NEUTRAMS: Neural Network Transformation and Co-design under Neuromorphic Hardware Constraints

Yu Ji*, YouHui Zhang*‡, ShuangChen Li†, Ping Chi†, CiHang Jiang*, Peng Qu*, Yuan Xie†, WenGuang Chen*‡
Email: zyh02@tsinghua.edu.cn, yuanxie@ece.ucsb.edu
*Department of Computer Science and Technology, Tsinghua University, PR.China
† Department of Electrical and Computer Engineering, University of California at Santa Barbara, USA
‡ Center for Brain-Inspired Computing Research, Tsinghua University, PR.China

***Abstract*—With the recent reincarnations of neuromorphic computing comes the promise of a new computing paradigm, with a focus on the design and fabrication of neuromorphic chips. A key challenge in design, however, is that programming such chips is difficult. This paper proposes a systematic methodology with a set of tools to address this challenge. The proposed toolset is called NEUTRAMS (Neural network Transformation, Mapping and Simulation), and includes three key components: a neural network (NN) transformation algorithm, a configurable clock-driven simulator of neuromorphic chips and an optimized runtime tool that maps NNs onto the target hardware for better resource utilization. To address the challenges of hardware constraints on implementing NN models (such as the maximum fan-in/fan-out of a single neuron, limited precision, and various neuron models), the transformation algorithm divides an existing NN into a set of simple network units and retrains each unit iteratively, to transform the original one into its counterpart under such constraints. It can support both spiking neural networks (SNNs) and traditional artificial neural networks (ANNs), including convolutional neural networks (CNNs) and multilayer perceptrons (MLPs) and recurrent neural networks (RNNs). With the combination of these tools, we have explored the hardware/software co-design space of the correlation between network error-rates and hardware constraints and consumptions. Doing so provides insights which can support the design of future neuromorphic architectures. The usefulness of such a toolset has been demonstrated with two different designs: a real Complementary Metal-Oxide-Semiconductor (CMOS) neuromorphic chip for both SNNs and ANNs and a processing-in-memory architecture design for ANNs.**

I. Introduction

Although a consensus on how the brain works is yet to be reached, the great potential of neural systems has aroused research enthusiasm [1]–[3]. In addition to the discovery of the brain's computational paradigm, these studies explore the possibility to implement neuromorphic circuits with higher parallelism and lower power, compared to the traditional von Neumann computer architecture.

Recently, very-large-scale integration (VLSI) systems have been widely employed to mimic neuro-biological architectures (called *neuromorphic engineering*). A multi-core processor (CMP)[1] with a Network-on-Chip (NoC) has emerged as a promising platform for neural network simulation, motivating many recent studies of neuromorphic chips [4]–[10]. The design of neuromorphic chips is promising to enable a new computing paradigm.

One major issue is that neuromorphic hardware usually places constraints on NN models (such as the maximum fan-in/fan-out of a single neuron, the limited range of synaptic weights) that it can support, and the hardware types of neurons or activation functions are usually simpler than the software counterparts. For example, one neurosynaptic core of the IBM's TrueNorth chip [4] has 256 axons, a 256×256 synapse crossbar, and 256 neurons, which limits the maximum number of synapses that one neuron can be connected to directly. Furthermore, it puts constraints on a neuron's synaptic weight, and the neuron model supported is a variant of the LIF (leaky integrate-and-fire, a simple neuron model). For Embrace [8], one neuron can be connected to up to 1024 synapses and the neuron model is also LIF. In contrast, such limitations do not exist for software simulation.

One way to avoid the limitation on the connection number is to decouple synaptic weight storage from processing logic and (often) uses software to control the memory I/O and resource re-usage, which has been adopted by some custom architectures [9]–[12] to accelerate artificial intelligence (AI) algorithms. However, with the scale increase, access to weight data may become the system bottleneck, as this method inherently separates computation and storage, which is just one of cruxes of the traditional von Neumann computer. Conversely, tightly coupling relationship between weight storage and processing logic (for example, each logical synapse mapped to a hardware synapse respectively) is conducive to the usage of new storage technologies (like memristors) to integrate storage and computation in the same physical location, e.g., a simple and transistor-free crossbar structure of metal-oxide memristors may achieve extremely high density [13] (even

[1]The *core* can refer to some general-purpose CPU to simulate neurons in software or some dedicated ASIC unit(s) to emulate neurons in hardware.

higher than that of biological prototypes). Related studies include [14]–[21].

Consequently, it is necessary to design software tools to bridge the gap between applications and neuromorphic hardware. A transparent method is preferred; namely, the toolchain should not be bound on concrete neuromorphic hardware. In contrast, existing solutions either are hardware-specific [4], [6]–[8], or use general-purpose CPU cores to complete neural computing (such as SpiNNaker [5]) with lower efficiency and scalability.

In this paper, we propose NEUTRAMS (Neural network Transformation, Mapping and Simulation), a toolchain of neuromorphic chips, and its usage for hardware/software (HW/SW) co-design. The discussion offers the following contributions:

• A hardware-independent representation layer of neural networks is proposed to describe high-level NN models to satisfy the hardware constraints of a neuromorphic chip.
• A training-based transformation algorithm is designed to transform an existing NN described by the above representation into its counterpart under the target hardware's constraints.
• A configurable, clock-driven simulator is implemented to achieve performance and power simulation on the micro-architecture level, supporting different types of neuromorphic cores and memory technologies. Trained SNNs can run on this simulator with diverse configurations. Thus, the relationship between network error-rates and hardware constraints and consumptions is explored.
• An optimized strategy to map the trained NNs onto neuromorphic hardware is proposed, using graph partitioning algorithm to put densely communicating neurons together.
• Such a toolchain for a real CMOS neuromorphic processor (*TIANJI* [22]) and for a processing-in-memory architecture design for ANNs (*PRIME* [23]) has been implemented. It can also guide the optimization of hardware design.

II. Related Work

• **Neuromorphic Chips.** The existing neural network hardware accelerators are inspired by two different disciplines: machine-learning or neuroscience. Recently, Du et al. [24] did comparisons among the two types of hardware implementation, and concluded that each has its advantages and disadvantages.

TrueNorth [4], [25] is a digital neuromorphic chip that includes 4096 neurosynaptic cores connected via a 2D-mesh NoC. Moreover, it has proposed a toolchain, including the programming paradigm, *Corelet* [26]. In addition, it provides an optimized strategy to map logical NNs to physical cores [27]. From open publications we deduce that the Corelet description matches the organization of the hardware substrate; thus, it is bound to the hardware platform. Neurogrid [6] is an analog/digital hybrid system with a customized network [28]. It uses the Neural Engineering Framework (NEF) [29] to configure neuromorphic chips to implement target functions. This allows the designer to work on a higher level of abstraction and yet still produce a detailed model using spiking neurons. Namely, Neurogrid supports the NEF on hardware and system levels. EMBRACE [8] is a compact hardware SNN architecture for embedded computing. It follows the Modular Neural Network (MNN) computing paradigm [30]. The FACETS [7] project and its successor BrainScaleS have produced wafer-scale IC systems. It developed a graph model to map a complex biological network to the hardware network. This strategy requires that both have the same underlying structure; thus, it is also hardware-specific. SpiNNaker [5] has a toolchain based on the CMPs of ARM cores [31]. Thus, its neural computing is completed by software. The drawback is that efficiency will be lower than dedicated hardware.

Other studies [13]–[21] have proposed to utilize emerging memory technologies (such as memristor crossbar) to mimic synaptic behaviors for higher density and energy efficiency. They are usually focused on prototype construction without sufficient software tools' support.
• **Software SNN Simulators.** A growing number of software tools have been developed to simulate SNNs, such as NEURON [32], Nengo [33], Brian [34] and CARLSIM [35]. For large-scale networks, they often neglect the geometric and biophysical complexity of individual neurons. Namely, neural networks can be described as weighted, directed graphs.
• **NN Training.** Quite a few research efforts [36]–[39] focused on temporal coding SNNs as they differ from traditional ANNs, in that neurons propagate information by the timing of spikes. On the other hand, several studies have been done regarding rate coding SNNs, which try to graft successes in rate-coding NNs in engineering to SNNs, including [40]–[44]. In addition, approximate computing [45]–[47] has used training to produce a neural network to replace the original computing function. These studies supported multi-layer perceptrons (MLPs) and some [46], [47] considered hardware constraints on precision and/or network topology. In contrast, we have designed a transformation algorithm that can be applied to more NN topologies, including both ANN and SNN. Moreover, the original NN's topology is maintained roughly (rather than being transformed into an MLP), for better convergence.
• **Others.** To decouple an NN application's algorithmic specification from the execution substrate, there are some related studies. NISA (neuromorphic instruction set architecture) [48] is such an XML-based language used to describe a cortical network. PyNN [49] is a simulator-independent language for building NN models.

In summary, for existing chips, their toolchains are either specific to hardware (or to some specific computing paradigm) or use general-purpose CPU cores to complete neural computing (such as SpiNNaker [5]). In contrast, we propose a solution that is not dependent on a specific hardware design and can support many types of NN applications.

