Operator Graph

A program in POG is defined as a directed graph named *operator graph* (OG) (Extended Data Fig. 1-b). In OG, each node represents an *operator* and a directed edge means a precedence relationship between the *operators*. For example, for an OG $G(V, E)$ in which V is the set of vertices and E is the set of edges, a vertice $v \in V$ is a dedicated *operator* and an edge $e_{v_1,v_2} \in E$ means that there is a precedence relationship between v_1 and v_2. That is, one or more of the outputs of v_1 are the input(s) of v_2 or v_2 should be computed after v_1..

An event t on edge e_{v_1,v_2} means that the precedence relationship between v_1 and v_2 is satisfied. The event may have various types, ranging from only a pure signal that indicates a dependency relationship to a complicated structure that provides concrete data. There are three important types of events: the first is the data event which provides concrete data $value$ as the input i of the destination *operator*, expressed as $t_{i=value}$; the second is the signal event t_s which indicates that the precedence relationship is satisfied; and the last one is the *parameter event* that updates dedicated parameters (Supplementary Information 3.3.1).

3.2. The Program Execution Model

The program execution model (PXM) not only defines what a program is, it also consists of an activity model that defines how a program is executed, a memory model for data management and accesses, and a synchronization model that deals with the interaction between concurrent activities. A Turing machine is a typical PXM, composed of a finite state machine (FSM) based activity model, an infinite tape as the memory model and a synchronization model based on the sequential operations of the read/write head. The PXM of POG contains an operator-based activity model and a memory model that collocates processing and storage and eliminates the necessity of synchronization model.

3.2.1. Activity model

The active model of POG is a finite state operator graph (FSOG) which is expressed as a five-elements tuple $\psi = (G, \mathrm{T}, \delta, q_0, F)$. G is the *operator graph*, T is the set of available events, δ is the transition functions, q_0 is the start state, and F is the set of the accepting states.

Each state of the FSOG is defined as a three tuple: $S = (T, E, P)$, where T is the set of all the received events, E is the set of the edges that receives the events, and P is the set of the values of all the modifiable parameters. The transition function of the state S is defined as a mapping

$$f: ((t, e),\ p) \rightarrow ((t', e'), p'), \qquad t \in T, e \in E, p \in P$$

In one state, there will be a set of *enabled operators*. The subsequent state transition is carried out as the following: These *operators* will be scheduled and consume all the received data and carry out computations in parallel. Then the outputs are generated and transferred to the destination *operators*. Afterwards, the FSOG enters a new state. Namely, it needs to execute a series of transition functions to transfer from one state to a new state.

When executing, the POG will start at the state q_0 which is specified by the FSOG, and then it will process all the enabled *operators*, carry out their computation and deliver the output events to the target *operators*; if *parameter updater* (Supplementary Information 3.3.1) exists the corresponding parameters are also updated. Afterwards, a new state will be achieved. The POG will repeat this procedure until there is no enabled *operators* or an accepting state is achieved.

3.2.2. **Memory model**

The POG possesses a memory model that collocates storage and processing. This not only reflects a significant characteristic of brain-inspired computing for higher efficiency[11], but also is beneficial to achieve higher universality as it can be implemented on traditional chips that separate processing and storage, besides the processing-in-memory systems that naturally collocate storage and processing, e.g. memristor-based systems[12][13][14][15].

In detail, each *operator* possesses an infinite amount of private memory to store parameters or buffer the inputs and outputs. As the data transferring between different *operators* can only be achieved through output-to-input connections and the private memory can only be accessed by the owner, the interactions between *operators* are statically defined. Thus, no synchronization model is necessary.

3.3. **Assistant Operators**

Our POG design also provides several assistant operators to help the users express brain-inspired computing operations easily and efficiently. Such as the *parameter updater* that updates the internal parameters of the operators, the control-flow operators which can construct common control-flow schemas, and several neuromorphic related operators. With these assistant operators, one can build complicate programs such as SNNs, DNNs and so on.

3.3.1. **Parameter Updater**

The parameter(s) of an *operator* can be viewed as its private data that is rarely modified during the execution. The parameters can provide performance optimization hints to the compiler/mappers as it is more efficient to deploy them to the on-chip memory of the hardware that collocates storage with processing. But this paradigm may burden the model/algorithm designers if they must modify the parameters (e.g. during the learning/training procedure, synapse weights must be updated).

To this extent, we propose the notion of *parameter updater* (Extended Data Fig. 1-c). The *parameter updater* is a special input edge that indicates the modification of the corresponding parameters. The event on *parameter updater* is termed *parameter event*, which has the meaning of "update the parameter p to a new $value$ $\left(t_{u:p=v}\right)$" or "no change (t_u)". When an *operator* which has *parameter updater* is executed, it will first update its parameters according to the parameter events and then carry out the computation of the *operator*. For an *operator* without any *parameter updater*, its parameters are immutable.

The *parameter updater* enables fine-grained modification on coarse-grained parameters, like updating a dedicated column of elements of a matrix parameter. For example, this feature is important for learning algorithms which may invoke sparse operations to speed up the processing procedure. For example, spike-time-dependent-plasticity (STDP)[16] algorithm only updates the synaptic weights related to the fired neurons.

There is another way to achieve similar functionality of the *parameter updater*, i.e. a self-loop *operator* with an additional input edge. But this form treats the parameter as another common input, which leads to the inability to distinguish between the two and further lose the advantages and succinctness to provide better adaptability to hardware.

3.3.2. Control-Flow Operators

The control flow *operators* are the *decider*, the *conditional merger* and the *true/false gates* (Extended Data Fig. 1-d). The decider transfers the input token to only one of the output edges according to the condition while the conditional merger selects one output token from two input tokens according to the condition. The true/false gates decide whether to let the input token pass through according to the condition.

Based on these *operators*, we can define the branch and loop operations (Extended Data Fig. 1-e) in common programming languages easily: The branch and the loop body are sub-OGs that contain various operations (including computation *operators*,

control flow *operators* and their combination) . For the simple true/false branch, the programmer just needs to replace the decider/conditional merger with the true/false gates. It is necessary to note that the output token of the decider must be applied as the input edge of every entry node in the sub-OG. Moreover, the output events of all the output *operators* in the sub-OG should be combined to one output event through a merger *operator*.

3.3.3. Neuromorphic Related Operators

In POG, typical operators are enabled when all the input tokens are satisfied. But in most neuromorphic models, the neuron may have input connections from many other neurons and its internal states should be updated every time a spike arrives. This pattern requires that the computation of the neuron driven by any single input token instead of all the input tokens. To this extent, the POG provides a dedicated operator to deal with the input spikes, named synapse (Extended Data Fig. 1-f). The synapse operator carries out a simple weighted-sum operation, but this operator is enabled when any of the input token arrives. Namely, if only one input token arrives, it will carry out a simple multiply operation, and it will do the weighted-sum when multiple input token arrives.

With the synapse operator, it will be much easier to build neuromorphic models, for example, the LIF model (Fig. 2-a). In this case, we use two synapse operators to deal with the excitory and inhibitory input connections. The major computation operator LIF is also a weighted-sum operator. We also use a conditional merger to reset the membrane potential of a fired neuron. Moreover, a parameter updater is introduced to update the internal membrane potentials.

3.4. Composability

As the *operator* of the POG may contain complicated instructions or even a whole algorithm, a sub-OG can be viewed as an *operator*.

This feature is known as *composability*, which is important for programming convenience because different hardware implementations may provide hardware operators with multi-granularity: It is possible to describe a neural network application as an OG that only contains basic computational and control-flow *operators* and then compose some sub-graphs of this OG to new *operators* to suit multi-granularity hardware operations. Thus, the application described as a POG may suit different hardware without modification.

From the perspective of hardware and software co-design, this property is also very useful: we can customize the programming *operator* sets based on the specific operations provided by the underlying hardware, or vice versa. Owing to the composability, these modifications can only occur up to the POG level, which will not affect the upper programming paradigms.

3.5. Proof of Turing Completeness

The POG has control flow operators and operators that may update their own parameters; combining with the underlying active model, they can achieve the Turing completeness. Accordingly, here we propose a method to represent an FSM of the Turing machine with a corresponding POG which contains a limited number of operators, to prove that the POG is Turing complete.

Firstly, we build two special types of *operators*. The first is the tape *operator* (Supplementary Figure 1-a). It has a parameter list P, which contains infinite updatable parameters and the value of each parameter may be one from $\{0, 1, blank\}$ (the same alphabet of the Turing machine). When an input data token $t_{d:index=i}$ arrives (the index is used to located the corresponding parameter), the tape *operator* will provide the value of the corresponding parameter ($P[index]$) as its output This *operator* also has a *parameter updater*, when the parameter token $t_{u:P[i]=v}$ arrives, it will update the value of the corresponding parameter $P[i]$ to the new value v. The tape *operator* servers as the Turing tape: Each read operation can be converted into issuing a data token to the corresponding tape *operator* and the write operation is inputting a parameter token to the *parameter updater*.

The second is the map *operator* (Supplementary Figure 1-b). It works like the *map* class in C++, which can give an output data token according to the two input data tokens. The map *operator* can also be constructed as a simple POG that contains control flow *operators* and other *operators*. Then, we combine two map *operators* to build a transfer *operator* that servers as

the transfer functions which convert the input data token $t_{d:state=s}$ and $t_{d:head=i}$ to output data token $t_{d:state=s'}$ and $t_{d:head=i'}$ according to the other input data token $t_{d:input=c}$. With these *operators*, we can convert the transfer function

$$\delta = \{f | f: q, c \to q', c', m\}$$

in the Turing machine to the transfer function

$$\delta^{NC} = \{(f_{state}, f_{move}) | f_{state}: \left((t_{d:state=q}, e), p[head] = c\right) \to (t_{d:state=q'}, e'), \quad p[head] = c')$$

and

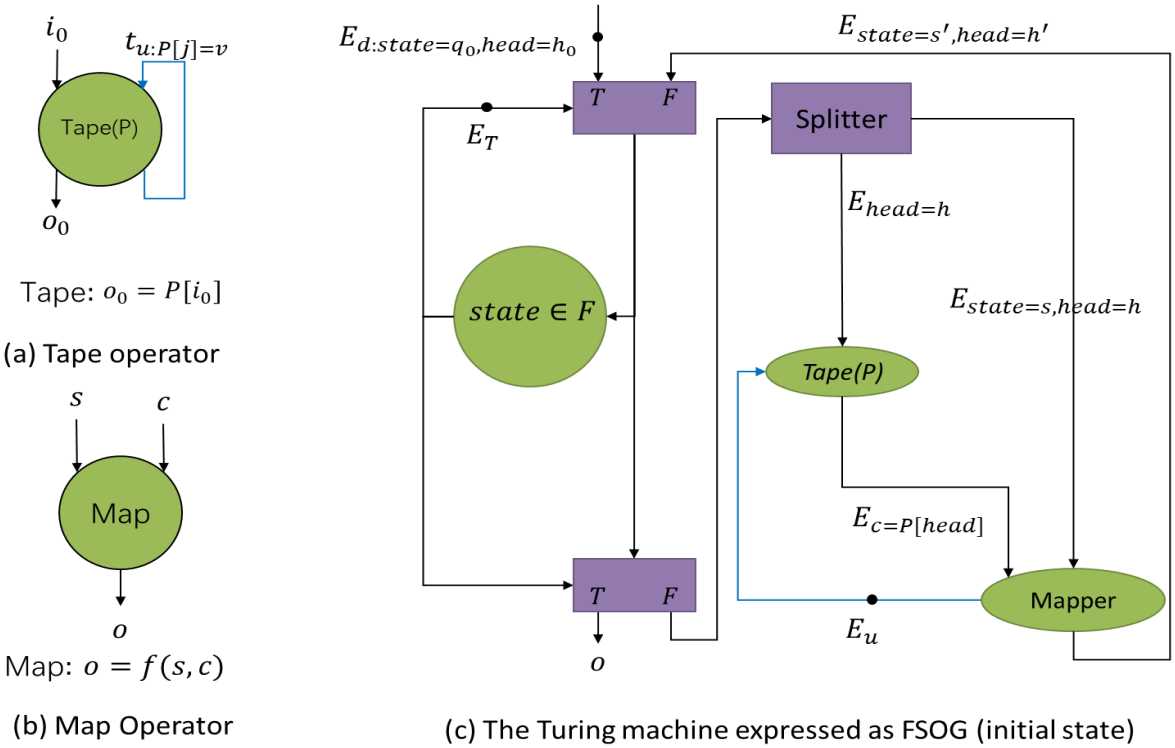
$$f_{move}: \left((t_{d:head=i}, e), \quad \right) \to \left((t_{d:head=i'}, e'), \quad \right), \quad i' \in \{i+1, i-1, i\}\}$$

in the POG. Furthermore, Q and Σ of the five-element tuple of Turing FSM are expressed as the parameters of the transfer *operator*.

Secondly, a loop graph is defined to loop through the states of the FSM, and the loop-body subgraph is defined as composed of four parts: Read the value of $P[head]$ from the tape operator, change the value of $P[head]$ if necessary, and update the value of the $state$ and $head$ accordingly (Supplementary Figure 1-c).

Thirdly, the loop condition is defined as whether the state is one of the accepting states in the FSM. Thus the F in the Turing machine is transferred into $F^{NC} = ((t_s, e_{out}),)$ in the NCFSM. And the q_0 will be converted to $q_0^{NC} = \left((t_{d:state=q_0}, \ e_{in}), \ P = init_{values}\right)$.

In this way, we convert $\psi = (Q, \Sigma, \delta, q_0, F)$ to $\psi = (G, \mathrm{T}, \ \delta^{NC}, \ q_0^{NC}, \ F^{NC})$. It means that we can mimic the Turing machine through POG, namely the POG is Turing complete.



Supplementary Figure 1. The Turing machine expressed as OG

3.6. **Relationship with Dataflow**

3.6.1. Dataflow Model

The *dataflow* schema represents the logical procedure of a program in a form that specifies the sequencing of the computations and the data transferred between them[17]. A program in the dataflow schema is described as a directed graph, in which every

node represents an instruction and every edge represents the data transfer.

Dataflow schema outlines the data dependence between different instructions directly, thus it is beneficial to exploit fine-grained parallelism and to achieve high throughput.

The coarse-grained dataflow model[18] follows the basic computation patterns of the original dataflow schema and further leases the requirement of the basic unit of computation. In coarse-grained dataflow, each unit of computation can be a set of instructions that should be executed atomically, that is, the node can be a set of instructions instead of a single instruction.

As the programs of the dataflow schema are described as directed graphs, the programmer may define an ill-conditioned graph that contains deadlock, unreachable nodes and so on. To guide the programmer, the notions of well-behaved and well-structured are proposed[19].

A well-behaved node must have no circular dependencies. It should be enabled when all the input edges have events (i.e. tokens), and it must be self-cleaning (i.e. the graph must return to its original state after a node is executed). Thus, the execution of a well-behaved node will consume all matching events on their input edges and produce events for all of the output edges, and it will not affect the state of any other node.

The notion of well-structured is closely related to well-behaved. There is a set of construction rules for building larger well-behaved graphs based on a set of smaller well-behaved graphs:

A regular dataflow node is a well-structured graph. A conditional graph is a well-structured if the subgraph of each branch is well-structured. A loop graph is a well-structured graph if the loop-body is a well-structured graph. A graph constructed with reference to a directed acyclic graph (DAG) is a well-structured graph if each node of the DAG is well-structured. A well-structured dataflow graph is guaranteed to be well-behaved if every node of this graph is well-behaved.

The dataflow schema supports parallelism inherently. As it is a pure activity model, thus, there are several studies that try to combine it with the memory model and the synchronization model to build parallel PXMs, such as parallel Turing machine[20].

3.6.2. Advancement

The POG is highly inspired by the dataflow model. Like dataflow, it is an asynchronous parallel model driven by the events. Accordingly, the OG is inspired by the dataflow graph of the traditional dataflow model, and it also has the well-behaved and well-structured property.

However, a fundamental difference between the POG and the dataflow model is that POG includes the memory model. The dataflow model is a pure activity model; no memory model or synchronization model is involved. What's worse, the dataflow model usually advocates some extension models that are built upon some basic notions of von Neumann architecture, e.g. instructions and the single-assignment shared memory model. Thus, they are not suitable for the brain-inspired computing hardware. In contrast, the POG proposes a memory model that collocates the storage and processing, which avoids the conflict caused by concurrent accesses to the same memory location in the traditional memory model; thus, it is more suitable.

Another important difference between the POG and the dataflow is the notion of *event*. Although *event* of the POG is similar to the availability of data in the dataflow model, it can also act as a simple relationship of precedence, not specific data. Thus, it provides greater flexibility and expressiveness. The event may even represent a data/signal generated from outside the computation itself, which allows the POG to describe the computation in an open environment. Moreover, the dataflow and all derived models depend on the strict equivalence based on the Turing completeness.

4. Execution Primitive Graph

The position of *execution primitive graph* (EPG) in the brain-inspired computing system hierarchy is similar to instruction sets in modern computer systems. It is close to hardware implementations while providing a degree of abstraction. We use a two-tiered graph representation to express it. Supplementary Figure 2 shows an example of the two-tiered graph to represent a LIF neuron model.

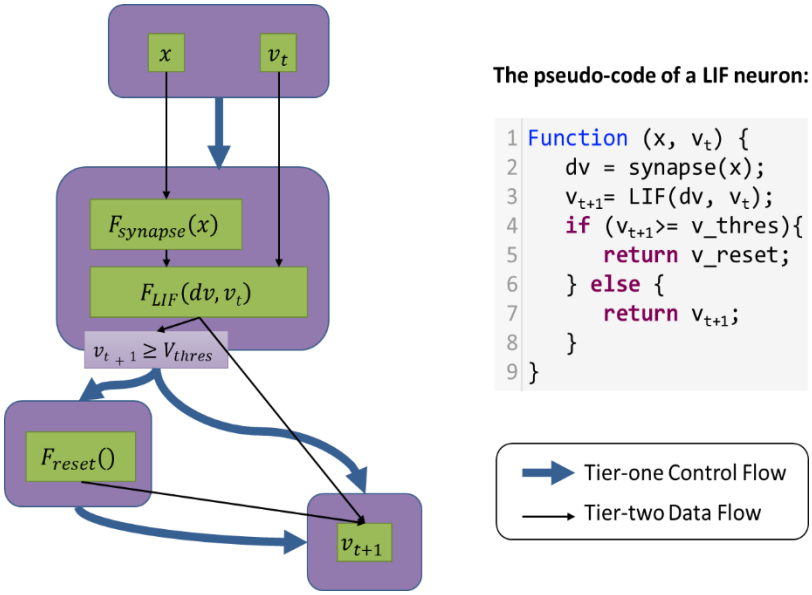
The tier-one is a control-flow graph (CFG). In the CFG, each node represents a basic block that contains pure computations defined by the tier-two graph. Each edge represents a control transfer (e.g. jump). A basic block can be optionally attached with a branch condition to choose the next basic block to execute if it has multiple successors.

