A Portable, Automatic Data Quantizer for Deep Neural Networks

ABSTRACT

With the proliferation of AI-based applications and services, there are strong demands for efficient processing of deep neural networks (DNNs). DNNs are known to be both compute- and memory-intensive as they require a tremendous amount of computation and large memory space. Quantization is a popular technique to boost efficiency of DNNs by representing a number with fewer bits, hence reducing both computational strength and memory footprint. However, it is a difficult task to find an optimal number representation for a DNN due to a combinatorial explosion in feasible number representations with varying bit widths, which is only exacerbated by layer-wise optimization. Besides, existing quantization techniques often target a specific DNN framework and/or hardware platform, lacking portability across various execution environments. To address this, we propose *libnumber*, a portable, automatic quantization framework for DNNs. By introducing Number abstract data type (ADT), *libnumber* encapsulates the internal representation of a number from the user. Then the auto-tuner of *libnumber* finds a compact representation (type, bit width, and bias) for the number that minimizes the user-supplied objective function, while satisfying the accuracy constraint. Thus, *libnumber* effectively separates the concern of developing an effective DNN model from low-level optimization of number representation. Our evaluation using eleven DNN models on two DNN frameworks targeting an FPGA platform demonstrates over 8× (7×) reduction in the parameter size on average when up to 7% (1%) loss of relative accuracy is tolerable, with a maximum reduction of 16×, compared to the baseline using 32-bit floating-point numbers. This leads to an geomean speedup of 3.79× with a maximum speedup of 12.77× over the baseline, while requiring only minimal programmer effort.

1 INTRODUCTION

Deep neural networks (DNNs) are the key enabler for emerging AI-based applications and services. For example, convolutional neural networks (CNNs) are widely deployed for computer vision applications, such as image classification [22, 31] and feature detection [49, 55]. For natural language processing, recurrent neural networks (RNNs) [37, 48] and memory networks [51, 56] are used for question answering [47], voice recognition [13], and image captioning [39]. Recently, Deep Q-Network (DQN) has been proposed to combine DNNs with reinforcement learning to achieve near- or super-human performance in playing Atari games [17].

Although these networks can perform complicated tasks, they require a tremendous amount of computation. For example, ResNet-152 [22], one of the deepest CNNs, requires 22.6 giga-operations (GOPs) to process one image in ImageNet [11]. They also require large memory space up to hundreds of megabytes to store network parameters alone [19]. Thus, there are many proposals to exploit algorithmic characteristics and data locality to reduce both computational intensity and memory footprint [28].

Quantization is a popular method to achieve this by using a reduced-precision format to represent layer inputs, weights, or both. Representing a number with fewer bits accelerates computation and reduces memory footprint, hence improving overall energy efficiency. Since DNNs often have a lot of redundancy in network parameters, an 8-bit fixed-point type, instead of the 32-bit full-precision floating-point type, may be sufficient to execute them without noticeable degradation of the output quality [19, 34]. Log2 quantization [38, 52] represents a value with its exponent only (in power of two) to provide a very large dynamic range for a given number of bits and reduce the computational strength by substituting a floating-point multiply with an integer add. Due to intrinsic trade-offs between accuracy and performance, it is crucial to find an optimal quantization scheme for a given DNN.

However, finding suitable number representations for a given network is difficult due to a combinatorial explosion in feasible number representations with varying bit widths. Furthermore, an optimal representation may vary layer by layer as different layers have different degrees of performance and accuracy sensitivity to representational changes [26]. Existing quantization frameworks for DNNs have various limitations as they fail to support multiple types [8, 14, 25, 34, 38, 40, 58], layer-wise optimization [19, 40], and are bound to a specific hardware and DNN framework [1, 14, 40, 58] (e.g., Caffe on GPU [19, 59]). Besides, the optimization goal is often hard-coded in the tuning algorithm, thus failing to accommodate various deployment scenarios with different accuracy-performance trade-offs.

To address this challenge we propose *libnumber*, a portable, automatic data quantization framework for DNNs. *libnumber* provides a *portable* API that allows the user to specify her

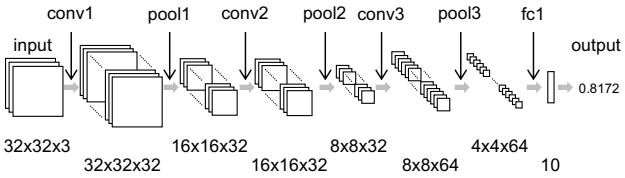
PACT-27 (Under Review), 2018
Unpublished working draft. Not for distribution.

optimization goals for a variety of DNN frameworks and hardware platforms. At the heart of the *libnumber* API is Number abstract data type (ADT), which subsumes most of the popular number representations for DNNs. Instead of specifying a concrete type and bit width (e.g., `float`), the user can declare those data she wants to quantize in a layer (e.g., inputs, weights, or both) as `Number` type. Then the auto-tuner of *libnumber* takes inputs from the user (for target layers, accuracy constraints, and optimization goals) and the platform engineer (for a list of supported types by the target platform) to find a suitable (type, bias, bit width) tuple for each layer efficiently, while satisfying the accuracy constraints. Using *libnumber*, we separate the concern of developing an effective DNN model from the concern of the low-level optimization of number representation.

To evaluate *libnumber* we use nine CNNs and two multilayer perceptrons (MLPs) on two DNN frameworks, Caffe [24] and DarkNet [45], targeting an FPGA platform. Unlike CPU and GPU, whose supported number types are already fixed, FPGA can flexibly accommodate a variety of types and bit widths, to make it an ideal platform to demonstrate the effectiveness of *libnumber*. According to our evaluation, *libnumber* reduces the size of the network parameters by 8.28× for the eleven models on average with a maximum reduction of 16.00× . While the size of the valid configuration space ranges from 7.40×10^4 to 1.65×10^{29}, the proposed auto-tuner finds a compact representation after navigating only through a tiny fraction of the configuration space (an order of 10^3 configurations in the worst case). Finally, our performance estimation via high-level synthesis (HLS) producing an FPGA bitstream from a quantized C++ code shows a geomean speedup of 3.79× with a maximum speedup of 12.77× over the 32-bit floating-point baseline for convolution layers, which account for over 90% of total execution time.

In summary, *libnumber* satisfies all of the following design goals:

- *High coverage* - It supports multiple data types (floating-point, fixed-point, and exponent in particular) with varying bit widths, biases, and layer-wise optimization to maximize bit savings.
- *Search efficiency* - The auto-tuning algorithm efficiently navigates through huge configuration space with a two-pass search to find an effective representation quickly.
- *Portability* - The C++ API provides a simple interface that can flexibly support multiple DNN frameworks and hardware platforms.
- *Ease of use* - It takes minimal programmer effort to port a new DNN framework to *libnumber*. The auto-tuner offers a simple Python-based interface for easy reconfiguration.



(a) Structure of Cuda-convnet

```
float output[out_c];
float input[in_c][h][w];
float filter[out_c][in_c][h][w];

// Computation of fully connected layer
for (int i=0; i<out_c; ++i) {
  for (int j=0; j<in_c; ++j) {
    for (int k=0; k<h; ++k) {
      for (int l=0; l<w; ++l) {
        output[i]+=filter[i][j][k][l]*
                   input[j][k][l];
}}}}
```

(b) FC layer reference code (simplified)

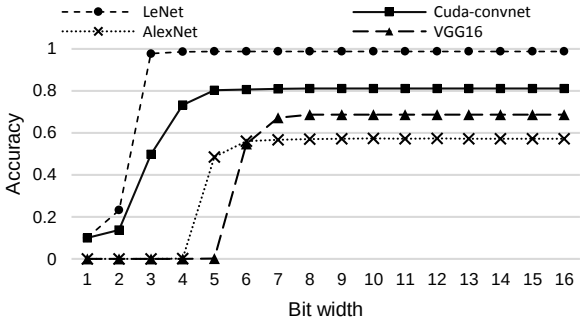
Figure 1: Cuda-convnet structure and fully-connected layer

The rest of this paper is organized as follows. Section 2 provides preliminaries for DNNs and motivates this work. Section 3 overviews the structure of *libnumber*, followed by the details of the API design (Section 4) and the auto-tuner (Section 5). Section 6 presents the evaluation methodology and the results. Finally, we briefly survey related work to ours (Section 7) and conclude the paper (Section 8).