III. Framework of the Toolchain

This work is inspired by the hierarchy of traditional computer systems with the following levels (see left part of Figure 1). For the neuromorphic system, we believe that its software stack should contain analogous levels, including neuromorphic applications and language (representation) and compiler (training tool) and runtime system (for mapping) and hardware (as well as equivalent simulator). Targeted neural computing applications include those from computational neuroscience (SNNs) and those converted from traditional ANNs. Our proposed toolchain includes the following:

- NN representation layer (Section III-A);
- Training-based transformation algorithm for NNs under constraints (Section III-B);
- Mapping algorithm (Section III-C).
- Configurable clock-driven simulator (Section III-D);

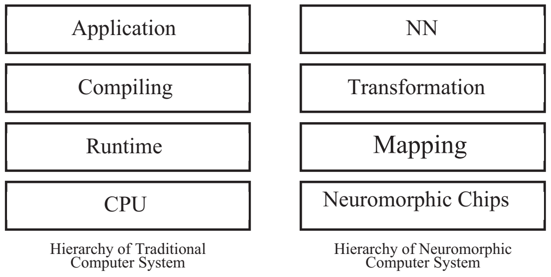

Application
Compiling
Runtime
CPU
NN
Transformation
Mapping
Neuromorphic Chips
Hierarchy of Traditional Computer System
Hierarchy of Neuromorphic Computer System

Fig. 1. The analogous hierarchies.

A. Representation Layer

Existing neural system simulators often provide programming interfaces for users to develop SNN models. For large-scale NN, the model representation usually operates on the *Population* (group of homogeneous neurons)/*Projection* (bundle of single connections between populations) level rather than on the single-neuron/ connection level.

```
// Two populations have been created; the neuron
// number and type are set
pre = Population(1000,one-neuron-type)
post = Population(500,another-neuron-type)
// An all-to-all-connection projection between two
// populations has been created; the weight and
// latency of each connection are set.
excitatory_connections = Projection (pre, post,
     AllToAllConnector (), StaticSynapse (weight=0.13))
excitatory_connections.set (delay=0.4)
                              ......
```

Fig. 2. Example of SNN representation.

For example, Figure 2 presents the pseudo-code that creates two neuron-populations and a projection connecting them. Besides the simple mode in Figure 2, more connection modes and attribute-configurations have been supported. Thus, under users' guidance, a simulator will create the target NN that has the following information[2]: (1)*Node information*, which contains neuron/synapse types, as well as the population to which it belongs. Each is identified by an integer ID. (2)*Edge information*, containing the IDs of its source and target nodes, the weight and delay (SNNs can take the arrival time of spikes into operating model) or other attributes describing the projection from source to target.

Accordingly, a representation layer is given to describe this information; the default connection mode between populations is *all-to-all*. Thus, edge information can be maintained in the population granularity. For other connection modes (like the random connection mode in which the existence of a connection of two neurons from different populations depends on a probability), 0-weight edges will be introduced. In this manner, all modes can be normalized as *all-to-all*. Connection modes can be defined by customized functions.

This method can also be used to describe a traditional ANN: for MLPs, it is quite suitable, as each layer fully connects to the next; for CNNs, because the connection between any two successive layers is sparse, quite a few 0-weight edges will be introduced.

B. Training-Based Transformation

We first present the outline of the backpropagation (BP) algorithm for SNNs. Second, training steps of a single-layer perceptron (SLP, the basic training unit) are presented in detail. Third, based on the *divide-and-conquer* principle, a general workflow on how to split a complicated NN into training units is designed. Thus, our algorithm is more general than existing work [45]–[47].

1) BP for SNNs: All SNN models supported by the toolchain are rate coding and the firing rate contains all of the information transferred between neurons. Thus, the spike count in a time interval can represent a numerical value (in a certain range). Moreover, we know inputs of a spiking neuron are spikes from pre-neurons; after synaptic computation, they are converted into the sum of currents that will be processed by the neural model further. These two functions are briefly described as follows:

- **Synaptic computation.** A spike issued at a given frequency, p ($p \in [0, 1]$), is input into a synapse. The latter produces a steady current, denoted as $I(p)$.
- **Neural computation.** As the input current is stable to I, the neuron's steady firing rate is denoted as $f(I)$. The rate is defined as the reciprocal of the average firing period (gap between two continuous issues). The latter is the elapsed time as the neuron membrane potential increases from the reset value to the threshold.

Based on the computation model of simulated neurons[3], these two functions usually own good continuity and are derivable. Therefore, the popular BP method can be used here for training SNN. The original NN (whether an ANN or SNN) is used as the golden model to generate infinite training data as long as there is enough input data. If the original is an SNN, training data is neurons' firing rates.

2) Training a unit NN under hardware constraints: As mentioned in Section III-A, we describe an NN model on the level of neuron populations. Accordingly, the training objective is to adjust connections between two populations (including the weights and connectivity) to meet hardware limitations. Without loss of generality, the connection matrix of the two neighboring populations (P_a and P_b; P_a is the pre-node of

[2]Although the neural network can be hierarchical, in the representation layer, it is regarded as a *flat* graph.

[3]Most existing neuromorphic chips support the Leaky Integrate-and-Fire (LIF) neuron model or its variants.

P_b) is denoted as M_{ab} (the size is $m \times n$), which is called P_b's connection matrix; its width equals the neuron number of P_b.

The training procedure can be divided into four steps: (1) training the NN (whose parameters has been initialized to corresponding values of the golden model) without any limitation to bridge the gap caused by different neuron models and then get the *perfect model*; (2) connection sparsification, to alter the perfect model to meet hardware limitations on connection number; (3) scaling for lower precisions; (4) layer insertion. For Steps (2), (3), and (4), details are given as follows:

• **Connection sparsification.** To meet the hardware limitation on connection number, we tend to make the connection matrix sparser by discarding those connections with tiny weight values. Namely, the objective is to maximize the sum of absolute weight-values of remaining connections. Without loss of generality, we assume that both m and n exceed the hardware limitations on the fan-in and fan-out of a single neuron (denoted as k and g, respectively). We designed a sparsification algorithm whose principle is straight: Initially it constructs several sub-matrices (the height of each matrix is not larger than k and the width is not larger than g) along the diagonal (Figure 3). Then, we repeat exchanging rows and columns of the whole matrix or moving rows or columns from a submatrix to another till no more exchange can increase the sum of absolute values of elements of all sub-matrices. Finally, all elements outside these sub-matrices are fixed to zero and the new network is trained again.

$$\begin{pmatrix} W_{11} & W_{12} & \dots & W_{1n} \\ W_{21} & W_{22} & \dots & W_{2n} \\ \dots & \dots & \dots & \dots \\ W_{m1} & W_{m2} & \dots & W_{mn} \end{pmatrix} \xRightarrow[\text{or columns}]{\text{swap rows}} \begin{pmatrix} (M_{11}) & 0 & \dots & 0 \\ 0 & (M_{22}) & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & (M_{pp}) \end{pmatrix}$$

Fig. 3. Matrix sparsification.

• **Scaling for lower precisions.** A number of recent studies [50]–[52] were carried out to assess the impact of the precision of computations on the final error of NN training. For example, [52] indicated that fixed point numbers of finite width are enough to represent gradients and storing weights. We referenced existing work to make NNs meet the hardware demand on precisions. The principle is as follows: after the normal training (floating points are used) and sparsification, all relevant parameters will be checked. Based on the statistics, some proper scaling factors will be chosen to convert the original data into the nearest integers of the preset width. This network will be initialized with the scaled parameters and be trained again, while during the back propagation and parameter update phase, we still use high precisions to improve the accuracy.

• **New layer(s).** A new population will be inserted between the original two, fully connected to both. This mechanism can improve the NN capacity to make up for the possible loss caused by the previous steps. Usually, its neuron number is set to $(m+n)/2$; tests show that in most cases this configuration has good convergence. Then, the new NN will be trained, following the aforesaid steps. This procedure can repeat until the error reaches the expected.

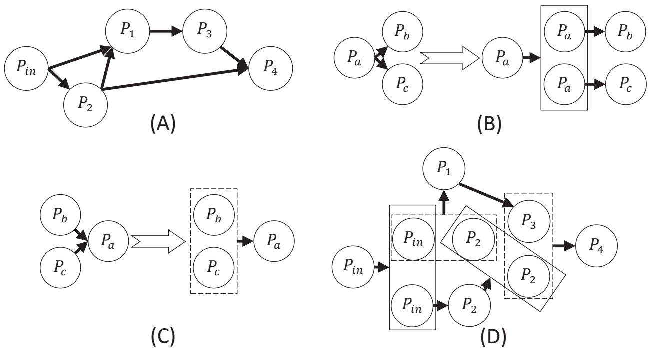


Fig. 4. Data flow diagram. (A) The original NN graph. (B) The solid line box means the n copies of pre-node are concatenated as one (C) The dashed line box means all pre-nodes are concatenated as one. (D) Original NN graph can be split into NN units. Each arrow represents an NN unit.

3) General transformation workflow: A more complex NN can be regarded as a directed acyclic data-flow graph and each graph node represents a population. Accordingly, the training order does adhere to the direction of data flow. For a given population (or more precisely, for its connection matrix), the necessary and sufficient condition in which it can be trained is that inputs of all its pre-nodes have been ready; here the *ready* status of a node means that it has been trained (the input one is considered to be ready from the beginning).

