Lan HY, Wu LY, Zhang X *et al.* DLPlib: A library for deep learning processor. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 32(2): 286–296 Mar. 2017. DOI 10.1007/s11390-017-1722-2

DLPlib: A Library for Deep Learning Processor

Hui-Ying Lan [1,2,3], Lin-Yang Wu [1,2,3], *Student Member, CCF*, Xiao Zhang [1,2], Jin-Hua Tao [1,2], Xun-Yu Chen [1,2] Bing-Rui Wang [1,2,4], Yu-Qing Wang [1,2,4], Qi Guo [1,2], *Member, CCF*, and Yun-Ji Chen [1,2]

[1] *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences Beijing 100190, China*

[2] *Microprocessor Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China*

[3] *University of Chinese Academy of Sciences, Beijing 100049, China*

[4] *Department of Computer Science, University of Science and Technology of China, Hefei 230026, China*

E-mail: {lanhuiying, wulinyang, zhangxiao, taojinhua, chenxunyu, wangbingrui, wangyuqing, guoqi, cyj}@ict.ac.cn

Received November 2, 2016; revised February 13, 2017.

Abstract Recently, deep learning processors have become one of the most promising solutions of accelerating deep learning algorithms. Currently, the only method of programming the deep learning processors is through writing assembly instructions by bare hands, which costs a lot of programming efforts and causes very low efficiency. One solution is to integrate the deep learning processors as a new back-end into one prevalent high-level deep learning framework (e.g., TPU (tensor processing unit) is integrated into Tensorflow directly). However, this will obstruct other frameworks to profit from the programming interface. The alternative approach is to design a framework-independent low-level library for deep learning processors (e.g., the deep learning library for GPU, cuDNN). In this fashion, the library could be conveniently invoked in high-level programming frameworks and provides more generality. In order to allow more deep learning frameworks to gain benefits from this environment, we envision it as a low-level library which could be easily embedded into current high-level frameworks and provide high performance. Three major issues of designing such a library are discussed. The first one is the design of data structures. Data structures should be as few as possible while being able to support all possible operations. This will allow us to optimize the data structures easier without compromising the generality. The second one is the selection of operations, which should provide a rather wide range of operations to support various types of networks with high efficiency. The third is the design of the API, which should provide a flexible and user-friendly programming model and should be easy to be embedded into existing deep learning frameworks. Considering all the above issues, we propose DLPlib, a tensor-filter based library designed specific for deep learning processors. It contains two major data structures, tensor and filter, and a set of operators including basic neural network primitives and matrix/vector operations. It provides a descriptor-based API exposed as a C++ interface. The library achieves a speedup of 0.79x compared with the performance of hand-written assembly instructions.

Keywords deep learning processor, API, library, DLPlib

1 Introduction

Today, deep learning has been applied to solve various tasks (e.g., image classification, object recognition, speech recognition, natural language process). With the model becoming more sophisticated, the computation workload is also increasing rapidly. Deep learning processors have been proved to be a good solution accelerating the computing processes. Right now, developers program the deep learning processors through writing assembly instructions by bare hands, which requires large amount of workload and possible unnecessary repeating implementation by different programmers. Therefore, a programming environment which can make the development of deep learning processor more productive, is imperative.

Regular Paper

This work is partially supported by the National Natural Science Foundation of China under Grant Nos. 61432016, 61472396, 61473275, 61522211, 61532016, 61521092, 61502446, 61672491, 61602441, and 61602446, the National Basic Research 973 Program of China under Grant No. 2015CB358800, and the Strategic Priority Research Program of the Chinese Academy of Sciences under Grant No. XDB02040009.

One solution of designing such a programming interface is to directly integrate the deep learning processors as a new back-end device into one prevalent high-level deep learning framework. This has proved to be a valid solution: researchers at Google built the tensor processing unit (TPU) that is an accelerator for machine learning, integrated it as a back-end device into the Tensorflow framework, and achieved an order of magnitude improvement of performance. The drawback of such a solution is that programmers can use the deep learning processors through only one framework, Tensorflow, and it is impossible for other frameworks to obtain benefits from such programming interface. An alternative approach is to design the programming interface as a framework-independent low-level library as the deep learning library cuDNN for GPUs. In this fashion, it will be rather easy to integrate the hardware into existing frameworks through calling functions of the library to finish computations. As a programming interface designed specific for deep learning processors and aiming at releasing end-users from writing low-level assembly instructions, we envision it as a low-level library instead of a back-end implementation of an existing framework.