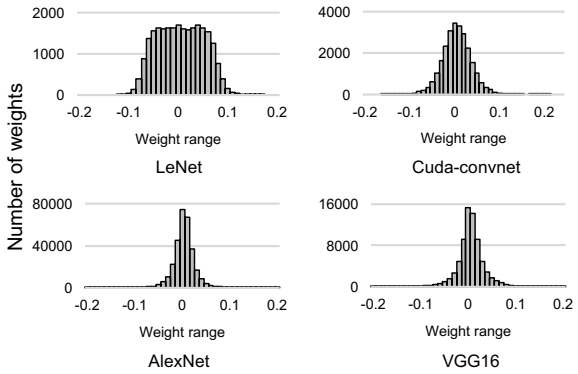
2 MOTIVATION

2.1 Redundancy in Deep Neural Networks

Figure 1(a) shows the structure of Cuda-convnet, which is used for image classification [29]. It takes as input an image file of 32×32 pixels with three input channels (for RGB color components) from CIFAR-10 dataset [30]. The image passes through a pipeline of kernels (layers), composed of three alternating pairs of convolution and pooling layers, followed by a fully-connected (FC) layer at the end. Each layer performs distinct matrix computations to extract more complex features toward the end of the pipeline. Finally, the FC layer is a classifier layer, which computes a vector of the probability for each of the 10 classes. In CNNs the convolution layers are known to be the most compute-intensive (consuming over 90% of total GOPs), while the FC layers account for a disproportionately large share of network parameters due to their full connectivity [29].



(a) Top-1 accuracy-bit width



(b) Distribution of weight parameters

Figure 2: Redundancy in CNN weight parameters

Figure 1(b) is a sequential C reference code for the FC layer. This kernel takes a three-dimensional matrix representing 64 4×4 feature maps as input to generate a one-dimensional probability vector for the 10 classes. The operations being performed are multiplying an input activation (`input[j][k][l]`) by a weight (`filter[i][j][k][l]`) and accumulating it into the output vector (`output[i]`). This multiply-accumulate (MAC) computation is the most common operations not only for the FC layer but also for the convolution layers.

However, it is well known that the network parameters in DNNs have a significant amount of redundancy [21] and we can improve computational efficiency by eliminating it. Figure 2(a) shows the top-1 accuracy of four popular CNNs with varying fixed-point bit widths from 1 through 16. All of the four CNNs fully recover their accuracy by using 8 or more bits. Moreover, LeNet requires only 3 bits to achieve the full accuracy.

Thus, there are ample opportunities for quantization to compress the network parameters and boost computational efficiency by using lower-precision arithmetic. Figure 2(b) shows the distribution of the weights for the four CNNs. The distribution features a relatively short dynamic range of the

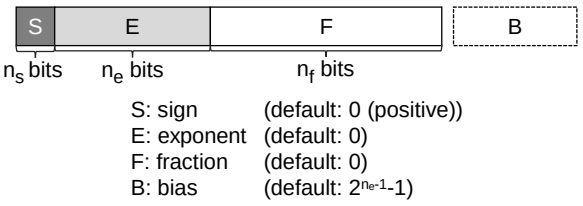


Figure 3: Canonical number format based on the IEEE 754 standard

values. If a wide dynamic range is unnecessary, simpler (and faster) fixed-point computation can replace expensive full-precision floating-point computation. To exploit these opportunities on a GPU, Nvidia has recently introduced mixed-precision support for their state-of-the-art CUDA 9 [15]. Microsoft's BrainWave deploys a custom 9-bit floating-point format, called *ms-fp9*, to achieve a sub-millisecond latency of a DNN inference task on their FPGA fabric [6].

There are two approaches to DNN quantization: training a quantized network from scratch [8, 9] and applying quantization to a pre-trained network (optionally followed by re-training to recover accuracy) [20, 21]. Although reported some success, training a quantized network is still a difficult problem and demonstrated to work only for relatively small networks [8, 9]. In contrast, the second approach is more attractive to many practitioners with lots of pre-trained models already available in the public domain [20]. Thus, we assume the second approach (i.e., quantizing a pre-trained model) for our work.

2.2 Optimizing Number Representations

Table 1 surveys popular number representations used by DNNs. Floating-point, fixed-point, and exponent are the three most widely used types for quantization. (The binary type is a special case of any of the three.) We observe that all of the three types can be represented by the canonical number format based on IEEE 754 Standard as shown in Figure 3. The canonical format consists of four fields: sign (S), exponent (E), fraction (F) and bias (B). If the lengths of the S, E, and F fields are denoted by n_s, n_e, and n_f, respectively, a number format can be characterized by a 4-tuple of $\langle n_s,n_e,n_f,B\rangle$. Assuming $E = \sum_{i=0}^{n_e-1} 2^i \cdot e_i$, $F = \sum_{i=0}^{n_f-1} 2^i \cdot f_i$,, where e_i and f_i denote the i-th bit of E and F, respectively, the number can be interpreted as follows:

$$\begin{cases} (-1)^S(1 + F \cdot 2^{-n_f})\, 2^{E-B} & if\ E \neq 0 \\ (-1)^S F \cdot 2^{-n_f} \cdot 2^{1-B} & if\ E = 0 \end{cases} \quad (1)$$

More specifically, the three popular number types are mapped to the the canonical format as follows. (1) N-bit *floating-point* (`FLOAT(N)`) is represented by a 4-tuple of $\langle 1,$

Format	Number Type	Bit width	Network	Quantized Layer	Canonical Form $\langle n_s, n_e, n_f, B \rangle$
FLOAT32	floating-point	32	Default for all DNNs	CONV, FC	⟨1, 8, 23, 127⟩
Half-precision		16	CNN, LSTM, DCGAN, DeepSpeech2 [15]	CONV, FC	⟨1, 5, 10, 15⟩
MS-fp9		9	DNNs on Brainwave [6]	CONV, FC	⟨–, –, –, –⟩
FIXED16	fixed-point	16	CaffeNet, VGG16, VGG16-SVD [40]	CONV, FC	⟨1, 0, 15, 0⟩
FIXED11		10	CNN [8]	CONV	⟨1, 0, 10, –⟩
FIXED8		8	MLP, CNN, LSTM [14]	CONV, FC	⟨1, 0, 7, –⟩
LogQuant4	exponent	4	AlexNet, VGG16 [38]	CONV, FC	⟨1, 3, 0, 3⟩
BIN	binary	1	BinarizedNN [9]	CONV, FC	⟨1, 0, 0, 0⟩

Table 1: Survey of popular number representations for DNNs (– means not disclosed.)

$n_e, n_f, B\rangle$, where $N = 1 + n_e + n_f$ and $B = 2^{n_e-1}$-1. For example, single-precision (`FLOAT(32)`) and half-precision (`FLOAT(16)`) types correspond to $\langle 1, 8, 23, 127\rangle$ and $\langle 1, 5, 10, 15\rangle$, respectively. (2) N-bit *dynamic fixed-point* (`FIXED(N)`) is represented by a 4-tuple of $\langle 1, 0, n_f, B\rangle$, where B is used to control the position of the radix point (zero by default). Unlike the floating-point type, the fixed-point type has a narrow dynamic range, but is easier to implement using integer arithmetic. (3) N-bit *exponent* (`EXP(N)`) is represented by a 4-tuple of $\langle 1, n_e, 0, B\rangle$, where $B = 2^{n_e-1}$-1 by default. The exponent type has the widest dynamic range at the cost of reduced precision.

