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

Yu Ji
jiy15@mails.tsinghua.edu.cn
Department of Computer Science and Technology
Tsinghua University
China

Youhui Zhang*
zyh02@tsinghua.edu.cn
Department of Computer Science and Technology
Tsinghua University
China

Wenguang Chen
cwg@tsinghua.edu.cn
Department of Computer Science and Technology
Tsinghua University
China

Yuan Xie
yuanxie@ece.ucsb.edu
Department of Electrical and Computer Engineering
University of California at Santa Barbara
USA

ABSTRACT

Different from developing neural networks (NNs) for general-purpose processors, the development for NN chips usually faces with some 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. We decouple the NN applications from the target hardware by introducing a compiler that can transform an existing trained, unrestricted NN into an equivalent network that meets the given hardware's constraints. We propose multiple techniques to make the transformation adaptable to different kinds of NN chips, and reliable for restrict hardware constraints.

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

CCS CONCEPTS

• **Computer systems organization → Neural networks**;

KEYWORDS

Neural Network, Accelerator, Compiler

ACM Reference Format:
Yu Ji, Youhui Zhang, Wenguang Chen, and Yuan Xie. 2018. Bridging the Gap Between Neural Networks and Neuromorphic Hardware with A Neural Network Compiler. In *ASPLOS '18: 2018 Architectural Support for Programming Languages and Operating Systems, March 24–28, 2018, Williamsburg, VA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3173162.3173205

1 INTRODUCTION

Designing custom chips for NN applications with the Very-Large-Scale-Integration (VLSI) technologies has been investigated as a power-efficient and high-performance alternative to general-purpose computing platforms such as CPU and GPU. However, programming these chips is difficult because of some hardware-specific constraints: ① Due to the utilization of hardware resources for digital circuits or the capability of analog computing for some memristor-based designs [18, 31, 40, 44, 55], the precision of input and output signals of neurons is usually limited, as well as ② the precision of NN parameters, such as synaptic weights. ③ The present fabrication technology limits the fan-in and fan-out of one neuron, which constrains the computation scale. ④ The diversity of nonlinear functions or neuron models supported by the hardware is also limited. For example, for TrueNorth chips [52], the maximum matrix that one synaptic core can handle is 256×256, and it supports only a simplified leaky–integrate-and-fire (LIF) neuron model.

One straightforward approach to this problem is to expose the hardware details and limitations to the NN developer directly. For instance, IBM has provided a TrueNorth-specific training mechanism [22]. The mechanism constructs the whole NN model from scratch to satisfy all hardware limitations, and then trains the NN model. This method has several drawbacks. First, it binds NN models to the specific target hardware. Developers can hardly benefit from existing NN models from the machine-learning community. Second, it limits the power of the NN algorithm. The constraints make it more difficult to converge and reach better accuracy for larger models. Third, training the specific NN model from scratch may take a very long time.

*Corresponding author

ASPLOS '18, March 24–28, 2018, Williamsburg, VA, USA

ACM ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/10.1145/3173162.3173205

Another approach is to make hardware satisfy software requirement by consuming more hardware resources to reduce the constraints on NN models [12, 15, 21, 47], such as using 16-bit precision rather than 1-bit spiking signals in TrueNorth. This approach can gain less performance improvement from NN quantization and compression technologies. For some memristor-based designs, some constraints due to analog computing are physical limitations, which are difficult to overcome even with more hardware resources consumed.

A third approach is to introduce a domain-specific Instruction Set Architecture (ISA) for NN accelerators, such as the Cambricon [48] ISA. This approach requires both hardware and software to satisfy the ISA. However, this approach still does not solve the gap between programming flexibility required by NNs and hardware efficiency that can be gained from NNs' redundancy. If we use high-precision instructions that do not have any constraints, the hardware can gain less benefit from NN's redundancy. In contrast, if we use low-precision instructions with many constraints, the NN developer should take these constraints into consideration when developing NN models.

In addition to these approaches, there are also some work to utilize the NNs' redundancy for performance and provide flexible programming interface by introducing a transforming procedure. EIE [27] is such an instance: it extensively uses deep compression to squeeze the redundancy, and design custom chip, EIE, to run the compressed NN model. NEUTRAMS [35] also use NNs' redundancy to adapt the original model to satisfy hardware constraints. However, these methods highly depends on the redundancy in NN models. Different NN models may have different minimum requirement (precision, connectivity, etc.) on hardware. Thus, transforming procedure is not a general method, especially for NN models with less redundancy and hardware with severe constraints.

In this paper we propose a new method with flexibility, better applicability, and easy convergence. First, we decouple the neuromorphic computer system into two levels for better flexibility, software programming model and hardware execution model. We use computational graph (CG), which is widely used in many popular NN frameworks [1, 4, 36], as the programming model for NN models. We also provide the hardware/software (HW/SW) interface and the minimum hardware functionality that an NN hardware should provide. We propose a transformation workflow to convert a trained NN, expressed as a CG, into an equivalent representation of HW/SW interface through the fine-tuning method.

To make the transformation workflow general and reliable for different cases, we employed two principles.

• **Trade Scale for Capability.** As the operations supported by NN hardware is not comparable to their software counterparts due to the constraints, it is reasonable to enlarge the graph scale and complicate the topology properly to improve the model capability, especially under strict conditions.

• **Divide and conquer.** We fine-tune the entire model part by part according to a topological ordering. Each part is a smaller graph that is more easier to converge. We also fine-tune each part with several phases to introduce different constraints, which also facilitates the fast convergence.

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 SW/HW interface), and the transformation tool could be called an NN compiler. As a summary, this paper has achieved the following contributions:

- An NN transformation workflow is presented to complete the aforementioned technologies to support different types of NNs. The SW/HW interface is easy to be adapted to different NN hardware.
- Such a toolchain is implemented to support two different hardware designs' constraints, a real CMOS neuromorphic chip for ANN&SNN, TianJi [60], and a PIM design built upon metal-oxide resistive random access memory (ReRAM) for ANN, PRIME [18].
- 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.

2 BACKGROUND

2.1 NN basis

NNs are a set of algorithms, modeled loosely after the human brain, that are designed to recognize patterns. Traditional NNs consist of multiple layer of neurons. Each layer performs the computation as shown in Equation 1 where X is the input vector, Y is the output vector, W is the weight matrix, B is the bias, and σ is a non-linear activation function, which is typically the Rectified Linear Units (ReLU) function [54].

$$Y = \sigma(W \cdot X + B) \tag{1}$$

This kind of NN is also known as multilayer perceptron (MLP), which has been proved to be a universal approximator [30]. Modern NNs are more complicated. The topology is a graph rather than a simple chain, and the types of operations are richer than matrix-vector multiplication. Most deep learning frameworks [1, 2, 4, 13] use computational graph (CG), a directed acyclic graph, to represent NN computations. Vertices in the graph represent operations (e.g., dot-product, convolution, activation function, pooling) and immutable/mutable states [1] (e.g., the weight parameters associated). Edges represent the data dependency between vertices. Both vertices and edges process or carry tensor data (multi-dimensional arrays).