Taking Figure 4(A) as an example, the order is P_2, P_1, P_3 and P_4. For the population to be trained, its input data is just the output from its latest trained pre-node(s), rather than the golden model's, to prevent error accumulation. Namely, each unit is retrained iteratively.

If a population (denoted as P_a) has more than one post-node (the number of post-nodes is denoted as n), we insert behind P_a a dedicated population that contains n copies of P_a; each copy connects to one post-node respectively (Figure 4(B)). Accordingly, the connection matrix between P_a and its multicast population can be trained as a unit NN; no population has more than one post-node now.

If a population has more than one pre-node, its connection matrix is represented as the combination of all matrices between this population and all pre-nodes (Figure 4(C)). In this way the merged one can be trained as a unit NN, too.

Therefore, the whole workflow is divided into multiple iterative phases, while each phase can be regarded as training a single-layer perceptron with the BP algorithm (the division of the graph in Figure 4(A) is given in Figure 4(D)). The final result is a set of connected populations and each connection submatrix (after sparsification) meets hardware limitations.

In addition, all above methods do not depend on concrete neuron and synapse models (as long as the synaptic and neural computations are both derivable).

4) Recurrent network: If the original NN is an RNN, the whole flow is slightly different. The input data should be a sequence. For a recurrent ANN, each layer will generate a sequence as the training data. For a recurrent SNN, we break some edges in the graph to remove all recurrences (the set of all broken edges is denoted as E). Then, the input begins to issue spikes step by step; at the same time, the firing rate of each neuron is being recorded. For $e \in E$ at step t, all spikes from its pre-node are recorded as $S_{e,t}$ and spikes of $S_{e,t-1}$ are injected into its post-node. Thus, we can get sequences of training data of all populations. During the transformation phase, the condition in which one population p can be trained is also different: If $e \in E$ is one of its input edges, we use the corresponding training data from the golden model.

5) Training and design space: Although the training method can be likened to the compiler of traditional computer systems, there is a significant difference between them: The compiler is precise and conservative; it must behave according to the language specifications strictly and to ensure the equivalence of language translation. In contrast, because of the inherent ambiguity and robustness of NNs, training is reasonable to be used as the transparent transformation method although some errors may be introduced. Many factors may affect the performance error of NNs. Thus, the tradeoff between errors and hardware-resource consumption is interesting, which enlarges the HW/SW co-design space.

C. Mapping

Mapping is a process assigning neurons into cores and presents connection relationship between them in synaptic weight arrays and fills routing tables. Optimized mapping is not only beneficial to utilize on-chip resources efficiently but also related to correctness [53]. SpiNNaker uses a simple sequential mapping scheme [31]: Neurons are uniformly allocated to NoC nodes in order; thus, neurons from a population will be distributed into one core or into several nearby cores (FACETS [7] employed the similar strategy). TrueNorth formulates the mapping as a wire-length minimization problem in VLSI placement [27].

We propose a mapping algorithm that tends to put densely-communicating neurons close together. Because we draw SNN models from existing software simulators and/or train models to adapt to the target substrate, we can statistically get communication patterns through software simulation and/or offline training.

As described in Section III-B, the transformation result is a set of connected populations and each connection submatrix meets hardware limitations. Thus, all neurons have been distributed into virtual cores and we map them into physical cores on the chip.

We use the Kernighan-Lin (KL) partitioning strategy for NoCs (here we take the 2D-mesh as the example). It abstracts the mapping as a graph partition problem and tries to minimize communications across partition boundaries.

First, a randomly generated initial mapping distribution is given. Second, the KL algorithm bipartitions the mapped NoC nodes: highly communicated modules are kept in one partition. This procedure is applied until only the closest two nodes are left in any of the final partitions in a 2D-mesh. During this phase, partitions that minimize the communication cost between cores will remain.

After mapping, one trick is to renumber neurons to make IDs of all neurons inside a core successive, which enables the router to access the routing table directly.

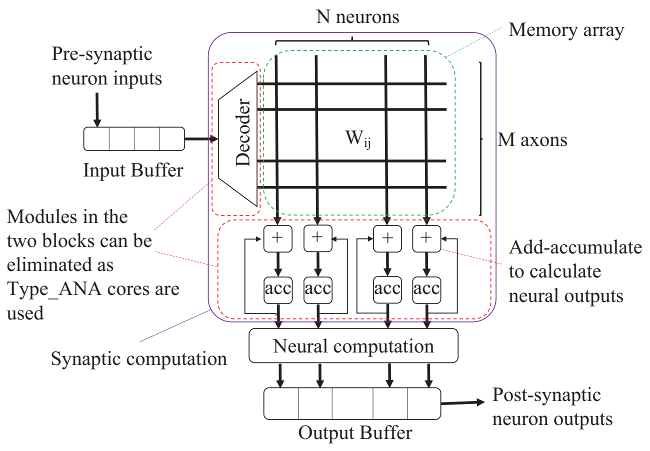

N neurons
Memory array
Pre-synaptic neuron inputs
Decoder
W_{ij}
M axons
Input Buffer
Modules in the two blocks can be eliminated as Type_ANA cores are used
+
acc
Add-accumulate to calculate neural outputs
Synaptic computation
Neural computation
Post-synaptic neuron outputs
Output Buffer

Fig. 5. Logical diagram of the proposed core architecture (including the central crossbar and peripheral circuits).

D. The Simulator

The design is modular, which abstracts neuromorphic chips into three common logical modules: neuromorphic cores and memory and NoC.

1) Cores: Neuromorphic cores mimic neurons. Figure 5 shows a basic logic diagram of the proposed core architecture (architectures of quite a few existing chips, like TrueNorth and EMBRACE and TIANJI, are similar). Each core processes a collection of N neurons, with each neuron having up to M input axons. The synaptic weights ($W_{i,j}$, $0 \leq i < N$ and $0 \leq j < M$) are stored in a memory array. These weight values are multiplied with the pre-synaptic input values (just 0 or 1; thus no multiplying unit is needed) and accumulated with the current membrane potential, and then are handled by a simple leak-integration function[4]. If an output spike is generated, it will be sent to the local router for further processing. Input and output buffers store pre-synaptic inputs and post-synaptic outputs, respectively.

The processing procedure of a simulated neuron is abstracted into three steps: (1) to get current input spike(s), as well as the corresponding synaptic weight(s); (2) to calculate and update the neuron's state (the computation paradigm is presented in Figure 6) and (3) to issue a new spike if the membrane potential reaches a predefined threshold (for each neuron, this value can be set separately). If there is more than one synchronous input, Steps (1) and (2) will repeat.

From the perspective of simulation, timing and functional simulations are separated. The concrete calculation is simulated as a single software function (annotated with timing information), while input/output queues are accurately simulated on the architectural level to reflect possible accessing conflicts. Memory access for synaptic weight is also a part of the whole processing procedure, which is handled by the memory module. Moreover, time division multiplexing is also simulated to improve resource utilization.

Owing to the modular design, now three types of typical neuromorphic cores [54] have been supported, which use dif-

[4]Here the most-widely-used LIF model is used as the example.

$$V_i(t) = V_i(t\text{-}1) + \sum_{j=0}^{N-1} S_j(t) W_{ij} - L$$

($V_i(t)$ denotes the membrane potential of Neuron$_i$ at Cycle t; $S_j(t)$ is the current spike (0 or 1) from the j^{th} axon, and L is the value of leakage current).

Fig. 6. Neuron computation paradigm.

ferent memory technologies for synaptic information, SRAM and two kinds of memristors (digital and analog cores):

For the first two (denoted as Type_SRAM and Type_DIG respectively), the functional procedures are just the same as mentioned above; the difference lies in that memristors own the higher density that will be reflected in the aspects of power-consumption and memory access latency.

For the third (Type_ANA), the high resistance range in memristor devices allows them to model synapses in analog form. In this form, calculations of pre-synaptic spikes and synaptic weights can be carried out using memristor resistance values and current summation, which reduces circuit area and power consumption apparently. Specially, two crossbars are needed to represent the positive and negative weights respectively; their outputs will be fed into a subtraction unit to get the difference signal. In addition, analog-to-digital converters (ADCs) are needed for analog computing to transform multi-bit-width signals. Related designs have been presented in a number of existing studies [13]–[15]. As we separate timing and functional simulations, this procedure is just simulated as a software function (annotated with timing and power consumption information that can reference existing studies). Detailed configurations are provided in Section IV.

Moreover, the crossbar design allows all synaptic values to be handled simultaneously to boost computing speed. Several components in the previous types could be eliminated, including the memory array decoder and accumulation circuits (Figure 5).

2) Memory: Besides parameters of the neuron model, memories are mainly used as the synaptic weight array to hold the data for $M\times N$ synapses. As the bit-width of a synapse is set to K, the whole storage consumption is $M\times N\times K$ bits. Different core-types use SRAM arrays and tiled crossbar arrays respectively. An asynchronous interface is implemented to integrate this simulator with other third-party architectural RAM simulators, for accurate timing information and power consumption of memory accesses.

3) NoC: It connects cores together to transfer spikes (each core is attached to a local NoC router). Currently the simulator adopts the 2D-mesh topology and source-based multicasting routing mechanism.

