# Bridging the Gap Between Neural Networks and Neuromorphic Hardware with A Neural Network Compiler

Yu Ji*, YouHui Zhang*‡, WenGuang Chen*‡, Yuan Xie†
*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*—Different from training common neural networks (NNs) for inference on general-purpose processors, the development of NNs for neuromorphic chips is usually faced with a number of hardware-specific restrictions, such as limited precision of network signals and parameters, constrained computation scale, and limited types of non-linear functions.**

**This paper proposes a general methodology to address the challenges. It can transform an existing trained, unrestricted NN (usually for software execution substrate) into an equivalent network that meets the given hardware constraints, which decouples NN applications from target hardware. The original NN is expressed as a computational graph (CG) that will be fine-tuned gradually according to a topological ordering, transforming to the target CG. Multiple techniques are proposed to address the architecture-specific restrictions, including a multilayer-perceptron(MLP)-based universal approximator, a data re-encoding method, a split-and-merge network reconstruction method, and a multi-phase weight-tuning algorithm.**

**We have built such a software tool that supports both spiking neural networks (SNNs) and traditional artificial neural networks (ANNs). Its effectiveness has been demonstrated with a fabricated neuromorphic chip and a processing-in-memory (PIM) design. Tests show that the extra inference error caused by this solution is insignificant and the transformation time is much shorter than the retraining time. In addition, parameter-sensitivity evaluations have been studied to explore the tradeoffs among network error and resource utilization for different transformation strategies, which could provide insights for co-design optimization of neuromorphic hardware and software.**

## I. Introduction

Designing neuromorphic chips with the state-of-the-art Very-Large-Scale-Integration (VLSI) technologies have been investigated as power-efficient and high-performance alternatives to implement neural network applications on general-purpose computing platform such as CPU/GPU or FPGA. However, programming such neuromorphic chips are much more difficult, because the hardware-specific constraints are very different from their general computing counter parts: ① Due to the utilization of hardware resource and/or the capability of analog computing (e.g. some memristor-based designs [1], [2], [3], [4], [5], [6]), the precision of input and output signals of neurons is usually limited, as well as ② the precision of network parameters (like synaptic weights). In contrast, general-purpose processor could use floating-point numbers. ③ The present fabrication technology limits the fan-in and fan-out of one neuron (and thus limits the corresponding computation scale). ④ Moreover, the diversity of supported nonlinear function (or neuron model) is limited. For example, for TrueNorth chips [7], the maximum matrix that a core can handle is $256 \times 256$ and only the leaky integrated-and-fire (LIF) neuron model is supported.

One straight approach to this problem is to expose the hardware topology and limitations to the software developer directly. For instance IBM has provided a TrueNorth-specific training mechanism [8] to construct the whole NN from top to bottom, which is "constraining and training". This method has several drawbacks: ① It binds NN development and application to the specific target hardware. ② Developers hardly benefit from existing massive networks produced by mature training methods for general-purpose processors, while re-training the entire NN may take a very long time.

Another approach is to introduce a domain-specific Instruction Set Architecture (ISA) for NN accelerators, like the Cambricon [9] ISA from the DianNao chip family [10], [11], [12], [13]. But this approach does not resolve the problem on how to create an NN program using this ISA.

In addition, NN compression is a related technique; a large number of accelerator designs employ this method to get tradeoff between hardware consumption, software adaption and inference error. EIE [14] is such an instance: it extensively uses deep compression techniques [15] to reduce the NN size (including reducing the bit-width of synaptic weights), which is conducive to adapt NNs to chip. Another related work is NEUTRAMS [16] that mainly uses network compression technologies to convert the original network to adapt to the hardware. But compression is not a general programming methodology, especially for networks under severe constraints.

Consequently, in this paper we proposes a new method with flexibility, better applicability, and easy convergence.

First, it decouples the neuromorphic computer system into two levels for better flexibility. ① The first level is located between programming and the execution substrate. NN experts could develop and train NN using any existing NN development framework [17], [18], [19] without considering hardware constraints. After the training, NN information such as neural model type/parameters and network topology, can be extracted to construct the corresponding computational graph (CG) as the input of our method. ② The second level decouples

software and hardware to make our proposal applicable to a variety of NN chips. We abstract the target hardware as a set of connected *physical cores*; the operation that a single core can accomplish is defined as *core_op*. Without loss of generality, in this paper, *core_op* refers to vector-matrix multiplication plus activation function (or spiking neuron model) that meets hardware restrictions, as they are the main computational components of many existing NN chips. Many chips face all (or part) of the similar constraints, as presented in Table I. Note that our methodology can be used for other chips not listed in the table, as discussed later in Section IV-F. Accordingly, NN inference by hardware is abstracted as the forward process of a CG and all the vertices represent *core_ops*.

We propose a transform workflow to convert a trained, unrestricted NN (expressed as a CG, which is referred to as *golden model*) into an equivalent CG of *core_ops* through the NN fine-tuning method. This method is called "training and constraining". Afterwards, the obtained CG will be mapped onto *physical cores* of target hardware efficiently. In this way, our solution decouples NN application's specification from execution substrate. To make the NN fine-tuning easy to converge under strict conditions, we also propose multiple techniques as described below:

The essential is to construct a CG of *core_ops* to approach the golden model. As *core_ops* are not comparable to their software counterparts, it is reasonable to enlarge the graph scale and/or complicate the topology properly to improve the capability, especially under strict conditions. Accordingly, some techniques, including an MLP-based universal approximator, data re-encoding based on *autoencoder* (a particular type of NN), and a *split-and-merge* method, are proposed to deal with issues on the unsupported functions, the limited IO precision and the constrained computation scale respectively, which may increase the graph scale. To some extent, the principle of 'trade scale for capability' is used.

In addition, it is necessary to note that this principle is not contradictory to network compression: on one hand, usually there is significant redundancy for DNN models, and compression can remove it. On the other hand, diverse types of NN and dataset have different requirements for optimized weight-/IO-precision of the underlying hardware. It is difficult to determine in advance whether or not the limited hardware precision are enough for any given NN. Thus, both are useful and orthogonal. We can compress the network first, then convert it, or vice versa.

Another point is to set parameters of *core_ops* properly to make the constructed CG equivalent to the golden model. Thus, after the above processing, the CG of *core_ops* will be fine-tuned part by part according to a topological ordering of the golden model to reduce transformation error, under the premise of satisfying the hardware weight precision.

In contrast, from the point of view of system abstraction, the EIE and EIE-like solutions support the general abstraction of NN models and the Cambricon ISA is a hardware abstraction (the specific way of abstraction is different from ours), while we introduce the two abstractions both. Moreover, this transformation procedure could be viewed as *compilation* of traditional computer systems that converts high-level programs (the hardware-independent, trained NN models) into instructions that hardware can understand (the equivalent CG of *core_ops*), and the transformation tool could be called an NN compiler. As a summary, this paper has achieved the following contributions:

1) An NN transformation workflow is presented to complete the aforementioned technologies to support different types of NNs. The result is a CG of *core_ops* that meets hardware constraints, which would be mapped onto the target hardware.
2) Such a toolchain is implemented to support two different hardware designs' constraints, a real CMOS neuromorphic chip for ANN&SNN, TianJi [20], and a PIM design built upon metal-oxide resistive random access memory (ReRAM, a kind of memristor that has been widely studied for NN acceleration) for ANN, PRIME [6]. The general optimization mapping strategy is given, too.
3) We complete quite a few evaluations of various metrics. The extra error caused by this process is very limited and time overhead is much less (compared to the whole training process of the original NN). In addition, its sensitivity to different configurations and transformation strategies has been explored comprehensively.

| Chip | Weight | IO | Scale | Nonlinear |
|---|---|---|---|---|
| TianJi [20] | 8-bit | 8-bit | $256^2$ | Configurable |
| PRIME [6] | 8-bit | 6-bit | $256^2$ | ReLU<br>Max Pooling |
| DianNao [10] | 16-bit | 16-bit | $16^2$ | Configurable |
| TPU [21] | 8-bit | 8-bit | None | ReLU<br>Max Pooling<br>etc. |
| TrueNorth [7] | 2-bit | Spiking | $256^2$ | LIF |

TABLE I
HARDWARE LIMITATIONS OF NN CHIPS

The rest of this paper is organized as follows. Section II presents related work and Section III introduces the concepts of NN, training, fine-tuning and CG. Section IV is the main part, which gives the outline of the whole transformation & mapping methodology and details of key technologies, from the point of view of CG conversion. Section V presents the implementation and evaluations. Section VI concludes.