For clarity, in this paper, dot-product, bias-addition, and convolution are categorized as *weighted-sum* operations. 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 we can view it as the immutable state of the vertex.

2.2 NN Chips

There are two types of NN chips. The first type focuses on the traditional ANNs. They are custom architectures [11, 12, 15, 21, 23, 24, 38, 39, 46, 47, 56, 57, 59, 61, 64, 67, 68] to accelerate mature ANN models. We usually call this type NN accelerators. The second is

Chip	Weight	I/O	Scale	Nonlinear
TianJi [60]	8-bit	8-bit	256^2	Configurable
PRIME [18]	8-bit	6-bit	256^2	ReLU Max Pooling
DianNao [12]	16-bit	16-bit	16^2	Configurable
TPU [37]	8-bit	8-bit	None	ReLU Max Pooling etc.
TrueNorth [52]	2-bit	Spiking	256^2	LIF

Table 1: Hardware limitations of NN chips

neuromorphic chips, which usually supports SNNs to yield higher biological reality [7, 9, 10, 25, 51, 52, 60, 66].

These chips usually consist of a lot of processing elements (PEs) that can efficiently perform dot-product operations because this operation is the main body of most NN models. Different chips put different constraints on the operations they support. Table 1 shows the constraints of some existing NN chips. Most NN chips employ low precision numbers to represent weights and input/output (I/O) data instead of floating-point numbers. The scale of computation that each PE can process is usually fixed. PRIME [18] and DianNao [12] have extra adders to support larger scale computations. However, NN chips such as TianJi [60] and TrueNorth [52] do not have extra adders, and the output of their PE can connect to only one input port of another PE. For these chips, the scale of computation is also a problem. Despite the widely-supported dot operation, many other operations required by NNs usually lack for support.

3 PROBLEM DESCRIPTION

To bridge the gap between NN applications and NN chips, we decouple the whole system stack with a software programming model and a hardware execution model. The former is the programming interface for NN experts to develop NN applications. The later is the SW/HW interface that NN chips can executed directly.

Software Programming Model. The machine-learning community has already employed *Computational Graph* (CG) as the programming model. It is a data-flow graph $G = (V, E)$ which represents a number of operations with vertices V and represents data dependencies between these operations with edges E. Most deep-learning frameworks [1, 2, 4, 13] adopt CG to build NN models. And the set of supported operations is F.

Each vertex in V is an operation $y = f(x_1, \ldots, x_n)$, where $f \in F$, y represents the output edge, and $\{x_1, \ldots, x_n\}$ represent input edges. Thus, the entire model can be expressed as a composite function $Y = H(X)$, where X represent all input vertices and Y represents all output vertices.

We also adopt CG as the programming model with a slight modification. The difference is that we regard model parameters as immutable states of the corresponding vertices instead of normal edges of the vertices. Namely, we regard an operation $f(x, \theta)$ as $f^{\theta}(x)$, where θ is model parameters and x is an input operand. Thus, it can only be a trained NN model that all parameters have been already determined.

Hardware Execution Model. The computation model that hardware could execute is also a data-flow graph $G' = (V', E')$. It has a supported operation set F', denoted as *core-op set*. However, the supported operations are very limited, and these operations have many limitations. In addition, some hardware also has constraints on the interconnection subsystem. For example, TianJi and TrueNorth does not support multi-cast; one output port of each PE can only be connected to one input port. The hardware execution model forms a composite function $Y' = H'(X')$.

Thus, our goal is to build G' from G so that H' is approximately equivalent to H.

Minimum Hardware Requirement. We define a minimum set of operations C that $C \subset F'$ has to be satisfied to use our NN compiler. It only contains one operation, denoted as *dot–core-op*. Namely, the operation dot–core-op must belong to the core-op set. Equation 2 shows the computation of the dot–core-op where X and Y are the input and output vectors, respectively; N and M are their sizes; W is the weight matrix of size $M \times N$; and σ is a nonlinear activation function.

$$Y_j = \sigma(\sum_i W_{ji} X_i) \quad (1 \le j \le M, 1 \le i \le N) \tag{2}$$

In addition, the I/O data precision is B bits.

Formally, the dot–core-op meets the following constraints:

- N,M,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 S^P with some parameters P.
- Without loss of generality, only ReLU function (σ) is supported.

We choose this dot–core-op as the minimum requirement for hardware because it can cover most existing NN chips (e.g., those listed in Table 1). Thus, our NN compiler can support most existing NN chips.

4 TRANSFORMATION METHODOLOGY

In this section, we will introduce the transformation methodology to transform the software programming model into an approximately equivalent hardware execution model.

4.1 The workflow outline

The proposed workflow involves 4 steps as shown in Figure 1.

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. An example is shown in Figure 2(a). In addition, we can also get the operation set F supported by the deep learning framework.

Graph Reform. It constructs an intermediate CG, $\hat{G} = (\hat{V}, \hat{E})$. The corresponding operation set $\hat{F}$ contains all operations that are *core-op–like*. These operations share the similar form as core-ops or can be achieved by core-ops easily, but do not meet those hardware constraints. Figure 2(b) shows an example of the graph with only core-op–like operations. The detailed transformation from G to $\hat{G}$ is in Section 4.2.

Graph Tuning. In this step, we further transform $\hat{G}$ to G'. Every vertex $\hat{v} \in \hat{V}$ is traversed in a topological ordering of $\hat{G}$ to form corresponding vertices in G'. As shown in Figure 1, we have multiple

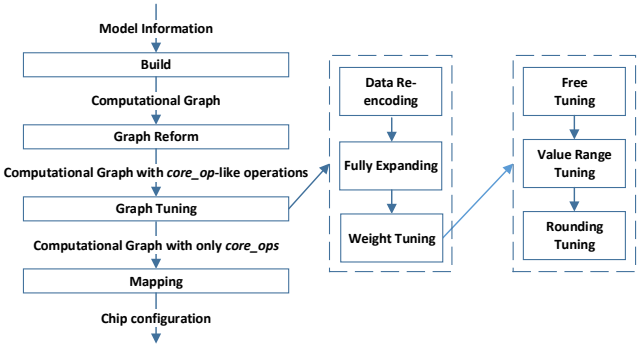

Figure 1: Workflow of our proposal, from the input model to the output for chip. Step 'Graph Tuning' contains 3 sub-steps for different hardware restrictions respectively and the third sub-step has 3 fine-tuning phases.