The routing strategy is straight: a spike from the source neuron is represented by one NoC packet from the source core to the destination (the X-Y routing strategy is used by default). Usually the digital identity of the spiking neuron is contained by the packet, which is called the address event representation (AER) method [55]. Before issuing, the source router will look up a local routing table to locate the target NoC node(s). If N ($N\geq 1$) nodes are located (it means that the set of target neurons has been distributed into more than one core), N packets will be issued. Accordingly, if M neurons can be occupied in one core, the size of a local routing table is $M\times N$. On the other side, on receipt of a spike, a core will decode the ID of the source and deliver it to all internal neurons, according to the aforesaid core steps. Apparently, the fan-in of one neuron is limited by the axon number (denoted as L) of a core's memory array, while the maximal fan-out is $L\times N$.

The NoC packet is simple, which just represents the arriving of a spike from some neuron. Correspondingly, a packet consists of only one network *flit* (32-bit width): the first 20-bit is the neuron ID and the subsequent 7 bits illustrate the issuing time (a simulated SNN cycle is used as the unit; we believe it is impossible that a spike will be delayed by more than 128 SNN cycles[5]), so that the receiver can judge whether the incoming spike is in time or not. The remaining 5 bits are reserved, which can be extended to support inter-chip communication.

In addition, the common scratchpad memory can be used as the routing table rather than expensive CAM (content addressable memory): neuron IDs are defined as a series of consecutive integers; thus, they can be used as memory addresses to access scratchpad.

IV. Implementation and Evaluation

In this section, we demonstrate the use of our toolsets on two different hardware platforms. The first one is a fabricated neuromorphic chip named **TIANJI** [22], and the second is a processing-in-memory (PIM) architecture design named **PRIME** [23].

• **TIANJI** [22] is an experimental CMOS neuromorphic processor, produced by the $110nm$ technology. The running frequency is $100MHz$ and the total dynamic power consumption is $120mW$. A chip contains 6 cores connected by a 2×3 mesh NoC; each core supports 256 simplified LIF neurons in the time division multiplexing mode. On-chip SRAM is used to store synaptic weights and other parameters. Moreover, multiple chips can be connected by the Serial Peripheral Interface to construct a larger mesh. Both the fan-in and fan-out of one neuron are 256. By default, the bit-width of synaptic weight is 8 and the calculating precision is 16-bit. According to the description of NoC in Section III-C, the size of a routing table is 256×1.

• **PRIME** [23] is a novel PIM architecture built upon emerging metal-oxide resistive random access memory (ReRAM) for NN applications. It exploits how crossbar structured ReRAM has both data storage and NN computation capabilities. Then, in the ReRAM based main memory design, a portion of the ReRAM crossbar arrays in each bank is enabled to work as an accelerator for NN applications with supported peripheral circuit design. Its processing scheme is similar to that of the analog core (in Section III-C). Since PRIME is designed for ANNs, we mainly verify the training function on it.

[5]Biological neurons tend to fire at a slow rate (measured in terms of hertz and usually less than 1000Hz), while modern electronics operates at multiple gigahertz or several hundred megahertz. Thus one SNN cycle usually includes 10^6 or 10^5 chip cycles.

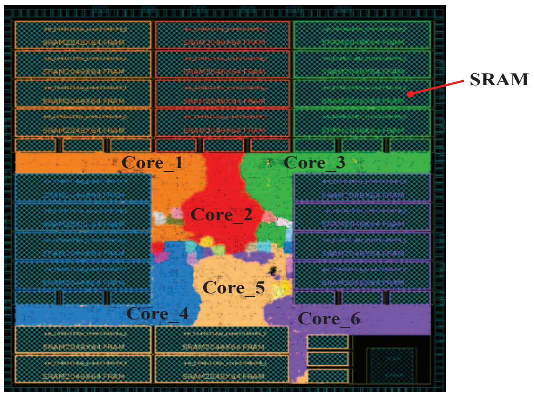


Fig. 7. The TIANJI CMOS neuromorphic processor [22].

A. Simulator

A software clock-driven simulator has been achieved to support features described by Section III-C. Especially, we reference the Noxim [56] to implement the NoC module, including all detailed key components. As a NoC packet contains only one flit, the router queue management has been simplified.

The simulator has two input files: The first describes configurations of the simulated SNN (after *mapping*), including the distribution of neurons/axons among cores and NoC, and the corresponding synaptic weight values, etc. The second contains input spikes, which are either spike sequences for input neurons or some predefined spike generating functions (like a *Poisson* generator).

The following parameters can be configured: the scale of the simulated chip (including the number of cores per chip and the mesh topology, number of neurons/axons per core), the running frequency and the number of chip cycles per SNN cycle, the core type and corresponding timing information (in cycles) and NoC parameters. Moreover, it supports to connect several chips together to simulate a larger scale NN. The simulator integrates with *NVMain* (a cycle accurate main memory simulator for emerging non-volatile memories on the architectural level) [57] to get memory-related runtime information of time and power consumption (for cores of Type_DIG). To do so, the simulator implements an asynchronous software interface to generate memory access requests of *NVMainRequest* (defined by NVMain), as well as the corresponding callback function to handle responses.

For power consumption of the NoC, we use McPAT [58] as the estimation tool; for lookup tables in routers and on-chip SRAM synaptic arrays (for cores of Type_SRAM), the CACTI cache tool [59] is used (components of a typical cache that would not be needed have been excluded).

For cores, the corresponding parameters can be configured as real values from the TIANJI chip. The exception is the analog core model. We reference [54] for timing and power information. In detail, [54] designed a memristor-based computation unit with the scale of 256×1024 and synapses of 4-bit-wide; the peripheral circuit (including the DAC and driver) is included. Its crossbar array can be divided into two pairs of 256×256 (4-bit values) arrays: each pair can construct an 8-bit computing unit for the positive or negative weights respectively; their outputs will be fed into a subtraction unit to get the difference signal. Larger arrays can be constructed in this way as each input signal is just 0 or 1. Main simulation configurations are shown in Table I.

TABLE I
MAIN SIMULATION CONFIGURATIONS (45NM CMOS TECHNOLOGY, 200MHZ)

Item	**Properties**	**Description**
Core number	4∼16	Connected by the 2D-mesh
Crossbar scale	256×256×8 512×512×8 1024×1024×8	axon number × neuron number × weight width
Storage array/core	64KB∼1MB	Depending on the neuron & axon number/core and weight width
Calculation latency and power of a LIF neuron	8, 9, 10 cycles/neuron; 0.33, 0.78, 1.54 nJ/neuron	Values for three crossbar scales are given. Valid for Type_*SRAM&DIG*. Accessing latency for weight values is excluded
Latency and power of a lookup (router-) table	256∼1024 entries; 0.96∼1.36 pJ/access;	Each router has a lookup table and each table entry represents a core ID (6-bit)
Average processing latency and energy consumption of a packet on a router	0.151 nJ/packet	N/A
Transfer latency of one NoC hop	1 cycle	The transfer width is 32-bit (one flit)
Average processing latency and energy consumption of a core of Type_*ANA*	3 cycles/neuron; 0.03 nJ/neuron	Two 256×256 (8-bit values) array

B. NN models and transformation tool

Quite a few NN models have been used. The first two are ANNs of the MNIST [60] handwritten digit recognition (widely used as a benchmark for machine-learning studies): one is MLP-style and the other is CNN-style. The next four SNNs are extracted from the Nengo simulator [33] (it follows the NEF [29] framework to map a wide range of neuro-computational dynamics to SNNs). Among these four SNNs, Nengo_basal_ganglia has a complex topology with 15 populations, which can demonstrate the capability of our method to deal with such complex cases, while Nengo_integrator is a recurrent network. The seventh is a benchmark SNN from the Brian simulator [34]. The last one is an example of computational neuroscience, a neural circuit of multisensory information integration [61]. It is also a recurrent SNN. All NNs are presented in our description way; the information is in Table II.

We have developed the transformation tool, including the functions of low precision computation and connection sparsification and layer insertion. For mapping, we just distribute highly-communicating neurons onto nearby cores.