## II. Related Work

There are two types of neural network chips. The first type focuses on the traditional ANNs: they are custom architectures to accelerate mature artificial intelligence (AI) algorithms, especially machine learning algorithms of DNN. The second usually supports SNNs to yield higher biological reality (more information will be given in Section IV-F). In order to distinguish, we call the former NN accelerators and the latter neuromorphic chips in the following content.

### A. NN compression

NNs are both computational intensive and memory intensive, making them difficult to deploy on hardware systems with limited resources. At the same time, there is significant redundancy for deep learning models [22]. Therefore,

quite a few studies have been carried out to remove the redundancy, which can be divided into three types: *pruning neurons*, *pruning synapses* and *pruning weights*. Both *weight-quantization* [23], [24], [25] and *weight sharing* [26] are pruning weights. For pruning synapses, deep compression [15] prunes the network by retaining only important connections. Diversity Network [27] prunes neurons, which selects a subset of diverse neurons and fuses the redundant neurons into the selected ones.

Moreover, there are some research efforts that study extremely compact data representations (including the I/O precision or weight precision or both) for NN computation. Binarized neural networks [1,3] that investigates the use of 1-bit data types for weight and ternary neural networks [4,5] using 2 bits belong to this category, which have achieved comparable accuracies to state-of-the-art full precision networks for some data sets. However, these methods are effective for specific networks, not common development scenarios.

### B. NN accelerators

EIE [14] extensively employs the above compression techniques and then proposes a dedicated engine to perform inference on the compressed network model. ESE [28] is its follow-up work that implements a speech recognition engine with compressed Long-Short-Term-Memory model on FPGA.

The DianNao chip family is a series of state-of-the-art NN accelerators. DianNao [10] designs an accelerator for DNNs and CNNs that exploits data reuse with tiling. The inherent sharing of weights in CNNs is explored in ShiDianNao [13]. PuDianNao [11] supports seven machine learning algorithms. DaDianNao [12] is a custom multi-chip machine-learning architecture. From the aspect of internal data representation and computation, they support fixed-point computation rather than 32-bit floating-point to reduce hardware cost. Accordingly, some corresponding retraining or tuning process is needed to adapt software for hardware. They use a load-store ISA [9] to decouple synaptic weight storage from neuron processing logic to avoid the limitation on connection number and improve the utilization of processing logic. This family also supports compressed, sparse NNs through a dedicated accelerator, Cambricon-X [29].

Minerva [30] uses fine-grained data type quantization and dynamic operation pruning to further reduce the resource consumption of DNN hardware. Strip [31] relies on bit-serial compute units to enable support for per-layer, dynamically configurable computation precision for DNN. DNPU [32] supports dynamic fixed-point with online adaption and weight quantization, too. Other studies include Origami [33], Convolution Engine [34], RedEye [35], NeuroCube [36], neuFlow [37] and quite a few FPGA-based designs [38], [39], [40], [41].

All the above studies are based on the traditional Complementary Metal-Oxide-Semiconductor (CMOS) technology. Another category is using novel nonvolatile memory devices (usually memristors) to perform neural computations in memory. PipeLayer [42] is such an accelerator for CNNs that supports both training and inference, while PRIME [6] and ISAAC [43] are for inference. Other work includes stand-along accelerators [1], [44], [45], co-processor [46] and many-core or NoC [47], [48] architecture. Since the new memory technology is not very mature, there is no systematic programming method.

The main computational components of these chips usually contain vector-matrix-multiplication units and nonlinear functions, which is the basis of our hardware abstraction.

Moreover, there are quite a few development frameworks for neural networking computing. Some (like Theano [18], TensorFlow [19], CNTK [49], Torch [50], MXNet [51], etc.) describe NN as computation graphs. For a trained NN, the complete information can be extracted from whichever of them.

### C. Neuromorphic chips

TrueNorth [7], [52] is a digital neuromorphic chip for SNNs, based on a structure of tiled crossbar (each crossbar is of size $256 \times 256$, and supports binary-valued neurons and ternary-valued synapses). The programming paradigm, *Corelet* [53], is bound to the hardware platform: its recent study [8] proposes a TrueNorth-specific training mechanism to construct the whole NN from top to bottom. The parameters learned are then mapped to hardware [54] using *Corelet*.

EMBRACE [55] is a compact hardware SNN architecture, with the limited fan-out of individual neuron circuits. From the programming aspect, it follows the Modular Neural Network (MNN) computing paradigm [56]. Thus, it is not a general solution. Neurogrid [57] and FACETS [58] (including its successor BrainScaleS [59]) are analog/digital hybrid systems, whose development methods are both hardware-specific. SpiNNaker [60] is different: its toolchain is not bound to any computing paradigm. The reason is that it is based on the chip multiprocessor (CMP) of ARM cores. Thus, its neural computing is completed by software. The drawback is that the efficiency will be lower than the dedicated hardware. TianJi [20] is an experimental CMOS neuromorphic chip based on tiled crossbars and supports hybrid computing of ANN and SNN. A toolchain NEUTRAMS [16] is developed to map various NN models onto the chip. It also decouples application from hardware and completes SW/HW co-design for optimization. But its methodology is based on NN compression and is not suitable for large-scale network under strict constraints.

## III. Background

### A. Neural network, training and fine-tuning

Neural networks are a set of algorithms, modeled loosely after the human brain, that are designed to recognize patterns. They have in common ① multiple layers of nonlinear processing units (neurons) and ② the supervised or unsupervised training of feature representations in each layer. In this paper, the term 'training' always refers to supervised training.

Without loss of generality, we take an MLP as the example: an MLP consists of multiple layers of neurons, each layer fully connected to the next one. The forward computation of the each layer is $Y = f(W \cdot X + b)$, where $W$ is the weight matrix, $X$ is the input vector from the previous layer, $b$ is

the bias term and $f$ is an non-linear activation function (e.g. Rectified Linear Units, ReLU). From a computational point of view, it completes a dot-product operation and a bias addition before the activation. Moreover, we can merge the bias vector into the weight matrix and append 1 to the input vector to combine the first two operations together.

Other DNN types include convolutional neural network (CNN) that contains convolutional layers, pooling layers (e.g. max pooling, average pooling) and fully connected layers, and recurrent neural network (RNN), a class of NN where connections between neurons form directed circles.

For DNN, the most commonly used training method is back-propagation (BP), which employs the stochastic gradient descent (SGD) training algorithm to find the optimized weight values that can successfully minimize the error between predictions and real values. The training of NN models includes forward phase and backward phase: data samples move forward to get the predictions and errors, then the errors move backward to update weight parameters.

Sometimes weight values of trained models can be reused for other tasks or should be adjusted to satisfy some new constraints. Instead of training the model from scratch, we could reuse the learned model as the initial one and then train it with new samples or under new constraints. This process is called *fine-tuning*. In short, fine-tuning is training pre-trained networks to adapt to new requirement.

### B. Computational graph

Quite a few popular development frameworks [18], [19], [49] use CG, a directed acyclic graph, to represent NN computations. In a CG, vertices represent operations (e.g. dot-product, convolution, activation function, pooling, etc.) and immutable/mutable states [19] (e.g. the weight parameters associated), while edges represent the data dependency between vertices; both process or carry tensor data (multi-dimensional arrays).

For clarity, in this paper dot product, bias-addition and convolution are categorized as *weighted-sum* operation, and all of them can be computed in the form of matrix vector multiplication. For the first two, the computation of this form is straight, while details of the third are given in Section IV-D.

Moreover, any constant operand, including the trained weight matrix for any vertex of weighted-sum operation, is considered as part of the corresponding vertex as it can be viewed as the immutable state of the vertex.

Figure 1(a) shows an example CG of a CNN, which consists of convolution layers, pooling layers, and fully connected layers. Each layer in the graph is represented by one or multiple vertices, which represent the corresponding operations respectively, denoted as *Conv(weight)* (convolution operation with weight matrix), *Add(bias)*, *Dot(weight)* (dot-product operation with weight matrix), *ReLU* or *Max pooling*, etc.

## IV. Transformation Methodology

We first give the problem description formally. Then the whole workflow is outlined, before the details of key steps and techniques.