In this setup, the task of optimizing number representations is reduced to finding an optimal set of the four parameters (n_s, n_e, n_f, and B) in the 4-tuple. However, this task easily becomes intractable due to a combinatorial explosion in feasible number formats. Suppose an FPGA device that can accommodate the three number types: floating-point (32 and 16 bits), fixed-point (1 through 32 bits), and exponent (2 through 9 bits). For example, if we are to find an optimal representation out of the 42 formats for both activations and weights, and each of the nine convolution layers in DarkNet [45], the search space is as large as $(42^2)^9 = 1.65\times10^{29}$.

To avoid this cost, prior proposals for quantization sacrifice coverage of the search space by either tuning the bit width only for a fixed number type [8, 14, 25, 34, 38, 40, 58] or turning off layer-wise optimization [19, 40]. This can lead to a suboptimal result. Furthermore, the proposed techniques often target a specific DNN framework and hardware device [1, 14, 40, 58], lacking portability across different execution environments for DNNs.

Thus, we need a quantization framework for DNNs with broad coverage in both optimization space and execution environments. The auto-tuner should find a suitable number format efficiently for each target layer. Besides, it is highly desirable for the framework to provide a portable, easy-to-use API that can flexibly support a variety of DNN frameworks and hardware platforms.

3 OVERVIEW

Figure 4 shows the overall operation flow with *libnumber*. The output of the quantization framework is a DNN framework optimized for a given network model by applying quantization to target layers as identified by the DNN developer. The two main components of *libnumber* are `Number` abstract data type (Number ADT) and an auto-tuner for number representations, which are presented in greater details in Section 4 and Section 5, respectively. The rest of this section sketches a flow of operations to quantize the activations and weights of each layer with *libnumber*.

Step 1 (Initializing the Auto-tuner). At the core of *libnumber* is the auto-tuner ❶ around which multiple components interact with each other. It can be flexibly configured using two descriptor files written in Python. The first one is a *platform descriptor* ❷ containing a key-value pair, where the key is a hardware device name and the value is a list of supported number formats. The second one is a *network descriptor* ❸, which specifies the optimization goal (i.e., accuracy tolerance, objective function) and a list of target layers among others. Using this information the auto-tuner determines search space, and enters the tuning loop to find a configuration that *minimizes* the objective function, denoted by f_{obj}, subject to satisfying the accuracy constraint.

Step 2 (Entering the Tuning Loop). Once initialization is completed, the auto-tuner selects the first configuration to try, and generates a Number configuration file ❹ in text format. This file contains a list of object name-number format pairs (e.g., `conv1.weight FLOAT 32`, `fc1.weight EXP 8 127`). Then the auto-tuner launches an inference job by invoking the *libnumber*-applied DNN framework ❺ in which data to be quantized (e.g., layer inputs, weights, or both) are declared as `Number` type. A `Number` type subsumes all number formats that can be represented in the canonical format in Figure 3. At runtime all `Number` objects in the DNN framework are bound to concrete number formats as specified by the Number configuration file.

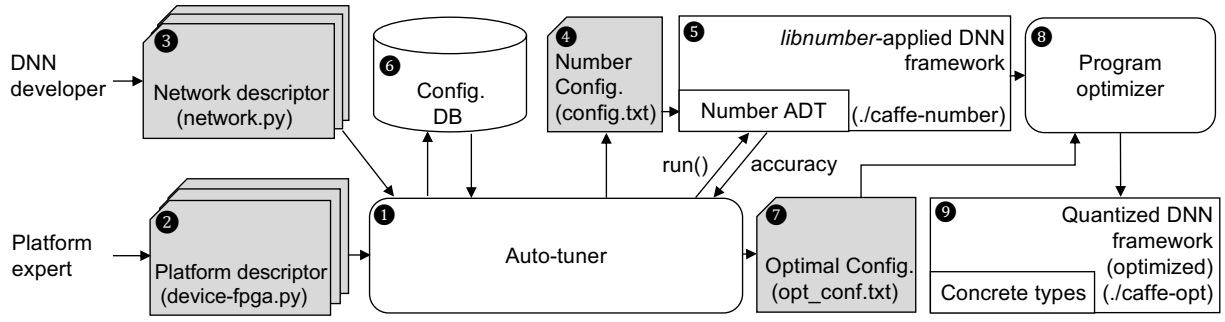


Figure 4: Overall operation flow of the *libnumber* quantization framework.

Step 3 (Assessing the Configuration). When the DNN inference is finished, the auto-tuner first updates the configuration database ❻ with accuracy and the value of $f_{obj}(cfg)$ for the current configuration `cfg`. If a configuration that minimizes f_{obj} is found, the auto-tuner reports the configuration in a plain text file (`opt_conf.txt`) ❼ and exits the loop. Otherwise, it selects the next candidate configuration and repeats the process in Step 2. The auto-tuning algorithm is presented in greater details in Section 5.

Step 4 (Generating Optimized DNN Kernels). Once the optimal configuration is found, the program optimizer ❽, either a human or compiler, generates optimized DNN kernels ❾ by applying quantization to the target layers. Since the optimal configuration file contains a list of object name-optimal format pair, the optimizer may simply replace each `Number` object with an object of the corresponding concrete type. Furthermore, reducing the bit width can expose additional optimization opportunities, which may not be feasible in the out-of-the-box kernels. At this point this transformation and optimization is done manually, but we believe this can be automated in the future.

4 NUMBER ABSTRACT DATA TYPE

API Design. Table 2 summarizes the C++ API of the `Number` ADT. The methods are divided into three categories: configuration, computation, and type conversion. The `parseConfig` method takes a `Number` configuration file generated by the auto-tuner as input and builds a dictionary of (object name: (type, format)) pairs (e.g., `conv1.weight: (EXP, <1, 7, 0, 63>)`). This dictionary is a global structure shared by all `Number` objects. `setName(key)` sets the name of the object (e.g., `conv1.weight`), which will be used as a key to access the dictionary, and initializes the type (`_type`) (e.g., `EXP`) and format attributes (e.g., `<1, 7, 0, 63>`) for the object. If a dictionary lookup fails, the full-precision 32-bit floating-point format is used by default.

There are three computational operators for the `Number` ADT: assignment, multiplication, and addition. When an assignment is made from another `Number` object, both the value and format attributes are copied over from the object. If the right-hand side (RHS) operand has a native type (e.g., `float`), its format attributes are set appropriately (e.g., (`FLOAT`, `<1, 8, 23, 127>`). For multiplication and addition, we generally follow the operation rules in the IEEE 754 standard. However, a special care should be given when the two operands have different format attributes. In this case, these methods internally convert both objects into full-precision 32-bit floating-point values, compute the result, and put it back to a `Number` object by taking the format attributes from the object with a wider dynamic range, except that we favor `FLOAT` type over `EXP` for higher precision. In this way we can prevent value saturation.

```
1  supported_formats = {
2       'FPGA':{                 # Platform name
3       'FLOAT': [16, 32],       # FLOAT 16 / 32 bits
4       'FIXED': range(2, 33),   # FIXED 2 to 32 bits
5       'EXP'  : range(2, 10)}} # EXP   2 to 9  bits
```

Figure 5: Example platform descriptor file

Finally, the `Number` ADT API provides a couple of type-conversion methods: `asFloat()` and `asInt()`. These methods extract the value from a `Number` object and export it as either `float` or `int` value.