The tier-two is a simplified dataflow graph: Each node is an *execution primitive*, which is a basic computational function that can be executed directly in the hardware. A node can have multiple input ports and output ports and can also have optional parameters of computation. Each edge connects the output of one node to the input of another node, which represents a data dependency between them.

Further, if a node has more than one successor nodes (different nodes are in different branches, that is, in different basic blocks), its output port(s) will be connected to the corresponding input ports of all the successors. For a node has more than one predecessors located in different branches, the situation is similar.

Thus, the execution order of the whole EPG is controlled by the tier-one CFG: Only one basic block is executing at any time, and only the *execution primitives* inside the executing block will be executed. Another prerequisite for execution is that all corresponding input ports have received data. A basic block finishes the execution when all *execution primitives* inside it have completed, and the control will be transferred to the next basic block according to the branch condition (if existing).

The two-tiered graph representation makes EPG composable, and the approximation capability is provided by its primitives. We define a basic set of *execution primitives* accordingly.



Supplementary Figure 2. Execution primitive graph

4.1. Basic Execution Primitives

The *basic execution primitives* just contain two primitives:

- **Weighted-sum:** It has one input x, one output y and two parameters w and b. Each element in the output vector is the sum of all the corresponding weighted inputs.

$$y_i = \sum_j w_{ij} x_j + b_i$$

- **Rectified Linear Unit (ReLU):** It is an element-wise operation that outputs zero if the input is negative and output the input value otherwise. It is the simplest non-linear function that only provides non-linearity at zero point.

$$y_i = \max(x_i, 0)$$

According to the universal approximation theorem[8], these two primitives is enough to approximate any function.

Note that, the EPG with *basic execution primitives* is just an instance. There are also other primitive sets that are capable of

forming universal approximators and make the EPG *neuromorphic complete*. Moreover, real neuromorphic hardware can provide more *execution primitives* for efficiency, just like that instruction sets in modern processors usually provide much more instructions than what are necessary to build a universal Turing machine.

4.2. **Wide Compatibility of the Basic Execution Primitives**

The *basic execution primitive* set is widely compatible with chips for either artificial neural networks (ANNs)[21][22], spiking neural networks (SNNs)[23][24][25], or the hybrid of the two[26].

The weighted-sum and ReLU are the most widely used operations in ANN. So, almost all deep learning chips support the two primitives directly.

For SNN hardware, Leaky-Integrate-and-Fire (LIF) neuron model is one of the simplest spiking neuron models; most SNN hardware can support it (or its variants). The following is proof that the LIF model essentially contains these two basic primitives.

In a typical LIF model, each spike that arrives at the i-th synapse will accumulate electric charge w_i in the capacitor. If the firing rate at the i-th synapse is s_i, the average current input to the neuron will be

$$I = \sum_i s_i x_i$$

The current will increase the membrane potential $V_m(t)$ according to

$$C_m \frac{dV_m(t)}{dt} = \max(I - I_{leak}, 0)$$

where C_m is the membrane capacitor, and I_{leak} is the leakage current. Thus, the membrane potential will be

$$V_m(t) = \max\left(\sum_i \frac{s_i}{C_m} x_i - \frac{I_{leak}}{C_m}, 0\right) t$$

When V_m reaches a threshold V_{th}, the neuron will fire a spike and reset V_m. Thus, the interval to issue a spike should be

$$\frac{V_{th}}{\max\left(\sum_i \frac{s_i}{C_m} x_i - \frac{I_{leak}}{C_m}, 0\right)}$$

Then, we can get the output firing rate y as follows

$$y = \max\left(\sum_i \frac{s_i}{V_{th} C_m} x_i - \frac{I_{leak}}{V_{th} C_m}, 0\right)$$

It contains the weighted-sum and the ReLU operations needed by the *basic execution primitives*; the weight and bias terms are $w_i = s_i / V_{th} C_m$ and $b = I_{leak} / V_{th} C_m$ respectively.

For a complicated graph, the adjacent linear operations can always be merged into one linear operation. Thus, the entire graph can be expressed as many linear operations separated by the non-linear operations. We can connect the linear operations with the following non-linear operation and implement them with LIF neurons supported by hardware.

SNN hardware also introduces a lot of constraints in the computation of the LIF neuron model, which have been considered to support by us.

4.2.1. Support for Positive Bias

If I_{leak} is constrained to be positive, the positive bias term can be implemented with an additional synapse that has a constant

input firing-rate and a positive weight.

4.2.2. Support for Negative Inputs

Firing-rates are naturally positive. To support negative inputs, we use a new encoding method in which one value x is represented with two positive firing-rates x^+ and x^- as following

$$x = x^+ - x^-$$

Thus, the computation of x can be implemented as following

$$\max\left(\sum_i w_i x_i + b, 0\right) = \max\left(\sum_i w_i x_i^+ + (-w_i) x_i^- + b, 0\right)$$

We only need two synapses with opposite weights to process x^+ and x^- respectively.

4.2.3. Support for Negative Outputs

Similarly, if the output without the ReLU operation is required, we can also use y^+ and y^- to represent y as following

$$y = y^+ - y^-$$

With this encoding method, to compute

$$y = \sum_i w_i x_i + b$$

Means we should compute y^+ and y^- separately as following

$$y^+ = \max\left(\sum_i w_i x_i + b, 0\right)$$

$$y^- = \max\left(\sum_i (-w_i) x_i - b, 0\right)$$

4.2.4. Support for More Bits

If the number of bits to represent w and x are constrained in hardware, we can support higher bits with encoding. Without loss of generality, we assume that the number of bits for w and x are n. We can support $n+1$ bits by representing w and x with w^H, w^L and x^H, x^L respectively as following

$$w = w^H + w^L$$

$$x = x^H + x^L$$

Where w^H, w^L, x^H, and x^L are all n-bit. The computation of $(n+1)$-bit of w and x can be implemented as following

$$\max\left(\sum_i w_i x_i + b\right) = \max\left(\sum_i w_i^H x_i^H + w_i^L x_i^H + w_i^H x_i^L + w_i^L x_i^L + b\right)$$

Note that, the aforementioned discussion of implementing the *basic execution primitives* is only based on the functionality of a simple LIF neuron model. Thus, it is widely applicable and then universal. Real chips are more complicated and provide richer functionality to make it easier to support the *basic execution primitives*. This is obviously true for ANN hardware as well.

4.3. Proof of Neuromorphic Completeness

The EPG can approach any Turing-computable functions in different ways: We can reduce the whole EPG to a single MLP that can approximate the entire function directly. Or, we first achieve basic arithmetic functions used in some Turing-complete system (e.g. basic *Boolean* operations) and then compose any function precisely with them, then the EPG becomes a Turing-complete system that approaches the function with composable algorithms. Moreover, the EPG can hybrid the two approaches by approximating more coarse-grained functions with MLPs and combine them to achieve the given function accurately. The granularity can range from basic Boolean operations to the entire function.

Here, we provide a constructive proof of how to approximate arbitrary functions with the *basic execution primitives*.

4.3.1. Construction of Unary Functions

For any unary function $y = f(x)$, its input range is $[x_1, x_2, \dots, x_n]$. We need to construct function $y = F(x)$ with only the *basic execution primitives* such that for a given $\epsilon \geq 0$, $\|f(x) - F(x)\| \leq \epsilon$ is satisfied for all $x \in [x_1, x_2, \dots, x_n]$. We first consider the strictest case, $\epsilon = 0$, and then we will discuss how to reduce the number of primitives used when $\epsilon > 0$.We first construct the function $g(x, a, b)$ as following

$$g(x, a, b) = \frac{1}{b-a}\max(x-a, 0) - \frac{1}{b-a}\max(x-b, 0)\,, a < b$$

It only contains these two primitives and it will return zero when $x \leq a$, return one when $x \geq a$ and return $(x-a)/(b-a)$ when $a < x < b$. Then, suppose $x_1 < x_2 < \cdots < x_n$, we can construct the function $F(x)$ as following

$$F(x) = f(x_1) + \sum_{i=1}^{n-1}[f(x_{i+1}) - f(x_i)]g(x, x_i, x_{i+1})$$

All the coefficients of g are constant weights since the input range and function f are known. Thus, $F(x)$ is constructed with only the two primitives. It is easy to ensure that $F(x_i) = f(x_i)$ for $1 \leq i \leq n$.

The constructed function $F(x)$ is a piecewise linear function with $\left(\mathrm{x_i}, f(x_i)\right)$ as the endpoints of each segment.

4.3.2. Construction of Binary Functions

For any binary function $z = f(x, y)$, we also have the input range $[x_1, \dots, x_n]$ and $[y_1, \dots, y_m]$ for x and y. Similarly, we need to construct function $z = F(x, y)$ with the *basic execution primitives* such that for a given $\epsilon \geq 0$, $\|f(x, y) - F(x, y)\| \leq \epsilon$ is satisfied for all $x \in [x_1, \dots, x_n]$ and $y \in [y_1, \dots, y_m]$. We also first consider the strict case when $\epsilon = 0$ and then discuss how to reduce the number of primitives used when $\epsilon > 0$.

We construct function

$$h(x, y, a, b, c, d) = g(g(x, a, b) + g(y, c, d), 1, 2)$$

It will return zero when $x \leq a$ or $y \leq c$, return one when $x \geq b$ and $y \geq d$. Otherwise, it will increase from zero to one linearly.

Then, suppose $x_1 < \cdots < x_n$ and $y_1 < \cdots < y_m$, we can construct the function $F(x, y)$ as following

$$F(x,y) = f(x_1,y_1) + \sum_{i=1}^{n-1}[f(x_{i+1},y_1) - f(x_i,y_1)]h(x,y,x_i,y_1,x_{i+1},y_1)$$

$$+ \sum_{i=1}^{m-1}[f(x_1,y_{i+1}) - f(x_1,y_i)]h(x,y,x_1,y_i,x_1,y_{i+1})$$

$$+ \sum_{i=1}^{n-1}\sum_{j=1}^{m-1}[f(x_{i+1},y_{j+1}) - f(x_i,y_{j+1}) - f(x_{i+1},y_j) + f(x_i,y_j)]h(x,y,x_i,y_j,x_{i+1},y_{j+1})$$

All the coefficients are constant weights since the input range and function f are known.

The constructed function $F(x,y)$ is also a piecewise linear surface. Each$\left(x_i,y_j,f(x_i,y_j)\right)$, $\left(x_{i+1},y_j,f(x_{i+1},y_j)\right)$, and $\left(x_i,y_{j+1},f(x_i,y_{j+1})\right)$ forms a small triangle and each $\left(x_{i+1},y_{j+1},f(x_{i+1},y_{j+1})\right)$, $\left(x_{i+1},y_j,f(x_{i+1},y_j)\right)$, and $\left(x_i,y_{j+1},f(x_i,y_{j+1})\right)$ also forms a small triangle.

4.3.3. Construction of n-nary Functions

For a more general case, such as an n-ary function $y = f(X)$ where $X \in \mathbb{R}^n$, suppose we expect the function to pass through a set of possible inputs $I = \{X_1, X_2, \dots, X_m\}$ and the corresponding y is $\{f(X_1), f(X_2), \dots, f(X_m)\}$. We can construct such a function $y = F(X)$ composed of *basic execution primitives* by induction.

We define a sequence of sets $\hat{I}_1 \subset \cdots \subset \hat{I}_m = I$ and functions $F_1(X), \dots, F_m(X)$ that pass through all points in the corresponding sets. All the sets $\hat{I}_i$ satisfy the condition that none of $X \in I - \hat{I}_i$ are in the convex hull of $\hat{I}_i$. We define $\hat{I}_1 = \{X_{(1)}\}$, where $X_{(1)}$ is any point in I, and $F_1(X) = f(X_{(1)})$ is a constant function. Obviously $\hat{I}_1$ and $F_1(X)$ satisfy the induction conditions. Then, if $\hat{I}_i$ and $F_i(X)$ satisfy the induction condition, we will construct $\hat{I}_{i+1}$ and $F_{i+1}(X)$ that satisfy the induction condition as follows.

We find $X_{(i+1)} \in I - \hat{I}_i$ that none of $X \in I - \hat{I}_i - \{X_{(i+1)}\}$ are in the convex hull of $\hat{I}_i \cup \{X_{(i+1)}\}$. Then, we define $\hat{I}_{i+1} = \hat{I}_i \cup \{X_{(i+1)}\}$. Obviously, $\hat{I}_{i+1}$ satisfies the induction condition. Since $X_{(i+1)}$ is not in the convex hull of $\hat{I}_i$, we can find a surface $W_{i+1}X + B_{i+1} = 0$ such that $\hat{I}_i$ and $X_{(i+1)}$ are on the each side of the surface. Assuming $W_{i+1}X_{(i+1)} + B_{i+1} > 0$ and $W_{i+1}X + B_{i+1} \leq 0$ for all $X \in \hat{I}_i$, we just define $F_{i+1}(X)$ as follows:

$$F_{i+1}(X) = F_i(X) + \frac{f(X_{(i+1)}) - F_i(X_{(i+1)})}{W_{i+1}X_{(i+1)} + B_{i+1}}\max(W_{i+1}X + B_{i+1}, 0)$$

When $X \in \hat{I}_i$, the second term will be zero, and $F_{i+1}(X) = F_i(X) = f(X)$. When $X = X_{(i+1)}$, the second term will become the gap $f(X_{(i+1)}) - F_i(X_{(i+1)})$ and F_{i+1} also satisfies the condition.

Repeat this process until $F_m(X)$ has passed all points in I, and then we can use $F_m(X)$ to approximate the n-ary function.

4.4. Support of Learning Rules

Brain-inspired computing hardware that does not support learning (including training) natively usually implement the weighted-sum operations as

$$f(x|w,b) = \sum_i w_i x_i + b$$

where w and b are pre-configured parameters. Learning is to update w with some learning rules. Thus, if learning is required, the weighted-sum operation should be programmed as

$$f(x, w, b) = \sum_i w_i x_i + b$$

Where w and b becomes inputs that will dynamically change during learning. The weighted-sum operation becomes a ternary function and can be implemented with the method in Supplementary Information 4.3.3.

However, this implementation is really inefficient with the *basic execution primitives*. We could add an additional *learning primitive* in the EPG for hardware to support learning efficiently. The *learning primitive* is the ternary version of weight-sum primitive. It corresponds to the weight-sum operation with the *parameter updater* (Supplementary Information 3.3.1) in POG: An update event in the POG corresponds to the new input w and b, for the *learning primitive*.

5. Compilation Guidelines

The compiler is used to transform the POG to the EPG. A major difference between them are the *operators* (*primitives*) used. Thus, the compiler needs to transform any *operator* in the POG to the *execution primitive(s)* to meet *neuromorphic complete*. Moreover, the control flow in POG is determined by the events while in EPG, it is determined by the tier-one CFG. Thus, we need to extract the control flow from the POG and transform it into the CFG.

5.1. Basic Compilation Flow

The first step is to extract the control flow: We split the POG into basic blocks by locating control-flow *operators.* The *decider* and *true/false gates* start a branch and the *condition merger* ends the branch.

Then, we transform all the other *operators* into *execution primitives*, which includes two major steps: Template Matching and General Approximation.

5.1.1. Template Matching

We first conduct a template matching to find all *operators* that can be merged by one or multiple *execution primitives*. Most linear operators can be replaced with the *weighted-sum* primitive directly. Moreover, the hardware usually provides more than the basic set. If any *operators* can be transformed precisely with the *execution primitives*, it is also handled in this step.

5.1.2. General Approximation

For the remaining *operators*, we will split or merge them into a proper granularity to approximate and use the weighted-sum and ReLU primitives to approximate them according to the constructive proof mentioned before. A proper granularity is that with acceptable complexity of the corresponding approximator.

The complexity depends on the number of points to pass through. The number usually depends on the number of possible inputs, which increases exponentially with the number of operands increasing. For example, a coarse-grained operator contains multiple weight-sum-operations and non-linear function can have exponential complexity to approximate. Thus, in our compilation flow, we avoid approximating at this granularity.

So far, we give the basic flow that uses general approximation to cover all unsupported operators. However, there are still many optimization technologies to reduce the number of primitives consumed.

5.2. Optimized with Boolean Logic

The number of *execution primitives* used to approximate the binary functions in our constructive proof is $O(n)$ where n is the number of valid inputs. When the number of bits to represent operands increases, the complexity will increase exponentially.

However, there is another way to form a complicated function: We can use the approximation method to construct basic Boolean functions and then construct any function with Boolean logic. The complexity is usually not exponential to the number of bits since modern computers are built based on this principle.

Thus, we can optimize the general approximation with a combination of the two ways. Some low-precision functions can be

implemented with the general approximation. Afterwards, we combine them with precise Boolean logic to form a high-precision function. The compiler could determine which one to use by comparing the complexity. For example, to form a 4-bit multiplier, we could use general approximation to approximate a 2-bit multiplier and use four 2-bit multipliers to construct a 4-bit multiplier accurately.

5.3. Optimization with Expression Simplification

A complicated function built upon Boolean logic or general approximation can be further simplified by combining the linear parts. Since the EPG is composed of only two kinds of primitives, the simplification methods are intuitive.

- **Combine adjacent weighted-sum operations:** Adjacent weighted-sum operations can be merged into one weighted-sum operation.
- **Combine like terms:** weighted-sum operations with like terms can be merged.

For example, the approximation function for a unary function (Supplementary Information 4.3.1) can be combined as

$$F(x) = f(x_1) + \sum_{i=1}^{n-1} \frac{[f(x_{i+1}) - f(x_i)]}{x_{i+1} - x_i} [\max(x - x_{i+1}, 0) - \max(x - x_i, 0)]$$

$$= f(x_1) + \sum_{i=2}^{n-1} \left\{ \frac{[f(x_i) - f(x_{i-1})]}{x_i - x_{i-1}} - \frac{[f(x_{i+1}) - f(x_i)]}{x_{i+1} - x_i} \right\} \max(x - x_i, 0) - \frac{[f(x_2) - f(x_1)]}{x_2 - x_1} \max(x - x_1)$$

$$+ \frac{[f(x_n) - f(x_{n-1})]}{x_n - x_{n-1}} \max(x - x_n)$$

The combination reduces the number of primitives used by two times. The combination can also be performed across individual general approximation functions in a complicated function.

5.4. Optimization with Approximation

The Boolean logic optimization still keeps the error gap $\epsilon = 0$. If $\epsilon > 0$ is allowed, the EPG can be simplified significantly.