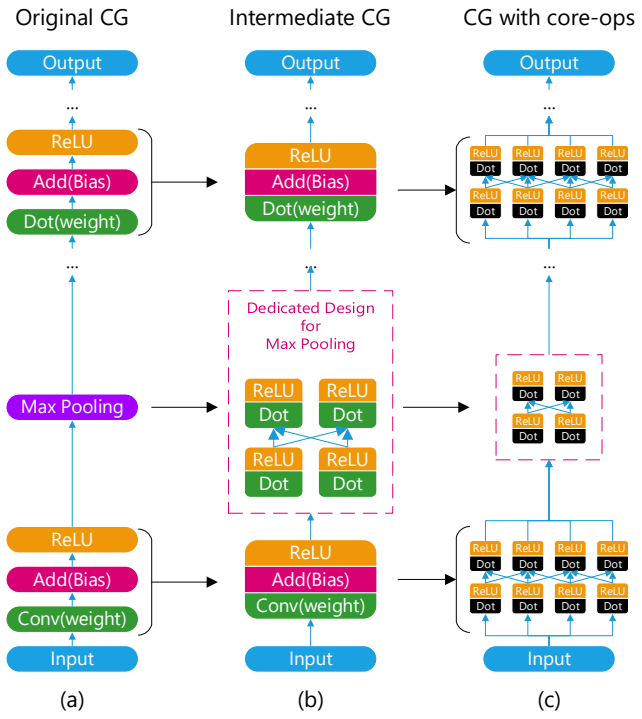


Fig. 1. A transformation example

### A. Problem description

As input, the trained NN is expressed as a CG, $G(V, E)$. The set of vertices is $V = \{v_1, \ldots, v_n\}$ and the edge set is $E = \{e_1, \ldots, e_m\} \in V \times V$.

The output is a CG including only *core_ops*, equivalent to $G(V, E)$. Each *core_op* performs the computation of $Y = f(W \cdot X)$, where $X$ and $Y$ are the input and output vectors and their sizes are $N$ and $M$ respectively. $N$ and $M$ are also the maximum fan-in and fan-out of one neuron constrained by hardware, which limits the computation scale. $W$ is its weight matrix of size $M \times N$, and $f$ is the activation function. Moreover, the I/O precision of all vector elements is $B$-bit.

Formally, *core_op* meets the following constraints:

- $N$, $M$ and $B$ are fixed.
- The value range of each element in $W$ is a finite set $S$. $S$ is either a fixed set or a configurable set with some parameter(s).
- Without loss of generality, only *ReLU* function ($f$) is supported.

Accordingly, the goal of transformation is to construct a CG of *core_ops*, including setting all the weight matrices properly, to make it equivalent to $G(V, E)$. Afterward, it will be mapped onto the target chip.

### B. The workflow outline

The proposed workflow involve 4 steps (Figure 2).

**Building CG.** According to the above description, it constructs $G(V, E)$ based on the input NN's information that includes the trained parameters, network topology, vertex information and training dataset.

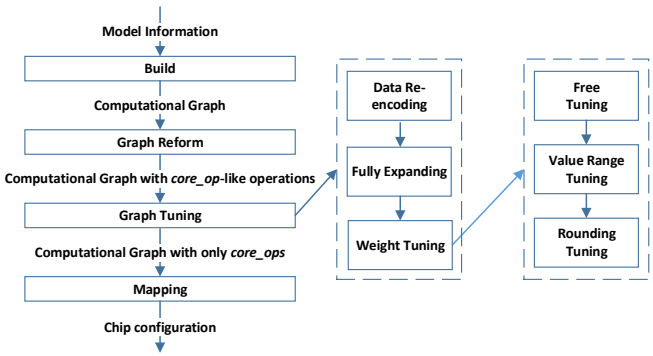


Fig. 2. Workflow of our proposal, from the input model to the output for chip. Step 'Graph Tuning' contains 3 sub-steps for different hardware restrictions respectively and the third sub-step has 3 fine-tuning phases.

**Graph Reform.** It constructs an intermediate CG, $G'(V', E')$, equivalent to $G(V, E)$; any vertex of $G(V, E)$ corresponds to a subgraph in $G'$. The operation of every vertex in $G'$ is in the form of weighted-sum plus activation function. In another word, all operations are *core-op*-like (as shown in Figure 1(b)), as they can be achieved by *core-ops* that support vector-matrix multiplication and activation, while hardware constrains have not yet been met. It is beneficial for subsequent processing. Details are in Section IV-C

**Graph Tuning.** Every vertex of $G(V, E)$ is traversed according to a topological ordering. In this order, the corresponding subgraph of $\boldsymbol{G'}$ is processed sequentially in the following sub-steps:

- Re-encoding transmitted signals on each edge of the subgraph to solve the precision problem of I/O;
- Processing all vertices of the subgraph to deal with the limitation on the scale of vector-matrix multiplication.

  After these two sub-steps, operations of the current subgraph have been achieved by one or more *core_ops* (Figure 1(c)). Accordingly, the next sub-step is
- Fine-tuning the weight matrix(matrices) of the *core_op(s)* to minimize transformation error, under the premise of satisfying the hardware weight precision.

Details are in Section IV-D

**Mapping.** Now we have built an equivalent CG composed of *core_ops* that meet hardware constraints, which will be mapped onto the target hardware efficiently.

### C. Graph Reform

Initially, $G(V, E)$ is slightly adjusted: any vertex that represents an activation function is merged with its previous vertex(vertices) whose computation is in the form of weighted-sum to form a *core_op*-like operation. Examples are presented in Figure 1: multiple vertices in one convolutional layer (Figure 1(a)) are merged into one vertex (Figure 1(b)), as well as the fully connected layer.

Afterwards, for all remaining vertices whose operations are neither in the form of weighted-sum nor supported by hardware directly, each of them will be approximated with an MLP, because the latter is a universal approximator; the activation function used is *ReLU* as the target hardware provides it. In another word, we will train an MLP to achieve the operation(s) of the original vertex; some common functions can be trained in advance.

Moreover, for those vertices that are not suitable to be approached this way because of huge resource consumption, some dedicated designs could be proposed.

For example, the resource consumption of max pooling increases exponentially with scale, if we just use such a general approximator. Consequently, a dedicated design is given. Max pooling can be built with max functions, each of which is completed as following: $\max(a, b) = \frac{1}{2}\operatorname{ReLU}(a + b) + \frac{1}{2}\operatorname{ReLU}(a - b) + \frac{1}{2}\operatorname{ReLU}(-a + b) + \frac{1}{2}\operatorname{ReLU}(-a - b)$

Thus, a simple max function with two variable can be implemented with a neural network of four hidden neurons. We could use this neural network as a basic building block to construct max functions with more variables.

In this way, we can construct an equivalent CG, $G'(V', E')$. Vertices of $G$ are either copied into $G'$ (including those after merge), or replaced with the equivalent MLPs. For the latter, the corresponding weight matrices are trained (or set up) in floating-point numbers.

Finally, the initial 'merge' operation is performed against $G'$. Then all vertices in $G'$ are just *core_op*-like operations. An example is given in Figure 1(b): besides the vertex merge, the max-pooling vertex is replaced with a dedicated subgraph of only *core_op*-like operations.

### D. Graph Tuning

$G(V, E)$ is traversed in a topological ordering (as a CG is directed acyclic graph); for every accessed vertex (if it has been merged with adjacent vertices in $G$ by the previous step, then they are handled as a whole), the following steps will be performed against the corresponding sub-CG of $G'$ in order.

**1.Data Re-encoding:** Since the golden model is trained using floating point numbers, direct rounding data to hardware I/O precision may distort or lost information. Thus, this sub-step aims to re-encode the data transmitted by each output edge of the vertex(vertices) of the current sub-CG with hardware I/O precision. *Autoencoder* is used to improve the situation.

An autoencoder is an NN with one hidden layer that can learn a constrained representation of a set of input data. The output layer has the same dimension as the input layer and is trained to approach the input: the computation from input to the hidden layer is encoding input data to the hidden layer's representation, while the computation from the hidden layer to output is decoding. Here as we use it to represent the original floating-point signals with the hardware I/O precision, the neuron number of the hidden layer may be greater than that of input/output layer; the specific value can be configured manually. Usually, a large hidden layer will improve the accuracy, but consume more resources. The tradeoff is evaluated in Section V.

For clarity, we take a vertex of vector-matrix multiplication plus activation function as the example to describe the process.