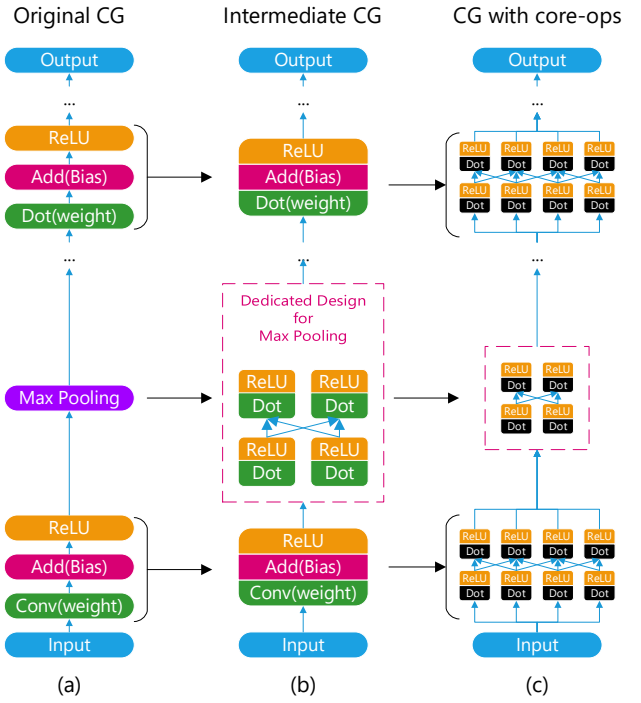

Figure 2: A transformation example

sub-steps for the transformation of each vertex $\hat{v}$. We can also transform a subgraph with multiple vertices as a whole. We transform the whole graph part by part to have a better convergence since smaller graph are easier to be approximated. It is where we employ the *divide-and-conquer* principle. The sub-steps are as following.

- **Data Re-encoding.** Re-encode I/O data on each edge of the subgraph to solve the precision problem of I/O data. This sub-step is where we employ the *trade-scale-for-capability* principle. It enlarges the computation scale, but does not change the operation type, each vertex is still a core-op–like operation.
- **Fully Expanding.** Since core-op–like operations are easy to be implemented with core-ops, in this sub-step, we fully expand each core-op–like operation with multiple core-op operations to solve the limitation on the computation scale. After this sub-step, the subgraph only contains core-ops.
- **Weight Tuning.** This step aims to fine-tune the weight matrices of core-ops in the subgraph to minimize transformation error, under the premise of satisfying the hardware weight precision. As shown in Figure 1, we also introduce three phases of fine-tuning to make it more reliable and easier to converge. It is also where we employ the *divide-and-conquer* principle.

Figure 2(c) shows an example of the transformed graph G' with only core-ops. Detailed transformation are in Section 4.3.

Mapping. Now we have built an equivalent Graph G' with only core-ops that meet hardware constraints, which will be mapped onto the target hardware efficiently. This part highly depends on the hardware's interconnection implementation. In Section 4.4 we will introduce the basic principle of mapping the hardware execution model onto target chips.

4.2 Graph Reform

In this step, we need to transform a CG G into a graph $\hat{G}$ with only core-op–like operations. The operation set $\hat{F}$ includes all operations that could be combined by core-ops in F' without any precision constraints. For example, the corresponding core-op–like operations for dot–core-op are all operations that in the form of weighted sum with activation function, denoted as *dot-like* operations.

To replace all vertices represented with F into operations in $\hat{F}$, we take the following three steps in order.

① First, we find all subgraphs that match the computation of any $\hat{f} \in \hat{F}$, and replace them with $\hat{f}$. For example, in Figure 2(b), we merge the dot-product, add-bias and ReLU operations into one operation, which is a dot-like operation.

② Then, we can also have some customized mapping from a subgraph in G into a subgraph formed of operations in $\hat{F}$, and apply these dedicated designs here. For example, max-pooling operation can be built with max functions. A simple max function with two operands can be achieved with as Equation 3, which includes multiple dot-like operations.

$$\begin{aligned}\max(a,b) = &\frac{1}{2}[\mathrm{ReLU}(a+b) + \mathrm{ReLU}(a-b) \\ &+ \mathrm{ReLU}(-a+b) + \mathrm{ReLU}(-a-b)]\end{aligned} \tag{3}$$

We can use the max function with two operands to form max function with more operands.

③ Finally, for the left operations in G, we provide a default transformation strategy: we use multiple dot-like operations to form MLPs to approximate them since MLP is proved to be an universal approximator [30].

After the transformation, we form an graph $\hat{G}$ with only core-op–like operations.