Example of Applying `Number` ADT. Figure 6(a) shows a transformed reference code of the fully-connected layer taken from Figure 1(b), with modified part shown in gray. In this example, we assume the user wants to quantize both weight parameters and input activations; thus, we declare the `filter` and `input` arrays in `Number` type instead of `float` (Line 4 and 5). In Line 1, `parseConfig()` class method is invoked to load the format configuration from a `Number` configuration file. Figure 6(b) is an example of the file. It contains a list of object name and format pairs. The format field has two parameters, type and bit width, and an optional third parameter for bias (in Line 5, for example). By invoking `parseConfig`, a dictionary is created as shown in Figure 6(c).

Method	Descriptions
void Number::parseConfig (string filename)	Parse the Number configuration file to build a global object-format dictionary
void Number::setName (string key)	Set the name of the object and initialize the 4-tuple format attributes $\langle n_s,n_e,n_f,B\rangle$ for it
void Number::operator=()	Assign a value from another [Number|float|int] value
Number& Number::operator+()	Add a [Number|float|int] value to the Number value
Number& Number::operator*()	Multiply the Number value by a [Number|float|int] value
float Number::asFloat()	Export the Number value as a float type
int Number::asInt()	Export the Number value as an int type

Table 2: Number ADT API in C++

```
1  Number::parseConfig("config.txt");
2
3  float output[out_c];
4  Number input[in_c][h][w];
5  Number filter[out_c][in_c][h][w];
6  // Format setting
7  // 'layername' is passed by caller
8  setArrayName((Number*)filter,
9      out_c*in_c*h*w, layername+".weight");
10 setArrayName((Number*)input,
11     in_c*h*w, layername+".input");
12
13 // Computation of fully connected layer
14 for (int i=0; i<out_c; ++i) {
15   for (int j=0; j<in_c; ++j) {
16     for (int k=0; k<h; ++k) {
17       for (int l=0; l<w; ++l) {
18         output[i]+=(filter[i][j][k][l]*
19                     input[j][k][l]).asFloat();
20 }}}}
```

(a) FC layer with Number ADT

```
1  // config.txt
2  conv1.weight     EXP 8
3  conv1.input      EXP 6
4  ...
5  fc1.weight       FIXED   15  -3
6  fc1.input        FLOAT   32
7  ...
```

(b) Configuration file

```
1  // dictionary produced by Number::parseConfig
2  {
3      conv1.weight     : (EXP, <1,7,0,63>),
4      conv1.input      : (EXP, <1,5,0,15>),
5      ...
6      fc1.weight       : (FIXED, <1,0,14,-3>),
7      fc1.input        : (FLOAT, <1,8,23,127>),
8      ...
9  }
```

(c) Dictionary of object name (key) and format pairs

Figure 6: Application of Number ADT to fully-connected layer reference code (originally from Figure 1(b))

Lines 8-11 set the object configuration of each element in the filter and input arrays, respectively, by invoking setArrayName(), which internally calls the setName() method. In this example, object names are fc1.weight and fc1.input; thus, the format attributes will be set to be $\langle n_s,n_e,n_f,B\rangle$ = ⟨1, 0, 14, -3⟩ with _type = FIXED and $\langle n_s,n_e,n_f,B\rangle$ = ⟨1, 8, 23, 127⟩ with _type = FLOAT. The loop body in Lines 18-19 requires only minor modifications as the multiply operator (*) handles a multiply of a Number operand by a float operand. As the result has a Number type, it needs to be converted to a float value by invoking the asFloat method. Overall, it only takes minimal effort to port an existing DNN kernel to the Number ADT.

5 AUTO-TUNER DESIGN

5.1 Configuring the Auto-tuner

The auto-tuner takes two descriptor files to configure it: platform and network descriptors. The platform descriptor specifies a supported_formats attribute as shown in Figure 5. It contains a key-value pair, where the key is a device name and the value is a list of supported number formats. There can be multiple platform descriptor files to support multiple hardware devices. Besides, one can specify different supported types for activations and weights by appending .activation and .weight suffix to the device name.

The network descriptor file specifies various network parameters as summarized in Table 3. We need two command lines (cmd_smallset and cmd_fullset) to invoke the DNN framework with small and full test sets, respectively, as the tuning algorithm in Section 5.2 uses both. The user specifies an accuracy constraint and optimization goal by setting err_margin and f_obj. At this point we only use the default objective function, which is a sum of bit width-layer size products for all target layers, to minimize overall storage requirements for activations and weights. However, one can override this function to customize the optimization goal. A regular expression is provided by get_result_regex to extract the accuracy number from the console output at the end of execution of every run in the tuning loop. The user can also control layer-wise optimization for both activations and weights by setting the layerwise_opt field.

Platform descriptor		
Parameter	**Description**	**Example**
supported_formats	Key-value pair of device name and list of supported number formats	Refer to Figure 5

Network descriptor		
Parameter	**Description**	**Example**
platform	Device name	"FPGA"
cmd_smallset	Command line to run small test set	"./caffe test -model LeNet.prototxt -weight Lenet.caffemodel -iterations 10"
cmd_fullset	Command line to run full test set	"./caffe test -model LeNet.prototxt -weight Lenet.caffemodel -iterations 100"
err_margin	Error margin relative to baseline accuracy	0.07
f_obj	Objective function to specify optimization goal	"Default" (sum(bit_width[i]*layer_size[i]))
layer_name	Name of target layers in Python list format	["conv1", "conv2"]
sub_layer_name	Name of branches in same layer in Python list format	[[],["branch1", "branch2"...]...]
layerwise_opt	Enable layer-wise optimization [BOTH|WEIGHT-ONLY|ACTIVATION-ONLY|NONE]	BOTH
wgt_size	Size of target layers (weight count) in Python list format	[500, 25000]
act_size	Size of target layers (activation count) in Python list format	[784, 2880]
act_max_abs	Maximum absolute values of activations for target layers in Python list format	[0.996094, 5.05664]
wgt_max_abs	Maximum absolute values of weights for target layers in Python list format	[0.5794976, 0.16474459]
get_result_regex	Regular expression to extract accuracy from standard output	"Accuracy : (.*)\n"

Table 3: Auto-tuner configuration parameters

The remaining parameters are self-explanatory, except that `sub_layer_name` is used to group multiple sublayers using the same format. This feature is useful for optimizing networks with branches such as ResNet [22] and GoogleNet [16].

5.2 Tuning Algorithm

To achieve high quality and efficiency at the same time, the auto-tuner takes a two-pass approach, where each pass consists of two phases. During the first pass, the auto-tuner uses a small test set (specified by `cmd_smallset`) to reduce not only the number of iterations but also the cost of each iteration. Using a smaller dataset can yield an order of magnitude reduction in execution time, hence leading to significant savings in total search time, especially for large and deep networks. Starting from the best configuration of the first-pass, the second pass further tunes (type, bit width, bias) tuples using the full test set (specified by `cmd_fullset`). The rest of this section presents the details of the four phases described in Algorithm 1 and 2 using a real example of auto-tuning LeNet in Figure 7.