As shown in Figure 3(a&b&c), we add an autoencoder after the current vertex. The activation of its hidden layer is a round

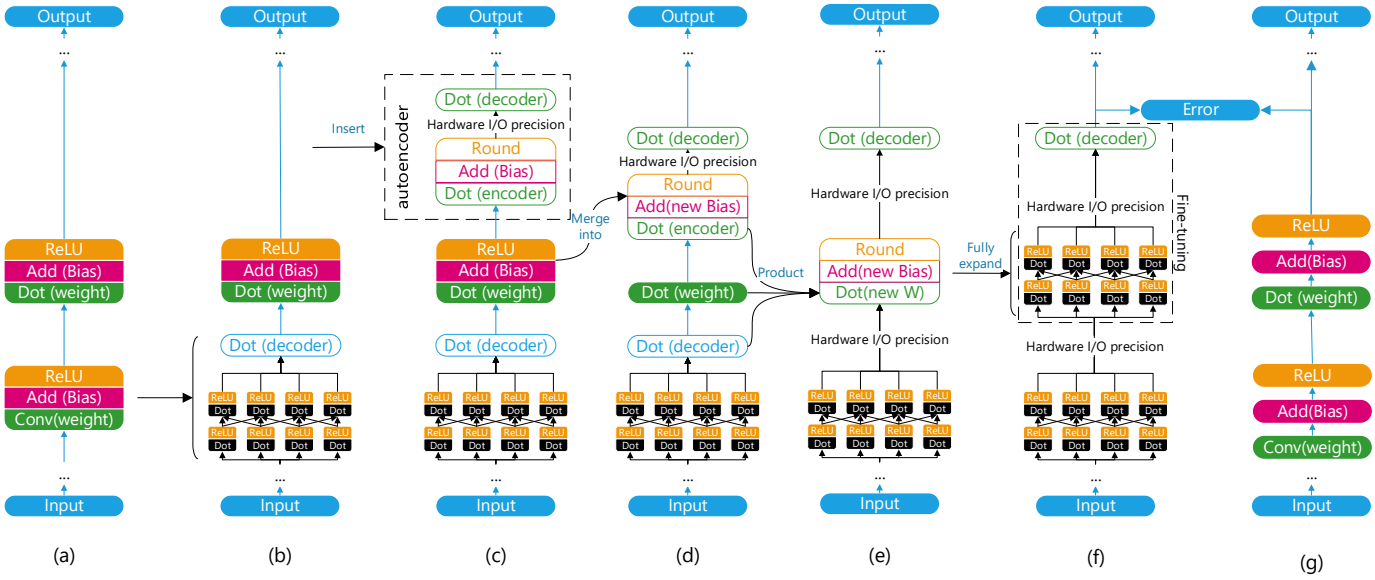


Fig. 3. Graph tuning of one vertex. (a) $G'$ with only *core_op*-like operations. (b) 'Graph Tuning' has been performed against all previous vertices. (c) Insert an 'autoencoder' after the current vertex. (d) Merge the bias and activation of the current vertex into the hidden layer of the autoencoder. (e) Multiply the three weight matrices together. (f) Fully expand the vertices with *core_ops*. (g) The original CG, $G$. The current subgraph (including the decoder of the current vertex) is fine-tuned to approach the corresponding vertex of $G$, with output of the previous tuned vertex as input. Repeat (b) to (g) for the rest vertices.

operation [1] which rounds the output to hardware I/O precision. Moreover, we design an optimized initialization method for the encoder and decoder as following.

For the input vector $X = \{x_1, \ldots, x_n\}$ (i.e. the output of the previous ReLU, as illustrated by Figure 3(c)), we build an autoencoder network with a hidden layer of $m \times n$ neurons ($m$ is a configurable integer). Suppose the $n$ dimensions of $X$ are independent and equally important, then each $x$ has $m$ hidden neurons to represent its low precision encoding; connections from $x$ to other hidden neurons are initialized to zero. A hidden neuron linearly scales the corresponding $x$ to $w^{(e)}x + b^{(e)}$ ($w^{(e)}$ and $b^{(e)}$ are the weight and bias term of the encoder respectively) and then rounds it to the I/O precision ($\{0, \ldots, 2^N - 1\}$). Any $x < \frac{-b^{(e)}}{w^{(e)}}$ will be rounded to 0 and any $x > \frac{2^N - 1 - b^{(e)}}{w^{(e)}}$ will be rounded to $2^N - 1$. Thus, one hidden neuron can well represent $x \in [\frac{-b^{(e)}}{w^{(e)}}, \frac{-2^N - 1 - b^{(e)}}{w^{(e)}}]$.

Now we have $m$ hidden neurons for each dimension; thus the best way to represent $x \in [0, x_{max}]$ (the output of ReLU is positive) with the $m$ hidden neurons is to divide the data range into $m$ adjacent and non-overlapping intervals, each of which corresponds to one hidden neuron. Namely, we properly initialize $w_i^{(e)}$ and $b_i^{(e)}$ for the the encoder of the $i$-th neuron of the $m$ to adapt $[\frac{-b_i^{(e)}}{w_i^{(e)}}, \frac{2^N - 1 - b_i^{(e)}}{w_i^{(e)}}]$ to the corresponding interval $[\frac{i x_{max}}{m}, \frac{(i+1) x_{max}}{m}]$. Thus $w_i^{(e)} = \frac{(2^N - 1)m}{x_{max}}$ and $b_i^{(e)} = -(2^N - 1)i$.

Accordingly, to decode and restore $x$, the decoder should scale the data back. Thus, its weight matrix is set to $w_i^{(d)} = \frac{1}{w^{(e)_i}} = \frac{x_{max}}{(2^N - 1)m}$.

Note that, if input $x < 0$ (outside of the encoded interval $[0, x_{max}]$), the initialized autoencoder will always return 0. It means that the autoencoder can also perform ReLU operation. Therefore, we could remove the redundant ReLU operation of the current vertex. Moreover, as shown in Figure 3(d), the bias term of the current vertex could also be encoded by the dot-product operation and merged into the bias term of encoder $b_i^{(e)}$. Namely, the new bias term of encoder becomes $b_i^{(e)} + w_i^{(e)} b$.

Finally, as shown in Figure 3(e), the decoder of the previous vertex, the dot-product and the encoder of the current vertex can be merged as one dot-product operation, whose weight matrix is the product of the three's matrices.

Till now, the input and output of the current vertex have been constrained to hardware I/O precision.

For vertices of convolution plus activation function, the process is similar. Instead of using dot-product operation as encoder and decoder, we use convolution instead, and the three convolutions can be merged into one as well. The initialization is also similar: the hidden layer has $m$ channels for each input channel, and only the center value of the encoder/decoder kernel is set to non-zero.

Owing to this step, we solve the limitation problem of I/O precision.

**2.Fully Expanding:** This step aims to achieve the operation of every vertex of the current sub-CG by *core_ops*, in order to avoid the hardware limitation on the vector-matrix-multiplication scale. We use the "split-and-merge" strategy: the weighted-sum operation of every vertex will be divided into several smaller operations, each of which can be held by a *core_op*.

[1]The round operation not only constrains the output precision, but also forces the output to be positive, which provides the non-linearity like the widely-used ReLU function.

For a vertex of vector-matrix multiplication plus activation function, as shown in Figure 3(f), we use two layers of smaller *core_ops* to construct an equivalent graph for the weighted-sum operation. The first layer is a computation layer, which stores weights and carries out the vector-matrix multiplications (after division). The second layer is a reduce layer, which gathers result from the former to output.

The division is straight: ① divide the weight matrix into small sub-matrices that satisfy the hardware limitation on scale; each is held by a *core_op* of the first layer. ② Divide the input vector into sub-blocks and transfer each sub-block to the corresponding sub-matrices(*core_ops*) at the same horizontal position and ③ gather results from the same column by the second layer.

For a convolutional case (suppose a kernel of size $k \times k$ convolves a $W \times H$ image from $m$ channels to $n$ channels), the $n$ channels of one pixel in the output side are fully connected to the $m$ channels of $k \times k$ corresponding pixels in the input side. This forms a small-scale vector-matrix multiplication of size $(m \times k^2) \times n$. There are $W \times H$ such small operations in the convolution case. Each input should be transferred to $k^2$ such small operations, while reduction is needless. If such a small operation is still too large for a *core_op*, we can divide the operation as the vector-matrix multiplication case does.

If there are some *core_ops* that are not fully used, we can distribute them onto one *physical core* to reduce resource consumption during the mapping step.

Till now, the computation of the current sub-CG has been achieved with *core_op(s)*; problems on the constrained I/O precision, unsupported functions, and the limited computation scale are solved. Next, the weight matrix(matrices) of the *core_op(s)* will be fine-tuned.

**3. Weight Tuning:** Based on previous steps, the initialization mode of the weight matrix(matrices) of the current sub-CG can be divided into two categories:

- If the sub-CG is not created by Step 'Graph Reform' (i.e. it is copied from $G$), the corresponding weight parameters from the golden model are used.
- For others, the initial values are set up by the 'Graph Reform' step.