The constructive proof provides a way to approximate any function with piecewise linear functions that pass through the given points; thus, the complexity highly depends on the number of points that the approximator has to pass through. When $\epsilon > 0$, the number of segments or surfaces used can be reduced accordingly. Therefore, there is large optimization space when using fewer points for approximation.

5.5. Optimization with Data Redundancies

The optimization with approximation is based on random input data. However, the data is usually highly redundant in practical applications. We can first approximate the function under a larger error gap and then fine-tune the graph to decrease the error gap in order to satisfy the error constraint. There are tons of work on neural network compression that exploits this property, which mainly includes two approaches, neural network quantization[27] and neural network pruning[28].

6. Abstract Neuromorphic Architecture

The *Abstract Neuromorphic Architecture* (ANA) is an abstract hardware model that supports EPG. It has four parts: processing units (PUs) for *executing primitives*, scheduling units (SUs) for control flow and execution scheduling, the private memory equipped with each PU for storing the parameters and the intermediate data, and the inter-connection network that connects all the above-mentioned units.

It should be pointed out that SU is a logical concept while there would be several physical ways to implement it: It may be a centralized component responsible for all the controlling and scheduling jobs, or SUs may be distributed and each is responsible for the part assigned to it. Even in some hardware implementations, there is no functionally independent SU, which does not contradict the proposed abstract model.

PUs are designed to be distributed; they can sustain the running of the EPG in parallelism, that is, any PU can perform *execution primitives* under the scheduling of SU. Further, a PU is equipped with one private memory that can be only accessed by the PU itself. This design makes specific hardware implementations of ANA be in favor of collocating storage and processing, which is one of the key features of brain-inspired computing, while still keeping the compatibility with the hardware whose memory and computing are separate. The execution of PU can be event-driven or not.

Further, hardware often puts many efforts on the design of the connection between cores or chips, which can contain various connection modes at different levels. The inter-connection network of ANA is an abstraction at the functional level.

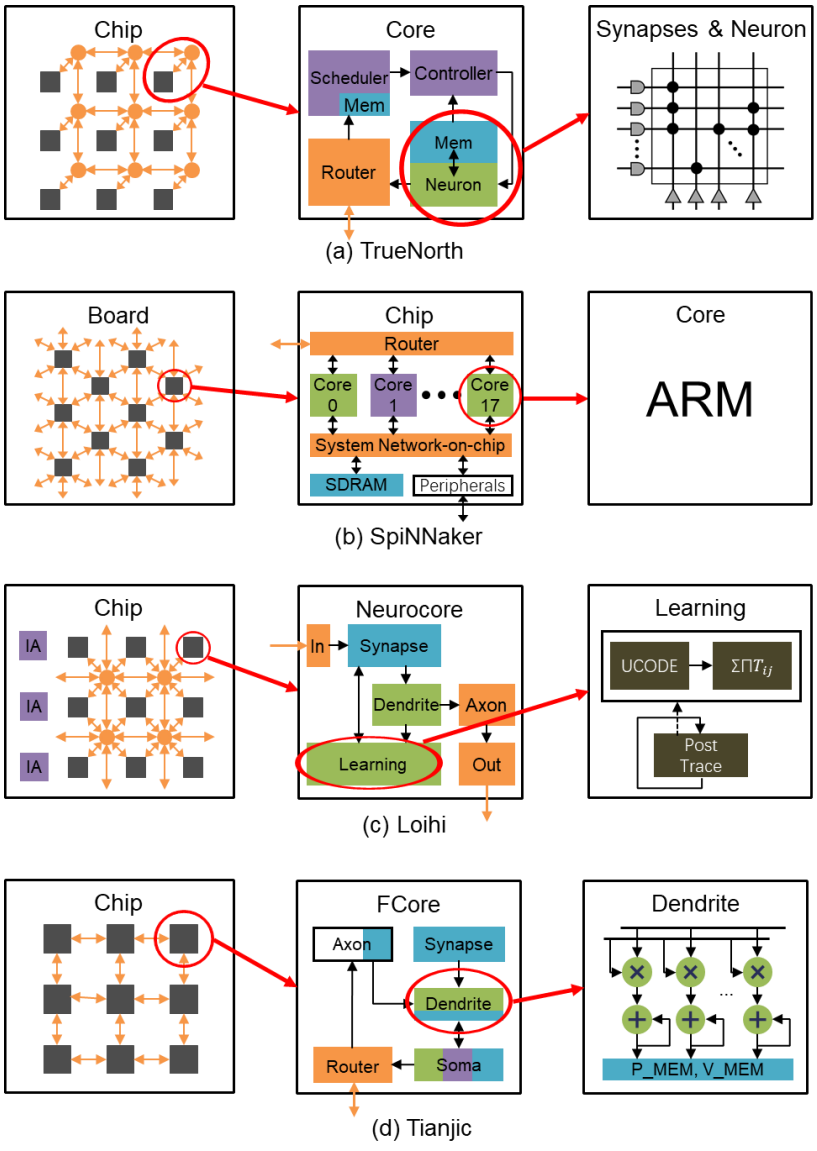
Thus, the memory abstraction of ANA is suitable for the collocation of storage and processing, distributed PUs are suitable for fine-grained parallel computing, and the separation design of SU and PU is conducive to event-driven computing. It is also intuitive and general enough to cover mainstream neuromorphic chips (Supplementary Information 6.1) and can give guidance for hardware design.

6.1. **Instantiations**

Specific hardware implementations of these units are diverse.

PU: The primary task of PU is to execute the *execution primitives*, so there is no constraint on the implementation form, e.g. it can be implemented as a control flow architecture or a dataflow architecture.

SU: SU implementations are very flexible, from the traditional control-flow soft cores (SpiNNaker[24]) to some customized



Supplementary Figure 3. The hardware instantiations

hardware, e.g. the lookup tables, the CLBs (configurable logic blocks) or some other configurable customized components, like Field Programmable Synapse Array (FPSA[29]). For some implementations that have no dedicated control flow hardware, the toolchain will convert the tier-one control flow schemas of the EPG into *execution primitives* that PU can achieve (Supplementary Information 7.2.3).

Private Memory: Some hardware adopts memory that collocates storage and processing, which merges the private memory and the PU together and applies in-situ computation. This is usually achieved based on non-volatile technologies, like memristor-crossbars, STT-RAM, etc. The general storage technologies, like SRAM, DRAM, etc., can also be used to achieve near-memory computing[30]. Some hardware separates the memory from the PU, while the memory module can be private, shared or global.

Inter-connection Network: Usually brain-inspired computing systems pursuit high scalability; thus, there are connections at different levels, like the core-level, chip-level, and board-level, which are implemented in a very flexible manner. One widely-used is the AER (asynchronous address event) communication system[31][32] that is suitable for event-driven computations.

Here we basically introduce four leadership-class brain-inspired computing platforms, including the hardware, software and programming paradigm. For hardware, we will explain how ANA is instantiated to these chips in detail.

6.1.1. TrueNorth

TrueNorth is a well-known brain-inspired computing chip proposed by IBM. It provides a comprehensive ecosystem for SNN computing, including hardware[23], support software[33] and programming paradigm[34].

Hardware (Supplementary Figure 3-a): A TrueNorth chip can be viewed as a pool of neurosynaptic cores. One neurosynaptic core contains 256 input lines (axons) and 256 outputs (LIF neurons) connected via 256-by-256 directed, programmable synaptic connections (private memory). Through on-chip and off-chip connections, neurosynaptic cores can form large-scale spiking neural networks.

In each neurosynaptic core, there is a neuron module that processes the main computation (PU) with co-located memory (private memory). The scheduler (with an input buffer) and the controller can be viewed as SU.

Software: Different software pipelines are provided for application runtime and development, as well as a TrueNorth simulator.

Programing Paradigm: A programming paradigm dedicated to TrueNorth architecture has been proposed, called *Corelet*. A *corelet* unit is an abstraction of one neurosynaptic core. Multiple *corelets* can form a composite *corelet* to achieve a more complicated function. Developers can repeat the composition procedure to construct a whole network based on many *corelet* units, which is compatible with the TrueNorth hardware.

6.1.2. SpiNNaker

The SpiNNaker project[24] of the University of Manchester aims to deliver a massively parallel million-core computer, which is suited to the modeling of large-scale spiking neural networks in biological real-time.

Hardware (Supplementary Figure 3-b): Each SpiNNaker chip is composed of 18 ARM968 cores, a shared memory of 128 MB, a router of network-on-chip, and other peripherals; a board contains 48 chips. All cores on the board are connected with a 2-D triangular mesh and more boards can form larger clusters (chip level inter-connected network). One of the unique features of SpiNNaker is the communication subsystem, which is customized for concurrent transmission of a large number of small packets and then adapts to the delivery of spikes.

In each chip, there are 17 soft cores. Usually, 16 cores are responsible for computation (PUs) and the last one is used to schedule these cores (SU). Cores communicate with each other through the system NoC (the core level inter-connection network). The system NoC also connects to an SDRAM for data storage (private memory).

Software: The software can be categorized into host software and SpiNNaker software (running on chips). SpiNNaker software is further divided into control software and application software: The former is a primitive operating system resident in the monitor processor of each chip, for task control, computation scheduling, I/O, etc., while the latter performs users' tasks.

Programing Paradigm: SpiNNaker provides an event-driven real-time programming model[36], which coincides with the hardware execution mode. Programmers can use PyNN, a simulator-independent language for building neuronal network models, to describe the target SNN and then map it onto the hardware using its toolchain software. SpiNNaker provides a series of pre-implemented neuron/synaptic models to facilitate development, based on fixed-point arithmetic; to improve performance, the ARM core it currently uses has no floating-point components.

The soft-core-based computing model of SpiNNaker brings programming flexibility while reducing execution efficiency.

6.1.3. Loihi

Loihi is an experimental chip[37] designed by Intel that supports SNNs to implement learning and inference.

Hardware (Supplementary Figure 3-c): A Loihi chip is composed of 128 neuromorphic cores and three embedded Intel Architecture (IA) cores. Between these cores, there is an asynchronous NoC for transporting messages (inter-connection network).

The three IA cores (SUs) are responsible for task assignment and scheduling of neuromorphic cores, and control the spike traffic into and out of the chip. They map the mimicked neuronal components onto the neuromorphic cores. In Supplementary Figure 3-c, *Dendrite* and the *learning module* can be viewed as the PU and *Synapse* as the private memory. The *In*, *Out* and *Axon* modules contain the routing logic which can be viewed as part of the inter-connection network.

Software: Loihi provides a compiler and a runtime library for deploying SNN models on chip. The compilation process includes preprocessing, resource allocation, and code generation. The runtime library is called during the step of code generation to produce the executable byte stream.

Programing Paradigm: Through Python-based APIs, Loihi provides some core programming primitives, including neuronal compartments, synaptic connections, synaptic traces, some neuron models that describe SNN dynamics and learning rules, to specify the SNN topology, the learning rule, etc.

6.1.4. Tianjic

Tianjic[26] is the first brain-inspired computing chip that supports the hybrid execution of ANN and SNN.

Hardware (Supplementary Figure 3-d): A Tianjic chip is formed by functional cores (FCores). One FCore is composed of units of *Axon*, *Dendrite*, *Synapse*, *Soma*, the NoC router, etc., which construct a neural computation pipeline: The *Axon* unit receives the input data from the router and buffers the data temporarily before sending it to the *Dendrite* unit. Then the *Dendrite* unit completes synapse computations (the weights are stored in the *Synapse* unit) using its internal MAC array; the membrane potential is stored in V_MEM (private memory) of the *Dendrite*. The computation result is further processed by the *Soma* (PU), a non-linear unit. The *Soma* has the internal memory for neuronal parameters and the configurable hardwired control logic. Finally, the router sends the output to other cores through a 2-D mesh network (inter-connected network).

Thus, from the aspect of ANA, the *Dendrite* and a part of the *Soma* achieve all computations; thus, they are the PU, while the *Soma* also completes the function of SU. Parameters and intermediate data are stored by the *Synapse* unit, the *Axon* unit and the *Soma* unit, that is, the private memory of an FCore is distributed across multiple units.

Software: Tianjic provides a compiler, which supports applications of ANN, SNN, and ANN-SNN hybrid. It can transform a trained neural network into an approximately equivalent network that complies with hardware constraints, and then map the latter onto the hardware.

Programing Paradigm: The *Axon*, *Dendrite*, *Synapse*, *Soma*, and *Router* need different configurations for different functions. By abstracting these configurations, Tianjic provides a series of functional primitives for these units. Programmers can use

these primitives to construct neural networks[38].

7. Design Guidelines of the Mapper

The mapper is used to deploy the EPG onto the specific hardware (this procedure is called *mapping*). The mapping result should be as efficient as possible under the premise of meeting the hardware restrictions, like the capacity limits of memory and network routing. Thus, the mapping guidelines include two main aspects: To meet the hardware constraints and to improve efficiency.

Hardware constraints: Physically, the capacity or capability of hardware resources is limited, like the memory, the routing, scheduling, etc. The mapper should take considerations of all these constraints and generate valid strategies.

Efficiency: An EPG may have unbalanced demand in computing, communication and storage: Taking a CNN (convolutional neural network) as an example, the convolutional layers are computationally expensive and use relatively few parameters, while the fully-connected (FC) layers are just the opposite. Considering the collocated computing and storage resources, a naïve mapping strategy would cause FC layers to occupy a lot of computing/storage resources, but the utilization of computing resources is very low.

7.1. General Mapping Framework

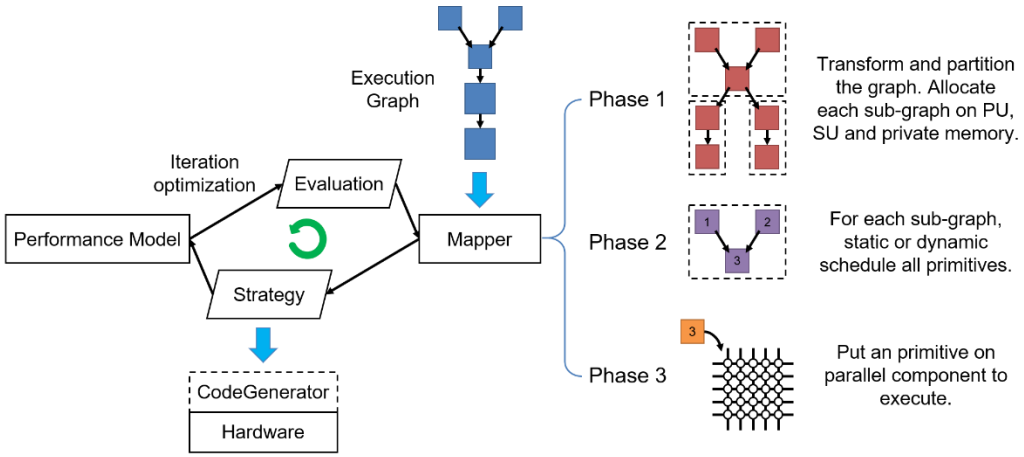
The mapping procedure generally contains three phases: Assign the EPG's tasks evenly to PUs, schedule the execution of primitives on each PU, and adapt each primitive to the PU.

In the first phase, the original EPG is partitioned into multiple sub-graphs and assign each sub-graph on one PU. A balanced partition should be made to approach the maximally pipelined fashion and a better load balance [39]. The corresponding control flow schemas and data storage of each sub-graph will also be assigned to the SU and private memory of its PU. The allocation strategy of the storage may be different for various hardware implementations (Supplementary Information 7.2.2).

In the scheduling phase, the static or dynamic scheduling can be done for each sub-graph. Static scheduling does not depend on runtime information, so it can determine the execution timing of each primitive before running. Dynamic scheduling is the opposite, which has to generate the scheduling rules during the runtime (e.g. using if/else or loop condition to control whether and in what order the primitives are executed on the fly). Anyway, the scheduling should satisfy the control and data dependencies, boost the whole dataflow execution and fully consider the consumption of hardware resources at run time (like the buffer resource for intermediate data, in Supplementary Information 7.2.4) to satisfy relevant constraints. Scheduling strategies will be carried out by the SU. If there is no SU or SU is not flexible enough, they can be accomplished using the general approximation method by the PU (Supplementary Information 7.2.3). The scheduling implementation also depends on whether the hardware is event-driven (Supplementary Information 7.2.1).

The last phase will deploy the specific primitives on the computing components inside PU. Usually the latter has parallel processing capability and provides encapsulated libraries for such primitives. Library interfaces can be called to finish the whole mapping procedure.

The above mapping procedure is just a general principle. Taking the hardware complexity and variety into consideration, in practice a behavioral level simulator for the target hardware could be implemented as a supplementary measure; thus, we can evaluate the effectiveness of a mapping scheme and adjust the scheme accordingly. This process can be repeated and the direction of optimization can be adjusted by some heuristic algorithms[40][41](Supplementary Figure 4).



Supplementary Figure 4. The framework of the mapper and its mapping procedure

7.2. **Mapping on Various Hardware**

The above content is just a general framework, while the specific mapping and optimization strategies are closely related to the functional characteristics of the target hardware, especially in the following four parts.

7.2.1. Event-Driven or Not

The underlying hardware may execute in the event-driven mode or not. Each tire-two block in the EPG is a dataflow sub-graph without any control-flow schema, i.e., no dynamic scheduling is needed for all tier-two blocks (only control flow could result in the dynamic scheduling). Thus, if the hardware is not event-driven, we can schedule the dataflow blocks statically (that is, the order of all primitives is fixed before running) and then allocate a time window for the execution of each primitive.

If event-driven is enabled, primitives will be triggered by the data dependency dynamically. In this case, the scheduling strategy can comply with the event-driven paradigm and use the SU(s) to check the data dependency in runtime; the predetermined sequence information is not required.

Actually, most of the brain-inspired computing hardware[23][24][25][26] claim to support the event-driven mode in some form. The real mapping strategy should be designed carefully based on the specific hardware implementation. We only give a preliminary discussion on this aspect, as well as the following.

7.2.2. Memory

Any mapping strategy must satisfy the data storage requirement of each *execution primitive*.

Collocated memory and computation is an important feature for brain-inspired computing. Some hardware follows this principle inherently, like the in-situ computation modules based on non-volatile technologies. Some hardware adopts the near-memory processing strategy; each memory module is adjacent to and monopolized by a PU (e.g. like the synapse and neuron modules in each neurosynaptic core of TrueNorth[23], or the similar memory structures in the cores of Loihi[25] and Tianjic[26]). In this case, the mapping strategy is similar to the hardware that collocates memory and computation inherently because the ratio of computing power to memory capacity of a PU is fixed. These approaches have the advantage of reducing data movement overhead and the disadvantage of having a less flexible allocation strategy.

Some hardware provides shared memory at the chip-level, that is, memory shared by all PUs on a chip (like SpiNNaker[24]). In this case the mapper can use these shared memories synergistically based on the amount of memory required for each primitive.