To design such a library, we consider three major issues. The first one is the design of data structures. The library should include only a few data structures which are capable of representing all types of data. Fewer data structures make the optimization towards data structures easier; at the same time, we should make sure the generality is not compromised. The second issue is the selection of operations that is a trade-off between flexibility and efficiency. And the third one is the design of the API, which should support the data structures and the operations in a flexible and easy-to-use fashion. In addition, in order to allow popular frameworks to gain benefits from our library, the API should make it easy to be integrated into current mainstream frameworks such as Caffe, Tensorflow, and MXNet.

In this work, we propose DLPlib, a low-level library for deep learning processors to bridge the gap between the hardware and the programmers. In DLPLib, all data are represented as tensors or filters, and operators are considered as transformations of the tensors. Operators include memory operators and computational operators. The former is in charge of copying and allocating memories, and the latter includes a set of deep learning primitives (e.g., convolution, pooling) and common basic matrix/vector operations (e.g., matrix multiplication). We build the library on an architecture similar to the recently proposed accelerator, Cambricon-X. We achieve an average speedup of 0.79x on seven single-layer benchmarks and two entire network benchmarks. Although the performance of DLPlib is slightly slower than that of the hand-written instructions, the library still achieves a speedup of 4.1x compared with the GPU according to the comparison of Cambricon-X and GPU[1].

The contributions of this paper are as follows.

• We design and implement a library specific for deep learning processors. It is a tensor-filter-based library which includes a set of deep learning primitives and vector/matrix operations.

• We design a descriptor-based API which provides a flexible and easy-to-use programming model and allows the library to be conveniently integrated into existing frameworks. And we provide an implementation of integrating DLPlib into Caffe.

• We provide an experimental study which compares the performance of hand-written assembly code and the library on six single-layer benchmarks and two entire network benchmarks. The library achieves 0.79% performance of the hand-written code, which is much better than GPU.

The rest of this paper is organized as follows. Section 2 discusses the related work of deep learning algorithms, accelerators, and deep learning libraries and frameworks. Section 3 introduces the architecture and major components of DLPlib. Section 4 demonstrates the programming model of building a network by DLPlib. In Section 5, the implementation of the library is described. Section 6 presents the way of integrating DLPlib into a very popular deep learning framework, Caffe. In Section 7, the performance of the library is evaluated. Finally, Section 8 makes the conclusion of this paper.

2 Related Work

2.1 Deep Learning Algorithms

In recent years, deep learning algorithms (i.e., deep neural network algorithms) have achieved great success in lots of fields[2-6]. There are two tendencies of the development of neural networks. Firstly, the networks are becoming more sophisticated and having larger scales, which leads to a rapid increasing requirement of computational workloads. Such computation-demanding tasks are the ground of the prevalence of neural network accelerators. Another development tendency of deep learning is that with more and more fields

starting adopting deep learning technology, more new and unique algorithms are proposed. In addition to visual tasks which achieved great success of applying deep learning, other fields such as speech recognition and natural language processing also began to adopt deep learning techniques (e.g., RNN, LSTM), and obtained rather promising results. This situation requires the deep learning processors to fit in more complex and various tasks. DLPlib is in the right position of enabling the processor to have enough generality.

2.2 Deep Learning Processors

Conventionally, neural network algorithms are implemented on general-purpose processors, e.g., GPUs/CPUs. However, there are special operations, e.g., convolution, that are not fit for GPUs/CPUs. In addition, today, the neural network models are becoming increasingly sophisticated, and the computational workloads are also increasing. All networks that were introduced previously require large computational workloads and thus usually take days or even weeks to train by using GPUs/CPUs as back-ends. In order to accelerate this process, researchers have begun to explore the possibilities of using different hardware to implement deep learning algorithms (e.g., FPGA[7-8], ASIC[9-11]). Chakradhar *et al.*[12] proposed a dynamically configurable coprocessor for convolution neural networks, which supports various scales of network through signals configuration. Chi *et al.*[13] designed an architecture called PRIME, which adopts ReRAM crossbar array as both the main memory and neural network accelerators. Shafiee *et al.*[14] proposed an architecture that uses crossbar array to not only store input and weight but also perform dot-product operations.