Of course, in either case, all the weight matrices have been processed by subsequent steps, i.e. 'Data Re-encoding' and 'Fully expanding'.

However, some methods, including autoencoder and the MLP-based unsupported function handling, introduce transformation errors. In addition, activation functions used by $G(V, E)$ may be different from the hardware counterpart (i.e. the activation function of *core_op*), which also makes the initialization inaccurate. Therefore, some fine-tuning phases have to be taken to minimize the error, under the premise of satisfying the hardware weight precision constraint.

Further, it should be noted that the fine-tuning unit is a sub-CG (as marked out in Figure 3(f)). Fine-tuning will make its behavior approximate to the corresponding vertex in $G$.

Because a sub-CG is composed of one or more vertices of *core_ops* and each is equipped with a weight matrix, it can be fine-tuned as a common DNN: as shown in Figure 3(f&g), we use the output of the previous sub-CG as input to fine-tune the current sub-CG to approach the output of the corresponding vertex (*ReLU*) of the golden model, in order to avoid error accumulation. The output of the golden model and the previous sub-CG can be generated on demand (or cached in advance to improve the transformation speed).

Specially, target hardware usually puts strict constraints on weight storage since the latter occupies most of the hardware resources. Here we present a formal description of the weight matrix $W$: the value of each element $W_{ij}$ should be assigned to the closest value in a finite set $S$. $S$ is either a fixed set or a configurable set $S^P$ with parameter(s) $P$.

Three kinds of typical weight encoding methods, which have been widely used by real NN chips, are presented as following (in all cases, the size of $S$ is $N$):

- **Dynamic fixed-point**: $S^P = \{\frac{-2^{N-1}}{2^P}, \ldots, \frac{0}{2^P}, \ldots, \frac{2^{N-1}-1}{2^P}\}$ where $P$ represents the point position. This method is used by DNPU [32], Strip [31], TianJi-ANN [20], etc.
- **Fraction encoding**: $S^P = \{\frac{-2^{N-1}}{P}, \ldots, \frac{0}{P}, \ldots, \frac{2^{N-1}-1}{P}\}$, where $P$ is the threshold of the spiking neuron or the scale factor of the result. It is used by PRIME [6], and TianJi-SNN [20].
- **Weight sharing**: $S^{P_1,\ldots,P_N} = \{P_1, \ldots, P_N\}$, used by EIE [14].

Without loss of generality, suppose $W_{ij}$ is rounded to the $k_{ij}$-th element in $S^P$, denoted as $S^P_{k_{ij}}$. This step aims to find the best $P$ and to set $k_{ij}$ for each element in the weight matrix properly to minimize the transformation error. It is similar to *weight quantization* of network compression. Our contribution is that we generalize it to typical hardware cases and introduce several fine-tuning phases to deal with different parameter-setting issues separately.

For a sub-CG, three fine-tuning phases are taken in order: The first is to reduce the initialization error. The second is to determine the best value range of weight matrix (i.e. to choose the best $P$) and the last is to determine the best value from $S^P$ for each element (i.e. to choose the best $k_{ij}$). Each phase gets parameters from the previous one and fine-tunes them under certain constraints.

1) Free tuning

As mentioned above, the previous transformation steps have introduce error. Thus a fine-tuning operation without any constraint on weight precision is conducted first to reduce any existing error. In this procedure, all parameters and signals are processed as floating-point numbers, while the hardware activation function is used.

2) Value-range tuning

Now the precision constraint on weight is introduced. Accordingly, we need to choose the best value-range of the weight matrix (namely, the best $P$). Apparently, we will minimize $J(k, P) = \sum_{ij}(W_{ij} - S^P_{k_{ij}})^2$, which can be achieved by an iterative expectationmaximization (EM) algorithm:

- E-step: fix the current parameter $P^{(t)}$ and calculate $k_{ij}^{(t)} = \arg\min J(k|P^{(t)})$.
- M-step: fix $k_{ij}^{(t)}$ and calculate $P^{(t+1)} = \arg\min J(P|k^{(t)})$.

Then $W_{ij}$ is replaced with $S^P_{k_{ij}}$ where $k_{ij}$ is fixed and $P$ is the parameter (i.e. each element in the matrix is a function of $P$).

After the initialization, we fine-tune the sub-CG to optimize $P$. During this process, we maintain the precision of $W_{ij}$ first and then round it to $P_{k_{ij}}$ at every time $P$ is updated.

Further, for the weight sharing case mentioned above, the EM algorithm is just reduced to the k-means algorithm. If $S^P$ is a fixed set without any configurable parameter, this phase could be omitted.

3) Rounding tuning

The data set of weight value $S^P$ is fixed now. This procedure adjusts each weight matrix element to a proper element in this set. In another word, it aims to choose the best index $k_{ij}$ for $W_{ij}$. The training mechanism is: parameters are stored as floating point number. In the forward phase, any parameter is rounded to the closest element in $S^P$. During the backward phase, floating-point number is used. This mechanism is also employed by the above 'value-range tuning' phase.

After processing all the sub-CGs, we have transformed the golden model into an equivalent CG composed of *core_ops* that satisfies all the constraint conditions.

Finally, it is worth adding that we can fine-tune several successive sub-CGs in $G'$ simultaneously according to a topological ordering, and the above steps are still applicable. We will evaluate the effect of fine-tuning-granularity on transformation results in Section V.

### *E. Mapping*

The final equivalent CG needs to be deployed on the target hardware efficiently, which is a hardware-specific problem. Thus, we give the optimization principle here.

For NN chips that bind the neural computation and storage with *physical cores* (it is called the *weight stationary* computing mode, classified by [61]), this is a mapping problem to assign *core_ops* to *physical cores*. Moreover, several *core_ops* that are not fully used can also be distributed onto one *physical core*, as long as there are no data conflicts.

For chips whose *physical cores* are computing engines with flexible memory access paths to weight storage (usually work in time division multiplex mode), it is a mapping and scheduling problem to schedule each *core_op*'s task onto *physical cores*. Multiple *core_ops* could be mapped onto one core to increase resource utilization.

As we can get data dependencies and communication patterns between *core_ops* through the transformed CG, we could use these information to optimize the mapping or scheduling to minimize transmission overhead, e.g. putting densely-communicating cores close. TrueNorth has designed such an optimized mapping strategy [54].

Moreover, for those *core_ops* sharing weights (e.g. convolution vertices can be fully expanded to a lot of *core_ops* sharing the same weight matrix), we could map (or schedule) them to the same *physical core* to reduce data movement.

### *F. Others*

(1) SNN models

SNN, called the third generation of ANN, is a widely-used abstraction of biological neural networks. In addition to neuronal and synaptic states that traditional ANN has featured, it incorporates the timing of the arrival of inputs (called spikes) into the operating model to yield higher biological reality. SNNs of rate coding can emulate ANNs:

The spike count in a given time window can represent a numerical value within a certain range, like a traditional ANN does. Accordingly, the input of a synapse is a spike sequence of certain firing rate from the pre-neuron. After synapse computation, it is converted into the sum of currents that will be computed by the post-neuron. For those widely-used SNN models, the functions of their synapse and neuron computations usually own good continuity and are derivable in rate coding domain. Therefore, the popular SGD method can be used for training SNN: several recent studies [62], [63] have used the SGD algorithm to train SNNs directly or indirectly and achieved the state-of-the-art results for some object recognition tasks.

As our workflow is not dependent on the concrete NN type (ANN or rate-coding SNN), it can support SNN hardware and SNN models, too. For SNN models, the training data is the firing rate of each neuron.

(2) RNN models

RNN is an NN with some cycle(s). We could transform and fine-tune each operation inside an RNN as normal, and add an additional step to fine-tune the entire RNN after that.

(3) Adaptability analyses

Extra hardware features could be utilized. The utilization depends on the specific hardware and we present some examples as following:

- If hardware supports *reduction communication* or provides additional *adders*, the reduce layer in 'Fully expanding' would be needless.
- If some special function (e.g. max function) is supported, it can also be viewed as a type of *core_op*. Accordingly, the corresponding *core_op*-like operations can include max pooling and others that can be easily achieved by existing *core_ops*. It is no longer required to be approximated by the dedicated design or the MLP-based universal approximator.

## V. Implementation and Evaluation

### *A. Implementation*

We have implemented the tool to support different hardware constraints, including those of TianJi [20] and PRIME [6].