C. Evaluation

1) Evaluation metrics.: Besides error-rates and energy consumption, one essential metric that we can get from the

TABLE II
NN MODELS

Name	**Descriptions**
MNIST MPL	An MLP (784 × 50 × 10 × 10)
MNIST CNN	A seven-layer CNN: the first is a 28 × 28 input layer; the 2nd is a convolutional layer with 2 convolution kernels (5 × 5) and its output is 2×24×24; the 3rd is sum pooling and the output is 2 × 12 × 12; the 4th is another convolutional layer with 2×8×8 output; the 5th is another sum pooling with 2 × 4 × 4 output, fully connected to the 6th layer (120 neurons); Layer 6 and output layer (10 neurons) are fully connected.
Nengo addition	An SNN contains three populations, which add two floating-point inputs (represented by the two populations) together; the value range is between 0 and 1. Each population has 800 neurons.
Nengo squaring	A floating-point-squaring SNN contains two populations. The first stands for a time varying input connected to the second; the latter squares the input value. The range is between -1.0 and 1.0. Each population has 800 neurons.
Nengo integrator	An SNN of one-dimensional neural integrator contains two populations. The first population stands for a time varying integer input and is connected to the second. The latter is a recurrently connected population that constantly sums the input. Each population has 800 neurons.
Nengo basal ganglia	It models basic behaviors of the basal ganglia [62], which can be regarded as an action selector that chooses the maximum among three inputs. There are in total 1500 neurons and 15 populations.
Benchmark	It contains multiple populations, connected to each other randomly
Continuous attractor neural network (CANN)	It is a biologically realistic neural network to combine visual and vestibular cues to infer heading direction. This model consists of two reciprocally connected neural networks; each network includes 500 neurons

toolchain is *effective_chip_speed*, which is inversely proportional to the number of chip cycles per simulated SNN cycle. For example, a real-time neuromorphic system means the simulated network is working as 'fast' as the real biological system, usually up to 1000Hz; one simulated SNN cycle contains chip-frequency/1000 chip cycles. Apparently, with the speed increase, more spikes per unit time will be inserted into the NoC and may cause traffic congestion to inversely hinder the further improvement. Thus, the maximum *effective_chip_speed* (for a given chip and NN) means that if the speed is faster than this value, some spikes cannot reach the target node in one SNN cycle.

2) Transformation results: We have trained all models of Table II and results are given in Table III. For each model, three types of hardware constraints have been considered. The first is about the TIANJI chip: the fan-in and fan-out of one neuron are both 256 and the bit-width of synaptic weight is 8, which is expressed as '256×256×8' in the table. The second is about PRIME; its bit-width of synaptic weight is 8[6]. The third is based on the configurations of our simulator, which usually contains three types ('256×256×8', '512×512×8' and '1024×1024×8'; the first one is just the same as TIANJI).

We also present the NN scale after transformation, which is expressed in the form of '$v_1/v_1/v_1/v_1$'. Each value stands for the total number of neurons, layer number, the number of occupied cores[7] and the status of layer insertion respectively. For the fourth item, '0' means there is no extra layer. Or, a parenthesized sequence of '1' and '2' is used to represent the NN layers in order; '2' means that there is a new layer inserted before the current position. Moreover, it is necessary to mention that, in Table III, the error rate of MNIST cases is the recognition error rate (comparing with its golden model).

TABLE III
TRANSFORMATION RESULTS

Name	**Hardware constraints**	**Scale (transformed)**	**Error rate**
MNIST MLP	256 × 256 × 8	70/3/6/0	3.87%
		541/4/8/(2,1,1)	0.90%
	512 × 512 × 8	70/3/4/0	2.54%
		541/4/5/(2,1,1)	0.63%
	1024×1024×8	70/3/3/0	2.46%
		541/4/4/(2,1,1)	0.40%
	256 × 256 × 8 (for PRIME)	7/3/6/0	10.88%
		581/4/10/(2,2,2)	1.45%
	1024×1024×8 (for PRIME)	70/3/3/0	7.22%
		581/4/4/(2,2,2)	1.34%
MNIST CNN	256 × 256 × 8	2450/7/17/ (1,2,1,1,1,1)	1.94%
	2048 × 2048 × 8 (for PRIME)	1951/9/9/ (1,1,1,2,2,2)	7.42%
Nengo addition	256 × 256 × 8	100/1/1/0	6.50%
	512 × 512 × 8	200/1/1/0	5.50%
	1024 × 1024 × 8	400/1/1/0	3.77%
	256 × 256 × 8 (for PRIME)	250/2/2/(2)	6.64%
	1024 × 1024 × 8 (for PRIME)	1000/2/2/(2)	5.64%
Nengo squaring	256 × 256 × 8	200/1/1/0	1.56%
	512 × 512 × 8	400/1/1/0	1.24%
	1024 × 1024 × 8	800/1/1/0	1.08%
	256 × 256 × 8 (for PRIME)	400/1/2/0	4.78%
	512 × 512 × 8 (for PRIME)	400/1/1/0	1.32%
Nengo integrator	256 × 256 × 8	100/1/1/0	8.51%
	512 × 512 × 8	200/1/1/0	2.46%
	1024 × 1024 × 8	400/1/1/0	1.66%
	256 × 256 × 8 (for PRIME)	400/1/2/0	4.57%
	512 × 512 × 8 (for PRIME)	400/1/1/0	4.57%
Nengo basal ganglia	256 × 256 × 8	2100/15/18/(3 layers inserted)	1.79%
	256 × 256 × 8	2100/15/18/(3 layers inserted)	2.38%
CANN	256 × 256 × 8	4000/5/20/0	12.05%
	512 × 512 × 8	4000/5/10/0	12.01%
	1024 × 1024 × 8	4000/5/7/0	10.7%
	256 × 256 × 8 (for PRIME)	4000/5/20/0	14.1%
	512 × 512 × 8 (for PRIME)	4000/5/10/0	13.2%
	1024 × 1024 × 8 (for PRIME)	4000/5/7/0	13.2%

[6]In PRIME, the data width of input and output signals between layers is 6-bit by default and there is no constraint on the fan-in and fan-out of one neuron. But if the connection number is larger than 256, the simulation speed will be impaired.

[7]For PRIME, one "core" refers to a memristor crossbar.

For other cases, the relative absolute error of output values has been used. The first error type is not sensitive to the second. For example, for the best MNIST case, the relative output error is over 5% while its recognition error rate is only 0.40%. Some preliminary conclusions or findings can be gotten:

(1) From the aspect of error rate, layer insertion (by software) is more effective than expanding the hardware core.

Taking MNIST_MLP as the example, the rate of the case of $256\times256\times8$ with one layer inserted is 0.9%, while the rate of $1024\times1024\times8$ without any new layer is 2.46%. Even for quite a few examples, it is almost impossible to achieve acceptable accuracy without any new layer. We believe the reason lies in that, hardware constraints also include the difference of neuron models. Thus an MLP with hidden layers can bridge this gap, while a SLP is not always feasible.

(2) The strategy of training the current layer with data generated from the trained pre-node(s) can prevent error accumulation.

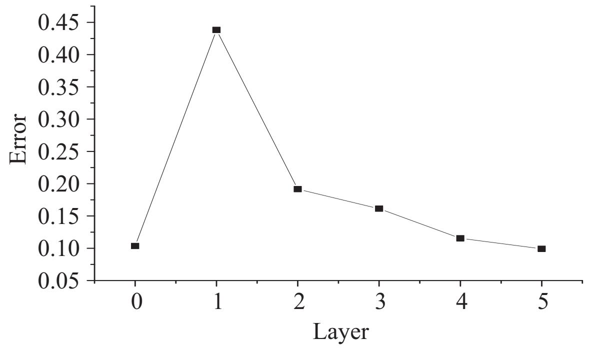


Fig. 8. Error rates along the transformation flow

We even find that if some middle layer has a large error, it is still possible for the following layers to decrease it with this strategy. Figure 8 shows the relative absolute errors of each layer of MNIST_CNN: Although the convolution layers and pooling layers have large errors (up to 44.8%), the last two fully connected layers decrease the error to 9.9% (the corresponding recognition error rate is 1.94%).

(3)Connection sparsification precedes the step of scaling for lower precisions.

The reason lies in that sparsification may cause larger errors and then it is more beneficial to train the NN with floating points. Figure 9 is such an example of a unit NN. The initial error rate in Step 2 increases (comparing with the error after Step 1) due to sparsification. Fortunately, this rate can be still reduced through the training with floating points in this step. In contrast, lower precisions have less impact on the rate, which can be seen from Step 3 in Figure 9: its initial rate approximately equals with the error after Step 2 and is kept stable during this epoch.

(4)Usually there are some neural parameters that can be scaled synchronously without affecting output.

Taking TIANJI as an example, the firing rating will remain unchanged as the weight, threshold and leakage current increase proportionally, while its expression accuracy can be improved. Accordingly, before the step of scaling, we use this method to choose some proper factor to scale all parameters into the range that matches the hardware constrains best. From the aspect of scaling, one advantage of TIANJI lies in that its scaling mechanism is more flexible (as the LIF model has the threshold parameter to control scaling), while for PRIME the scaling factor can only be the integer power of 2.

3) Simulation for TIANJI: Here we have completed two kinds of tests. The first is about the verification of simulation precision. We configure our simulator to mimic the target chip (Type_SRAM cores are used) and then compare behaviors of some NNs running on the simulator and on the accurate RTL model of TIANJI, respectively. Results show that it is basically consistent with the latter.

The second is to use simulation results to guide the setting of *effective_chip_speed*: we try to increase this speed under the premise of meeting the time requirement. From the accurate simulator, we can get detailed logs of spike transmissions. Then, we calculate the average value of the maximum transmission delay of all spikes in each SNN cycle and multiply the average by a factor to determine the maximum speed. In practice, the factor is usually set to 1.5; almost no spike will be delayed.

Without loss of generality, speed information of benchmark NNs with different scales (other neural configurations are the same, including the connection probability between populations, input pattern, etc.) illustrated in Figure 10(A); all results have been normalized. From the result, we can see that with the increase of the NN scale (represented by the total number of neurons), the maximum *effective_chip_speed* decreases gradually. The reason is that the connection number is proportional to the square of the number of neurons; thus, with the scale increase, the spike-number per unit time (defined as the active degree illustrated in Figure 10(A)) will increase much faster, which may cause traffic congestion and then reduce the effective speed.