Phase 1 (Coarse-grained Search). The first phase of the algorithm is described by Line 1-10 in Algorithm 1. This phase performs a coarse-grained search to reduce the bit width starting from the baseline configuration of using `FLOAT(32)` for all layers (Line 1) and a small test set (Line 3). The coarse-grained search procedure (`CoarseGrainedSearch` in Line 9) reduces the bit width by half as long as the accuracy constraint is satisfied. Once the search hits the minimum bit width for `FLOAT`, it switches to the `FIXED` type and continues. If it hits a configuration that violates the accuracy constraint, it selects the configuration in the middle of the last two configurations to perform a binary search. Iterations 1-7 in

Algorithm 1 Auto-tuning algorithm

Input: Error margin err_margin
Output: Best configuration best_cfg

```
1:  init_cfg ← InitializedConfig()
2:
3:  test_set ← SmallSet
4:  threshold = GetAccuracy(init_cfg) × (1 - err_margin) // small set
5:
6:  /* Phase 1 (small set): Coarse-grained search */
7:  cfg ← init_cfg
8:  while GetAccuracy(cfg) >= threshold do
9:      cfg ← CoarseGrainedSearch(cfg)
10: end while
11:
12: /* Phase 2 (small set): Fine-tuning bias, bit width and type */
13: cfg ← FinetuneBiasBitWidthAndType(cfg)
14:
15: test_set ← FullSet
16: threshold = GetAccuracy(init_cfg)×(1 - err_margin) // full set
17:
18: /* Phase 3 (full set): Accuracy recovery */
19: if GetAccuracy(cfg) < threshold then
20:     cfg_set ← GetConfigsWithOneMoreBit(cfg) ∪
21:               GetConfigsWithDecrementedBias(cfg)
22:     while IsAllBelowThreshold(cfg_set) do
23:         cfg ← SelectBestConfig(cfg_set)
24:         cfg_set ← GetConfigsWithOneMoreBit(cfg) ∪
25:                   GetConfigsWithDecrementedBias(cfg)
26:     end while
27: end if
28:
29: /* Phase 4 (full set): Fine-tuning bias, bit width and type */
30: best_cfg ← FinetuneBiasBitWidthAndType(cfg)
```

Figure 7 illustrate this phase when tuning the two convolution layers (L1 and L2) of LeNet. As a result, 3-bit fixed-point

Algorithm 2 Fine-tuning bias, bit width and type

Input: Configuration cfg
Output: Best configuration best_cfg

```
1:  while True do
2:      cfg_set ← GetConfigsWithIncrementedBias(cfg)
3:      while ¬ IsAllBelowThreshold(cfg_set) do
4:          cfg ← SelectBestConfig(cfg_set)
5:          cfg_set ← GetConfigsWithIncrementedBias(cfg)
6:      end while
7:      cfg_set ← GetConfigsWithOneFewerBit(cfg)
8:      while ¬ IsAllBelowThreshold(cfg_set) do
9:          cfg ← SelectBestConfig(cfg_set)
10:         cfg_set ← GetConfigsWithOneFewerBit(cfg)
11:     end while
12:     cfg_set ← GetConfigsWithTypeChange(cfg)
13:     if IsAllBelowThreshold(cfg_set) then
14:         break
15:     else
16:         cfg ← SelectBestConfig(cfg_set)
17:     end if
18: end while
19: return cfg
```

(I3) is selected as best configuration thus far (Iteration 6). The procedure also attempts to further reduce the bit width using the EXP type but fails (Iteration 7). Note that the biases surrounded by parentheses are initialized based on value profiling, as summarized by `act_max_abs` and `wgt_max_abs` parameters in Table 3.

Phase 2 (Fine-tuning Bias, Bit Width and Type). The second phase performs a fine-tuning of bias, bit width and type as described in Algorithm 2, starting from the best configuration from Phase 1. First, the algorithm attempts to tune bias for fixed-point and exponent types (Line 2-6), which is crucial to balance the precision and dynamic range of a quantized number. The goal is to minimize the dynamic range for each layer (i.e., maximizing bias), while satisfying the accuracy constraint. Stripes [27] performs similar tuning to minimize the number of integer bits. In Line 2, we first collect a set of all configurations (`cfg_set`) whose bias is incremented by one from the current configuration at a target layer (for either activations or weights. For example, Iterations 8-11 in Figure 7 show the four elements in `cfg_set`. Among them the third one (I3(0)-I3(3) for weights and I3(0)-I3(-3) for activations in Iteration 10) is selected due to highest accuracy, where I3(0) denotes 3-bit fixed-point (I) with bias 0. This process is repeated until no configuration satisfies the accuracy constraint.

Second, the algorithm proceeds to reduce bit width (Line 7-11). In Line 7, it first collects a set of all configurations (`cfg_set`) that use one fewer bit than the current configuration for either activations or weights. In Figure 7, if the

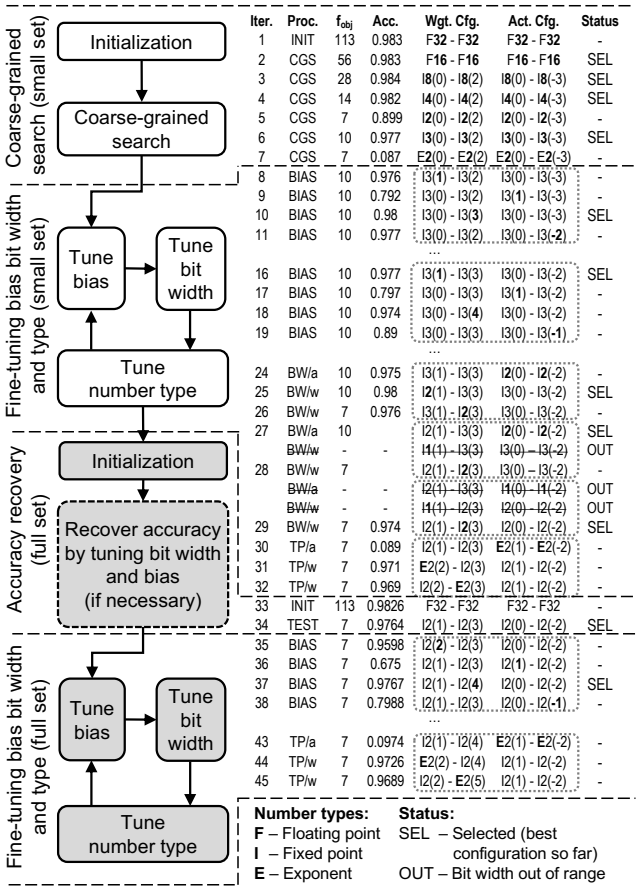


Figure 7: Auto-tuning the two CONV layers of LeNet with maximum accuracy loss of 1% (accuracy threshold: 0.9728). Biases (*B*) are shown in parentheses. Layer-wise optimization is enabled only for weights.

current configuration is I2(1)-I3(3) for weights and I3(0)-I3(-2) for activations (selected at Iteration 25), the three configurations in Iterations 27-28 represent `cfg_set`. Then the configuration that gives the most benefit (i.e., maximizing $\Delta f_{obj}(cfg)$) at the minimum cost of accuracy loss (i.e., minimizing $\Delta accuracy(cfg)$) is selected. This process is repeated until no configuration satisfies the accuracy constraint (Line 8-11). For example, the minimum bit widths are found at Iteration 29 in Figure 7 (I2(1)-I2(3) for weights and I2(0)-I2(-2) for activations).

Finally, we perform fine-tuning of the number type (Line 12-17). In this step, we first collect a set of all configurations (`cfg_set`) that have only one type different from the current configuration. In Figure 7, if the current configuration is I2(1)-I2(3) for weights and I2(0)-I2(-2) for activations (from Iteration 29), the configurations in Iterations 30-32 represent `cfg_set`. If no configuration satisfies the accuracy constraint, the algorithm exits the loop; otherwise, it selects the one with the highest benefit-to-loss ratio as in the previous step, and

Framework	Network	Dataset	Metric
Caffe	MLP-M [2]	MNIST [32]	Accuracy
	MLP-C [53]	CIFAR-10 [30]	Accuracy
	LeNet [33]	MNIST [32]	Accuracy
	Cuda-convnet [29]	CIFAR-10 [30]	Accuracy
	AlexNet [31]	ImageNet [11]	Accuracy
	NiN [35]	ImageNet [11]	Accuracy
	VGG-16 [50]	ImageNet [11]	Accuracy
	GoogleNet [16]	ImageNet [11]	Accuracy
	ResNet-50 [22]	ImageNet [11]	Accuracy
DarkNet	DarkNet-ref [45]	ImageNet [11]	Accuracy
	Tiny YOLO [46]	VOC2007 [18]	mAP

Table 4: Network models for evaluation

performs tuning of bias and bit width again. In the example of LeNet, the aforementioned configuration from Iteration 29 is selected as the best one.