TianJi is fabricated with 120nm CMOS technology. The running frequency is 100MHz and the total dynamic power consumption is 120mW. TianJi chip supports both ANN and SNN modes. The numerical accuracy of weight value is 8-bit fixed-point and the scale of vector-matrix-multiplication is

$256 \times 256$. For ANN mode, the I/O precision is 8-bit that is cut from the 24-bit internal computation output; the cut range is configurable, thus its weight encoding strategy is dynamic fixed-point. For SNN mode, the minimal I/O precision is 1-bit, which can be extended to $n$-bit with $2^n$ cycles as the sampling window (as described in Section IV-F). The neuron model is a simplified LIF neuron model with a constant leakage and a threshold for firing; the weight encoding method can be viewed as fraction encoding. PRIME [6] is a memristor-based PIM architecture for ANN. The weight precision is 8-bit and the I/O precision is 6-bit. The scale of vector-matrix-multiplication is $256 \times 256$, too. The output range can be configured with an amplifier; thus its weight encoding can also be viewed as fraction encoding.

Quite a few NN applications, an MLP for MNIST dataset (784-100-10 structure, $98.2\%$ accuracy of full precision), LeNet-5 [64] for MNIST dataset ($99.1\%$ accuracy), a CNN [65] for CIFAR-10 dataset ($84.64\%$ accuracy [2]), AlexNet [66] and VGG16 [67] for ImageNet, have been respectively transformed and then deployed onto TianJi [20] and PRIME [6] to show the validation. The first three networks are trained by Theano [18]. Parameters of the next two CNNs for ImageNet are extracted from trained models of the Caffe Model Zoo directly, as well as the inference accuracies of full precision in Table II.

Without loss of generality, we take the mapping of the LeNet-5 for MNIST onto the TianJi system as an example.

One TianJi chip contains 6 cores connected by a $2 \times 3$ mesh NoC; each core supports 256 simplified LIF neurons and works in the weight stationary mode. The main body of a TianJi system is a Printed Circuit Board (PCB) including 16 chips. On the whole, all of the cores form a $12 \times 8$ 2D-mesh network, and there is a great gap between the delay/bandwidth of intra-/inter-chip communications.

The transformed CG consists of 497 *TianJi_core_ops*. Taking into account the weight reuse of convolution, 19 physical cores are actually occupied. We use the heuristic Kernighan-Lin (KL) partitioning algorithm for the mapping problem. It abstracts the mapping as a graph partition problem and tries to minimize communications across partition boundaries. Take the bipartition as an example: the input to the algorithm is a graph; the weight of each edge is the communication delay. The goal is to partition the graph into two disjoint subset $A$ and $B$ of equal size, in a way that minimizes the communication cost of the subset of edges that cross from $A$ to $B$.

First, a randomly generated initial mapping distribution is given.Second, the KL algorithm bi-partitions the mapped cores repeatedly till only the closest two cores are left in any of the final partition in the 2D-mesh. During this phase, partitions that minimize the communication cost between cores are remained; here the cost of an edge across boundary refers to its weight multiplied by the number of transmissions, as we can get the communication statistics from the transformed CG, including those information about the reused cores.

[2]As described by [65], with some special initialization method, the CNN accuracy can exceed 90%. Here we ignore it for simplicity, which does not affect our evaluation.

### B. Evaluation

The inference accuracies before and after transformation for TianJi and PRIME are given in Table II, which are given in the sixth column and the seventh column respectively. This table also shows the transformation accuracies of our tool (in the brackets of the seventh column), which takes the inference results of golden models as the ground truth.

We can see that they are very limited: all relative error rates are less than 3.9%. Moreover, it is necessary to note that, except for the NNs for TianJi-SNN, the number of occupied hardware neurons of every transformed NN is equal to the neuron number of the corresponding original NN, which means that our tool could achieve high transformation accuracy without increasing the NN scale. For SNN mode, as the I/O constraints are very strict, 4x hardware neurons are used for these two cases.

This toolchain also improves the development efficiency remarkably. For example, it can transform AlexNet in about 1 hour, while training it from scratch will take about 3~4 days. Specially, training the whole NN requires millions of iterations to converge, while our method only costs thousands of iterations to converge for each step, that is, takes 5 to 10 minutes to transform one layer. The reason lies in that, after partitioning, we fine-tune each unit one by one; each one is well initialized and much easier to converge than training the entire model. All evaluations are completed on a common PC server equipped with one Tesla P100 GPU.

In addition, as the large-scale NNs (e.g. those for ImageNet) cannot be occupied by the TianJi system because of the physical limit of chip capacity (a TianJi system contains 16 chips and a chip can occupy 1536 neurons), those related results are drawn from the cycle-accurate TianJi chip simulator.

(1) Accuracy vs. fine-tuning-granularity

We conduct experiments to explore the relationship between accuracy and fine-tuning-granularity. In another word, in the step of 'Graph tuning', we can fine-tune one or more successive sub-CGs in $G'$ simultaneously, even fine-tune the entire model as a whole. It looks like that increasing the fine-tuning scale per time will increase the search space, which may lead to a lower error rate, while consuming more computations and more time to converge. However, results show that coarse grained fine-tuning does not result in improved accuracy. For example, for CIFAR10-CNN, the inference accuracy (the I/O precision is 7 bits and the weight precision is 4 bits) is 83.07% as the fine-tuning-granularity is two sub-CGs (the accuracy is 83.14% as the granularity is one). If we tune the whole NN together, the accuracy is just 83.24%. Thus, one by one fine-tuning is an optimal strategy, which also means that the problem of error accumulation has been well solved. In this section, unless specifically noted, the fine-tuning-granularity is just one.

(2) Accuracy vs. resource consumption

From the transformation workflow, we can see that the step of 'Data Re-encoding' may introduce the most additional

| NN model | Chip | Weight encoding | Weight | I/O | FP accuracy | Accuracy (relative) |
|---|---|---|---|---|---|---|
| MNIST-MLP | TianJi-ANN | Dynamic fixed-point | 8-bit | 8-bit | 98.2% | 98.15%(99.95%) |
| MNIST-MLP | TianJi-SNN | Fraction encoding | 8-bit | 1-bit | 98.2% | 96.59%(98.36%) |
| MNIST-MLP | TianJi-SNN | Fraction encoding | 8-bit | 2-bit | 98.2% | 97.63%(99.42%) |
| MNIST-MLP | PRIME | Fraction encoding | 8-bit | 6-bit | 98.2% | 98.14%(99.94%) |
| LeNet-5 | TianJi-ANN | Dynamic fixed-point | 8-bit | 8-bit | 99.1% | 99.08%(99.98%) |
| LeNet-5 | PRIME | Fraction encoding | 8-bit | 6-bit | 99.1% | 99.01%(99.91%) |
| CIFAR10-CNN | TianJi-ANN | Dynamic fixed-point | 8-bit | 8-bit | 84.64% | 84.02%(99.26%) |
| CIFAR10-CNN | PRIME | Fraction encoding | 8-bit | 6-bit | 84.64% | 83.57%(98.74%) |
| AlexNet-ImageNet(top1) | TianJi-ANN | Dynamic fixed-point | 8-bit | 8-bit | 57.4% | 56.9%(99.13%) |
| AlexNet-ImageNet(top1) | PRIME | Fraction encoding | 8-bit | 6-bit | 57.4% | 55.2%(96.17%) |
| VGG16-ImageNet(top1) | TianJi-ANN | Dynamic fixed-point | 8-bit | 8-bit | 70.5% | 69.6%(98.72%) |
| VGG16-ImageNet(top1) | PRIME | Fraction encoding | 8-bit | 6-bit | 70.5% | 68.2%(96.74%) |

TABLE II
ACCURACY FOR NNS UNDER DIFFERENT RESTRICTIONS

resource overhead: when the neuron number of the hidden layer of autoencoder is $n\times$ as much as that of the input/output layer, the number of crossbar consumed of the whole NN will be $n^2\times$. The latter can also be considered as a direct indicator of area consumption and runtime overhead. Thus, we conduct experiments to explore the effect of *autoencoder*.

We use the MLP for MNIST dataset. Although this network is relative small, its conclusion is general for large-scale NNs because we fine-tune NNs part by part and each part is a small CG. To show the effect, we compare the inference accuracies without or with different scales of autoencoder. For the former, we simply scale and round the I/O signal values to make them suitable for I/O precision.

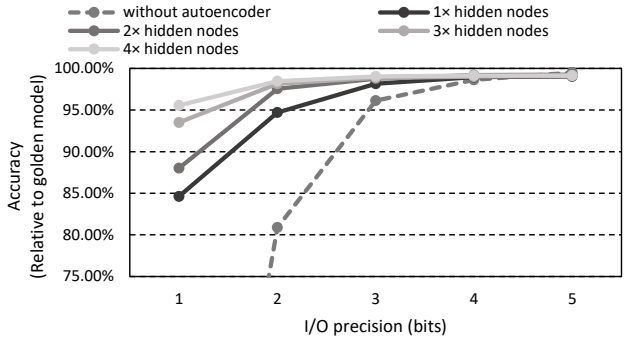


Fig. 4. Accuracy v.s. I/O precision under different transformation strategies.