The effect of the active degree on the speed has been confirmed further. For the biggest NN (it contains 1536 neurons and has occupied all 6 cores), we give the relationship between these two factors in Figure 10(B). As the ratio of the average weight to the spiking threshold is roughly proportional to the active degree, it is used to stand for the latter. Thus, this figure

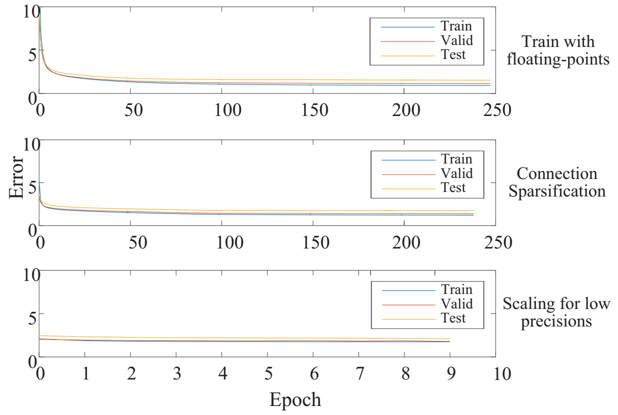
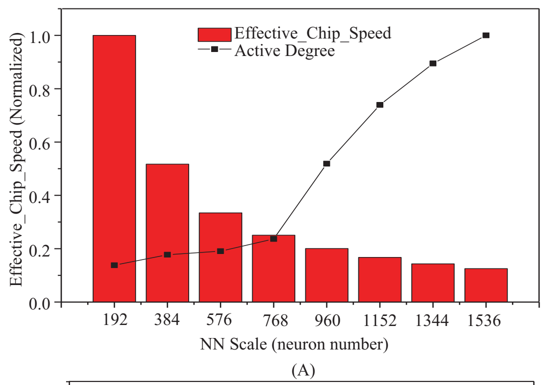


Fig. 9. Error rates in Training steps 1&2&3. (The y-axis stands for error rates and the x-axis represents the training epochs.)



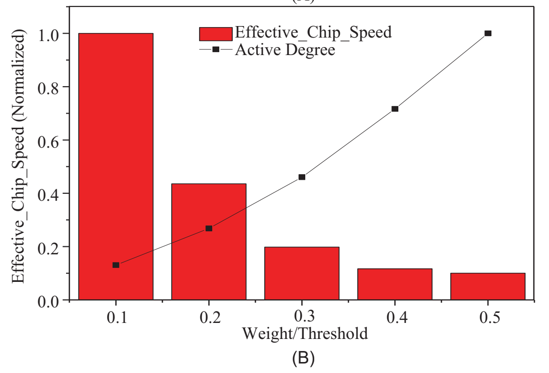
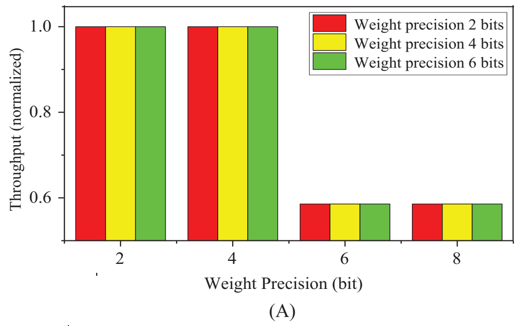


Fig. 10. Effective chip speed and active degree.)



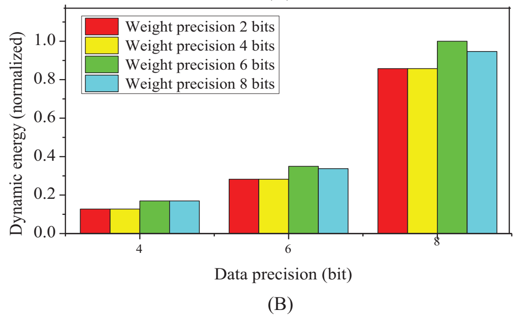


Fig. 11. Throughput and dynamic energy under different precisions for PRIME.)

TABLE IV
ERROR RATES OF MNIST_MLP WITH DIFFERENT WEIGHT/DATA PRECISIONS

Weight \ Data	4 bis	6 bits	8 bits
8 bits	2.71%	1.34%	1.24%
6 bits	2.92%	1.68%	1.54%
4 bits	4.49%	3.30%	2.13%
2 bits	72.03%	30.36%	27.67%

shows the expected relationship.

4) Co-design for PRIME: Due to the split-merge mapping mechanism, there is no constraint on the NN connectivity in PRIME. Thus we explore the correlation between error rates and hardware precisions (including the data widths of input / output signals and weight values) and consumption, which is beneficial to optimize this morphable architecture's configuration. Without loss of generality, we present recognition errors of MNIST_MLP with different precisions in Table IV. It is as expected that the higher the precisions, the lower the error rate. Moreover, higher precisions will introduce more hardware overheads; the corresponding results of circuit simulation are illustrated in Figure 11. We can see that if the weight precision changes from 6 bits to 4 bits, the throughput is increased by 71.15% (Figure 11(A)); if the data precision changes from 6 to 8, dynamic energy is increased by 2 times. On the other hand, the impact of data precisions on the throughput is small, while the impact of weight precisions on energy consumption is limited.

5) Co-design for TIANJI architecture: We configure the simulator to emulate neuromorphic chips with different core types and NoC scales; cores' hardware constraints can be set on demand, too. Therefore, we are able to run corresponding transformed NNs in Table III on the simulator with a proper configuration. Afterward, runtime information of speed and power can be obtained, which is helpful to present insights on neuromorphic architectures.

Without loss of generality, here we demonstrate such information of MNIST_MLP and Nengo_add/ Nengo_ squaring/Nengo_integrator. In the upper part of Figure 12(A), along the x-axis, the maximum effective_chip_ speed of each case with different layer insertion and different core types (Type_SRAM&_DIG&_ANA) and different hardware constraints ('256×256×8', '512×512×8' and '1024×1024×8') is presented; in the lower part, the information of power is given, too. All results are normalized. Moreover, to compare the effect of layer insertion with that of no insertion, corresponding error rates have been given at the bar top. In Fig.11B, such information of three NN models from Nengo is presented. From simulation results, some preliminary conclusions can be drawn:

(1) Although from the aspect of error rate, layer insertion is more effective than expanding the hardware core (as mentioned above), from the aspect of co-design, the software solution will cause more energy consumptions and lower the effective speed apparently.

Taking MNIST_MLP on cores of Type_SRAM as an example, compared with the case of 256×256×8 (no layer insertion), the error rate of the case of 256×256×8 (layer insertion) has been reduced to 23.26% while the power dissipation increases about 300% and the maximum speed is about 13.8% of the original. In contrast, the error rate of the case of 512×512×8 (no layer insertion) has been reduced to 66%; its power dissipation increases about 109% and the maximum speed is about 88.3%. Thus, the energy consumption of the case of 256×256×8 (layer insertion) is about an order of

magnitude higher. We think the reason is that the performance per watt of the hardware-based solution is usually much higher than that of the software solution (just as in the similar situation in the traditional computer architecture).
(2) Cores of Type_ANA (analog cores based on the memristor crossbars) are the most effective.

Compared with other types with the same configuration, this type owns the highest speed and the lowest power dissipation: its speed can reach 2 to 2.66 times as much as that of the other two, while the core power dissipation is only 1/13 to 1/25 of the others.
(3) NoC optimization is essential.

For Type_SRAM, the ratio of NoC power to the total is from 64.25% to 9.9% with the scale increase (256×256 to 1024×1024); for Type_DIG, the ratio is from 78.23% to 18.03%; for the last type, it is from 97.90% to 74.10%. Accordingly, improving the NoC is more conducive to both the layer insertion strategy and the use of cores of Type_ANA. On the one hand, layer insertion tends to occupy more on-chip cores so that more communications will be caused. On the other hand, this improvement can further highlight the advantage of analog cores, especially when the scale is small.

D. Discussion

Although our simulator is based on the crossbar structure, our methodology is a generic one, not constrained by the underlining architecture types. This is because, from the aspect of the transformation workflow, a target hardware is abstracted as a dot-product processing unit with similar constraints, like the maximum fan-in/fan-out, limited precision, and various neuron models. Accordingly, several adaption steps will be used to transform the network under specific constraints.

In addition, the training-based transformation workflow and techniques are not constrained by the network size (the experiments on the relative small NNs are partially due to the fact that the scale of the prototype TIANJI chip is small). For larger NNs, we use a conservative strategy that fully expands the NN without any connection pruning. This strategy uses 3 layers of crossbars to occupy an NN unit, the in-between one is used for the blocked matrix-multiplication computation, while the first is for issuing inputs to the middle layer and the last for gathering outputs. It solves the problem on fan-in&fan-out limitations without any loss of information (in contrast, the current ‘connection sparsification’ strategy is relatively aggressive that prunes connections with tiny weights). In fact, we have employed the full expansion strategy for PRIME: as PRIME has special adders and supports flexible data transmission through the bus, connection sparsification has been omitted and we do perform blocked matrix multiplication. We also carefully transfer the weight value from original network as a good initialization. Other steps are the same, whether the NN is large or small. The preliminary result shows that for the large CNN case VGG16 [63] transformed under the constraints of TIANJI chip, the extra recognition error is about 2.36%.