Recently, the DianNao family[9-11,15], which is a series of architectures aiming to perform machine learning algorithms with higher efficiency and lower power, has been proposed. DianNao[9] is an accelerator of large-scale CNNs and DNNs with special consideration of the impact of memory, performance and energy. DaDianNao[10] was proposed as a multi-chip system, which further improves the speed of processing even larger scale models. ShiDianNao[15] designed an energy efficiency architecture by explored the important property of convolutional neural networks (CNNs) that a lot of weights are shared among neurons. Chen *et al.* proposed Cambricon[11], an instruction set architecture (ISA) designed specific for implementing neural network techniques, using matrix/vector/scalar operators to implement different algorithms. Zhang *et al.*[1] designed Cambricon-X, which is an accelerator for sparse neural networks.

2.3 Deep Learning Frameworks and Libraries

With the thriving of deep learning, GPU has become one of the most popular and efficient devices of implementing neural network algorithms. In order to further improve the performance of GPU-based deep learning algorithms, NVIDIA (a GPU vendor), proposed cuDNN[17], a library which includes a set of efficient deep learning primitives (e.g., convolution, pooling, LRN). It releases the researchers from implementing and optimizing their own version of basic neural network operations (e.g., Convolution). It provides a flexible, easy-to-use C-language API for deep learning workloads, and has been integrated into many popular deep learning frameworks (e.g., Caffe, Tensorflow).

Abadi *et al.*[17] recently published a paper about the prevailing machine learning system, Tensorflow, where a custom designed ASIC known as the tensor processing units (TPUs) is integrated as a computational backend into the system and achieves an order of magnitude improvement in performance-per-watt. This shows the great potential of deep learning processors: with a proper programming interface, the power of deep learning processors will be fully released.

3 Library Design

3.1 Overview

There are two major components of DLPlib, data structures and operators. In DLPlib, data is represented as a tensor or a filter, and operators are considered as the transformation of the tensors.

We design operators as a set of operations that can be used to construct neural networks conveniently. In DLPlib, a primitive (i.e., operator) is considered as an independent operating unit. By calling a sequence of operations one after another, a network is implemented. All operations are executed linearly, the processor devotes its entire resources to performing one task, and the later operation will not be invoked until the previous one is finished. This strict constraint makes sure the correctness while the calls are getting more complex, and allows us to focus on optimizing each operation.

Fig.1 represents the dataflow of a 3-layer network. Tensors are initially stored on CPU (host), and copied to the device before computation. Adjacent operations share the same block of neurons as the output or input.

For example, the output neurons of *Conv* operation are as the input neurons of the *Pooling* operation. Synapses also need to be resident on the device while performing the operation. When the computation of each layer in the network is finished, the final output data (i.e., output neurons of operation *FC*) should be copied back to CPU as the result.

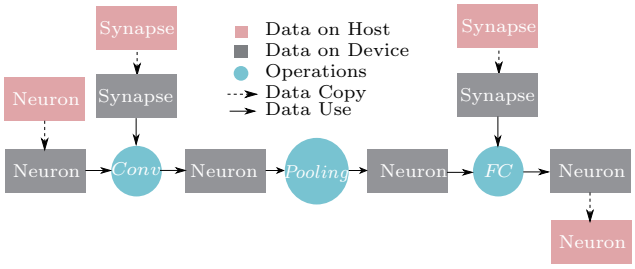
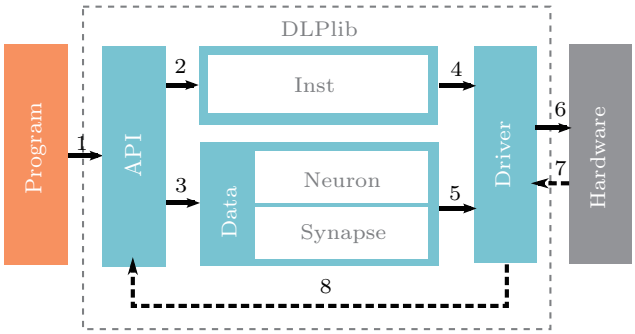


Fig.1. Dataflow of a 3-layer network.

Fig.2 demonstrates the overall flow of invoking the library to perform the inference of one layer, i.e., convolution, on deep learning processors.