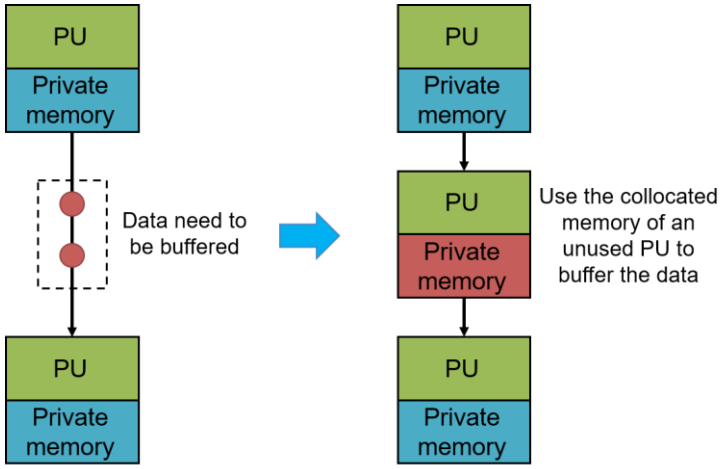
7.2.3. Support for Control-Flow

Currently, most mainstream brain-like chips support control-flow schemas in various forms, such as using general-purpose processors[24][25], configurable circuits[23][26] and so on.

If the target hardware does not have a flexible SU, we have to transform all the control flow schemas into primitives that PUs support (in the step of *General Approximation* of the compiler). These primitives will approximate the control schema and be mapped on PUs, like the normal primitives. Specifically, considering the branch caused by the condition (e.g. if/else) or iteration (i.e. a loop branch), since there is no SU for condition assertion, all the conditional branches should be mapped and executed on the hardware, and only one branch will generate valid data. The input of these PUs contains two parts, one is the data, and the other is the control signal which controls whether the PU performs valid calculations or not. For example, in a loop, the loop body will always be running and only generate valid results if the loop condition is satisfied.

7.2.4. Data Buffer

In an EPG, there is intermediate data between the nodes (i.e. primitives) of the tire-two dataflow graph. After the EPG is mapped to the hardware, intermediate buffers are needed to store them; the buffer size is directly related to the mapping algorithms. The larger and more flexible the buffer, the more flexibility the mapping strategy has. The theoretical minimal-intermediate-data scheduling can be viewed a scheduling on a *static dataflow graph*[42][43]. The *static dataflow graph* only allows at most one data token on each edge, i.e. any primitive can only be executed when all its dependencies are satisfied and its output edge is empty. The disadvantage is that it cannot fully exploit the parallelism. Given more buffers, the mapping strategy can be optimized accordingly to improve concurrency, like pipelining the computation or time division multiplexing PUs.



Supplementary Figure 5. Use a PU as the buffer for intermediate data

The most extreme situation is: the hardware does not have any configurable buffers (e.g. no any shared memory), which can only keep data in the memory module that is collocated with computation and keep data in transit (i.e. in the inter-connection network). In this case, we can use the collocated memory of the PU as a buffer (Supplementary Figure 5).

7.3. Others

So far, we've introduced the general flow of mapping and the ideas of optimization in four major areas. But, the final code generation and optimization is highly dependent on the underlying target hardware. In our toolchain (Supplementary Information 8), a preliminary adaption interface has been provided to allow designers to implement them for specific hardware.

For instance, the routing architecture of FPSA[50] is similar as the island-style architecture in FPGA chips; thus the mapping problem is the same as FPGA's placement and routing. Then, we directly use some existing open source EDA tool from the FPGA community. A typical mapping flow in FPSA includes two part: one is placement and another is routing. The placement is implemented with simulated annealing algorithm. Its cost function is estimated with the distance of communication. The routing is generated with multiple phases of Dijkstra's algorithm. In each phase, the algorithm will increase the weight of the

paths with conflicts to avoid conflicts.

Further, we should point out that the compilation and mapping cannot efficiently deal with transient activities during runtimes that are unknown at compile time or are very small probability, such as many spikes converging simultaneously on a particular group of neurons. They need to be optimized by the runtime system. This can be part of future work.

8. Implementation Instance of the Toolchain

We have implemented a toolchain software in comply with the proposed hierarchy, including the corresponding compiler and the mapper, which supports three typical hardware platforms:

(1) General-purpose graphic processing units (GPGPUs). It is a Turing complete general-purpose computing platform, widely used in deep learning and some SNN simulation fields.

(2) FPSA[50]. It is a DNN accelerator based on emerging non-volatile memory devices. Owing to the extremely high weighted-sum computing density and computing performance, it is used as a platform that achieves the *neuromorphic completeness* through the general approximation using the *basic execution primitives*. Accordingly, by the compiler, we have explored the relationship between different approximate granularities, resource consumption and performance.

(3) Tianjic[26], a state-of-the-art brain-like computing chip. In terms of how *neuromorphic completeness* is implemented, it is between the first two.

In the following content, we give a brief introduction to FPSA first. Then we present the toolchain's compilation and mapping processes.

8.1. FPSA

Non-volatile technologies are usually compared to biological synapses. There have been many studies[44][45][46][47][49][50] that utilize their characteristics, e.g. synaptic efficacy and synaptic plasticity[61] to mimic synapses. In the simplest form, incoming signals are multiplied by the stored weights of synapses, which are usually represented as programmable, analog, non-volatile resistance. Supplementary Figure 6 illustrates how such operations can be accomplished in-situ using a crossbar with resistive random-access memory (ReRAM, a widely used type of memristor), and more crossbars can be connected to form a dense, large-scale processor featuring in-situ computations. There is a ReRAM cell in each intersection of the crossbar. An input voltage vector V_i is applied to the rows and is multiplied by the conductance matrix of ReRAM cells G_{ji}. The resulting currents I_j are summed across each column. The output current vector can be calculated by $I = GV$.

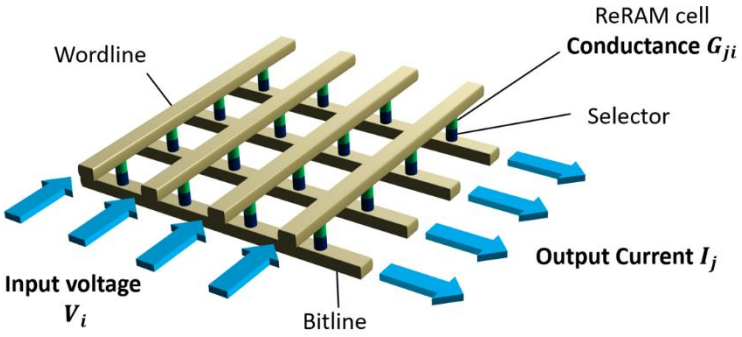
FPSA is such a DNN accelerator architecture that uses ReRAM-crossbars to process weighted-sum operations. It uses analog circuits to implement the ReLU function to eliminate the ADC (analog-digital-converter) overhead between the crossbar and ReLU, as both of them are analog circuits. Moreover, the spike encoding schema is used to reduce the consumption of ADCs/DACs that connect the analog computing parts to the digital communication sub-system on chip. The above two technologies can greatly increase the computing density and performance.

Other challenges for the chip design and their solutions are as follows:

(1) The high efficiency of ReRAM-crossbars puts high demands on the communication performance on chip. A ReRAM-based reconfigurable inter-connection[51] is employed to provide rich wiring resources on chip.

(2) The effects of crossbar non-idealities, including the ReRAM-device variation, the IR-drop along the word-line, etc., will affect the neural network accuracy. Accordingly, FPSA proposes the 'add' method[50] that can effectively improve the accuracy of multiple ReRAM cells representing a weight, and uses the device-characteristics-aware training algorithm[52] to improve the end-to-end accuracy further.

(3) The support for control flow and data buffer. It uses reconfigurable logic units (CLBs) to achieve the control-flow and scheduling logic, and uses traditional static random-access memory (SRAM) as data buffer on chip.



Supplementary Figure 6. ReRAM crossbar

The detailed configuration of FPSA is presented in Extended Data Table 4. Compared to one of state-of-the-art ReRAM based neural network processors, PRIME[47], its computational density improves by 31×; for representative DNNs, its inference performance can achieve up to 1000× speedup.

From the computing functional perspective, FPSA only supports the ReLU and in-situ weighted-sum operations efficiently. They are the same as the *basic execution primitives*. Thus, FPSA is *neuromorphic complete*. With the proposed system hierarchy and the compilation technology, we can extend its application scope from DNNs to any Turing-computable functions, so that any model described with the POG can be mapped to FPSA.

8.2. Compilation and Mapping Processes

8.2.1. GPGPU Compilation

GPGPU provides a Turing complete ISA, and then it is possible to implement any Turing complete operations through these instructions. Moreover, the manufacturers also provide a series of tools, as well as programming interfaces at different levels (such as CUDA, cuBLAS, cuDNN and so on) for developers; thus, the EPG for the GPGPU may contain flexible control flow schemas and abundant computing primitives. In our design, the computing primitives in the EPG are the same as the operators of POG, and the conversion between them is merely transferring the control flow operators into control flow graph. For computing operations which are not supported by the GPGPU directly, we either implement the corresponding CUDA kernels manually or refer to some development frameworks for existing implementations (e.g., we use the PyTorch framework to provide DNN related operators). In summary, all *operators* of the POG are achieved by precise general-purpose computation.

8.2.2. GPGPU Mapping

GPGPUs features the single-instruction-multiple-thread (SIMT) execution model that provides massive parallelism, as well as flexible and hierarchical memory buffers; they also implement coarse-grained (i.e. at the warp level; a warp is the smallest unit of threads that execute simultaneously) control flow operations.

Thus, it is easy to map the control flow graph directly to the warp-grained control flow of GPGPU and then execute the dedicated computing operations with multiple threads to achieve higher parallelism. The data dependency inside the basic blocks can be converted to the order of the instructions statically. Although GPGPU doesn't collocate storage with processing, it provides massive on-chip memory; thus, it is feasible to provide parts of the global memory as the private memory of each operator.

8.2.3. Tianjic Compilation

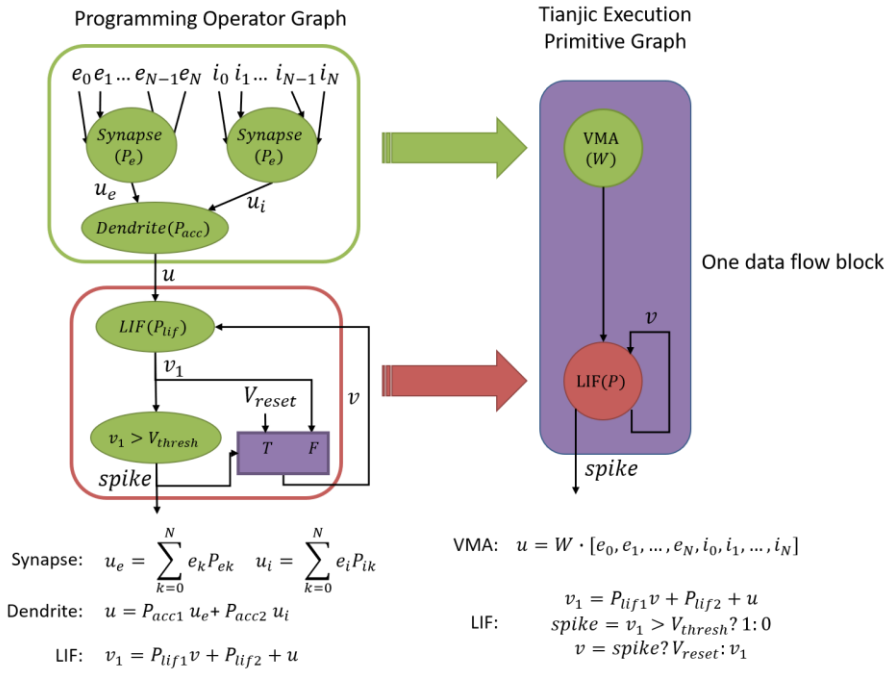
The Tianjic chip provides a series of primitives that aim to support ANN/SNN functionalities. Part of the primitives is listed in Extended Data Table 3; the compiler should convert the POG into the EPG composed of these primitives.

The compilation process adopts two methods: general approximation and universal computation. Tianjic primitives include some *basic execution primitives*, like Vector Matrix Multiplication (VMM), Vector Matrix Accumulation (VMA, the spike version of VMM), ReLU, that can be used to approximate any part of a POG with different granularities. In addition, Tianjic has the LUTs (look-up tables) as another way for function approximation. Tianjic also provides other primitives for precise computations. These primitive, as templates, can match and replace the *operators* or sub-graphs that have the same functionalities in the POG.

Supplementary Figure 7 gives an example of how to convert a POG to the corresponding EPG with the two methods. The VMA primitive in the EPG is used to approximate the green area of the POG. The weights of VMA, which are different from the original parameters, are produced by the constructive compiling procedure. Because the green part just contains linear operations, the approximation can be precise. Further, the orange part, i.e. the main part of LIF model, can be exactly matched and converted to the LIF primitive in the EPG by template matching. The LIF will have same parameters and same calculation steps as the original area.

8.2.4. Tianjic Mapping

Tianjic is designed with abundant PU-equivalent resources to achieve massive parallel execution of ANN, SNN and the hybrid. Its SU-equivalent components support the general control flow logic in the ANN/SNN through customized circuits. Further, Tianjic can expose its hardware configuration for programmers to implement some logic that is not covered by Tianjic primitives. Other complex logic can be approximated by basic *execution primitives* before being mapped on PUs.



Supplementary Figure 7. Convert a POG to an EPG for Tianjic. The POG describes a standard LIF model with two-synapse inputs. Each input data is accumulated in the operators of Synapse and Dendrite with specific weights. This accumulation part can be equally approximated by the Tianjic primitive: VMA. Then, the LIF model will first process the input membrane potential u in the LIF operator, generate the spike and then refresh the membrane potential by v, which is matched to the LIF primitive exactly as a whole.

Moreover, Tianjic adopts the near-memory computation mode, while the memory in an FCore can be shared by many primitives in the same core or used as a buffer for intermediate data. In addition, many different primitives are assumed by different hardware components; thus, after mapping, each primitive could be assumed by different components to form a computational pipeline in hardware naturally.

8.2.5. FPSA Compilation

The architecture of FPSA includes many compact and efficient ReRAM-based processing element (PE), which only supports the ReLU and in-situ weighted-sum operations. To match the throughput of PEs, its communication subsystem is an FPGA-like reconfigurable routing architecture with massive wiring resources. Moreover, it provides configurable on-chip buffer and configurable logic blocks (CLBs) to flexibly implement arbitrary control logic. Thus, the EPG for FPSA can match well that composed of the *basic execution primitive* (described in Supplementary Information 4.1).

The compilation flow is also the same as described in Supplementary Information 5. It heavily relies on the universal approximation capability of the *basic execution primitives*. Operators in the POG are merged or split into parts. For each part, we construct an approximator with the *basic execution primitives* as described in Supplementary Information 4.3. Moreover, the granularity of each part will affect the resource consumption significantly. The most fine-grained way is to approximate all the basic mathematic functions of individual *operators* in the POG. The most coarse-grained way is to regard the entire POG as one part and approximate it directly. However, both of them are not the most efficient. Some configurations between the two cases are much better.

We use the Universal Approximation (UA) method to construct a specific approximators. To construct an n-ary approximator for a given function $y = f(X)$ where $X \in \mathbb{R}^n$. We assume that it passes through m points $\{X_1, \dots, X_m\}$. The basic idea is to calibrate the approximator at the points one by one without affecting the approached points. If we can use a linear $n-1$ dimensional hypersurface to sperate the approached points and the new point, we can use the ReLU function to mask the approached ones and only adjust the approximator on the other side. Thus, we need to sort the m points into $X_{(1)}, \dots, X_{(m)}$ such that $X_{(i)}$ is outside the convex hull for $\{X_{(1)}, \dots, X_{(i-1)}\}$ for any i.

To sort the given points, directly find from the rest points the one that satisfies the condition is time consuming. In contrast, we start from $X_{(m)}$ and sort the points in a reversed order. We first construct a convex hull with all m points, and randomly pick one vertex of the convex hull. Then, we use it as $X_{(m)}$ and remove it from the points. The rest points can also form a convex hull. The facets facing $X_{(m)}$ can be used as a hypersurface to separate $X_{(m)}$ from other points, and we pick the one with the longest distance to $X_{(m)}$ as the separation hypersurface for $X_{(m)}$. Then, we random pick $X_{(m-1)}$ from the new convex hull and repeat the procedure until there are only $n+1$ points left. For a n dimensional space, the convex hull can only be formed when there are at least $n+1$ points. These points can satisfy our condition no matter what order they take. Thus, we remove them one by one and calculate the best hypersurface to separate the chosen one and the rest points.

The best hypersurface should be the one with the longest distance from the chosen point. Thus, if we select the chosen point as an origin, the normal vector of the hypersurface should fall in the linear subspace spanned by the rest points and the hypersurface will pass through all the rest points. Suppose the chosen point is X, the rest points are $\boldsymbol{X} = \{X_1, \dots, X_k\}$, and the normal vector of the hypersurface is N, then we have $N = \sum_i \alpha_i (X_i - X)$, which means N is in the subspace spanned by $X_i - X$, and $N(X_j - X) + b = 0$; it also means any rest point X_j is in the hypersurface, and $b \neq 0$. Combining the two equations, we have $\sum_i \alpha_i (X_i - X)(X_j - X) + b = 0$. In matrix form, it is $(\boldsymbol{X} - X)(\boldsymbol{X} - X)^{\boldsymbol{T}}\boldsymbol{\alpha} + b = 0$, where $\boldsymbol{\alpha} = \{\alpha_1, \dots, \alpha_k\}$. Since all points are constant values, we can initialize $b = 1$ and solve the linear equation to get α and then obtain N.

Now, we have sorted points, $X_{(1)}, \dots, X_{(m)}$, and the corresponding linear equations, $N_{(2)}X + b_{(2)} = 0, \dots, N_{(m)}X + b_{(m)} = 0$, to separate them from all the previous points. We can construct the approximator as follows

$$F_k(X) = f(X_{(1)}) + \sum_{i=2}^{k} \frac{f(X_{(i)}) - F_{i-1}(X_{(i)})}{N_{(i)}X_{(i)} + b_{(i)}} ReLU(N_{(i)}X + b_{(i)})$$

$F_m(X)$ will be our final approximator for this function.

After UA, we can get an initial approximator. Then, we will use the backpropagation algorithm to fine-tune the approximator to get a more precise one. For a specific function, the number of input variables is determined; thus, the only thing we can adjust is the number of nodes in the hidden layer. In general, the more nodes the hidden layer has (more cost), the higher

precise the approximator is. To focus on the relationship between granularity and cost, we stipulate a union error upper bound for each approximator. Under this error limit, we try our best to find an approximator with minimum cost.