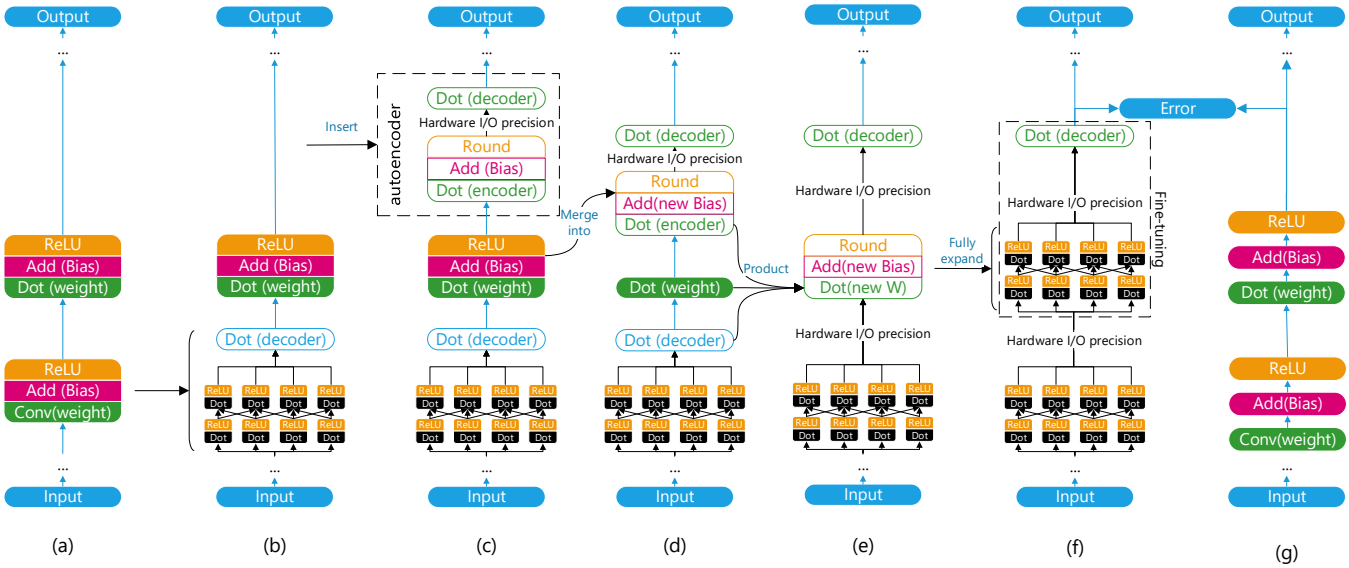

Figure 3: Graph tuning of one vertex. (a) G' with only *core_op*-like operations. (b) 'Graph Tuning' has been performed against all previous vertices. (c) Insert an 'autoencoder' after the current vertex. (d) Merge the bias and activation of the current vertex into the hidden layer of the autoencoder. (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.

4.3 Graph Tuning

In this step, we transform the intermediate graph $\hat{G} = (\hat{V}, \hat{E})$ to the hardware execution model $G' = (V', E')$. We use the original graph G to supervise the fine-tuning progress of the generated G'. The graph G can provide not only labels of the output but also supervised signals of all intermediate data between operations. Each edge $e \in E$ can provide supervised signal for graph G'. Thus, we can split $\hat{G}$ into parts, and transform it into G' part by part. To do so, first we find the edges $\hat{e} \in \hat{E}$ that correspond to the edges $e \in E$, and use them to split the graph $\hat{G}$. Then, we perform the following steps against each part one by one in a topological ordering of $\hat{G}$. Note that, we can also transform multiple adjacent parts as a whole each time.

• **Data Re-encoding.** Since the original model G usually use floating-point numbers for computation and data representation, directly rounding data to hardware I/O precision may distort and lost information. Thus, this sub-step aims to re-encode the input and output data of each vertex. To encode a floating-point vector with a low-precision vector, we employ an autoencoder to get the low-precision representation.

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 vector with the hardware–I/O-precision vector, it is reasonable that the neuron number of the hidden layer may be greater than that of the 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 5.

For clarity, we take a vertex of dot-like operation 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 function of its hidden layer is a round operation, which rounds the output to hardware I/O precision. In addition, the round operation not only quantizes the output to low-precision, but also forces the output to be positive, which provides the non-linearity as the widely-used ReLU function.

The initial weight parameters for the encoder and decoder of the autoencoder are set 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, where m is a configurable integer, denoted as *re-encoding factor*. Suppose the n dimensions of X are independent and equally important, then each x has m hidden neurons to represent its low precision encoding. Thus, we initialize the connections from x to other hidden neurons to zero. A hidden neuron linearly scales the corresponding x to $w^{(e)}x + b^{(e)}$, where $w^{(e)}$ and $b^{(e)}$ are the weight and bias term of the encoder respectively, and then rounds it to the I/O precision, N-bit integer ($\{0, \ldots, 2^N - 1\}$). 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 $[\frac{ix_{max}}{m}, \frac{(i+1)x_{max}}{m}]$. 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 operation 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. In this step, we only change the computation scale of the operations in $\hat{G}$.

• **Fully Expanding.** In this step, we will turn $\hat{G}$ into G'. Since the core-op–like operations $\hat{f} \in \hat{F}$ can be combined by core-ops in F', we expand all operations in $\hat{G}$ into individual subgraphs consisting of operations in F' to form the graph G'.

Take dot-like operations as an example. Dot-like operations can be represented as a fully-connected layer. As shown in Figure 3(f), to support the fully-connected layer of any scale with dot–core-ops, we use two layers of dot–core-ops to construct an equivalent graph. The first layer is a computation layer. The original dot-like operations are divided into smaller blocks and are performed with many dot–core-ops. 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 dot–core-op of the first layer. ② Divide the input vector into sub-vectors and transfer each sub-vector to the corresponding sub-matrices(*core_ops*) at the same horizontal position and ③ gather results from the same column by the reduce layer.

Regarding all dot-like operations as fully-connected layers are sometimes very inefficient. We can have dedicated division according to its connection pattern.

For example, 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 dot–core-op, we can divide the operation as the fully-connected–layer case does.

If there are some dot–core-ops that are not fully used, we can distribute them onto one physical PE to reduce resource consumption during the mapping step.

Till now, the computation of the current subgraph of $\hat{G}$ has been transformed to a subgraph of G', which consists of core-ops $f' \in F'$. Next, the weight matrices of the core-ops will be fine-tuned.

• **Weight Tuning.** In this step, we will fine-tune the parameters to make the generated subgraph of G' approximately equal to the corresponding subgraph of G.

As shown in Figure 3(g)(f), we use the corresponding supervised signal from graph G to fine-tune current subgraph of G'. The input to the current subgraph is from the output of previous transformed subgraphs instead of the corresponding supervised signal from the graph G. Thus, the transformation of current subgraph will consider the error from previous transformed subgraphs, which can avoid error accumulation. The output of previous transformed subgraph can be generated on demand or cached in advance to improve the transformation speed.

We will consider the hardware constraints on weight parameters in this step. Specifically, target hardware usually puts strict constraints on weight storage since it occupies most of the hardware resources. Here we present a formal description of the constraints on the weight matrix W: the value of each element W_{ij} should be assigned dependently from 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 2^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 [61], Strip [38], TianJi-ANN [60], 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 [18], and TianJi-SNN [60].
- **Weight sharing**: $S^{P_1,\ldots,P_{2^N-1}} = \{0, P_1, \ldots, P_{2^N-1}\}$, used by EIE [27].

Without loss of generality, suppose the floating-point parameter 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 subgraph, 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.

• **Free Tuning.** In previous steps, we use the parameters in the original graph G to initialize those parameters in the generated graph G'. However, some methods, including autoencoder and the MLP-based unsupported-function handling, introduce transformation errors. In addition, activation functions used by G may be different from the hardware counterpart, which also makes the initialization inaccurate. In addition, previous transformed subgraphs also have errors. Therefore, some fine-tuning phases have to be taken to minimize the error, under the premise of satisfying the hardware constraints on weight precision.

Thus we first fine-tune the subgraph of G' without any constraint on weight precision 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.