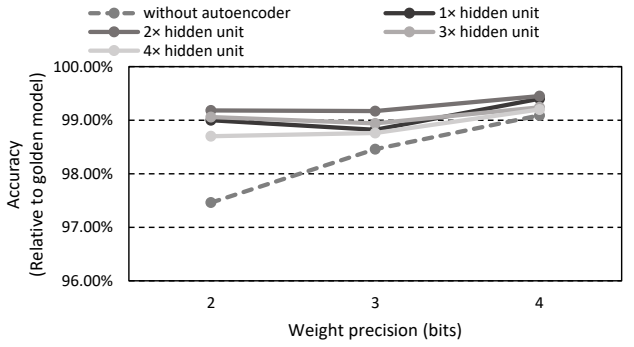


Fig. 5. Accuracy v.s. weight precision under different transformation strategies.

Figure 4 (Figure 5) lists the accuracy with different I/O (weight) precisions respectively (after transformation), without or with different scales of autoencoder, while no other constraints are introduced. Results show that this data re-encoding strategy is effective to improve the transformation accuracy under strict constraints.

In Figure 4, the accuracy of the transformed network without autoencoder drops significantly when I/O precision is less than 3 bits (different NNs may have different turning points). In contrast, when autoencoder is used (we assume the I/O limitation is only 1-bit and only 1x hidden neurons are used, namely, the most critical case), the accuracy is $84.63\%$ (if no autoencoder, the value is only $13.45\%$). With more hidden neurons, the accuracy continues to rise, which means our method could trade NN scale for capability.

Apparently, increasing the number of hidden neurons can only linearly increase the encoding ability, which is worse than increasing the I/O precision directly because the latter's encoding ability is $\propto (2^n)$. For example, using $2\times$ hidden neurons and 1-bit I/O does consume the same number of I/O ports with that of using $1\times$ hidden nodes and 2-bit I/O; the accuracy of the former is only $88.2\%$ while the latter is $94.71\%$. Thus, it looks like that the hardware had better provide enough I/O precision since rescuing the accuracy by software (using autoencoder) may cost more hardware resources, especially when the hardware I/O precision is less than the turning point. Anyway, this is a tradeoff between hardware consumption, software adaption and inference error.

Moreover, as illustrated by Figure 5, autoencoder is also able to rescue the accuracy loss caused by low weight precision. Compared with Figure 4, we can see that NNs are more tolerant of low weight precision than low I/O precision, since the latter can cause signal distortion directly. Figure 5 also shows that when the weight precision is 2-bit or more, using different scales of autoencoder does not change the accuracy apparently because it has already reached $99\%$.

(3) Impact of weight encoding methods

We have evaluated the weight tuning algorithm, as well as the three kinds of weight encoding strategies. Figure 6 shows that weight tuning can set the weight parameters well for all the three cases: with the increase of weight precision (all other constraints are not introduced), all of them can reach the upper bound accuracy. In Table III, we further give the effect of each phase of the weight tuning step (the weight precision is 2-bit,

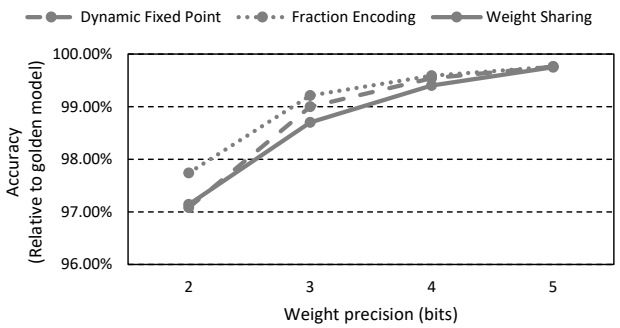


Fig. 6. Accuracy v.s. weight precision under different weight encoding strategies. For the weight sharing case, weight precision means the bit-width of weight indices.

| | dynamic fixed point | fraction encoding | weight sharing |
|---|---|---|---|
| **I** | 81.56% | 85.60% | 86.19% |
| **I+P** | 81.56% | 87.05% | 89.86% |
| **I+L** | 97.08% | 97.05% | 96.31% |
| **I+P+L** | 97.08% | 97.74% | 97.14% |

TABLE III
ACCURACY V.S. DIFFERENT WEIGHT ENCODING STRATEGIES UNDER 2-BIT WEIGHT PRECISION. I: INITIALIZATION WITH EM; P: PARAMETER RANGE FINE-TUNING (TO DECIDE $P$); L: LOW PRECISION FINE-TUNING (TO DECIDE $k_{ij}$)

without any other constraint).

With only the EM-based initialization of the value-range tuning phase (I in Table III), the accuracy of different weight encoding strategies depends on the latter's flexibility. The accuracy of *weight-sharing* is the highest as it is the most flexible: the precision just limits the bit-width of indices, not weight values. *dynamic-fixed-point* only allows weight values to be scaled by power of 2; thus it is the least flexible. *fraction-encoding* is positioned in the middle.

With the whole value-range tuning phase (including the initialization and fine-tuning, I+P in Table III), the accuracy increases a little for both *fraction-encoding* and *weight-sharing* and remains unchanged for *dynamic-fixed-point*. The reason is in the initialization we have already found a good $P$, further fine-tuning $P$ cannot increase the accuracy much, especially for *dynamic-fixed-point* with limited flexibility.

With the EM initialization and the rounding tuning phase (I+L in table III), the accuracy increases significantly, which means that NN can easily find suitable parameters from any well-initialized set of weight values.

With all phases employed (I+P+L in table III), the accuracy could still increase a little compared with the I+L case, except for *dynamic-fixed-point*.

Anyway, under every weight precision, there is no obvious capability difference between the three strategies.

### C. Discussion and future work

Now, some transformation stepsintroduce extra layers more than one time, which may exacerbate NN redundancy. Therefore it is necessary to strike a balance between the possible information loss and hardware-resource consumption. Anyway, we provide a framework, while the concrete workflow could be customized. Moreover, the interaction between network compression and transformation is interesting, and we will study it as the future work.

In addition, it is helpful to present insights into future neuromorphic architecture designs:

(1) It could give design tradeoff between the common computational components and special functional components. For some neural functions or layers that are relatively easy (in terms of hardware consumption) to be achieved by common *core_ops*, it is unreasonable to integrate such a dedicate component on chip. If not, a special component is worth to realize.

(2) After transformation, the data flow between *core_ops* is basically determined; we can analyze the communication pattern in detail, which is conductive to the balanced distribution of computing and communication resources on chip.

(3) Our solution regards NN inference as the forward process of a CG with fixed topology. To some extent, it is suitable for field programmable devices (especially in the aspect of on-chip connection); thus how to combine the device configurability with the flexibility of transformation is an interesting topic.

## VI. CONCLUSION

We present a programming solution for NN chips, which can transform a trained, unrestricted NN into an equivalent network to meet hardware constraints. Multiple techniques are proposed to reduce the transformation error and improve the processing speed. The solution is validated on a real neuromorphic chip and a PIM design for ANNs, as well as on different scales of NNs under different constraints. The evaluation shows that the the transformation methodology is very effective with insignificant error introduced and the transformation time is much faster than re-training the NN models for a specific neuromorphic hardware.

## REFERENCES


[1] 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.

[2] B. Li, Y. Shan, M. Hu, Y. Wang, Y. Chen, and H. Yang, "Memristor-based approximated computation," in *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*, pp. 242–247, IEEE Press, 2013.

[3] 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.

[4] 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, IEEE, 2013.

[5] 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, IEEE, 2013.

[6] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 27–39, IEEE Press, 2016.

[7] 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, l. 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.

[8] S. K. Esser, P. A. Merolla, J. V. Arthur, A. S. Cassidy, R. Appuswamy, A. Andreopoulos, D. J. Berg, J. L. McKinstry, T. Melano, D. R. Barch, C. di Nolfo, P. Datta, A. Amir, B. Taba, M. D. Flickner, and D. S. Modha, "Convolutional networks for fast, energy-efficient neuromorphic computing," *Proceedings of the National Academy of Sciences*, p. 201604850, 2016.

[9] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An instruction set architecture for neural networks," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 393–405, IEEE Press, 2016.