In general, the error can be defined as the area between the approximator and the approximated function. The error for an approximator A of a function f on a specific domain D can be calculated as:

$$Error(A) = \frac{\int_{x \in D} |f(x) - A(x)| dx}{\int_{x \in D} dx}$$

Because each approximator can have very different output range and this measurement is unfair for large-value functions, we use the relative error, instead of the absolute error:

$$Error(A) = \frac{\int_{x \in D} |\frac{f(x) - A(x)}{f(x)}| dx}{\int_{x \in D} dx}$$

For simplicity, we use the sum instead of the integral to calculate the error. This error should be not greater than E_{max} (the error upper bound):

$$Error(A) = \frac{\sum_{i=0}^{\min(D)+step \times i \in D} |\frac{f(\min(D) + step \times i) - \mathrm{A}(\min(D) + step \times i)}{f(\min(D) + step \times i)}|}{(\max(D) - \min(D))/step} \leq E_{max}$$

This $step$ is the sampling step. For avoiding zero in the denominator, we eliminate [-0.01, 0.01] in the input domain.

8.2.6. FPSA Mapping

To map the EPG onto FPSA, we use the on-chip buffer and CLBs to synthesize the tier-one control flow graph and use PEs to perform the *basic execution primitives*. Moreover, FPSA is a processing-in-memory (PIM) architecture that collocates computation and memory, i.e., the PEs not only compute the *basic execution primitives* but also store the parameters of them. To efficiently support an EPG contains massive primitives, we can map primitives that share the same parameters onto the same PEs and schedule the execution of them in a time-division multiplexing manner. The scheduling logic will be combined into the tier-one control flow graph and synthesized into on-chip buffers and CLBs as well.

8.3. Analyses and Discussions

From the perspective of general-purpose computer instruction sets, there are two types of processors: Complex Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC). The former has a large set of complex and specialized instructions, e.g. a single instruction can load data from memory, complete the arithmetic operation and store the result back. The later only implements a relatively small set of simple and general-purpose instructions and leverages software to achieve more complicated functions; its advantage is that the processor implementations can be more compact and efficient.

FPSA and the proposed toolchain can be regarded as the embodiment of the RISC principle in the field of brain-inspired computing hardware: With our proposal, FPSA has been enhanced from supporting only ANN to being able to efficiently support SNNs and more.

How to expand it further is an interesting question. One possible way is to use low-precision memristor-crossbars in combination with a high-precision digital processor or digital circuits for other types of commonly-used accurate computations: The approximation scheme based on the *execution primitives* is a trade-off between resource consumption and performance; thus, using digital hardware for common operations instead of approximation may be more economical in some cases. Furthermore, such hybrid hardware can be used in cooperation with hybrid spike-based learning approaches, like locally unsupervised learning followed by globally supervised back-propagation [54]. Some work [55] has illustrated that some locally unsupervised learning approaches based on spiking can be efficiently implemented through memristive crossbars. Anyway, hybrid approaches are considered to be a very important area of brain-like computing technologies [61].

9. Experiments

We carry out the experiments on three kinds of applications to demonstrate the outcomes and advantages of our proposal. The first is an SNN/ANN hybrid neural network for bicycle driving and tracking[26], which is composed of CNN, SNN, MLP, and other neural network structures. The second is the Boids model for bird flock simulation[56]. It is a widely-accepted model for biological flock simulation, not in the form of neural networks. Our hierarchy can support these applications in different platforms.

The third one is QR decomposition, a common mathematical algorithm widely used in solving linear algebra problems, which is also not in the form of neural networks. This case is technically used to show the design space exploration based on the new optimization dimension introduced by *neuromorphic completeness*, i.e. the approximation granularity (of course, the first two experiments can also demonstrate this effect to varying degrees).

9.1. Bicycle Driving and Tracking Model

To demonstrate the applicability of the toolchain, we transform the different types of neural networks into equivalent networks for each of the three hardware platforms by the toolchain.

CNN: It is for image processing and object detection, which has three convolutional layers with kernel size: 3, output channel number: 6, 12, 12 respectively, two max-pooling layers, and two fully connected layers. The detailed structure and parameters of the CNN are showed in Extended Data Fig. 2-a. The CNN takes a 70 by 70 grayscale image from the camera as the input, and outputs the coordinates of the human and obstacles.

SNN: The SNN is responsible for the recognition of the voice signal from the microphone, and outputs the corresponding control commands. The SNN has a 510-256-7 fully connected architecture (Extended Data Fig. 2-b). Each neuron is a LIF model defined as the following equations:

$$x_i^{t+1,n} = \sum_{j=1}^{l(n-1)} w_{ij}^n o_j^{t+1,n-1}$$

$$u_i^{t+1,n} = u_i^{t,n}\left(1 - o_i^{t,n}\right)\tau + x_i^{t+1,n}$$

$$o_i^{t+1,n} = gateFunc\left(u_i^{t+1,n}\right)$$

$$gateFunc(x) = \begin{cases} 1 & x \geq thresh \\ 0 & x \leq thresh \end{cases}$$

The $x_i^{t,n}$ is the input of LIF model, which is calculated by the matrix product of weights, w^n, and the previous layer's output, $o^{t,n-1}$. $u_i^{t,n}$ is the membrane potential that should be updated at each time step. If $o_i^{t,n} = 1$, which means the LIF neuron generates a spike at this time, $u_i^{t+1,n}$ will be set to the input, $x_i^{t+1,n}$. If there isn't any output spike at time t, the membrane potential will be updated by the leaky rule, i.e. plus $u_i^{t,n}\tau$ where τ is the decay factor.

CANN: The CANN (continuous attractor neural network) is designed for object tracking[48]. It can be viewed as a one-layer fully connected RNN. This layer contains 20 by 24 neurons. The basic computation in each time step is showed in Extended Data Fig. 2-c. The CANN receives the images clipped by the initial human coordinates from the object-detection CNN, and outputs coordinates of the tracked target.

MLP: The MLP takes in the motion information from the sensors and the related state signals from NSM, outputs data about bike balancing. The MLP has a three-layered (30-256-32-1) architecture (Extended Data Fig. 2-d).

NSM: The NSM controls these four networks. It performs as a finite state machine with six states and nine transition conditions by a neural network structure. The transition conditions are the inputs of the NSM, which are from the CNN (obstacle, no person, person), the SNN (follow, speed, turn, straight) or the signals about internal states respectively. The state transition and decision making are achieved by a series of linear operations and LIF neurons with a structure showed in Extended Data

Fig. 2-e. The LIF model is the same as the one of the above-mentioned SNN. The state signals generated by the NSM will be synthesized with other networks' outputs to realize the bike motion control.

The overall relationship of these five networks is showed in Extended Data Fig. 2-f. The proposed toolchain will compile this ANN/SNN hybrid example to the EPGs composed of primitives specific to the three target platforms (i.e. GPGPU, Tianjic chip and FPSA) respectively before optimized mapping, while the POG of each network is same for these platforms.

9.1.1. GPU Results

As GPU is a general-purpose platform, the primitives in the EPG for GPU can be as rich as the operators in the POG. The compilation is just to directly call the corresponding kernels for each operator in POG and the transformation is precise.

Fig. 3-d shows the throughput of these five models on the GPU. Here we use the NVIDIA Tesla P100 GPU card as the test platform. The GPU provides 12GB memory and 3584 CUDA cores running at 1.33GHz. As mentioned before, we directly mapped the computation blocks to the frameworks and libraries that based on the CUDA interface; thus, the performance of each model is directly related to its inherent computational complexity and parallelism. For example, the CNN model, which involves the convolutional operations instead of simple matrix multiplications, achieves higher latency and less throughput. Although the NSM has limited computing operations, it has the most sequential steps, which means the overall parallelism is quite limited; thus, it has the highest latency and the least throughput.

9.1.2. Tianjic Results

Tianjic is not as flexible as general-purpose processors, but it is more flexible than the basic execution primitives; many of the unsupported operations can be achieved by its LUT operation. The area and throughput for the five networks are reported in Fig. 3-d. All the networks at the POG level are quantized and the approximation precision is the same with the quantized input. So, there is no error introduced.

9.1.3. FPSA Results

The FPSA only provides the basic execution primitives in hardware; thus it heavily relies on the universal approximation to achieve operators that are not directly supported. In this experiment, we set the error bound to zero. Namely, the transformations are all precisely equivalent, including operators achieved by universal approximators. Thus, the application behaviors in FPSA are the same as GPU and Tianjic.

The area and throughput of FPSA for the five networks are reported in Fig. 3-d, which also shows that we can extend the application scope of FPSA from DNNs to different kinds of brain-inspired models.

The throughput of CNN and CANN are smaller compared to other models because we reuse many PEs to execute different *execution primitives* that share the same parameter to save the hardware consumption. For the CNN model, it is quite intuitive because convolutional layers reuse their weights for different regions in the input feature map. For CANN, approaching some element-wise vector operations requires many PEs and we also reuse the corresponding approximators for different elements in the vector to reduce the consumption. The resources used to approximate different kinds of operations are presented in Extended Data Fig. 3. These models contain a lot of weighted-sum operations while there are also many other functions. The costs for approximating these functions are acceptable.

Moreover, the granularity of computation to be implemented with the universal approximator has a significant impact on resource consumption. Neither the most fine-grained nor the most coarse-grained is the optimal choice. Fig. 3-e shows the area consumption for different granularities. We start from the most fine-grained way (the rightmost case for each model) that approximates the basic arithmetic functions and uses them to compute the model. Then, we gradually increase the granularity before the hardware consumption goes exponential. We can see that the hardware consumption decreases with a larger granularity.

9.2. Boids Model

The Boids model is one of the widely-accepted mathematical models based on flocks[56]. As a typical case of artificial life

simulation, it can simulate a biological flock at the behavioral level. The Boids model was used to research the bird flocking when it was first proposed [57]. In this model, each bird is called a boid and follows threes rules:

1) Separation rule. A bird tries to keep a certain distance from nearby birds (that is, not too close).

2) Cohesion rule: A bird tries to fly toward the center of all other birds.

3) Alignment rule: A bird tries to maintain the same speed as the surrounding birds.

These three simple rules can make the whole artificial flock act as a natural flock. [58] proposed a formal definition of the Boids model. There are N boids in a Euclidean vector space: $V = R^d$ (in general, $d = 2, 3$). Each boid has an internal state $q \in Q$:

$$Q = \{q | q =< \boldsymbol{p}, \boldsymbol{v}, r, fov, m, v_M, f_M >\}$$

$\boldsymbol{p} \in V$, $\boldsymbol{v} \in V$ are its position and velocity respectively. $r =< r_s, r_a, r_c >$ represents the separation, the alignment, and the cohesion perception distances. Correspondingly, $fov =< fov_s, fov_a, fov_c >$ represents the fields of view of the three rules. m is its mass, which is often 1 and then omitted by the model. v_m is the maximal speed (the maximum change in position per simulation frame). f_m is the maximal available force (maximal change in velocity per simulation frame, i.e. the acceleration in physics). Fig. 4-a gives an example of the 2D boids.

The three rules can be implemented in a variety of ways, and our realization refers to an open-source solution named XBoids[59]. XBoids is also chosen by SpiNNaker to show the generality of its simulation platform[60]. For simplicity, the mass of all the birds is 1 and the acceleration limitation is achieved by multiplying the velocity of each rule by a coefficient (we denote the coefficients as $< \alpha_s, \alpha_a, \alpha_c >$). We adopt the 2D simulation, and the size of the simulation space (i.e. canvas size) is 500×500. Base on the original Boids description, the perception distance of the cohesion rule is the whole space. This will let a bird fly toward the center of all other birds. For the separation rule and the alignment rule, the perception distance is limited, and the corresponding field of view is the whole circle ($[-2\pi, 2\pi]$). The left bottom of the Fig. 4-a gives a visualized demonstration of the perception distance and the field of view in our realization. The perception distances for the alignment rule (r_a) and the separation rule (r_s) are both limited, and the former is larger than later. For simplicity, the corresponding field of view is the whole circle (fov_a, fov_s). The perception distance for cohesion rule (r_c) is unlimited, and the corresponding field of view (fov_c) is the whole simulation space. The specific parameters are listed as follows:

$$r =< r_s, r_a, r_c >=< 25, 50, \infty >$$

$$fov =< fov_s, fov_a, fov_c >=< [-2\pi, 2\pi], [-2\pi, 2\pi], [-2\pi, 2\pi] >$$

$$m = 1$$

$$< \alpha_s, \alpha_a, \alpha_c >=< \frac{1}{1500}, \frac{1}{250}, \frac{1}{100N} >$$

$$f_M = 2$$

In each simulation frame, the velocity and the position of each bird will be updated; the velocity updating is based on three rules. The pseudocode is:

Pseudocode: The velocity update based on three rules

// **Separation Rule**

$\boldsymbol{v_s} = 0$

for i = 0 to N-1

  for j = 0 to N-1 // i, j means the i-, j-th bird

    if $Distance(\boldsymbol{p}[i], \boldsymbol{p}[j]) < r_s$ and $i \neq j$

      $\boldsymbol{v_s}[i] = \boldsymbol{v_s}[i] - (\boldsymbol{p}[j] - \boldsymbol{p}[i])$

// **Alignment Rule**

$\boldsymbol{v_a} = 0$

for i = 0 to N-1

  for j = 0 to N-1

    if $Distance(\boldsymbol{p}[i], \boldsymbol{p}[j]) < r_a$ and $i \neq j$

      $\boldsymbol{v_a}[i] = \boldsymbol{v_a}[i] + \boldsymbol{v}[j]$

// **Cohesion Rule**

$\boldsymbol{v_c} = 0$

for i = 0 to N-1

  for j = 0 to N-1

    if $i \neq j$

      $\boldsymbol{v_c}[i] = \boldsymbol{v_c}[i] + \boldsymbol{p}[j]$

  $\boldsymbol{v_c}[i] = \frac{v_c[i]}{N-1}$

// Update velocity

$\boldsymbol{v} = \max(\boldsymbol{v} + \alpha_s \boldsymbol{v}_s + \alpha_a \boldsymbol{v}_a + \alpha_c \boldsymbol{v}_c, \boldsymbol{v}_m)$

The position of each bird will be updated based on the new velocity:

$$\boldsymbol{p} = \boldsymbol{p} + \boldsymbol{v}$$

The Boids model is a non-neural network algorithm, selected to show the universality of *neuromorphic completeness* and the corresponding hierarchy. It is also an interesting case to show the effect of the degree of approximation on the algorithm. We deploy the Boids model onto three hardware platforms: Tianjic, FPSA, and GPU. All of the platforms evaluate the Boids model with the population size N of 20, 50, and 100, respectively.

Supplementary Figure 8 shows the detailed computation of the Boids model. It has four parts, and each involves many tensor operations.

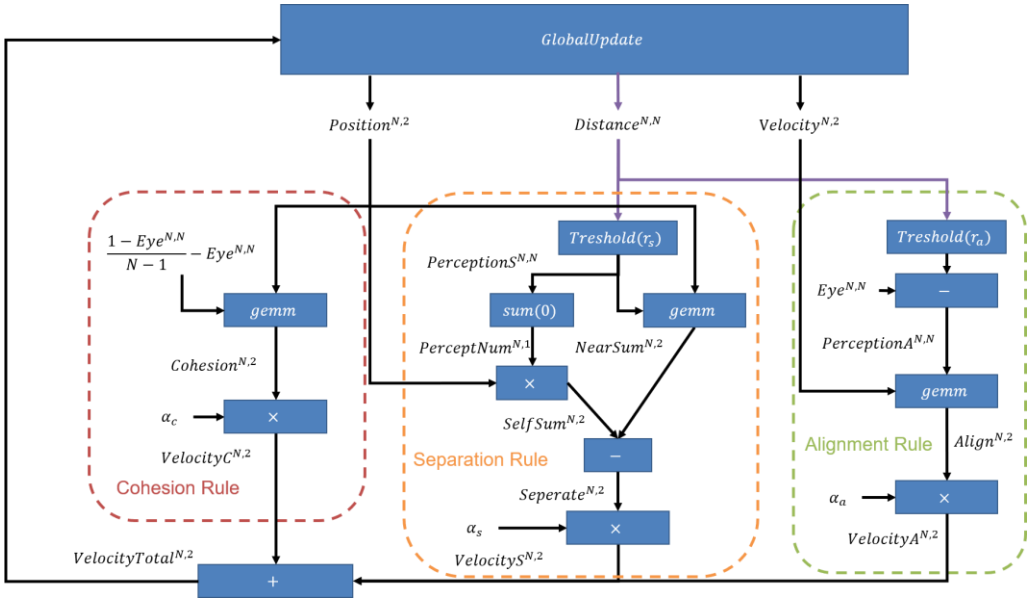
Cohesion Model

$$W^{N,N} = \frac{1 - Eye^{N,N}}{N - 1} - Eye^{N,N}$$

$$Cohesion^{N,2} = W^{N,N}.gemm(Position^{N,2})$$

$$VelocityC^{N,2} = \alpha_c \times Cohesion^{N,2}$$

Here Eye represents the unit matrix and $gemm$ represents the general matrix multiplication.



Supplementary Figure 8. Boid Model Details

Separation Model

$$Threshold(r_s, PerceptionS^{N,N}[i][j]) = 1\ if\ PerceptionS^{N,N}[i][j] < r_s\ else\ 0$$

$$PerceptNum^{N,1}[i] = \sum_{j=0}^{j<N} PerceptionS^{N,N}[i][j]$$

$$SelfNum^{N,2}[:][0] = PerceptNum^{N,1}.\times Position^{N,2}[:][0]$$

$$SelfNum^{N,2}[:][1] = PerceptNum^{N,1}.\times Position^{N,2}[:][1]$$

$$NearSum^{N,2} = PerceptionS^{N,N}.gemm(Position^{N,2})$$

$$Seperate^{N,2} = SelfNum^{N,2} - NearSum^{N,2}$$

$$VelocityS^{N,2} = \alpha_s \times Seperate^{N,2}$$

.× means element-wise multiplication.