• **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 $k^{(t)}_{ij} = \arg\min J(k|P^{(t)})$.
- M-step: fix $k^{(t)}_{ij}$ and calculate $P^{(t+1)} = \arg\min J(P|k^{(t)})$.

Then we replace W_{ij} with $S^P_{k_{ij}}$ where k_{ij} is fixed and P is the parameter.

After the initialization, we fine-tune the subgraph 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, we can omit this phase.

• **Rounding Tuning.** The data-range 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}. During the fine-tuning progress, parameters are stored as floating point number. In the forward phase, any parameter is rounded to the closest element in S^P. While during the backward phase, floating-point number is used to update W_{ij}. This mechanism is also employed by the above *Value-Range Tuning* phase if P can be set from a discrete set.

After processing all the subgraphs, we have transformed the original model G into an equivalent hardware execution model G' that satisfies all the constraint conditions.

4.4 Mapping

The generated graph G' will 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 in the *physical cores* (it is called the *weight stationary* computing mode, classified by [16]), 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 the 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 graph, 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 [3].

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.

4.5 Others

4.5.1 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 [33, 43] have used the stochastic gradient decent (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.

4.5.2 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.

5 IMPLEMENTATION AND EVALUATION

5.1 Implementation

We have implemented the tool to support different hardware constraints, including those of TianJi [60] and PRIME [18].

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 256×256. For ANN mode, the I/O

NN model	Chip	Weight Encoding	Weight Precision	I/O Precision	Re-encoding Factor	Top1 Accuracy (Accuracy Drop)
MNIST-MLP		Floating-point				98.2%
MNIST-MLP	TianJi-ANN	Dynamic fixed-point	8-bit	8-bit	1×	98.15%(-0.05%)
MNIST-MLP	TianJi-SNN	Fraction encoding	8-bit	1-bit	2×	96.59%(-1.61%)
MNIST-MLP	TianJi-SNN	Fraction encoding	8-bit	2-bit	2×	97.63%(-0.57%)
MNIST-MLP	PRIME	Fraction encoding	8-bit	6-bit	1×	98.14%(-0.06%)
LetNet-5		Floating-point				99.1%
LeNet-5	TianJi-ANN	Dynamic fixed-point	8-bit	8-bit	1×	99.08%(-0.02%)
LeNet-5	PRIME	Fraction encoding	8-bit	6-bit	1×	99.01%(-0.09%)
CIFAR10-VGG17		Floating-point				84.64%
CIFAR10-VGG17	TianJi-ANN	Dynamic fixed-point	8-bit	8-bit	1×	84.02%(-0.62%)
CIFAR10-VGG17	PRIME	Fraction encoding	8-bit	6-bit	1×	83.57%(-1.07%)
ImageNet-AlexNet		Floating-point				57.4%
ImageNet-AlexNet	TianJi-ANN	Dynamic fixed-point	8-bit	8-bit	1×	56.9%(-0.5%)
ImageNet-AlexNet	PRIME	Fraction encoding	8-bit	6-bit	1×	55.2%(-2.2%)
ImageNet-AlexNet	PRIME	Fraction encoding	8-bit	6-bit	4×	57.0%(-0.4%)
ImageNet-VGG16		Floating-pint				70.5%
ImageNet-VGG16	TianJi-ANN	Dynamic fixed-point	8-bit	8-bit	1×	69.6%(-0.9%)
ImageNet-VGG16	PRIME	Fraction encoding	8-bit	6-bit	1×	68.2%(-2.3%)
ImageNet-VGG16	PRIME	Fraction encoding	8-bit	6-bit	4×	69.5%(-1.0%)

Table 2: Accuracy for NNs under different restrictions

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 4.5.1). 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 [18] 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 256×256, 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, including an MLP for MNIST dataset (784-100-10 structure, 98.2% accuracy of full precision), LeNet-5 [42] for MNIST dataset (99.1% accuracy), a CNN [53] for CIFAR-10 dataset (84.64% accuracy [1]), AlexNet [41] and VGG16 [62] for ImageNet, have been respectively transformed and then deployed onto TianJi [60] and PRIME [18] to show the validation. The first three networks are trained by Theano [4]. Parameters of the next two CNNs for ImageNet are extracted from trained models of the Caffe Model Zoo directly. The inference accuracies of full precision are given in Table 2.

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 2×3 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 12×8 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's dot–core-op. 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.

5.2 Evaluation

The inference accuracies after transformation for TianJi and PRIME are given in Table 2. This table also shows the re-encoding factor, which indicate the number of hidden units for autoencoders, for each case.

We can see that the transformation errors introduced are very limited, for all NN models and chips tested, we can achieve less than 2% accuracy drop. For most cases, we only use 1× hidden unit

[1] As described by [53], with some special initialization method, the CNN accuracy can exceed 90%. Here we ignore it for simplicity, which does not affect our evaluation.

to re-encode data, which means that the NN scale is not changed. For the two TianJi-SNN cases, as the I/O constraints are very strict, we use 2× hidden units, thus 4× PEs will be used. For ImageNet cases on PRIME, if we only use 1× hidden units, the accuracy drop will be over 2%. With 4× hidden units, we can reduce the accuracy drop to less than 1%. We can always achieve better accuracy with more hardware resources employed.

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.

5.2.1 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 parts 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 but consume 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 subgraphs (the accuracy is 83.14% as the granularity is one). If we fine-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. Unless specifically noted, the fine-tuning granularity is just one.

5.2.2 Accuracy vs. Resource Consumption. From the transformation workflow, we can see that the step of *Data Re-encoding* may introduce the most additional 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 graph. Table 2 also shows the improvement for ImageNet on PRIME with a larger re-encoding factor. 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.

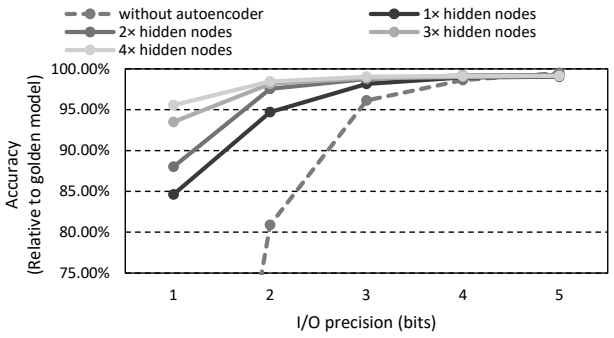

Figure 4: Accuracy v.s. I/O precision under different transformation strategies.