[10] 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 *ACM Sigplan Notices*, vol. 49, pp. 269–284, ACM, 2014.

[11] 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 *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 369–381, ACM, 2015.

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

[13] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 92–104, ACM, 2015.

[14] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 243–254, IEEE Press, 2016.

[15] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[16] Y. Ji, Y. Zhang, S. Li, P. Chi, C. Jiang, P. Qu, Y. Xie, and W. Chen, "Neutrams: Neural network transformation and co-design under neuromorphic hardware constraints," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, 2016.

[17] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, pp. 675–678, ACM, 2014.

[18] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermüller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky, Y. Bengio, A. Bergeron, J. Bergstra, V. Bisson, J. B. Snyder, N. Bouchard, N. Boulanger-Lewandowski, X. Bouthillier, A. de Brébisson, O. Breuleux, P. L. Carrier, K. Cho, J. Chorowski, P. Christiano, T. Cooijmans, M. Côté, M. Côté, A. C. Courville, Y. N. Dauphin, O. Delalleau, J. Demouth, G. Desjardins, S. Dieleman, L. Dinh, M. Ducoffe, V. Dumoulin, S. E. Kahou, D. Erhan, Z. Fan, O. Firat, M. Germain, X. Glorot, I. J. Goodfellow, M. Graham, Ç. Gülçehre, P. Hamel, I. Harlouchet, J. Heng, B. Hidasi, S. Honari, A. Jain, S. Jean, K. Jia, M. Korobov, V. Kulkarni, A. Lamb, P. Lamblin, E. Larsen, C. Laurent, S. Lee, S. Lefrançois, S. Lemieux, N. Léonard, Z. Lin, J. A. Livezey, C. Lorenz, J. Lowin, Q. Ma, P. Manzagol, O. Mastropietro, R. McGibbon, R. Memisevic, B. van Merriënboer, V. Michalski, M. Mirza, A. Orlandi, C. J. Pal, R. Pascanu, M. Pezeshki, C. Raffel, D. Renshaw, M. Rocklin, A. Romero, M. Roth, P. Sadowski, J. Salvatier, F. Savard, J. Schlüter, J. Schulman, G. Schwartz, I. V. Serban, D. Serdyuk, S. Shabanian, É. Simon, S. Spieckermann, S. R. Subramanyam, J. Sygnowski, J. Tanguay, G. van Tulder, J. P. Turian, S. Urban, P. Vincent, F. Visin, H. de Vries, D. Warde-Farley, D. J. Webb, M. Willson, K. Xu, L. Xue, L. Yao, S. Zhang, and Y. Zhang, "Theano: A python framework for fast computation of mathematical expressions," *CoRR*, vol. abs/1605.02688, 2016.

[19] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *CoRR*, vol. abs/1603.04467, 2016.

[20] 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.

[21] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, R. C. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," vol. abs/1704.04760, 2017.

[22] M. Denil, B. Shakibi, L. Dinh, M. A. Ranzato, and N. de Freitas, "Predicting parameters in deep learning," in *Advances in Neural Information Processing Systems 26* (C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, eds.), pp. 2148–2156, Curran Associates, Inc., 2013.

[23] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on cpus," in *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, vol. 1, p. 4, Citeseer, 2011.

[24] K. Hwang and W. Sung, "Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1," in *Signal Processing Systems (SiPS), 2014 IEEE Workshop on*, pp. 1–6, IEEE, 2014.

[25] S. Anwar, K. Hwang, and W. Sung, "Fixed point optimization of deep convolutional neural networks for object recognition," in *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pp. 1131–1135, IEEE, 2015.

[26] W. Chen, J. Wilson, S. Tyree, K. Weinberger, and Y. Chen, "Compressing neural networks with the hashing trick," in *International Conference on Machine Learning*, pp. 2285–2294, 2015.

[27] Z. Mariet and S. Sra, "Diversity networks," *CoRR*, vol. abs/1511.05077, 2015.

[28] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. J. Dally, "ESE: efficient speech recognition engine with compressed LSTM on FPGA," *CoRR*, vol. abs/1612.00694, 2016.

[29] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2016.

[30] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 267–278, IEEE Press, 2016.

[31] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2016.

[32] D. Shin, J. Lee, J. Lee, and H. Yoo, "Dnpu: An 8.1tops/w reconfigurable cnn-rnn processor for general purpose deep neural networks," in *International Solid-State Circuits Conference*, IEEE, 2017.

[33] L. Cavigelli and L. Benini, "A 803 gop/s/w convolutional network accelerator," *IEEE Transactions on Circuits and Systems for Video Technology*, 2016.

[34] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: balancing efficiency & flexibility in specialized computing," in *ACM SIGARCH Computer Architecture News*, vol. 41, pp. 24–35, ACM, 2013.

[35] R. LiKamWa, Y. Hou, J. Gao, M. Polansky, and L. Zhong, "Redeye: analog convnet image sensor architecture for continuous mobile vision," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 255–266, IEEE Press, 2016.

[36] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pp. 380–392, IEEE, 2016.

[37] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, pp. 109–116, IEEE, 2011.

[38] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "Cnp: An fpga-based processor for convolutional networks," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pp. 32–37, IEEE, 2009.

[39] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 161–170, ACM, 2015.

[40] L. Song, Y. Wang, Y. Han, X. Zhao, B. Liu, and X. Li, "C-brain: A deep learning accelerator that tames the diversity of cnns through adaptive data-level parallelization," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*, pp. 1–6, IEEE, 2016.

[41] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2016.

[42] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *High Performance Computer Architecture (HPCA),2017 23rd IEEE Symposium on*, IEEE, 2016.

[43] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 14–26, IEEE Press, 2016.

[44] M. Hu, J. P. Strachan, Z. Li, E. M. Grafals, N. Davila, C. Graves, S. Lam, N. Ge, J. J. Yang, and R. S. Williams, "Dot-product engine for neuromorphic computing: programming 1t1m crossbar to accelerate matrix-vector multiplication," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*, pp. 1–6, IEEE, 2016.

[45] Z. Chen, B. Gao, Z. Zhou, P. Huang, H. Li, and W. Ma, "Optimized learning scheme for grayscale image recognition in a rram based analog neuromorphic system," in *Electron Devices Meeting (IEDM), 2015 IEEE International*, IEEE, 2015.

[46] B. Li, Y. Shan, M. Hu, Y. Wang, Y. Chen, and H. Yang, "Memristor-based approximated computation," in *international symposium on low power electronics and design*, pp. 242–247, 2013.

[47] X. Liu, M. Mao, B. Liu, H. Li, Y. Chen, B. Li, Y. Wang, H. Jiang, M. Barnell, Q. Wu, and J. Yang, "Reno: A high-efficient reconfigurable neuromorphic computing accelerator design," in *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, pp. 1–6, IEEE, 2015.

[48] M. N. Bojnordi and E. Ipek, "Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pp. 1–13, 2016.

[49] A. Agarwal, E. Akchurin, and C. Basoglu, "An introduction to computational networks and the computational network toolkit," 2014.

[50] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *neural information processing systems*, 2011.

[51] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," 2015.

[52] A. S. Cassidy, P. Merolla, J. V. Arthur, S. K. Esser, B. Jackson, R. Alvarez-Icaza, P. Datta, J. Sawada, T. M. Wong, V. Feldman, A. Amir, D. B.-D. Rubin, F. Akopyan, E. McQuinn, W. P. Risk, and D. S. Modha, "Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores," in *Neural Networks (IJCNN), The 2013 International Joint Conference on*, pp. 1–10, IEEE, 2013.

[53] 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, IEEE, 2013.

[54] 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, no. 10, pp. 1537–1557, 2015.

[55] 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, no. 12, pp. 2451–2461, 2013.

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

[57] 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, no. 5, pp. 699–716, 2014.

[58] 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.

[59] K. Meier, "A mixed-signal universal neuromorphic computing system," in *Electron Devices Meeting (IEDM), 2015 IEEE International*, pp. 4–6, IEEE, 2015.

[60] 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, no. 12, pp. 2454–2467, 2013.

[61] 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.

[62] E. Hunsberger and C. Eliasmith, "Training spiking deep networks for neuromorphic hardware," *CoRR*, vol. abs/1611.05141, 2016.

[63] J. H. Lee, T. Delbruck, and M. Pfeiffer, "Training deep spiking neural networks using backpropagation," *Frontiers in Neuroscience*, vol. 10, 2016.

[64] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[65] D. Mishkin and J. Matas, "All you need is a good init," *arXiv preprint arXiv:1511.06422*, 2015.

[66] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[67] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.