Alignment Model

$$PerceptionA^{N,N} = Treshold(r_a, Distance^{N,N}) - Eye^{N,N}$$

$$Align^{N,2} = PerceptionA^{N,N}.gemm(Velocity^{N,2})$$

$$VelocityA^{N,2} = \alpha_a \times Align^{N,2}$$

Then, three components of velocity generated by these models will become one steering velocity as an input of the Global Update model.

$$VelocityTotal^{N,2} = VelocityC^{N,2} + VelocityS^{N,2} + VelocityA^{N,2}$$

Global Update Model

The Global Update model (Supplementary Figure 9) updates the velocity and position of each bird and calculates some auxiliary data for the next iteration. The detailed operations are:

$$VelocityNew^{N,2} = VelocityTotal^{N,2} + Velocity^{N,2}$$

$$Velocity\_2^{N,1}[i] = (VelocityNew^{N,2}[i][0])^2 + (VelocityNew^{N,2}[i][1])^2$$

$$Velocity\_2\_sqrt^{N,1}[i] = \sqrt{Velocity\_2^{N,1}[i]}$$

$$Velocity\_2\_sqrt\_1^{N,1}[i] = \frac{1}{Velocity\_2\_sqrt^{N,1}[i]}$$

$$VelocityRatio^{N,1} = v_m \times Velocity\_2\_sqrt\_1^{N,1}$$

$$Velocity^{N,2}[:][0] = VelocityRatio^{N,1}.\times Velocity^{N,2}[:][0]$$

$$Velocity^{N,2}[:][1] = VelocityRatio^{N,1}.\times Velocity^{N,2}[:][1]$$

$$Position^{N,2} = Velocity^{N,2} + Position^{N,2}$$

$$DiffX^{N,N} = diff(Position^{N,2}[:][0])$$

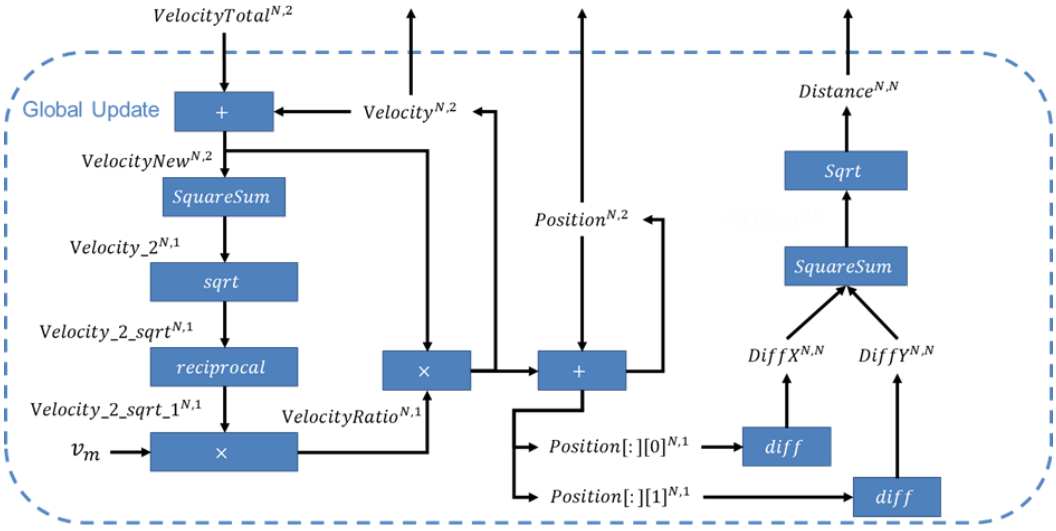
$$DiffY^{N,N} = diff(Position^{N,2}[:][1])$$

$$Distance^{N,N} = \sqrt{DiffX^{N,N}.\times DiffX^{N,N} + DiffY^{N,N}.\times DiffY^{N,N}}$$

$$x.repeat(N,1) - (x.repeat(N,1))^T$$

$diff(x)$ is defined as $x.repeat(N,1) - (x.repeat(N,1))^T$. $x.repeat(N,1)$ represents the operation of repeating N time of x along with dimension *1*. For example, for $x = [1,2]^T$, $x.repeat(2,1)$ represents:

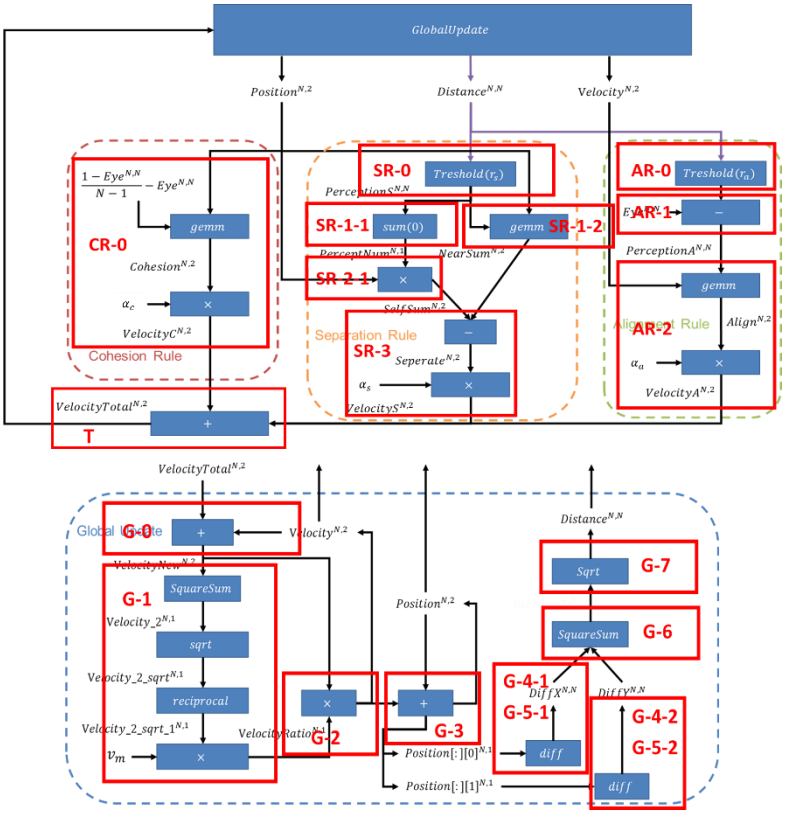
$$\begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix}$$



Supplementary Figure 9. Global Update model

9.2.1. Tianjic Results

For Tianjic, some non-linear operators, like *sqrt* and *reciprocal*, are not supported directly, which can be implemented with the look-up table (LUT). Other operations can be directly supported by Tianjic primitives or the combination of them. Supplementary Figure 10 shows how to map the EPG of Boids model onto Tianjic cores:



Supplementary Figure 10. The mapping of Boids model on Tianjic

In Supplementary Figure 10, each red rectangle represents a core group, which means that the operations in the red rectangle will be mapped on this group of cores. The number of cores in one group is determined by the resource consumption of the operation. Because the algorithm is fully *unrolled* on the Tianjic cores (it means that none of the cores are reused by different parts), each core in one group executes different parts of the same operations (same primitives) in parallel. For example, the core-group SR-1-2 executes the general matrix multiplication (gemm) operation. If the operation size is small enough, just one core can cover it. With the operation size becoming larger, we should divide the gemm operation and map it onto multiple cores. Because all the cores run in parallel, the latency of this core group doesn't change. Supplementary Table 1 shows the configuration of the whole mapping procedure. 20, 50, 100 in the 'Core Number' column are the population size.

Supplementary Table 1. Configuration of the Boids model on Tianjic

Module	**Operation**	**Output**	**Core Group**	**Dendrite Primitive**	**Soma Primitive**	**Core Number**		
						20	50	100
Cohesion	gemm	$Cohesion^{N,2}$	CR-0	VMM	LUT(αX)	1	1	1
	scale	$VeloctiyC^{N,2}$						
Seperation	Threshold	$PerceptionS^{N,N}$	SR-0	VB	LUT(0.1)	2	10	40
	sum	$PerceptNum^{N,1}$	SR-1-1	VVA	LUT(X)	2	10	50
	×	$SelfSum^{N,2}$	SR-2-1	VVM	LUT(X)	2	2	2
	gemm	$NearSum^{N,2}$	SR-1-2	VMM	LUT(-X)	7	50	200
	-	$Seperate^{N,2}$	SR-3	VVA	LUT(βX)	1	1	2
	scale	$VelocityS^{N,2}$						
Alignment	Threshold	$PerceptionT^{N,N}$	AR-0	VB	LUT(0.1)	2	10	39
	-	$PerceptionA^{N,N}$	AR-1	VVA	LUT(X)	4	20	79
	gemm	$Align^{N,2}$	AR-2	VMM	LUT(γX)	7	50	200
	scale	$VelocityA^{N,2}$						
	+	$VelocityTotal^{N,2}$	T	VVA	LUT(X)	1	2	3
Global Update	+	$VelocityNew^{N,2}$	G-0	VVA	LUT(X^2)	1	1	2
	square	$VelocitySquare^{N,2}$						

	sum	$VelocitySum^{N,1}$	G-1	VVA	LUT(vm/sqrt(X))	1	1	1
	sqrt	$VelocitySqrt^{N,1}$						
	1/x	$VelocityReci^{N,1}$						
	scale	$VelocityRatio^{N,1}$						
	×	$Velocity^{N,2}$	G-2	VVM	LUT(X)	2	2	2
	+	$Position^{N,2}$	G-3	VB	LIF (Vth = 0, keep V, fire V)	1	1	2
	diff	$DiffX^{N,N}$	G-4-1	VMM	LUT(X)	4	20	79
	square	$DiffXSquare^{N,N}$	G-5-1	VVA	LUT(X^2)	4	20	79
	diff	$DiffY^{N,N}$	G-4-2	VMM	LUT(X)	4	20	79
	square	$DiffYSquare^{N,N}$	G-5-2	VVA	LUT(X^2)	4	20	79
	+	$DiffSum^{N,N}$	G-6	VVA	LUT(sqrt(X)	2	10	40
	sqrt	$Position^{N,2}$						

Fig. 4-b shows the throughput and area-consumption on Tianjic.

In the Tianjic design, the algorithm has been fully unrolled onto the cores. So, the longest path from the input to the output determines the throughput of the execution. The increase in the population size doesn't affect the throughput of one core group. The problem size is not so large so that the calculation time can cover all the communication time. So, the overall throughput is the same for all the population sizes of 20, 50, and 100.

9.2.2. FPSA Results

For FPSA, the non-linear operations are achieved by the MLPs that are generated by the Universal Approximation Method (Supplementary Information 8.2.5). We use the universal approximator at the granularity of each operation (i.e. the most fine-grained), and we only merge some constant linear scaling and the element-wise operations in the Global Update module.

The main configurations of the universal approximators are listed in Supplementary Table 2. Since most operations are tensor operations and the universal approximator is designed for scalar operations, we duplicate them to process the whole tensor operations in parallel. For example, a universal approximator with n inputs and m points to pass forms an MLP of structure $n \to m-1 \to 1$. To process such computations in a tensor mode with the input size of k, we need k duplications of the MLP. A processing element (PE) in FPSA can process computation of size 256×256. We use a set of PEs to achieve the k duplications of the $n \to m-1$ part and another set of PEs for the k duplications of the $m-1 \to 1$ part. Since n is usually less than $m-1$, the number of duplications that one PE can map is $\lfloor 256/(m-1) \rfloor$. Thus, the total number of PEs required for the operation is $\lceil 2k/\lfloor 256/(m-1) \rfloor \rceil$.

Supplementary Table 2. Configuration of the Boids model on FPSA

Function	Input shape	Points passed	Number of duplications
Threshold(x)	[N, N]	3	N^2
Multiply in Gemm(x,w) w is Boolean	[N, 2], [N, N]	3	$2N^2$
Multiply(x, y) x is sum of N Boolean numbers	[N, 1], [N, 1]	N + 2	N
Square(x)	[N, 1]	33	N
ElementWise(x)	[N, 1]	33	N
Multiply(x, y)	[N, 1], [N, 1]	257	N
Square(x)	[N, N]	33	N^2
Sqrt(x)	[N, N]	33	N^2

Fig. 4-b shows the throughput and area-consumption on FPSA.

The hardware consumption for each part of the Boids model is listed in Supplementary Table 3.

Supplementary Table 3. Area consumption of the Boid model in FPSA (mm^2)

N (population size)	20	50	100
Cohesion	0.0442	0.0442	0.0442
Separation	0.7735	3.7791	16.5308
Alignment	0.5304	2.652	10.9616
Global Update	9.0831	48.2001	181.6178

From Supplementary Table 3, we can see that the Global Update modular takes up most of the area. The reason is, the square and the square root operations of it require N^2 universal approximators.

9.2.3. GPU Results

Fig. 4-b shows the latency and throughput of the Boids model with a population size of 20, 50, and 100 respectively. The GPU is same as the one of the bicycle task experiment. The experiment has been repeated six times for the average. Because the problem size is not large enough for the GPU, the performance is almost the same at different scales.

We also study the effect of the degree of approximation on the Boids model. We construct three $sqrt$ approximators with the relative errors of 0.1%, 1%, 10% respectively to replace the original and precise $sqrt$. Results of these models are obtained and then compared with the precise calculation.

Fig. 4-c shows the results of the Boids model with each of these $sqrt$ approximators respectively (every small, blue triangle in the figure represents a bird). The black triangles are the precise calculation result for comparison. All pictures of the bird flock are captured from the 500th frame. We can see that the greater the error, the bigger the gap with the behaviors of the exact calculation. Because of the chaotic feature of this model, the small error has affected the behaviors of the whole system while attributes of flock movement are still maintained (especially when the approximation error is limited).

This example visually demonstrates the relationship between approximation errors and program performance, which also means that, for some applications with fault tolerance, the optimization space introduced by *neuromorphic completeness* is especially worth considering.

9.3. QR Decomposition by Given Rotation

QR decomposition is a procedure that decomposes a matric A into an orthogonal matrix Q and an upper triangular matrix R. QR decomposition has a lot of numerical solutions. Here, we use Givens rotation[1] to do the QR decomposition. Givens rotation can be used to zero out an individual element in a matrix. For example, a Givens matrix is:

[1] https://www.sciencedirect.com/topics/mathematics/givens-rotation

$$G_{ij}(\theta) = \begin{bmatrix} 1 & & & & & & & & & & \\ & \ddots & & & & & & & & & \\ & & 1 & & & & & & & & \\ & & & i,i:\cos\theta & & & & i,j:\sin\theta & & & \\ & & & & 1 & & & & & & \\ & & & & & \ddots & & & & & \\ & & & & & & 1 & & & & \\ & & & j,i:-\sin\theta & & & & j,j:\cos\theta & & & \\ & & & & & & & & 1 & & \\ & & & & & & & & & \ddots & \\ & & & & & & & & & & 1 \end{bmatrix}$$

Let $x = [x_1, x_2, \dots, x_n]^T$,$y = G_{ij}(\theta)x = [y_1, y_2, \dots, y_n]^T$. Then:

$$\begin{cases} y_i = cos\theta x_i + sin\theta x_j \\ y_i = -sin\theta x_i + cos\theta x_j \\ y_k = x_k \qquad (k \neq i, j) \end{cases}$$

When $x_i^2 + x_j^2 \neq 0$, let:

$$cos\theta = \frac{x_i}{\sqrt{x_i^2 + x_j^2}}, \qquad sin\theta = \frac{x_j}{\sqrt{x_i^2 + x_j^2}}$$

We will have:

$$\begin{cases} y_i = \sqrt{x_i^2 + x_j^2} > 0 \\ y_j = 0 \end{cases}$$

That will annul the x_j. Using the Givens rotation one by one, we can upper-triangularize one matrix (get the matrix R):

$$\begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix} \xrightarrow{G_1} \begin{bmatrix} m & m & m \\ 0 & m & m \\ \times & \times & \times \end{bmatrix} \xrightarrow{G_2} \begin{bmatrix} m & m & m \\ 0 & \times & \times \\ 0 & m & m \end{bmatrix} \xrightarrow{G_3} \begin{bmatrix} m & m & m \\ 0 & \times & \times \\ 0 & 0 & m \end{bmatrix}$$

The orthogonal matrix $Q^T = G_1 G_2 G_3$.

The pseudocode of this procedure is:

Pseudocode: QR Decomposition by Givens Rotation

Input: A square matrix A
Output: An orthogonal matrix Q and an upper triangular matrix R, which let A=QR
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]
        $x_i^2 = x_i \times x_i$
        $x_j^2 = x_j \times x_j$ } ①

$x_{ij}^2 = x_i^2 + x_j^2$ ②

$x_{ij}^2\_sqrt = \sqrt{x_{ij}^2}$ ③

$x_{ij}^2\_sqrt\_1 = \frac{1}{x_{ij}^2\_sqrt}$ ④

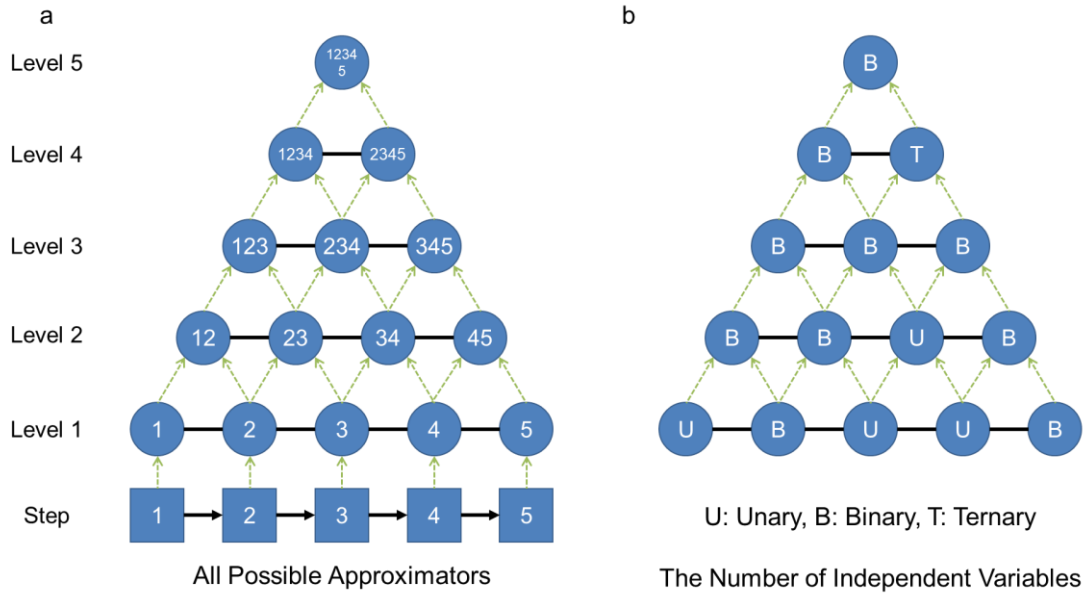
$cos\_ = x_i \times x_{ij}^2\_sqrt\_1$
$sin\_ = x_j \times x_{ij}^2\_sqrt\_1$ ⑤

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

There are some non-linear operations in Step 1 to 5, each of which is called a **basic operation**. The non-linear operations can't be directly supported by some brain-inspired hardware, which makes the task hard to be deployed. However, by introducing *neuromorphic completeness*, we can approximate all operations by *dot-product* and *ReLU* (the proposed basic execution primitives). In this experiment, we use *dot-product* and *ReLU* to construct an MLP to replace the non-linear operations and make the task meet the requirements of *neuromorphic completeness*. This MLP is called an **approximator**. During this procedure, we will choose various combinations of approximators with different granularities to explore the design space introduced by *neuromorphic completeness*.