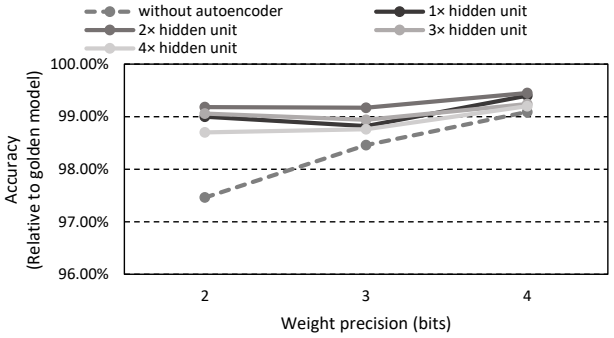

Figure 5: Accuracy v.s. weight precision under different transformation strategies.

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× hidden neurons and 1-bit I/O does consume the same number of I/O ports with that of using 1× 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

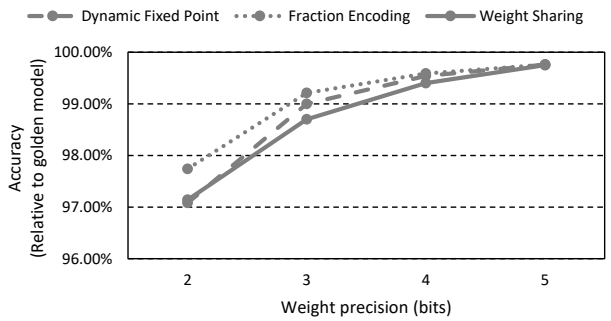

Figure 6: Accuracy v.s. weight precision under different weight encoding strategies. For the weight sharing case, weight precision means the bit-width of weight indices.

	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 3: 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})

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%.

5.2.3 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 the three cases: with the increase of weight precision (any other constraint is not introduced), all of them can reach the upper bound accuracy. In Table 3, we further give the effect of each phase of the weight tuning step (the weight precision is 2-bit, without any other constraint).

With only the EM-based initialization of the value-range tuning phase (I in Table 3), 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 3), 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 3), 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 3), the accuracy could still increase a little compared with the I+L case, except for *dynamic-fixed-point*.

Anyway, there is no obvious capability difference between the three strategies.

5.3 Discussion and future work

Now, some transformation steps introduce 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:

- 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.
- After transformation, the data flow between core-op 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.
- Our solution regards NN inference as the forward process of graph 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.

6 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. In order to distinguish, we call the former NN accelerators and the latter neuromorphic chips in the following content.

6.1 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 [20]. 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* [6, 34, 65] and *weight sharing* [14] are pruning weights. For pruning synapses, deep compression [28] prunes the network by retaining only important connections. Diversity Network [50] 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.

6.2 NN accelerators

EIE [27] extensively employs the above compression techniques and then proposes a dedicated engine to perform inference on the compressed network model. ESE [26] 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 [12] designs an accelerator for DNNs and CNNs that exploits data reuse with tiling. The inherent sharing of weights in CNNs is explored in ShiDianNao [21]. PuDianNao [47] supports seven machine learning algorithms. DaDianNao [15] 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 [48] 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 [68].

Minerva [57] uses fine-grained data type quantization and dynamic operation pruning to further reduce the resource consumption of DNN hardware. Strip [38] relies on bit-serial compute units to enable support for per-layer, dynamically configurable computation precision for DNN. DNPU [61] supports dynamic fixed-point with online adaption and weight quantization, too. Other studies include Origami [11], Convolution Engine [56], RedEye [46], NeuroCube [39], neuFlow [23] and quite a few FPGA-based designs [24, 59, 64, 67].

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 [63] is such an accelerator for CNNs that supports both training and inference, while PRIME [18] and ISAAC [58] are for inference. Other work includes stand-along accelerators [17, 32, 55], co-processor [45] and many-core or NoC [8, 49] 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 [4], TensorFlow [1], CNTK [2], Torch [19], MXNet [13], etc.) describe NN as computation graphs. For a trained NN, the complete information can be extracted from whichever of them.

6.3 Neuromorphic chips

TrueNorth [10, 52] is a digital neuromorphic chip for SNNs, based on a structure of tiled crossbar (each crossbar is of size 256×256, and supports binary-valued neurons and ternary-valued synapses). The programming paradigm, *Corelet* [5], is bound to the hardware platform: its recent study [22] proposes a TrueNorth-specific training mechanism to construct the whole NN from top to bottom. The parameters learned are then mapped to hardware [3] using *Corelet*.

EMBRACE [9] 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 [29]. Thus, it is not a general solution. Neurogrid [7] and FACETS [66] (including its successor BrainScaleS [51]) are analog/digital hybrid systems, whose development methods are both hardware-specific. SpiNNaker [25] 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 [60] is an experimental CMOS neuromorphic chip based on tiled crossbars and supports hybrid computing of ANN and SNN. A toolchain NEUTRAMS [35] 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 redundancy of NN models and is not suitable for large-scale network under strict constraints.

7 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 transformation methodology is very effective and only insignificant errors will be introduced. The transformation time is much faster than re-training the NN models for a specific neuromorphic hardware.

ACKNOWLEDGMENTS

The work is supported by the National Key Research and Development Program of China under Grant No. 2016YFB0200505 and by Beijing Municipal Commission of Science and Technology under Grant No. Z161100000216147.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR* abs/1603.04467 (2016). http://arxiv.org/abs/1603.04467

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

[3] Filipp Akopyan, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, Yutaka Nakamura, Pallab Datta, Gi-Joon Nam, Brian Taba, Michael Beakes, Bernard Brezzo, Jente B Kuang, Rajit Manohar, William P Risk, Bryan Jackson, and Dharmendra S Modha. 2015. 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* 34, 10 (2015), 1537–1557.