Finally, since we retrain each NN unit iteratively and each unit has fewer parameters to fit, fewer iterations are needed for convergence. The entire transformation is much faster than designing and training NN for specific hardware, especially for larger NN. Using VGG16 as an example, the runtime of transforming one layer is about 5 minutes to 1 hour (depends on layer size) using Theano (v0.7) over a GTX580 GPU. Designing and training an NN of that scale will cost weeks.

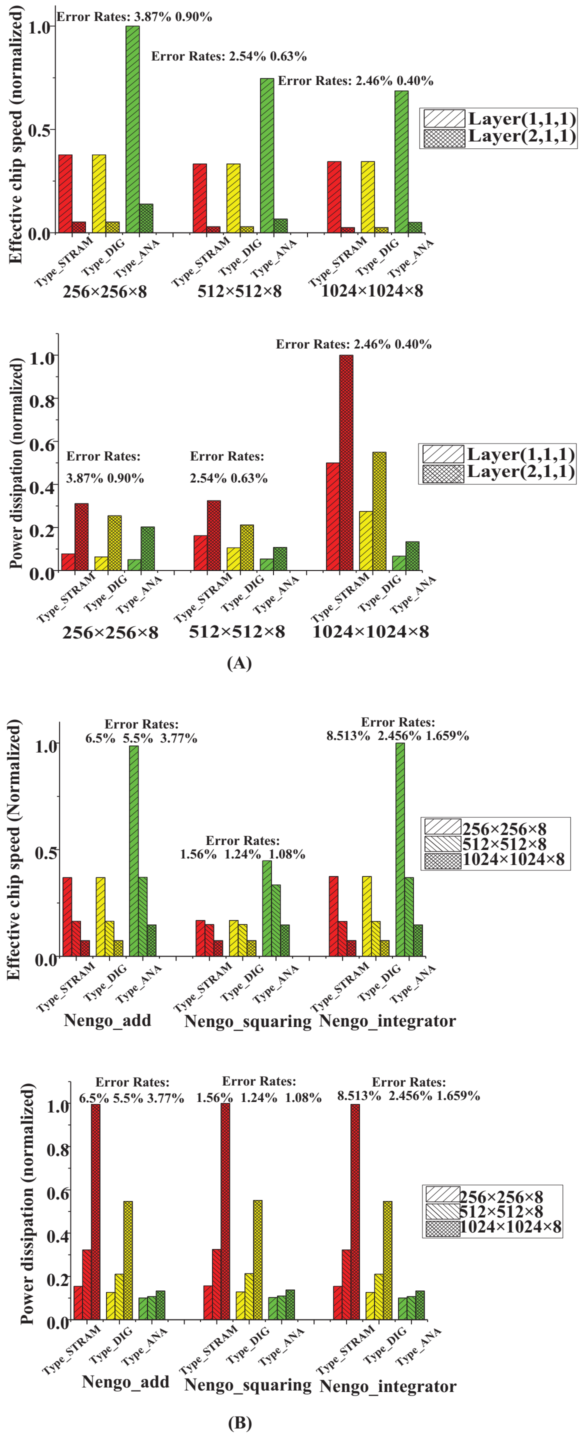


Fig. 12. Effective chip speed and power dissipation with various NN configurations.

V. Conclusion and Future Work

This paper presents a toolset for neuromorphic systems. Compared to existing methods, it is focused on the feature of decoupling NN applications from underlying execution substrates, such that it can support multiple types of NNs, including SNNs and traditional ANNs. The toolset has been validated on a real neuromorphic chip and a processor-in-memory architecture design for ANNs. It has been also used to explore the co-design space specific to neuromorphic chips, which is beneficial to present insights for the optimized design of neuromorphic architectures.

Moreover, we are working on the following studies on NN transformation: (1) The representation layer will be enhanced for users to provide some hints for the transformation procedure, such as the target error range or her/his inclination (preferring the lower error or fewer hardware consumptions), which can be used as the transformation target. (2) Quite a few NN pruning technologies will be used to improve efficiency; (3) More NN examples will be tested, not only to demonstrate the toolchain's capability but also to discover its limitations and sensitivities to diverse inputs.

Acknowledgment

The work is supported by the Science and Technology Plan of Beijing under Grant No. Z161100000216126, and the Brain Inspired Computing Research of Tsinghua University under Grant No. 20141080934.

References

[1] National Academy of Engineering, "Reverse-engineer the brain," 2012.

[2] W. G. amd H. Sprekeler and G. Deco, "Theory and simulation in neuroscience," *Science*, vol. 338, no. 6103, pp. 60–65, 2012.

[3] H. Markram, "The blue brain project," *Nature Reviews Neuroscience*, vol. 7, no. 2, pp. 153–160, 2006.

[4] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, 2014.

[5] S. B. Furber, D. R. Lester, L. A. Plana, J. D. Garside, E. Painkras, S. Temple, and A. D. Brown, "Overview of the spinnaker system architecture," *IEEE Transactions on Computers*, vol. 62, pp. 2454–2467, Dec 2013.

[6] B. V. Benjamin, P. Gao, E. McQuinn, S. Choudhary, A. R. Chandrasekaran, J. M. Bussat, R. Alvarez-Icaza, J. V. Arthur, P. A. Merolla, and K. Boahen, "Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations," *Proceedings of the IEEE*, vol. 102, pp. 699–716, May 2014.

[7] K. Wendt, M. Ehrlich, and R. Schüffny, "A graph theoretical approach for a multistep mapping software for the facets project," in *Proceedings of the 2Nd WSEAS International Conference on Computer Engineering and Applications*, CEA'08, (Stevens Point, Wisconsin, USA), pp. 189–194, World Scientific and Engineering Academy and Society (WSEAS), 2008.

[8] S. Carrillo, J. Harkin, L. J. McDaid, F. Morgan, S. Pande, S. Cawley, and B. McGinley, "Scalable hierarchical network-on-chip architecture for spiking neural network hardware implementations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, pp. 2451–2461, Dec 2013.

[9] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, "Pudiannao: A polyvalent machine learning accelerator," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, (New York, NY, USA), pp. 369–381, ACM, 2015.

[10] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, (Washington, DC, USA), pp. 609–622, IEEE Computer Society, 2014.

[11] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 269–284, ACM, 2014.

[12] Y. H. Chen, T. Krishna, J. Emer, and V. Sze, "14.5 eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 262–263, Jan 2016.

[13] M. Prezioso, F. Merrikh-Bayat, B. D. Hoskins, G. C. Adam, K. K. Likharev, and D. B. Strukov, "Training and operation of an integrated neuromorphic network based on metal-oxide memristors," *Nature*, vol. 24, no. 521, pp. 61–64, 2015.

[14] B. Li, Y. Shan, M. Hu, Y. Wang, Y. Chen, and H. Yang, "Memristor-based approximated computation," in *Low Power Electronics and Design (ISLPED), 2013 IEEE International Symposium on*, pp. 242–247, Sept 2013.

[15] G. Snider, R. Amerson, D. Carter, H. Abdalla, M. S. Qureshi, J. Leveille, M. Versace, H. Ames, S. Patrick, B. Chandler, A. Gorchetchnikov, and E. Mingolla, "From synapses to circuitry: Using memristive memory to explore the electronic brain," *Computer*, vol. 44, pp. 21–28, Feb 2011.

[16] S. Yu, Y. Wu, R. Jeyasingh, D. Kuzum, and H. S. P. Wong, "An electronic synapse device based on metal oxide resistive switching memory for neuromorphic computation," *IEEE Transactions on Electron Devices*, vol. 58, pp. 2729–2737, Aug 2011.

[17] M. Hu, H. Li, Q. Wu, and G. S. Rose, "Hardware realization of bsb recall function using memristor crossbar arrays," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pp. 498–503, June 2012.

[18] Y. Kim, Y. Zhang, and P. Li, "A reconfigurable digital neuromorphic processor with memristive synaptic crossbar for cognitive computing," *J. Emerg. Technol. Comput. Syst.*, vol. 11, pp. 38:1–38:25, Apr. 2015.

[19] M. Hu, H. Li, Y. Chen, Q. Wu, and G. S. Rose, "Bsb training scheme implementation on memristor-based circuit," in *Computational Intelligence for Security and Defense Applications (CISDA), 2013 IEEE Symposium on*, pp. 80–87, April 2013.

[20] B. Liu, M. Hu, H. Li, Z.-H. Mao, Y. Chen, T. Huang, and W. Zhang, "Digital-assisted noise-eliminating training for memristor crossbar-based analog neuromorphic computing engine," in *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pp. 1–6, May 2013.

[21] B. Liu, H. Li, Y. Chen, X. Li, T. Huang, Q. Wu, and M. Barnell, "Reduction and ir-drop compensations techniques for reliable neuromorphic computing systems," in *Computer-Aided Design (ICCAD), 2014 IEEE/ACM International Conference on*, pp. 63–70, Nov 2014.