To be specific, Step 1 to 5 are successive basic operations (most of them are non-linear operations, except for Step 2 which is a linear operation for continuity). Each basic operation can be achieved by an approximator, which is the most fine-grained case. Several successive operations can be viewed as a compositive operation and then can be approached by one approximator with coarser granularities. The most coarse-grained approximator is the one that covers all of the steps. Supplementary Figure 11.a lists all 15 possible approximators.



Supplementary Figure 11. All possible Universal approximators for QR

Each circle node represents a unique approximator. The value on the node represents the basic operations it covers. The value

is also used to identify one approximator, e.g., the approximator that covers basic operation 2 and 3 is denoted as A_{23}. Each approximator in level i covers i successive basic operations and the coverage scope of an approximator in level $i+1$ is the union of the scope of two adjacent approximators in level i. The function approached by one approximator can be unary, binary, or ternary, showed in Supplementary Figure 11.b. For example, A_{123} approximates binary function $f(x,y) = \sqrt{x^2+y^2}$. We call the representation in Supplementary Figure 11.a Fusion Space Network (FSN), which is used to enumerate all the approximators and all the approximation strategies. It can also be used to express the heuristic search algorithm. The FSN will be constructed based on any data-flow graph (a POG can be regarded as an extended data-flow graph).

Several approximators can form a complete approximation strategy of the task. For example, Fig. 4-e shows a strategy that is composed of A_{123} and A_{45}, denoted as $S\{123,45\}$. In this figure, the red triangle (**cover triangle**) represents the cover scope and the orange part represents the new dataflow after approximation. A valid strategy should ensure that the triangles can cover all the basic operations and there is no overlap. It is easy to calculate the number of all valid strategies, $2^{5-1} = 16$.

To study the trade-off between granularity and cost, we must define them first. We use $G(\cdot)$ and $C(\cdot)$ to represent the granularity and the cost of an approximator or a strategy respectively. We define $G(\cdot)$ of an approximator as the number of nodes contained by the cover triangular. For example, $G(A_{123}) = 6$ and $G(A_{45}) = 3$. For a given strategy, the granularity is the sum of the granularity of all the approximators, like $G(\mathrm{S}\{123,45\}) = G(A_{123}) + G(A_{45}) = 9$. This definition accords with our intuition and makes different strategies have different values. For example, the granularity of strategy $\mathrm{S}\{123,45\}$ is greater than strategy $\mathrm{S}\{12,3,45\}$, $G(\mathrm{S}\{123,45\}) = 9 > G(\mathrm{S}\{12,3,45\}) = G(A_{12}) + G(A_3) + G(A_{45}) = 3+1+3 = 7$.

An approximator can be viewed as a three-layer MLP with ReLU activation function. We assume that the number of nodes in the input, hidden, and output layer is m, n and 1 respectively (our approximated functions are all one-to-one or multi-to-one functions). We use the MAC (multiply-and-accumulate) number to represent the cost (t means this approximator is used t times during the computation):

$$C(A) = (m \times n + n) \times t$$

So far, we summarize the experiment as follows: We explore the trade-off between granularity G and cost C given a specific task (QR decomposition) to find a strategy S with an acceptable cost C, on the premise of satisfying some error limitation.

For each node in FSN, we use binary search to find the approximator with minimal cost. The pseudocode of this procedure is:

Pseudocode: Find approximator with the minimal cost given E_{max}

Input: Approximated function f, some hyper-parameters p (including E_{max}), min_hidden_number, max_hidden_number
Output: An approximator A

```
left, right = min_hidden_number, max_hidden_number
while left <= right
    mid = (left+right)/2
    approximator = ua(mid, f, p)
    approximator = fine_tune_with_back_propogation(approximator, f, p)
    if Error(approximator) < E_max
        best_approximator = approximator
        right = mid - 1
    else
        left = mid + 1
return best_approximator
```

The corresponding hyper-parameters are:

$\boldsymbol{E_{max}}$**:** 0.03 // We have evaluated that, under this limitation, the final result of QR decomposition will have an acceptable

deviation in most cases. Specifically, we have tested the minimal cost strategy, $S\{12,3,45\}$. The mean square error of the Q matrix is less than 0.1, and the R matrix is less than 0.5. It is acceptable considering the input is 10 random 4×4 matrixes with values ranging from -8 to 8 (i.e. $[-8,8]$).

first function input domain: (-8, -8) to (8, 8), the domains of other functions are deduced accordingly.

// BP relative hyper-parameters

batch number: 100000

max iteration: 200000 (50000 for unary function)

learning rate: 0.001

learning method: Adam

stage2 learning rate: 0.0001

state2 error: 0.1 // When the error is smaller than this value, BP will start the next stage of training (with a smaller learning rate).

function sample step: 0.2 (0.5 for the ternary function)

To show the optimization possibility brought by *neuromorphic completeness* intuitively, we also give a simple heuristic iterative algorithm in the granularity-cost space. This algorithm is based on two facts:

1. There may be too many valid strategies and it is impossible to transverse every strategy.
2. Constructing an approximator is time-consuming. It would be better if we could search for fewer approximators.

This algorithm only explores a few strategies and evaluates part of them and tries to find a good (but not optimal) strategy.

The algorithm starts with the most fine-grained strategy: $S\{1,2,3,4,5\}$. The selected approximators in the current strategy can form some trees: Two adjacent and selected approximators are the tree's leaves, and the lowest approximator in the FSN that covers them is the tree's root. For example, A_1 and A_2 with A_{12} as the root can form $Tree\{A_{12}: A_1, A_2\}$. So, $S\{1,2,3,4,5\}$ can form 4 trees: $Tree\{A_{12}: A_1, A_2\}$, $Tree\{A_{23}: A_2, A_3\}$, $Tree\{A_{34}: A_3, A_4\}$, and $Tree\{A_{45}: A_4, A_5\}$. We evaluate the cost of root approximators in trees and calculate the saved cost of each tree. The saved cost means if we replace the leaf approximators with the corresponding root approximator, how much would be saved. If the saved cost is positive, we can reduce the total cost by using the root approximator instead. So, we insert all the trees whose saved cost is positive into a priority queue; this queue is arranged in descending order of this cost.

Extended Data Fig. 4 shows the algorithm procedure on QR decomposition. Each step of Extended Data Fig. 4 has two parts: Approximators on FSN that have been evaluated and the current priority queue. The value on every circle node means the cost of that approximator. The red triangular represents the current strategy. In each step, the first tree in the queue is selected (green rectangle) and the leaves are replaced by the root approximator.

For example, from Step 1 to Step 2, A_4 and A_5 are replaced by A_{45} and form a new strategy: $S\{1,2,3,45\}$. In Step 2, a new approximator, A_{345}, is evaluated and a new tree, $Tree\{A_{345}: A_3, A_{45}\}$, is added into the queue. Step by step, we get the final strategy $S\{12,345\}$ (Step 4). In the last step, $Tree\{A_{12345}: A_{12}, A_{345}\}$ can't bring any positive saved cost and the priority queue is empty. The algorithm ends. The dash circles in Extended Data Fig. 4 represent the approximators that don't need to be constructed and evaluated. If the problem scale is bigger, we can evaluate a smaller percentage of approximators and save more time. The whole procedure only explores 4 (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.

More complicated algorithms can be constructed to find better strategies. Here, we just give an example to show the new optimization space and the decision-making choices that *neuromorphic completeness* brings to us.

This experiment finds the minimal-cost approximator that satisfies the condition E_{max} for each of all 15 nodes in FSN, and constructs 16 strategies by these approximators. The costs of all the approximators and strategies are calculated.

Fig. 4-e, f, g are the experiment results. Fig. 4-e shows the FSN of QR decomposition and Fig. 4-f shows the cost of each approximator in Fig. 4-e. The exact values are listed in Extended Data Table 1. Fig. 4-g shows the cost of each valid strategy and the values are in Extended Data Table 2. The green trajectory in Fig. 4-g is the heuristic search procedure.

Note that because there are some random steps in the UA and back-propagation; thus we do this experiment many times for the average.

The experiment can help understand the property of the universal approximators better. Here are some properties we draw from the experiment:

- For a workflow with n basic operations, the number of approximators is about $O(n^2)$, while the number of strategies can be $O(2^n)$. Thus, sometimes it is impossible to transverse every valid strategy.
- In general, the cost of a $n+1$-ary approximator is larger than a n-ary approximator. In theory, the cost increases exponentially with the number of input variables (in our experiments, because of the feature of the function, the cost of the only ternary approximator isn't very high, $C(A_{2345}) = 246$).
- The cost of an approximator is highly dependent on the approximated function, the error metric, and the approximator construction method. Generally, the smoother and more linear the function is, the easier to construct a low-cost approximator.
- Fig. 4-f shows that two fine-grained approximators can be replaced by a more coarse-grained approximator to reduce the cost. For example, $C(A_{45}) = 168 < C(A_4) + C(A_5) = 274$. But a coarse-to-some-degree-grained approximator may dramatically increase the cost, e.g., $C(A_{1234}) = 573 > C(A_{123}) + C(A_{234}) = 36$.
- Fig. 4-g shows that neither the most fine- nor the most coarse-grained strategy has the minimum cost. The most fine-grained strategy, $S\{1,2,3,4,5\}$ has a cost 340, and the most coarse-grained strategy, $S\{12345\}$, has a cost 660. In the case of QR decomposition, the optimal strategy is $S\{12,345\}$ with a cost of 171. The cost distribution of all of the strategies shows that the *neuromorphic completeness* brings a vast exploration space to the trade-off between cost and granularity.
- This is a theoretical experiment. In practice, the deployment should consider specific hardware implementation, like the primitive set and the cost for every primitive.

10. Discussion

This paper is focused on the perspective of the brain-inspired computing system. Here we discuss the relationship and implications of our proposal for other related aspects, and try to give an overall research framework, which would be conducive to the collaboration among researchers in different fields/disciplines.

10.1. Neural Networks and Turing Machine

It should be pointed out that Turing complete usually refers to that programming languages, or hardware (which is usually embodied as instruction set) can realize all the functions of the Turing machine, while the Turing machine itself is one kind of Turing complete system. Different Turing complete systems may vary widely, as well as their description schemas (which can often be represented by corresponding programming languages), but they can be equivalent in term of computability.

On the other hand, there have been some studies[62][63] [64][65] focusing on the relationship between the brain (including various neural networks) and the Turing machine, and a few proposed corresponding hyper-computation models[65][66] beyond the Turing machine. However, whether Turing machine works similarly as brain or not, or whether Turing machine is suitable for representing brain-inspired algorithms, it is not correlated with whether brain is Turing-computable. Whether the brain is some form of Turing machine (or its equivalent model) is a long-term open question. But in practical, almost every algorithm we can come up with is Turing-computable and the success of DNN also partially showed that based on Turing machine, we can also develop intelligent algorithms that work quite different from Turing machine.

Thus, regardless of the answer to the above questions, our hierarch is widely applicable as the proposed software abstraction, POG, is *Turing complete*.

10.2. **Codesign**

Codesign is considered an important means to overcome the challenges faced by brain-inspired computing[61], which is often reflected in two aspects:

One is to choose the appropriate algorithm to play (avoid) the characteristics (defects) of hardware. For example, algorithmic resilience can be used to counter hardware vulnerability, thereby achieving the optimal trade-off between energy efficiency and accuracy. This design principle has been largely reflected in the notion of *neuromorphic completeness*: As above-mentioned, the realization of a *neuromorphic complete* system is to find a balance between resource consumption and performance through various approximate granularities.

Second is to design dedicated hardware elements/primitives for important algorithms or operations, that is, to expose more functionalities of hardware to the programming level for utilization, in order to improve efficiency. The proposed hierarchy decouples programming languages and hardware, and it also benefits the codesign procedure. Owing to the *composability* of POG, we can abstract a complex operation supported by the hardware into a computation primitive of EPG and further wrap it as an *operator* of POG to replace the original sub-graph that achieves the equivalent function; meanwhile the properties of both types of POG remain unchanged. Moreover, the compilation process (the step of *Template Matching*) has fully considered this application scenario, too. Thus, it is possible to make full use of new types of hardware without affecting the up-level models, which reduces the corresponding programming.

For example, at present, there are a large number of material/device studies[67][68][69][70][71][72], aiming at the efficient implementation of "artificial" neural processing units, e.g. neurons, synapses, which can form the various underlying layers of ANA hardware instances. Usually, these neural processing units can support some complex computations directly, like the STDP (spike-timing-dependent plasticity) learning rule mimicked by memristive synapses. According to this rule[62], the conductance (i.e. weight) change of the memristive synapse depends on the relative timing of the electrical pulses delivered from the presynaptic neuron and the postsynaptic neuron. Usually we need to design a POG to implement the STDP rule; owing to the *composability*, we can use the so-called STDP *operator* instead, while ensuring little impact on the rest of the program.

Basically, our idea of codesign is based on the premise of ensuring *neuromorphic completeness*. Drawing on the history of traditional computers, we believe that decoupling software and hardware would lead to better programming flexibility and development productivity, while codesign is conducive to the performance of hardware. Further, the detailed comparison between this strategy and the one based entirely on codesign (e.g. the strategy adopted by [73]) is one of the future jobs.

10.3. **Vision of AGI**

Despite the superior performance in solving sub-problems in specialized domains with abundant data, solving complex dynamic problems with uncertainty or incomplete information associated with many systems remains a huge challenge for the current narrow AI. Therefore, there is an increasing trend to develop artificial general intelligence (AGI) to further enhance the AI capability.

So far, people have adopted different technical routes to develop AGI. Generally speaking, there are two main directions: computer-based and neuroscience-based. Both have advantages and disadvantages and are incompatible. It is well accepted that one highly promising solution for developing AGI is to combine these two approaches, which may produce a synergy between the two fields and mitigate their weaknesses, potentially bringing new breakthroughs. However, due to the radical differences in formulas and coding schemes, they have to rely on distinct and incompatible computing models and algorithms, platforms, software and systems, thereby impeding the combination.

Brain inspired computing is no doubt a better solution to integrate both approaches, and we believe that more neuromorphic research related to AGI will appear. Recently a cross-paradigm platform Tianjic chip[26] is reported to support a wide range

of both spiking and non-spiking neural network models from neuroscience (e.g. SNNs) and machine learning (e.g. ANNs) domains, which is expected to stimulate AGI development by paving the way to more generalized hardware platforms.

Here we report on the *neuromorphic completeness* and a general system hierarchy for brain-inspired computing: *Neuromorphic completeness* has a wider coverage by connecting the universal approximation theorem with the Turing completeness and the hierarchy has uniform software/hardware abstraction models based on the *neuromorphic completeness* and/or the Turing completeness. They can facilitate the realization of such a highly flexible and integrated computing platform. With such integrated platform we can speed up the integration of both computer-science-oriented and neuroscience-oriented approaches and the iterative evolution of AGI development, and can also be useful in many real-world applications. To the best of our knowledge no such a kind of platform has been reported.

10.4. **Relationship with Related Research**

For developers and researchers of the fundamental hardware and software systems of brain-inspired computing, the proposed hierarchy including the software abstraction (POG), hardware abstraction (EPG and ANA), and compilation, provides an overall design that is conducive to the realization of the three properties, i.e. language portability, hardware completeness and compilation feasibility, which has been highlighted by the above toolchain implementation and the application examples.

Besides, the proposal has the flexibility and composability to support rich application development modes and algorithm-hardware codesign (across multiple levels from the algorithm development to the hardware underlying). The clear and flexible interfaces between the various research aspects would benefit researchers, including but not limited to material scientists, device physicists, circuits engineers, computer scientists, neuroscientists, to better work together in this very multidisciplinary field.

Specifically, we decouple the software (algorithms/models/software frameworks) and the hardware. It provides greater flexibility to programming paradigm designer, because they do not have to consider the limitation and variety of hardware and may deploy the programming paradigm to different hardware substrates with less modifications. Because the POG is Turing complete, this means that, theoretically the programming methods of neural network models/algorithms that it can support are not limited. Moreover, POG is flexible and powerful in the embodiment of the main characteristics of neural networks (e.g., parallelism, event-driven and collocated memory and computation elements). We believe that this would be beneficial to develop more complicated and larger models and may help the development of new programming languages for non-expert users.

Finally, we should point out that, compared with other related studies that try to bridge the various brain-inspired software and hardware through domain specific languages[74] or development frameworks[75][76][77][78], our work (or, more specifically, the compilation-related part) is different from the design philosophy of the existing work.

(1) Our compilation layer has a solid theoretical basis to ensure the feasibility, that is, it is an equivalent transformation under the premise of the guarantee of completeness. Specifically, we propose a set of widely applicable basic execution primitives based on *neuromorphic completeness* to ensure the equivalent conversion from any Turing complete software to an equivalent on *neuromorphic complete* hardware. For existing work, it is infeasible to ensure the conversion of any Turing complete algorithm to non-general purpose computing system (it may be not Turing complete but *neuromorphic complete*) system.

(2) This compilation technique introduces a new optimization dimension, namely the tradeoff between the approximate granularity of the target function and the resource overhead. Thanks to the wider coverage of *neuromorphic completeness* and the adaptive and flexible hardware abstraction interface (EPG), our compilation can adjust the approximate granularity of any target function (it may not even be in the form of neural networks) for optimization. This is fully demonstrated in the experiments, and existing studies cannot do that.

(3) It expands the scope of applications supported by neuromorphic hardware. In essence, *neuromorphic completeness* relaxes the requirements on the equivalency of program conversion and execution, thus expanding the application range. Our toolchain experiments have shown that general applications (non-neural network forms), such as Boids model for

bird flock simulation[79], can be converted into an equivalent composed of the basic execution primitives and/or other primitives and run on neuromorphic hardware, e.g. Tianjic and FPSA.

Anyway, the proposed novel hierarchy is natively designed for the integration of the *neuromorphic completeness* (to assure compatibility of different systems and decouple hardware and software) and Turing completeness (to provide enough compatibility of existing systems).

But on the other hand, our hierarchy design is a compatible and extensible framework that tries to help bring the existing work from different fields (including software frameworks, hardware chips and even compilers) together, instead of replacing them. For example, the POG is a well-designed form to define the computation of brain-inspired computation, and then compatible with many existing neuromorphic development language/frameworks [74][75][76][77][78]. Our hierarchy also works as a compatibility assurance between different software/hardware designs.