Phase 3 (Accuracy Recovery). From this phase the algorithm enters the second pass to use the full test set for evaluating accuracy (Line 15-27 in Algorithm 1). Since we change the test set, there is no guarantee that the current best configuration still satisfies the accuracy constraint. If the current configuration satisfies the constraint, the algorithm just moves to the next phase; otherwise, it increases the bit width and/or decrements the bias until the constraint is satisfied (Line 22-26). This algorithm is the same as the fine-tuning algorithm for bit width in Phase 2 except that it increments (decrements) the bit width (bias) instead of decrementing (incrementing) it. This loop is repeated until the algorithm finds the first configuration whose accuracy is above the threshold. In the example of LeNet in Figure 7, we skip this phase as the accuracy constraint is already satisfied.

Phase 4 (Fine-tuning Bias, Bit Width and Type). Finally, we repeat the same fine-tuning process of bias, bit width and number type as in Phase 2, but using the *full* test set. In Figure 7 the algorithm finally outputs the best configuration for LeNet, which is I2(1)-I2(4) for weights and I2(0)-I2(-2) for activations.

6 EVALUATION

6.1 Methodology

To evaluate *libnumber*, we use nine CNN and two MLP models as summarized in Table 4. We port two popular DNN frameworks to *libnumber* to demonstrate its portability: Caffe [24] and DarkNet [45]. For Tiny YOLO we use a mean-average precision (mAP) metric for quantifying object detection accuracy; for the others image classification accuracy. Following the methodology of Stanford DAWNBench [12], we target 93% relative accuracy of the full-precision 32-bit floating-point type (i.e., `err_margin` = 0.07) by default. We have also tested 99% accuracy to observe similar results, which are omitted due to limited space. As discussed in Section 2.1, we quantize a pre-trained model for the given accuracy constraint [21, 59, 60].

We quantize both weights and activations of convolution layers (for CNNs) and fully-connected layers (for MLPs), but layer-wise optimization is applied to weights only. Otherwise, we would have to pay the cost of format conversion across layers, which may nullify performance benefits. We use the default objective function (f_{obj}) for the auto-tuner, which minimizes the total number of bits for both weights and activations, and hence storage requirements as well as computational strength.

We compare the quantization quality of *libnumber* against two well-known frameworks: Ristretto [20] and the Stripes quantizer [27]. We have faithfully reproduced both algorithms and validated them so that they perform at least comparably to or better than the reported results under the same constraints. Note that Stripes quantizes activations only and uses the `FIXED(16)` type for weights. For fair comparison we also run *libnumber* under the same constraints and compare speedups on a Stripes-like bit-serial hardware.

Our target hardware is a Xilinx Kintex UltraScale KU115 FPGA platform [23]. The FPGA can flexibly accommodate a variety of number formats featuring a large search space, to make it an ideal test vehicle for our work. For performance evaluation we use Xilinx Vivado High-Level Synthesis (HLS) tool Version 2017.4. The HLS tool compiles a C++ kernel to generate FPGA bitstream, from which we can calculate the cycle count and resource utilization of the kernel. The baseline implementation of HLS-generated kernels is optimized following the methodology of Zhang et al. [57] to achieve competitive performance to theirs. Once the best configuration is found by a quantizer, we manually apply a transformation to the original kernel, to replace the `Number` ADT with a concrete, quantized type. Then we re-optimize the tiling factors and apply a bit-packing optimization, which packs multiple weights into a single word for SIMD-style execution and memory bandwidth reduction. Note that this is a new optimization enabled by quantization, and cannot be applied to the baseline.

6.2 Results

Search Quality. Figure 8 compares the size of network parameters at the best configuration for 7% and 1% error tolerance. The results are normalized to the 32-bit floating-point baseline as in [20]. *libnumber* reduces the parameter size by 8.91× and 7.13× on average and total storage requirements (including activations) by 8.28× and 6.44× for 7% and 1% tolerance, respectively. These numbers translate to 69.6% (38.1%) reduction of storage space over Stripes (Ristretto) for 7% tolerance. Note that Stripes has a constant size of 16 bits as its weight format is fixed to `FIXED(16)` [27]. Both Ristretto and Stripes fail to tune NiN within 1% accuracy loss

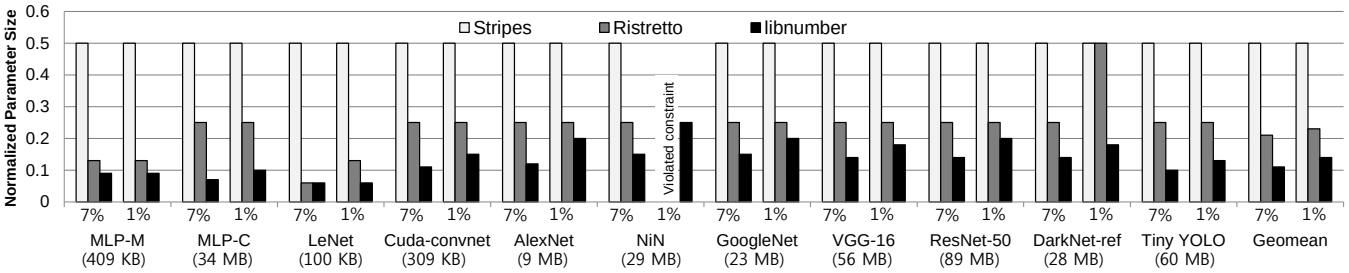


Figure 8: Model parameter size normalized to the `FLOAT(32)` baseline with 7% and 1% of accuracy tolerance. Lower is better. Network name is annotated with the baseline parameter size.

Network	Framework	Per Layer Format (Act/Weights)
MLP-M	*libnumber*	I3 / E3-E2-E3
	Ristretto	I2 / I4-I4-I4
MLP-C	*libnumber*	I4 / I2-E3-I3-I4
	Ristretto	I2 / I8-I8-I8-I8
LeNet	*libnumber*	I2 / E2-E2
	Ristretto	I2 / I2-I2
Cuda-convnet	*libnumber*	I3 / I4-E3-I4
	Ristretto	I2 / I8-I8-I8
AlexNet	*libnumber*	I5 / I5-I6-E4-E3-E4
	Ristretto	I4 / I8-I8-I8-I8-I8
NiN	*libnumber*	I6 / I4-I4-I6-E4-I5-I5-I6-I4-I6-E4-I5-I6
	Ristretto	I8 / I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8
VGG-16	*libnumber*	I8 / I5-I3-I7-I5-I5-I4-E4-I4-I4-I5-I6-I4-I3
	Ristretto	I8 / I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8
GoogleNet	*libnumber*	I6 / I6-I6-I5-I6-I4-I6-I5-I5-I5-I5-I4
	Ristretto	I8 / I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8
ResNet-50	*libnumber*	I7 / I6-I7-I5-I6-I6-I4-I5-I5-I5-I5-I4-I4-I5-I6-I5-I3-I4
	Ristretto	I8 / I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8
DarkNet	*libnumber*	I7 / I8-I7-I8-I6-I6-I6-E4-I5
	Ristretto	I8 / I8-I8-I8-I8-I8-I8-I8-I8
Tiny YOLO	*libnumber*	I8 / I6-I6-I7-I6-I5-I5-I3-E3-I5
	Ristretto	I8 / I8-I8-I8-I8-I8-I8-I8-I8-I8