(a)

1. Call API
2. Use model parameters to generate instructions
3. Prepare data
4∼6. Send instructions and data to hardware through the driver
7. Send the end signal back to host
8. Inform the library that execution is finished

(b)

Fig.2. (a) Overall flow of performing convolution by DLPlib. (b) Descriptions of the operation steps of the overall flow.

1) The host program is called, and thus invokes the library.

2) Through the API, a desired neural network model is constructed and all relative information is packed for instructions generation.

3) Instructions are generated by analyzing the parameters of the neural network model, and then sent to the hardware by the driver.

4) By invoking memory interfaces, data, e.g., neurons (including input and output) and synapses, are prepared and copied to the accelerator memory by the driver.

5) When the execution function is called, a start signal is sent to the device. After receiving the signal, instructions will be executed.

6) While the execution is done, an end signal will be sent to the host. By then, the whole process is finished.

3.2 Data Structure

In DLPlib, there are two major data structures, tensors and filters, which are derived from the two core concepts, neurons and synaptic weights in the neural network. Tensors refers to the input and output data of operations (e.g., convolution, pooling). In many frameworks (e.g., Tensorflow, Caffe), both neurons and synaptic weights are encapsulated in the tensor structure. In DLPlib, only neurons are represented as tensors, and synaptic weights are encapsulated as filters. Many deep learning frameworks use one data structure to represent both neurons and synaptic weights. For example, in Caffe, both neurons and weights are encapsulated in the Blob structure. In DLPlib, we define neurons and synaptic weights as two independent data structures (i.e., tensors and filters) due to the special architecture of the deep learning processor, Cambricon-X. In Cambricon-X[1], neurons and synaptic weights are stored in different buffers, neuron buffers (NBs) and synapse buffers (SBs) respectively, and the data arrangement and accessing pattern are very different.

For some operations (e.g., convolution, fully-connected), input and output neurons are connected by synaptic weights, which are learned through the training process. An operator takes a tensor as input and produces a tensor as output.

Tensor. In DLPlib, we define a data structure, tensor, which is a dense n-dimensional ($n \leqslant 4$) array, to represent neurons and bias (all data except for synapses) in DLPlib. Elements in a tensor are stored as a 1D-array (a sequence of memory), and the tensor is treated as an n-dimensional array. For example, convolutional operations take a 4D-tensor as input and a 4D-tensor as output, and fully-connected operations take a 2D-tensor as input and output a 2D-tensor.

Tensors are described by several attributes, and parameters of a 4D-tensor are showed in Table 1. *N, C, H, W* represent the size of the four dimensions, which denote the number of samples (mini-batch), the number of feature maps, the height of one feature map, and

the width of one feature map respectively. The data format is an enumeration type variable used to indicate the data layout of the tensor. The order of these letters implies the data arrangement of the tensor. For example, $NCHW$ indicates that the W stride is 1, the H stride is W, the C stride is $H \times W$, and the N stride is $C \times H \times W$. The data type indicates the data type of elements in the tensor.

Table 1. Parameters of Tensor Structure

Parameter	Description
N	Number of samples
C	Number of feature maps
H	Height of sample
W	Width of sample
Data format	$NCHW$, $NHWC$
Data type	Float32, Float16

For simplicity, we only provide a 4D-tensor data structure. Tensors with less dimensions are also represented by the same set of parameters. For example, a 2D-tensor can be regarded as a 4D-tensor which has the parameters H and W of 1.

Filter. The synaptic weight is a unique concept in neural networks. In DLPlib, synaptic weights are represented as a filter, which represents the learned synapses data of convolution and fully-connected operations. Similar to the tensor, the filter is also stored as a 1D-array and treated as an n-dimensional array. Table 2 shows the parameters of a convolutional filter. The four dimensions, OC, IC, K_h and K_w are used to indicate the number of output feature maps, the number of input feature maps, the height and the width of the kernels respectively.

Table 2. Parameters of Filter Structure

Parameter	Description
OC	Number of output feature maps
IC	Number of input feature maps
K_h	Height of kernel
K_w	Width of kernel
Data format	$NCHW$, $NHWC$
Data type	Float32, Float16

Although DLPlib has a host-called API, the input and output data of computational functions should be resident on the memory of the deep learning processor. The memory management fashion is similar to cuDNN and cuBLAS, i.e., tensors and filters are copied to the device before computation. The copying process is performed by explicitly calling the memory copy function provided by the library.