10.5. Range of Applications

Currently, the application subject of the brain-inspired computing systems is a variety of neural networks and a large number of tasks are related to AI. Here neural network refers to a computational paradigm rather than concrete AI applications. The computation properties include: massively parallelism, collocated processing and memory, relatively simple processing elements that communicate with each other, event-driven, sometimes approximation, etc. In some neuroscience-oriented fields, they might be called the computation achieved in the form of neural circuits.

On the other side, now some studies[77][78][80] are exploring the applications to general algorithms other than AI, like graph analysis, constraint satisfaction problem, scientific simulations, etc.. These applications could be divided into two classes roughly. One still uses neural networks to handle non-AI applications representations and executes them in neuromorphic hardware. Another class is the “non-neural network algorithms”, which adopts some features of neuromorphic hardware but has no conventional neural networks in the processing flow. For example, the Boids model for bird flock simulation can be deployed on the SpiNNaker[79]. It has features like massive parallelism, collocated processing and memory etc., which are consistent with some features of neuromorphic hardware.

Our proposal can support these two classes of non-neural network applications well. For the first class, because its computing paradigm is still neural network, it can be certainly supported. Further, because POG is Turing complete and is extended from the dataflow model, it can represent essentially any program, including the second class. In addition, the second has features like massive parallelism, collocated processing and memory, which is suitable to POG, too: Our experiments include non-AI and non-neural network applications, supported by the toolchain.

On the whole, Turing completeness enables our hierarchy to accommodate as many applications as possible in the rapid development of software, and *neuromorphic completeness* promises that software can be deployed on any compatible hardware. Having compatibility with ANN or other non-neural network applications demonstrates the strength of our design, as these applications are not necessarily supported when the system is designed.

10.6. Neuromorphic Completeness

In general, the concept of complete is a measurement of computation capability of a computing system (e.g. a processor or a programming paradigm). For example, Turing completeness reflects the capability of traditional computer and high-level programming languages; it helps to determine whether a processor or a programming language is general enough to compute all Turing-computable functions.

Similarly, *neuromorphic completeness* helps to determine whether a processor or a programming language is general enough to support any neuromorphic applications:

1) *Neuromorphic completeness* is a property of hardware or programming paradigm. It is not a criterion for evaluating a specific application.

2) *Neuromorphic completeness* determines if the hardware or programming paradigm is general-purpose. A dedicated neuromorphic hardware for a certain scope of neuromorphic applications may not be *neuromorphic complete*.

3) The scope of applications supported by *neuromorphic completeness* is all Turing-computable functions. But functions in that scope may not be neuromorphic.

4) Turing completeness requires the hardware or programming paradigm to achieve functions by a computing procedure. *Neuromorphic completeness* relaxes this requirement: A function can be achieved using universal approximation or other methods.

5) Achieving *neuromorphic completeness* does not require that the concrete implementation should be neuromorphic.

6) For a detailed neuromorphic application, the neuromorphic completeness ensures that it is executable for any neuromorphic complete system regardless of whether this application is considered when designing the system (i.e. software and hardware decoupling). For neuromorphic software paradigms or neuromorphic hardware implementations, neuromorphic completeness can determine whether they are general-purpose.

Finally, it is necessary to note that, from a concrete implementation perspective, universal approximation is not the only way to achieve approximation; other methods can be used (e.g. the Tianjic-based experiment uses the look-up table to achieve approximation).

As brain-inspired computing is a rapidly developing interdisciplinary field with new models, algorithms and hardware emerging constantly, the definition of neuromorphic completeness do not involve the requirement of the detailed implementations. Therefore, in the future, researchers can further integrate more specific properties to the computing systems to confine the specific implementation of software/hardware if necessary. For example, researchers may integrate more neural basis.

[1] Turing A M. On computable numbers, with an application to the Entscheidungs problem. Proceedings of the London Mathematical Society, 1936, S2-42(1): 230-265.

[2] Hemmerling A. Systeme von Turing-Automaten und Zellularr¨aume auf rahmbaren pseudomustermengen. Elektronische Informationsverarbeitung und Kybernetik, 1979, 15(1/2): 47-72. (in German)

[3] Wiedermann J. Parallel Turing machines. Technical Report RUU-CS-84-11, Department of Computer Science, University of Utrecht, The Netherlands, 1984.

[4] Ito T. Synchronized alternation and parallelism for three-dimensional automata [Ph.D. Thesis]. University of Miyazaki, 2008.

[5] Okinaka K, Inoue K, Ito A. A note on hardware-bounded parallel Turing machines. In Proc. the 2nd International Conference on Information, December 2002, pp.90-100.

[6] Ito T, Sakamoto M, Taniue A, Matsukawa T, Uchida Y, Furutani H, Kono M. Parallel Turing machines on fourdimensional input tapes. Artificial Life and Robotics, 2010, 15(2): 212-215.

[7] von Neumann J. First draft of a report on the EDVAC. IEEE Annals of the History of Computing, 1993, 15(4): 27-75.

[8] Hornik, Kurt. "Approximation capabilities of multilayer feedforward networks." Neural networks 4.2 (1991): 251-257.

[9] Blum, Edward K., and Leong Kwan Li. "Approximation theory and feedforward networks." Neural networks 4.4 (1991): 511-515.

[10] Kůrková, Věra. "Kolmogorov's theorem and multilayer neural networks." Neural networks 5.3 (1992): 501-506.

[11] Mead C, Ismail M (1989) Analog VLSI Implementation of Neural Systems. Springer.

[12] M. Prezioso, F. Merrikh-Bayat, B. Hoskins, G. Adam, K. K. Likharev, and D. B. Strukov, “Training and operation of an integrated neuromorphic network based on metal-oxide memristors,” Nature, vol. 521, no. 7550, pp. 61–64, 2015.

[13] A. Z. Stieg, A. V. Avizienis, H. O. Sillin, R. Aguilera, H.-H. Shieh, C. Martin-Olmos, E. J. Sandouk, M. Aono, and J. K. Gimzewski, “Self-organization and emergence of dynamical structures in neuromorphic atomic switch networks,” in Memristor Networks. Springer, 2014, pp. 173–209.

[14] T. Tuma, A. Pantazi, M. Le Gallo, A. Sebastian, and E. Eleftheriou, “Stochastic phase-change neurons,” Nature nanotechnology, vol. 11, no. 8, pp. 693–699, 2016.

[15] D. Negrov, I. Karandashev, V. Shakirov, Y. Matveyev, W. DuninBarkowski, and A. Zenkevich, “An approximate backpropagation learning rule for memristor based neural networks using synaptic plasticity,” Neurocomputing, 2016.

[16] Masquelier, T., Guyonneau, R. & Thorpe, S. J. Competitive STDP-based spike pattern learning. Neural Comput. 21, 1259–1276 (2009).

[17] Dennis J B, Fosseen J B, Linderman J P. Data flow schemas. In Proc. the International Symposium on Theoretical Programming, August 1972, pp.187-216.

[18] Jagannathan R. Coarse-grain dataflow programming of conventional parallel computers[C]//in Advanced Topics in Dataflow Computing and Multithreading. 1995.

[19] Zuckerman S, Suetterlein J, Knauerhase R, Gao G R. Using a “codelet” program execution model for exascale machines: Position paper. In Proc. the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era, June 2011, pp.64-69.

[20] Qu, P., Yan, J., Zhang, Y. H., & Gao, G. R. (2017). Parallel Turing Machine, a Proposal. Journal of Computer Science and Technology, 32(2), 269-285.

[21] Jouppi, Norman P., et al. "In-datacenter performance analysis of a tensor processing unit." 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2017.

[22] Chen, Tianshi, et al. "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning." ACM Sigplan Notices. Vol. 49. No. 4. ACM, 2014.

[23] Merolla, Paul A., et al. "A million spiking-neuron integrated circuit with a scalable communication network and interface." Science 345.6197 (2014): 668-673.

[24] Furber, Steve B., et al. "The spinnaker project." Proceedings of the IEEE 102.5 (2014): 652-665.

[25] Davies, Mike, et al. "Loihi: A neuromorphic manycore processor with on-chip learning." IEEE Micro 38.1 (2018): 82-99.

[26] Pei, Jing, et al. "Towards artificial general intelligence with hybrid Tianjic chip architecture." Nature 572.7767 (2019): 106-111.

[27] Anwar, Sajid, Kyuyeon Hwang, and Wonyong Sung. "Fixed point optimization of deep convolutional neural networks for object recognition." 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2015.

[28] Han, Song, Huizi Mao, and William J. Dally. "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding." arXiv preprint arXiv:1510.00149 (2015).

[29] Ji Y, Zhang Y, Xie X, et al. FPSA: A Full System Stack Solution for Reconfigurable ReRAM-based NN Accelerator Architecture[C]//Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, 2019: 733-747.

[30] Elliott, D., Stumm, M., Snelgrove, W. M., Cojocaru, C. & McKenzie, R. Computational RAM: implementing processors in memory. IEEE Des. Test Comput. 16, 32–41 (1999).

[31] Boahen, K. A. Point-to-point connectivity between neuromorphic chips using address events. IEEE Trans. Circuits Syst. II 47, 416–434 (2000).

[32] Serrano-Gotarredona, R. et al. AER building blocks for multi-layer multi-chip neuromorphic vision systems. In Advances in Neural Information Processing Systems Vol. 18 (eds Weiss, Y., Schölkopf, B. & Platt, J. C.) 1217–1224 (Neural Information Processing Systems Foundation, 2006)

[33] Sawada J, Akopyan F, Cassidy A S, et al. Truenorth ecosystem for brain-inspired computing: scalable systems, software, and applications[C]//SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2016: 130-141.

[34] Amir A, Datta P, Risk W P, et al. Cognitive computing programming paradigm: a corelet language for composing networks of neurosynaptic cores[C]//The 2013 International Joint Conference on Neural Networks (IJCNN). IEEE, 2013: 1-10.

[35] Furber S B, Lester D R, Plana L A, et al. Overview of the SpiNNaker system architecture[J]. IEEE Transactions on Computers, 2012, 62(12): 2454-2467.

[36] Sharp T, Plana L A, Galluppi F, et al. Event-driven simulation of arbitrary spiking neural networks on SpiNNaker[C]//International conference on neural information processing. Springer, Berlin, Heidelberg, 2011: 424-430.

[37] Davies M, Srinivasa N, Lin T H, et al. Loihi: A neuromorphic manycore processor with on-chip learning[J]. IEEE Micro, 2018, 38(1): 82-99.

[38] Deng L, Liang L, Wang G, et al. Semimap: A semi-folded convolution mapping for speed-overhead balance on crossbars[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2018.

[39] Moritz L B, Moritz L Y B. Safety and optimization transformations for data flow programs[J]. 1980.

[40] John Ruttenberg, Guang R Gao, Artour Stoutchinin, and Woody Lichtenstein. Software pipelining showdown: Optimal vs. heuristic methods in a production compiler.

[41] Chen T, Zheng L, Yan E, et al. Learning to optimize tensor programs[C]//Advances in Neural Information Processing Systems. 2018: 3389-3400.

[42] Moritz L B, Moritz L Y B. Safety and optimization transformations for data flow programs[J]. 1980.

[43] Gao G R. A pipelined code mapping scheme for static data flow computers[D]. Massachusetts Institute of Technology, 1986.

[44] Burr, G. W. et al. Neuromorphic computing using non-volatile memory. Adv. Phys. X 2, 89–124 (2017).

[45] Xia, Q., Yang, J.J. Memristive crossbar arrays for brain-inspired computing. Nat. Mater. 18, 309–323 (2019) doi:10.1038/s41563-019-0291-x.

[46] Snider, G. S. Spike-timing-dependent learning in memristive nanodevices. In Proc. Int. Symp. on Nanoscale Architectures 85–92 (IEEE, 2008).

[47] Chi, P. et al. Prime: a novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. In Proc. 43rd Int. Symp. Computer Architecture 27–39 (IEEE, 2016).

[48] Deng L, Zou Z, Ma X, et al. Fast Object Tracking on a Many-Core Neural Network Chip[J]. Frontiers in neuroscience, 2018, 12.

[49] Shafiee, A. et al. ISAAC: a convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In Proc.

43rd Int. Symp. Computer Architecture 14–26 (IEEE, 2016).

[50] Ji Y, Zhang Y, Xie X, et al. FPSA: A Full System Stack Solution for Reconfigurable ReRAM-based NN Accelerator Architecture[C]//Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, 2019: 733-747.

[51] Jason Cong and Bingjun Xiao. 2011. mrFPGA: A novel FPGA architecture with memristor-based reconfiguration. In Proceedings of the 2011 IEEE/ACM International Symposium on Nanoscale Architectures, NANOARCH 2011, San Diego, CA, USA, June 8-9, 2011. IEEE Computer Society, 1–8. https://doi.org/10.1109/NANOARCH.2011.5941476.

[52] Chakraborty, I., Roy, D. & Roy, K. Technology aware training in memristive neuromorphic systems for nonideal synaptic crossbars. IEEE Trans. Em. Top. Comput. Intell. 2, 335–344 (2018).

[53] Le Gallo, M. et al. Mixed-precision in-memory computing. Nature Electron. 1, 246 (2018).

[54] Krstic, M., Grass, E., Gürkaynak, F. K. & Vivet, P. Globally asynchronous, locally synchronous circuits: overview and outlook. IEEE Des. Test Comput. 24, 430–441 (2007).

[55] Prezioso, M., Mahmoodi, M.R., Bayat, F.M. et al. Spike-timing-dependent plasticity learning of coincidence detection with passively integrated memristive circuits. Nat Commun 9, 5311 (2018) doi:10.1038/s41467-018-07757-y.

[56] Zhang W, Yang Y. A survey of mathematical modeling based on flocking system[J]. Vibroengineering PROCEDIA, 2017, 13: 243-248.Bajec I L, Zimic N, Mraz M.

[57] Reynolds C W. Flocks, herds and schools: A distributed behavioral model[C]//Proceedings of the 14th annual conference on Computer graphics and interactive techniques. 1987: 25-34.

[58] Bajec I L, Zimic N, Mraz M. The computational beauty of flocking: boids revisited[J]. Mathematical and Computer Modelling of Dynamical Systems, 2007, 13(4): 331-347.

[59] Conrad Parker. XBoids, Feb 2002. GPL Version 2 Licensed. URL: http://www.vergenet.net/~conrad/boids/download[last accessed 2014-01-02].

[60] Araújo R, Waniek N, Conradt J. Development of a dynamically extendable spinnaker chip computing module[C]//International Conference on Artificial Neural Networks. Springer, Cham, 2014: 821-828.

[61] Roy K, Jaiswal A, Panda P. Towards spike-based machine intelligence with neuromorphic computing[J]. Nature, 2019, 575(7784): 607-617.

[62] Serrano-Gotarredona, T., Masquelier, T., Prodromakis, T., Indiveri, G., and Linares-Barranco, B. (2013) STDP and STDP variations with memristors for spiking neuromorphic learning systems. Front. Neurosci., 7 (2), 1–15.

[63] Zylberberg A, Dehaene S, Roelfsema P R, et al. The human Turing machine: a neural framework for mental programs[J]. Trends in cognitive sciences, 2011, 15(7): 293-300.

[64] Wegner P, Goldin D. Computation beyond Turing machines[J]. Communications of the ACM, 2003, 46(4): 100-102.

[65] Goldin, D. & Wegner, P. Minds & Machines (2008) 18: 17. https://doi.org/10.1007/s11023-007-9083-1.

[66] van Leeuwen, J. and Wiedermann, J. The Turing machine paradigm in contemporary computing. In B. Enquist and W. Schmidt, Eds., Mathematics Unlimited—2001 and Beyond. LNCS, Springer-Verlag, 2000.

[67] Peng Yao, Huaqiang Wu, Bin Gao, Sukru Burc Eryilmaz, Xueyao Huang, Wenqiang Zhang, Qingtian Zhang, Ning Deng, Luping Shi, H-S Philip Wong, and He Qian. 2017. Face classification using electronic synapses. *Nature Communications* 8 (2017).

[68] Kuzum, D., Jeyasingh, R. G., Lee, B. & Wong, H.-S. P. Nanoelectronic programmable synapses based on phase change materials for brain-inspired computing. Nano Lett. 12, 2179–2186 (2012).

[69] Krzysteczko, P., Münchenberger, J., Schäfers, M., Reiss, G. & Thomas, A. The memristive magnetic tunnel junction as a nanoscopic synapse–neuron system. Adv. Mater. 24, 762–766 (2012).

[70] Vincent, A. F. et al. Spin-transfer torque magnetic memory as a stochastic memristive synapse for neuromorphic systems. IEEE Trans. Biomed. Circuits Syst. 9, 166–174 (2015).

[71] Sengupta, A. & Roy, K. Encoding neural and synaptic functionalities in electron spin: a pathway to efficient neuromorphic computing. Appl. Phys. Rev. 4, 041105 (2017).

[72] Hu, M. et al. Dot-product engine for neuromorphic computing: programming 1T1M crossbar to accelerate matrix-vector multiplication. In Proc. 53rd ACM/EDAC/IEEE Annual Design Automation Conf. 21.1 (IEEE, 2016).

[73] Neckar, A., Fok, S., Benjamin, B. V., Stewart, T. C., Oza, N. N., Voelker, A. R., ... & Boahen, K. (2018). Braindrop: A mixed-signal neuromorphic architecture with a dynamical systems-based programming model. Proceedings of the IEEE, 107(1), 144-164.

[74] Davison, Andrew P., et al. PyNN: a common interface for neuronal network simulators. Frontiers in neuroinformatics 2 (2009): 11.

[75] Bekolay, Bergstra, Hunsberger, DeWolf, Stewart, Rasmussen, Choo, Voelker & Eliasmith. (2014) Nengo: a Python tool for building large-scale functional brain models. Frontiers in Neuroinformatics 7.

[76] Kasabov, N. NeuCube: A Spiking Neural Network Architecture for Mapping, Learning and Understanding of Spatio-Temporal Brain Data, Neural Networks vol.52 (2014), pp. 62-76, http://dx.doi.org/10.1016/j.neunet.2014.01.006.

[77] Aimone, James B., William Severa, and Craig M. Vineyard. "Composing neural algorithms with Fugu." Proceedings of the International Conference on Neuromorphic Systems. 2019.

[78] Lagorce, Xavier, and Ryad Benosman. "Stick: spike time interval computational kernel, a framework for general purpose computation using neurons, precise timing, delays, and synchrony." Neural computation 27.11 (2015): 2261-2317.

[79] Araújo R, Waniek N, Conradt J. Development of a dynamically extendable spinnaker chip computing module[C]//International Conference on Artificial Neural Networks. Springer, Cham, 2014: 821-828.

[80]Aimone, James B., et al. "Non-neural network applications for spiking neuromorphic hardware." Proceedings of the Third International Workshop on Post Moores Era Supercomputing. 2018.