Table 5: Best configuration: I*n* (`FIXED`), E*n* (`EXP`)

Network	# of target layers	# of iterations		
		Exhaustive	*libnumber*	Ratio
MLP-M	3	3.11×10^{6}	190	6.11×10^{-5}
MLP-C	4	1.31×10^{8}	359	2.75×10^{-6}
LeNet	2	7.41×10^{4}	74	9.99×10^{-4}
Cuda-convnet	3	3.11×10^{6}	208	6.68×10^{-5}
AlexNet	5	5.49×10^{9}	419	7.63×10^{-8}
NiN	12	1.27×10^{21}	2759	2.18×10^{-18}
VGG-16	13	5.31×10^{22}	3516	6.62×10^{-20}
GoogleNet	11	3.01×10^{19}	1783	5.92×10^{-17}
ResNet-50	17	1.65×10^{29}	5471	3.31×10^{-26}
DarkNet-ref	8	4.07×10^{14}	1130	2.78×10^{-12}
Tiny YOLO	9	1.71×10^{16}	1644	9.63×10^{-14}

Table 6: Comparison of search costs

as NiN requires careful bias tuning when using fixed-point numbers.[1]

Table 5 shows the best configurations found by both *libnumber* and Ristretto. *libnumber* yields more compact representation to reduce total storage requirements by up to 71.3% for MLP-C with an average of 38.1%. This performance gap is attributed to the difference in the coverage of the configuration space. Ristretto has a much narrower format coverage as it considers only `FIXED(`2^n`)` formats and does not perform layer-wise optimization. For example, the search space of *libnumber* has 1.65×10^{29} configurations for ResNet-50, whereas Ristretto has only 6.

Search Cost. Another important metric is the cost of search. Table 6 summarizes this cost for *libnumber* in terms of the number of iterations (i.e., tested configurations), which ranges from 74 to 5471, even if the search space can be as large as 1.65×10^{29}. The auto-tuning algorithm navigates through only a tiny fraction of the search space to find best configuration. Besides, the two-pass algorithm, using both small set and full set, effectively reduces the average cost per iteration. Using the small set yields a maximum speedup of 4.48× for Cuda-convnet with an average speedup of 2.52× over a version of *libnumber* using the full set only.

Speedups on FPGA. Figure 9(a) shows the speedups for the target layers of 11 DNNs, synthesized by HLS for an FPGA. *libnumber* achieves a 3.79× geomean speedup over the baseline `FLOAT(32)` version, which translates to 93% and 18% speedups over Stripes and Ristretto, respectively. By representing numbers with fewer bits for each layer, *libnumber* significantly improves resource utilization. In particular, this enables more weights and activations to be packed in a single 32-bit word, leading to fewer bank accesses for a given amount of computation. Besides, reduced bit widths allow us to increase parallelism and execute more operations in a SIMD style. The relative speedup of *libnumber* is more pronounced for Stripes than Ristretto. It is because Ristretto often uses just a few more bits per layer than *libnumber* to end up packing the same number of elements per 32-bit word. In all cases MLPs demonstrate greater speedups than CNNs as they have more parallelism for not using convolutions.

Ideal Speedups on Bit-serial Hardware. The benefits of quantization can be best realized by bit-serial hardware, whose latency for a MAC operation scales (almost) linearly to the number of bits [3, 27]. We use a very simple analytical

[1] Stripes [27] reports successful quantization of NiN within 1% accuracy loss, but their baseline (`FIXED(16)`) is different from ours (`FLOAT(32)`) having a higher baseline accuracy.

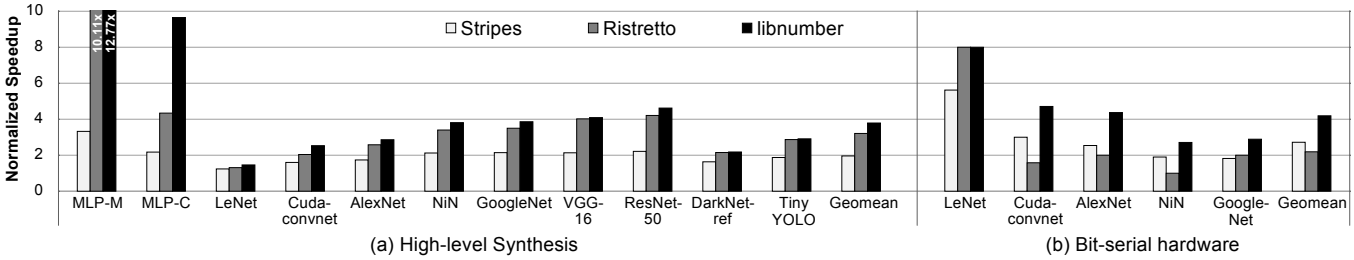


Figure 9: Normalized speedups using (a) High-level Synthesis (HLS) for FPGA and (b) bit-serial hardware

performance model to estimate ideal performance as in [27]. We assume a DNN accelerator like Stripes [27], which is based on DaDianNao [7] but serialized. Assuming the same constraints as Stripes (i.e., 16-bit fixed-point baseline, quantizing activations only using dynamic fixed-point type, 1% accuracy tolerance), we run *libnumber* to quantize the target layers. The corresponding bit widths for Stripes are taken from the original paper, instead of using our version of the Stripes quantizer. Assuming 100% PE utilization and 1-cycle latency for a 1-bit serial operation, the execution time of a kernel on DaDianNao is estimated to be the total GOPs of the MAC operations for the layer divided by the total number of PEs (in ops/cycle). We also assume a linear scaling of the throughput to reduction in bit width (i.e., speedup is $16/p$ if p is the bit width). Note that Stripes achieves speedups within 2% of the ideal speedups calculated similarly [27].

Figure 9(b) shows ideal speedups of Stripes, Ristretto, and *libnumber* over DaDianNao. We only use a subset of 5 DNN models that are also used for evaluating Stripes [27]. *libnumber* achieves a geomean speedup of 4.19×, which outperforms both Ristretto and the Stripes quantizer. In this setup, a reduction in bit width leads to performance boost more directly than FPGA, to demonstrate greater performance gap between *libnumber* and the other two quantizers.

Portability and Programmer Effort. Finally, we estimate the programmer's porting effort to *libnumber* by counting the number of modified lines of code (LoC). As we target an FPGA device, we use as baseline a sequential C++ version of the convolution kernel, which takes 696 (540) LoC for Caffe (DarkNet). The number of added/modified/deleted lines for porting is 64 and 57 for Caffe and DarkNet, respectively. Overall, porting takes only modest effort as the modified LoC is just a small fraction of the total LoC of the framework, which is 63,733 (25,144) for Caffe (DarkNet).

7 RELATED WORK

Quantization for DNNs. There are proposals to reduce the precision of data to improve performance of pre-trained DNNs with little degradation of accuracy [9, 15, 25, 26, 34, 36, 38, 44]. However, they have limited coverage by considering only one type of numbers [25, 26, 38] or supporting mixed types in a very limited way [15]. Deep compression [21] uses a value clustering technique to identify representative values. However, these techniques suffer from irregular memory accesses and inefficient computation due to an extensive use of high-precision values. In contrast, *libnumber* selects an suitable representation for each layer by considering both number type and bit width to effectively eliminate redundancy in numbers. Zhou et al.[59] propose a quantization technique that utilizes multiple number types, and demonstrate savings in bit count. However, they have much narrower type coverage by using zero and exponent types only, which can be problematic for complex networks. Besides, their work lacks evaluation on a realistic hardware platform.

Tuning Algorithms and Frameworks. Even before emergence of DNNs auto-tuning has been investigated in the research community for a long time, for compiler optimization [4, 5, 10, 41, 54], runtime parallelism adaptation [42, 43], and so on. While these frameworks may be used for tuning DNN workloads, their efficiency will be much lower without considering DNNs' algorithmic characteristics. In contrast, *libnumber* employs an efficient tuning algorithm customized for quantizing DNNs.