[4] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermüller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, Yoshua Bengio, Arnaud Bergeron, James Bergstra, Valentin Bisson, Josh Bleecher Snyder, Nicolas Bouchard, Nicolas Boulanger-Lewandowski, Xavier Bouthillier, Alexandre de Brébisson, Olivier Breuleux, Pierre Luc Carrier, Kyunghyun Cho, Jan Chorowski, Paul Christiano, Tim Cooijmans, Marc-Alexandre Côté, Myriam Côté, Aaron C. Courville, Yann N. Dauphin, Olivier Delalleau, Julien Demouth, Guillaume Desjardins, Sander Dieleman, Laurent Dinh, Melanie Ducoffe, Vincent Dumoulin, Samira Ebrahimi Kahou, Dumitru Erhan, Ziye Fan, Orhan Firat, Mathieu Germain, Xavier Glorot, Ian J. Goodfellow, Matthew Graham, Çaglar Gülçehre, Philippe Hamel, Iban Harlouchet, Jean-Philippe Heng, Balázs Hidasi, Sina Honari, Arjun Jain, Sébastien Jean, Kai Jia, Mikhail Korobov, Vivek Kulkarni, Alex Lamb, Pascal Lamblin, Eric Larsen, César Laurent, Sean Lee, Simon Lefrançois, Simon Lemieux, Nicholas Léonard, Zhouhan Lin, Jesse A. Livezey, Cory Lorenz, Jeremiah Lowin, Qianli Ma, Pierre-Antoine Manzagol, Olivier Mastropietro, Robert McGibbon, Roland Memisevic, Bart van Merriënboer, Vincent Michalski, Mehdi Mirza, Alberto Orlandi, Christopher Joseph Pal, Razvan Pascanu, Mohammad Pezeshki, Colin Raffel, Daniel Renshaw, Matthew Rocklin, Adriana Romero, Markus Roth, Peter Sadowski, John Salvatier, François Savard, Jan Schlüter, John Schulman, Gabriel Schwartz, Iulian Vlad Serban, Dmitriy Serdyuk, Samira Shabanian, Étienne Simon, Sigurd Spieckermann, S. Ramana Subramanyam, Jakub Sygnowski, Jérémie Tanguay, Gijs van Tulder, Joseph P. Turian, Sebastian Urban, Pascal Vincent, Francesco Visin, Harm de Vries, David Warde-Farley, Dustin J. Webb, Matthew Willson, Kelvin Xu, Lijun Xue, Li Yao, Saizheng Zhang, and Ying Zhang. 2016. Theano: A Python framework for fast computation of mathematical expressions. *CoRR* abs/1605.02688 (2016). http://arxiv.org/abs/1605.02688

[5] Arnon Amir, Pallab Datta, William P Risk, Andrew S Cassidy, Jeffrey A Kusnitz, Steve K Esser, Alexander Andreopoulos, Theodore M Wong, Myron Flickner, Rodrigo Alvarez-Icaza, Emmett McQuinn, Ben Shaw, Norm Pass, and Dharmendra S Modha. 2013. Cognitive computing programming paradigm: a corelet language for composing networks of neurosynaptic cores. In *Neural Networks (IJCNN), The 2013 International Joint Conference on*. IEEE, 1–10.

[6] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. 2015. Fixed point optimization of deep convolutional neural networks for object recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*. IEEE, 1131–1135.

[7] Ben Varkey Benjamin, Peiran Gao, Emmett McQuinn, Swadesh Choudhary, Anand R Chandrasekaran, Jean-Marie Bussat, Rodrigo Alvarez-Icaza, John V Arthur, Paul A Merolla, and Kwabena Boahen. 2014. Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proc. IEEE* 102, 5 (2014), 699–716.

[8] Mahdi Nazm Bojnordi and Engin Ipek. 2016. Memristive Boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. 1–13.

[9] Snaider Carrillo, Jim Harkin, Liam J McDaid, Fearghal Morgan, Sandeep Pande, Seamus Cawley, and Brian McGinley. 2013. Scalable hierarchical network-on-chip architecture for spiking neural network hardware implementations. *IEEE Transactions on Parallel and Distributed Systems* 24, 12 (2013), 2451–2461.

[10] Andrew S Cassidy, Paul Merolla, John V Arthur, Steve K Esser, Bryan Jackson, Rodrigo Alvarez-Icaza, Pallab Datta, Jun Sawada, Theodore M Wong, Vitaly Feldman, Arnon Amir, Daniel Ben-Dayan Rubin, Filipp Akopyan, Emmett McQuinn, William P Risk, and Dharmendra S Modha. 2013. Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores. In *Neural Networks (IJCNN), The 2013 International Joint Conference on*. IEEE, 1–10.

[11] Lukas Cavigelli and Luca Benini. 2016. A 803 gop/s/w convolutional network accelerator. *IEEE Transactions on Circuits and Systems for Video Technology* (2016).

[12] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ACM Sigplan Notices*, Vol. 49. ACM, 269–284.

[13] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*.

[14] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. 2015. Compressing neural networks with the hashing trick. In *International Conference on Machine Learning*. 2285–2294.

[15] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Teman. 2014. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 609–622.

[16] Y. H. Chen, T. Krishna, J. Emer, and V. Sze. 2016. 14.5 Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*. 262–263. https://doi.org/10.1109/ISSCC.2016.7418007

[17] Zhen Chen, Bin Gao, Zheng Zhou, Peng Huang, Haitong Li, and Wenjia Ma. 2015. Optimized learning scheme for grayscale image recognition in a RRAM based analog neuromorphic system. In *Electron Devices Meeting (IEDM), 2015 IEEE International*. IEEE.

[18] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. 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*. IEEE Press, 27–39.

[19] Ronan Collobert, Koray Kavukcuoglu, and Clement Farabet. 2011. Torch7: A Matlab-like Environment for Machine Learning. In *neural information processing systems*.

[20] Misha Denil, Babak Shakibi, Laurent Dinh, Marc' Aurelio Ranzato, and Nando de Freitas. 2013. 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.). Curran Associates, Inc., 2148–2156.

[21] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 92–104.

[22] Steven K Esser, Paul A Merolla, John V Arthur, Andrew S Cassidy, Rathinakumar Appuswamy, Alexander Andreopoulos, David J Berg, Jeffrey L McKinstry, Timothy Melano, Davis R Barch, Carmelo di Nolfo, Pallab Datta, Arnon Amir, Brian Taba, Myron D Flickner, and Dharmendra S Modha. 2016. Convolutional networks for fast, energy-efficient neuromorphic computing. *Proceedings of the National Academy of Sciences* (2016), 201604850.

[23] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. 2011. Neuflow: A runtime reconfigurable dataflow processor for vision. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*. IEEE, 109–116.

[24] Clément Farabet, Cyril Poulet, Jefferson Y Han, and Yann LeCun. 2009. Cnp: An fpga-based processor for convolutional networks. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. IEEE, 32–37.

[25] Steve B Furber, David R Lester, Luis A Plana, Jim D Garside, Eustace Painkras, Steve Temple, and Andrew D Brown. 2013. Overview of the spinnaker system architecture. *IEEE Trans. Comput.* 62, 12 (2013), 2454–2467.

[26] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William J. Dally. 2016. ESE: Efficient Speech Recognition Engine with Compressed LSTM on FPGA. *CoRR* abs/1612.00694 (2016). http://arxiv.org/abs/1612.00694

[27] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 243–254.