[22] L. Shi, J. Pei, N. Deng, D. Wang, L. Deng, Y. Wang, Y. Zhang, F. Chen, M. Zhao, S. Song, F. Zeng, G. Li, H. Li, and C. Ma, "Development of a neuromorphic computing system," in *2015 IEEE International Electron Devices Meeting (IEDM)*, pp. 4.3.1–4.3.4, Dec 2015.

[23] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Wang, Y. Liu, and Y. Xie, "Processing-in-memory in reram-based main memory," in *2016 Intl. Symposium on Computer Architecture, also as SEAL-Lab Technical Report - No.2015-001*.

[24] Z. Du, D. D. Ben-Dayan Rubin, Y. Chen, L. He, T. Chen, L. Zhang, C. Wu, and O. Temam, "Neuromorphic accelerators: A comparison between neuroscience and machine-learning approaches," in *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, (New York, NY, USA), pp. 494–507, ACM, 2015.

[25] S. K. Esser, A. Andreopoulos, R. Appuswamy, P. Datta, D. Barch, A. Amir, J. Arthur, A. Cassidy, M. Flickner, P. Merolla, S. Chandra, N. Basilico, S. Carpin, T. Zimmerman, F. Zee, R. Alvarez-Icaza, J. A. Kusnitz, T. M. Wong, W. P. Risk, E. McQuinn, T. K. Nayak, R. Singh, and D. S. Modha, "Cognitive computing systems: Algorithms and applications for networks of neurosynaptic cores," in *Neural Networks (IJCNN), The 2013 International Joint Conference on*, pp. 1–10, Aug 2013.

[26] A. Amir, P. Datta, W. P. Risk, A. S. Cassidy, J. A. Kusnitz, S. K. Esser, A. Andreopoulos, T. M. Wong, M. Flickner, R. Alvarez-Icaza, E. McQuinn, B. Shaw, N. Pass, and D. S. Modha, "Cognitive computing

programming paradigm: A corelet language for composing networks of neurosynaptic cores," in *Neural Networks (IJCNN), The 2013 International Joint Conference on*, pp. 1–10, Aug 2013.

[27] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G. J. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha, "Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, pp. 1537–1557, Oct 2015.

[28] P. Merolla, J. Arthur, R. Alvarez, J. M. Bussat, and K. Boahen, "A multicast tree router for multichip neuromorphic systems," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 61, pp. 820–833, March 2014.

[29] C. Eliasmith and C. H. Anderson, *Neural Engineering: Computation, Representation, and Dynamics in Neurobiological Systems*. The MIT Press, 2004.

[30] B. Happel and J. Murre, "Design and evolution of modular neural network architectures," *Neural Networks*, vol. 7, no. 6-7, pp. 985–1004, 1994.

[31] X. Jin, *PARALLEL SIMULATION OF NEURAL NETWORKS ON SPINNAKER UNIVERSAL NEUROMORPHIC HARDWAR*. Ph.D. thesis. University of Manchester, 2010.

[32] M. L. Hines and N. T. Carnevale, "Neuron: a tool for neuroscientists," *Neuroscientist*, vol. 7, no. 2, pp. 123–135, 2001.

[33] T. C. Stewart, B. Tripp, and C. Eliasmith, "Python scripting in the nengo simulator," *Frontiers in Neuroinformatics*, vol. 3, no. 7, 2008.

[34] M. Stimberg, D. F. M. Goodman, V. Benichoux, and R. Brette, "Equation-oriented specification of neural models for simulations," *Frontiers in Neuroinformatics*, vol. 8, no. 6, 2014.

[35] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau, and A. V. Veidenbaum, "A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors," *Neural Networks*, vol. 22, no. 5-6, pp. 791–800, 2009.

[36] H. Paugam-Moisy and S. Bohte, *Handbook of Natural Computing*, ch. Computing with Spiking Neuron Networks, pp. 335–376. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[37] J. J. Wade, L. J. McDaid, J. A. Santos, and H. M. Sayers, "Swat: A spiking neural network training algorithm for classification problems," *IEEE Transactions on Neural Networks*, vol. 21, pp. 1817–1830, Nov 2010.

[38] N. G. Pavlidis, O. K. Tasoulis, V. P. Plagianakos, G. Nikiforidis, and M. N. Vrahatis, "Spiking neural network training using evolutionary algorithms," in *Neural Networks, 2005. IJCNN '05. Proceedings. 2005 IEEE International Joint Conference on*, vol. 4, pp. 2190–2194 vol. 4, July 2005.

[39] Y. Xu, X. Zeng, and S. Zhong, "A new supervised learning algorithm for spiking neurons," *Neural Computation*, vol. 25, pp. 1472–1511, June 2013.

[40] Y. Cao, Y. Chen, and D. Khosla, "Spiking deep convolutional neural networks for energy-efficient object recognition," *International Journal of Computer Vision*, vol. 113, no. 1, pp. 54–66, 2014.

[41] P. Merolla, J. Arthur, F. Akopyan, N. Imam, R. Manohar, and D. S. Modha, "A digital neurosynaptic core using embedded crossbar memory with 45pj per spike in 45nm," in *Custom Integrated Circuits Conference (CICC), 2011 IEEE*, pp. 1–4, Sept 2011.

[42] Y. Cao and S. Grossberg, "Stereopsis and 3d surface perception by spiking neurons in laminar cortical circuits: A method for converting neural rate models into spiking models," *Neural Networks*, vol. 26, pp. 75–98, 2012.

[43] F. Folowosele, R. J. Vogelstein, and R. Etienne-Cummings, "Towards a cortical prosthesis: Implementing a spike-based hmax model of visual object recognition in silico," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 1, pp. 516–525, Dec 2011.

[44] J. A. Perez-Carrasco, C. Serrano, B. Acha, T. Serrano-Gotarredona, and B. Linares-Barranco, "Spike-based convolutional network for real-time processing," in *Pattern Recognition (ICPR), 2010 20th International Conference on*, pp. 3085–3088, Aug 2010.

[45] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," *IEEE Micro*, vol. 33, pp. 16–27, May 2013.

[46] R. S. Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger, "General-purpose code acceleration with limited-precision analog computation," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pp. 505–516, June 2014.

[47] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin, "Snnap: Approximate computing on programmable socs via neural acceleration," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 603–614, Feb 2015.

[48] A. Hashmi, A. Nere, J. J. Thomas, and M. Lipasti, "A case for neuromorphic isas," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, (New York, NY, USA), pp. 145–158, ACM, 2011.

[49] A. P. Davison, D. Bruderle, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger, "Pynn: a common interface for neuronal network simulators," *Frontiers in Neuroinformatics*, vol. 2, no. 11, 2009.

[50] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," *CoRR*, vol. abs/1502.02551, 2015.

[51] M. Courbariaux, Y. Bengio, and J. David, "Low precision arithmetic for deep learning," *CoRR*, vol. abs/1412.7024, 2014.

[52] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, "Neural networks with few multiplications," *CoRR*, vol. abs/1510.03009, 2015.

[53] S. Pande, F. Morgan, G. Smit, T. Bruintjes, J. Rutgers, B. McGinley, S. Cawley, J. Harkin, and L. McDaid, "Fixed latency on-chip interconnect for hardware spiking neural network architectures," *Parallel Computing*, vol. 39, no. 9, pp. 357 – 371, 2013. Novel On-Chip Parallel Architectures and Software Support.

[54] T. M. Taha, R. Hasan, C. Yakopcic, and M. R. McLean, "Exploring the design space of specialized multicore neural processors," in *Neural Networks (IJCNN), The 2013 International Joint Conference on*, pp. 1–8, Aug 2013.

[55] R. Paz, F. Gomez-Rodriguez, M. A. Rodriguez, A. Linares-Barranco, G. Jimenez, and A. Civit, *Computational Intelligence and Bioinspired Systems: 8th International Work-Conference on Artificial Neural Networks, IWANN 2005, Vilanova i la Geltrú, Barcelona, Spain, June 8-10, 2005. Proceedings*, ch. Test Infrastructure for Address-Event-Representation Communications, pp. 518–526. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.

[56] V. Catania, A. Mineo, S. Monteleone, M. Palesi, and D. Patti, "Noxim: An open, extensible and cycle-accurate network on chip simulator," in *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*, pp. 162–163, July 2015.

[57] M. Poremba, T. Zhang, and Y. Xie, "Nvmain 2.0: A user-friendly memory simulator to model (non-)volatile memory systems," *IEEE Computer Architecture Letters*, vol. 14, pp. 140–143, July 2015.

[58] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, (New York, NY, USA), pp. 469–480, ACM, 2009.

[59] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, (Washington, DC, USA), pp. 3–14, IEEE Computer Society, 2007.

[60] Y. LeCun, C. Cortes, and C. J. Burges, "The mnist database of handwritten digits," 1998.

[61] W.-h. Zhang, A. Chen, M. J. Rasch, and S. Wu, "Decentralized multisensory information integration in neural systems," *The Journal of Neuroscience*, vol. 36, no. 2, pp. 532–547, 2016.

[62] T. C. Stewart, X. Choo, and C. Eliasmith, "Dynamic behaviour of a spiking model of action selection in the basal ganglia," in *Proceedings of the 10th international conference on cognitive modeling*, pp. 235–40, Citeseer, 2010.

[63] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.