8 CONCLUSION

This paper introduces *libnumber*, a portable C++ API and auto-tuning framework that optimizes number representation for each layer of a DNN. By introducing the `Number` ADT, *libnumber* encapsulates the internal representation of a number, thus separating the concern for developing an effective DNN from the concern of optimizing the number representation at a bit level. While the task of quantization has been performed in an ad hoc manner, *libnumber* proposes a systematic approach to it by providing a common API and a flexible auto-tuner. The resulting quantization framework is easy to use, requiring minimal programmer effort, while producing a high-quality search for a suitable number representation for each layer of a DNN. We plan to port other DNN frameworks to *libnumber* and release it as open-source in the future.

REFERENCES

[1] TensorFlow: How to Quantize Neural Networks with TensorFlow. https://www.tensorflow.org/performance/quantization.

[2] TensorFlow mechanics 101. https://github.com/tensorflow/tensorflow/tree/r1.2/tensorflow/examples/tutorials/mnist.

[3] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. 2017. Bit-pragmatic Deep Neural Network Computing. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '17)*.

[4] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*.

[5] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*.

[6] Doug Burger. 2017. Microsoft unveils Project Brainwave for real-time AI. https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/. 2017.

[7] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. 2014. DaDianNao: A Machine-Learning Supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '14)*.

[8] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2014. Training deep neural networks with low precision multiplications. 2014. arXiv:1412.7024

[9] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. 2016. arXiv:1602.02830

[10] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. 2017. End-to-End Deep Learning of Optimization Heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT '17)*.

[11] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR '09)*.

[12] Cody A. Coleman et al. 2017. DAWNBench: An End-to-End Deep Learning Benchmark and Competition. https://github.com/stanford-futuredata/dawn-bench-entries. 2017.

[13] G. Hinton et al. 2012. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*2012.

[14] Norman P. Jouppi et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. 2017. arXiv:1704.04760

[15] Paulius Micikevicius et al. 2017. Mixed Precision Training. 2017. arXiv:1710.03740

[16] Szegedy et al. 2015. Going Deeper With Convolutions. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR '15)*.

[17] Volodymyr Mnih et. al. 2015. Human-level control through deep reinforcement learning. *Nature*2015.

[18] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. 2010. The pascal visual object classes (voc) challenge. *International journal of computer vision*2010.

[19] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. 2016. Hardware-oriented approximation of convolutional neural networks. 2016. arXiv:1604.03168

[20] Philipp Gysel, Jon Pimentel, Mohammad Motamedi, and Soheil Ghiasi. 2018. Ristretto: A Framework for Empirical Study of Resource-Efficient Inference in Convolutional Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems*2018.

[21] Song Han, Huizi Mao, and William J. Dally. 2015. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. 2015. arXiv:1510.00149

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR '16)*.

[23] Xilinx INC. Xilinx Kintex UltraSCALE FPGA Family. https://www.xilinx.com/products/silicon-devices/fpga/kintex-ultrascale.html.

[24] Yangqing et al. Jia. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia (ACMMM '14)*.

[25] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, Raquel Urtasun, and Andreas Moshovos. 2015. Reduced-precision strategies for bounded memory in deep neural nets. 2015. arXiv:1511.05236

[26] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M. Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Proteus: Exploiting numerical precision variability in deep neural networks. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*.

[27] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos. 2016. Stripes: Bit-serial deep neural network computing. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '16)*.

[28] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. 2015. Compression of Deep Convolutional Neural Networks for Fast and Low Power Mobile Applications. 2015. arXiv:1511.06530

[29] Alex Krizhevsky. 2012. cuda-convnet: High-performance c++/cuda implementation of convolutional neural networks. https://code.google.com/p/cuda-convnet/. 2012.

[30] Alex Krizhevsky and Geoffrey Hinton. 2009. Learning multiple layers of features from tiny images. 2009.

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *Advances in neural information processing systems*.

[32] Yann LeCun. 1998. The MNIST Database of Handwritten Digits. *http://yann.lecun.com/exdb/mnist/*1998.

[33] Yann LeCun et al. LeNet-5, convolutional neural networks.

[34] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. 2016. Fixed point quantization of deep convolutional networks. In *Proceedings of the 33rd International Conference on Machine Learning (ICML '16)*.

[35] Min Lin, Qiang Chen, and Shuicheng Yan. 2013. Network In Network. 2013. arXiv:1312.4400

[36] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. 2015. Neural Networks with Few Multiplications. 2015. arXiv:1510.03009

[37] T. Mikolov, S. Kombrink, L. Burget, J. Cernocky, and S. Khudanpur. 2011. Extensions of recurrent neural network language model. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '11)*.

[38] Daisuke Miyashita, Edward H. Lee, and Boris Murmann. 2016. Convolutional neural networks using logarithmic data representation. 2016. arXiv:1603.01025

[39] Cesc Chunseong Park, Byeongchang Kim, and Gunhee Kim. 2017. Attend to You: Personalized Image Captioning with Context Sequence Memory Networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR '17)*.

[40] Jiantao et al. Qiu. 2016. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate*

Arrays (FPGA '16).

[41] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*.

[42] Arun Raman, Hanjun Kim, Taewook Oh, Jae W. Lee, and David I. August. 2011. Parallelism orchestration using DoPE: the degree of parallelism executive. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*.

[43] Arun Raman, Ayal Zaks, Jae W. Lee, and David I. August. 2012. Parcae: a system for flexible parallel execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*.

[44] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. 2016. arXiv:1603.05279

[45] Joseph Redmon. 2013-2016. Darknet: Open source neural networks in c. http://pjreddie.com/darknet/. 2013-2016.

[46] Joseph Redmon and Ali Farhadi. 2016. YOLO9000: better, faster, stronger. 2016. arXiv:1612.08242

[47] Mengye Ren, Ryan Kiros, and Richard Zemel. 2015. Exploring models and data for image question answering. In *Advances in neural information processing systems*.

[48] Hochreiter S. and Schmidhuberm J. 1997. Long short-term memory. *Neural Computation*1997.

[49] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. FaceNet: A unified embedding for face recognition and clustering. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR '15)*.

[50] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. 2014. arXiv:1409.1556

[51] Sainbayar Sukhbaatar, arthur szlam, Jason Weston, and Rob Fergus. 2015. End-to-end memory networks. In *Advances in neural information processing systems*.

[52] Hokchhay Tann, Soheil Hashemi, R Iris Bahar, and Sherief Reda. 2017. Hardware-software codesign of accurate, multiplier-free Deep Neural Networks. In *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*. IEEE, 1–6.

[53] Rakesh Vasudevan. CIFAR-10 Classifier. https://github.com/vrakesh/CIFAR-10-Classifier.

[54] Zheng Wang, Dominik Grewe, and Michael F. P. O'boyle. 2014. Automatic and Portable Mapping of Data Parallel Programs to OpenCL for GPU-Based Heterogeneous Systems. *ACM Transaction on Architecture and Code Optimization*2014.

[55] Ouyang Wanli and Xiaogang Wang. 2013. Joint deep learning for pedestrian detection. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV '13)*.

[56] Jason Weston, Sumit Chopra, and Antoine Bordes. 2015. Memory Networks. In *Proceedings of the International Conference on Learning Representations (ICLR '15)*.

[57] 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 (FPGA '15)*.

[58] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. 2017. Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*.

[59] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. 2017. Incremental network quantization: Towards lossless cnns with low-precision weights. 2017. arXiv:1702.03044

[60] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. 2016. Trained ternary quantization. 2016. arXiv:1612.01064