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

[29] Bart LM Happel and Jacob MJ Murre. 1994. Design and evolution of modular neural network architectures. *Neural networks* 7, 6 (1994), 985–1004.

[30] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. 1989. Multilayer feedforward networks are universal approximators. *Neural networks* 2, 5 (1989), 359–366.

[31] Miao Hu, Hai Li, Yiran Chen, Qing Wu, and Garrett S Rose. 2013. BSB training scheme implementation on memristor-based circuit. In *Computational Intelligence for Security and Defense Applications (CISDA), 2013 IEEE Symposium on*. IEEE, 80–87.

[32] Miao Hu, John Paul Strachan, Zhiyong Li, Emmanuelle M Grafals, Noraica Davila, Catherine Graves, Sity Lam, Ning Ge, Jianhua Joshua Yang, and R Stanley Williams. 2016. Dot-product engine for neuromorphic computing: programming 1T1M crossbar to accelerate matrix-vector multiplication. In *Design Automation*

Conference (DAC), 2016 53nd ACM/EDAC/IEEE. IEEE, 1–6.

[33] Eric Hunsberger and Chris Eliasmith. 2016. Training Spiking Deep Networks for Neuromorphic Hardware. *CoRR* abs/1611.05141 (2016). http://arxiv.org/abs/1611.05141

[34] Kyuyeon Hwang and Wonyong Sung. 2014. Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1. In *Signal Processing Systems (SiPS), 2014 IEEE Workshop on*. IEEE, 1–6.

[35] YU Ji, Youhui Zhang, ShuangChen Li, Ping Chi, CiHang Jiang, Peng Qu, Yuan Xie, and WenGuang Chen. 2016. NEUTRAMS: Neural Network Transformation and Co-design under Neuromorphic Hardware Constraints. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*.

[36] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 675–678.

[37] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *CoRR* abs/1704.04760. http://arxiv.org/abs/1704.04760

[38] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, and Andreas Moshovos. 2016. Stripes: Bit-serial deep neural network computing. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–12.

[39] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. 2016. Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 380–392.

[40] Yongtae Kim, Yong Zhang, and Peng Li. 2015. A Reconfigurable Digital Neuromorphic Processor with Memristive Synaptic Crossbar for Cognitive Computing. *J. Emerg. Technol. Comput. Syst.* 11, 4, Article 38 (April 2015), 25 pages. https://doi.org/10.1145/2700234

[41] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. 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.). Curran Associates, Inc., 1097–1105.

[42] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[43] Jun Haeng Lee, Tobi Delbruck, and Michael Pfeiffer. 2016. Training deep spiking neural networks using backpropagation. *Frontiers in Neuroscience* 10 (2016).

[44] Boxun Li, Yi Shan, Miao Hu, Yu Wang, Yiran Chen, and Huazhong Yang. 2013. Memristor-based approximated computation. In *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*. IEEE Press, 242–247.

[45] Boxun Li, Yi Shan, Miao Hu, Yu Wang, Yiran Chen, and Huazhong Yang. 2013. Memristor-based approximated computation. In *international symposium on low power electronics and design*. 242–247.

[46] Robert LiKamWa, Yunhui Hou, Julian Gao, Mia Polansky, and Lin Zhong. 2016. RedEye: analog ConvNet image sensor architecture for continuous mobile vision. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 255–266.

[47] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. 2015. Pudiannao: A polyvalent machine learning accelerator. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 369–381.

[48] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. 2016. Cambricon: An instruction set architecture for neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 393–405.

[49] Xiaoxiao Liu, Mengjie Mao, Beiye Liu, Hai Li, Yiran Chen, Boxun Li, Yu Wang, Hao Jiang, Mark Barnell, Qing Wu, and Jianhua Yang. 2015. RENO: A high-efficient reconfigurable neuromorphic computing accelerator design. In *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*. IEEE, 1–6.

[50] Zelda Mariet and Suvrit Sra. 2015. Diversity Networks. *CoRR* abs/1511.05077 (2015). http://arxiv.org/abs/1511.05077

[51] Karlheinz Meier. 2015. A mixed-signal universal neuromorphic computing system. In *Electron Devices Meeting (IEDM), 2015 IEEE International*. IEEE, 4–6.

[52] Paul A Merolla, John V Arthur, Rodrigo Alvarez-Icaza, Andrew S Cassidy, Jun Sawada, Filipp Akopyan, Bryan L Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, Bernard Brezzo, Ivan Vo, Steven K Esser, Rathinakumar Appuswamy, Brian Taba, Arnon Amir, Myron D Flickner, William P Risk, Rajit Manohar, and Dharmendra S Modha. 2014. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science* 345, 6197 (2014), 668–673.

[53] Dmytro Mishkin and Jiri Matas. 2015. All you need is a good init. *arXiv preprint arXiv:1511.06422* (2015).

[54] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*. 807–814.

[55] Mirko Prezioso, Farnood Merrikh-Bayat, BD Hoskins, GC Adam, Konstantin K Likharev, and Dmitri B Strukov. 2015. Training and operation of an integrated neuromorphic network based on metal-oxide memristors. *Nature* 521, 7550 (2015), 61–64.

[56] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark A Horowitz. 2013. Convolution engine: balancing efficiency & flexibility in specialized computing. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 24–35.

[57] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 267–278.

[58] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 14–26.

[59] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–12.

[60] 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. 2015. Development of a neuromorphic computing system. In *2015 IEEE International Electron Devices Meeting (IEDM)*. 4.3.1–4.3.4. https://doi.org/10.1109/IEDM.2015.7409624

[61] Dongjoo Shin, Jinmook Lee, Jinsu Lee, and HoiJun Yoo. 2017. DNPU: An 8.1TOPS/W Reconfigurable CNN-RNN Processor for General Purpose Deep Neural Networks. In *International Solid-State Circuits Conference*. IEEE.

[62] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[63] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. 2016. PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning. In *High Performance Computer Architecture (HPCA),2017 23rd IEEE Symposium on*. IEEE.

[64] Lili Song, Ying Wang, Yinhe Han, Xin Zhao, Bosheng Liu, and Xiaowei Li. 2016. 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*. IEEE, 1–6.

[65] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. 2011. Improving the speed of neural networks on CPUs. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, Vol. 1. Citeseer, 4.

[66] Karsten Wendt, Matthias Ehrlich, and René Schüffny. 2008. 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)*. World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 189–194. http://dl.acm.org/citation.cfm?id=1373936.1373969

[67] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 161–170.

[68] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–12.