3.3 Operators

Operators are the fundamental compute unit in DLPlib. The design of operators is a trade-off between the efficiency and the flexibility of programming. We make selection by studying the current popular networks and picking the most commonly used operations, which will help the users to build a network more easily. In order to balance the flexibility, DLPlib also provides basic vector/matrix computations which allow users to implement new and more complex operations[18]. In addition, DLPlib provides a series of functions to catenate, split and reshape the data.

An operator takes m ($m > 0$) input tensors and n ($n > 0$) output tensors. For some operators, a set of attributes are provided to describe their computational behavior (e.g., *Conv., Pooling*). We introduce several representative operators and their attributes as follows.

Conv.. Convolution is the most important layer in convolutional neural networks (CNNs). It takes a 4D-tensor as input, and outputs a 4D-tensor. The output is computed by using a set of filters convolving across through the input. DLPlib supports the conventional 2D-convolution operation[19], whose attributes are listed in Table 3. S_h and S_w represent the sliding strides of the height and the width directions respectively. P_h and P_w indicate the padding sizes used to adjust the height and the width of the input tensor respectively. The output size is computed by (1).

$$H_o = \lceil \frac{H_i - K_h + 1 + 2P_h}{S_h} \rceil, \tag{1}$$

where H_o and H_i represent the output and the input height respectively. The output width is computed following the same manner.

Table 3. Parameters of Different Operators

Operator	Parameters
Conv.	P_h, P_w, S_h, S_w
Pooling	K_h, K_w, P_h, P_w, S_h, S_w, *pool_type*
LRN	k, N, α, β
Active	*Active func.*

Pooling. Pooling layer is widely used in neural networks for reducing the size of feature maps by down-sampling. In DLPlib, two types of down-sampling are provided: max-pooling and avg-pooling. The output is computed by performing the maximum or average operation to each local window of the input data. As listed

in Table 3, the *Pooling* operator has seven attributes, S_h and S_w represent the size of strides, and P_h and P_w represent the padding sizes, which are similar to the convolution attributes. K_h and K_w indicate the height and the width of the sliding window respectively, and *pool_type* indicates the operation to each window, maximum or average.

LRN. In neural networks, the local response normalization (LRN) layer is used to aid generalization[2,20-21]. The output is computed by

$$Nout^i_{x,y} = Nin^i_{x,y} / \left(k + \alpha \times \sum_{j=\max(0,i-N/2)}^{\min(i-1,i+N/2)} (Nin^j_{x,y})^2 \right)^{\beta},$$

where $Nin^i_{x,y}$ is the input neuron located at (x, y) on input feature map i, $Nout^i_{x,y}$ is the output neuron located at (x, y) on output feature map i, N is the number of related input feature maps, and k, α, β are constant parameters. Accordingly, k, N, α, β are the attributes of LRN operators.

Active. Non-linear functions are used in neural networks to transform the output neurons to a certain interval. DLPlib provides three most commonly used active functions, *Sigmoid*, *Tanh*, and *ReLU*. *Sigmoid* function applies $f(x) = \frac{1}{1+e^{-x}}$ to each neuron of input. *Tanh* activates the input neurons by $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. *ReLU* is a recently increasing popular active function for its outstanding performance. It applies non-saturating nonlinear function $f(x) = \max(0, x)$ to the input neurons. In DLPlib, an active operation is treated as an independent operator instead of a part of other operations in order to make the abstraction clearer and the programming more flexible.

Vector/Matrix Operation. Due to the fact that many neural network techniques can be decomposed into matrix/vector operations[16], we provide a set of basic vector/matrix operations to enhance the flexibility of DLPlib, including matrix multiply, element-wise addition, subtraction, multiplication, etc. For example, batch normalization operation[22], which has been used in some state-of-the-art networks[23], can be performed by addition and multiplication operators.

4 Implementation

We implement DLPlib as a C++-based library for portability and efficiency. In this section, we discuss the implementation of different modules of DLPlib.

In DLPlib, parameters of the tensor, filter and operator are all encapsulated in the descriptor structures. A descriptor is implemented as a *struct* type which contains all necessary parameters. Through these descriptors, different structures can be easily described, e.g., a block of data can be described as a tensor by attaching a tensor descriptor to it. Parameters of operators are also encapsulated in the descriptor structures. Each operator has a corresponding descriptor to store the related parameters, e.g., *convolutionDescriptor* for convolution operator.

The kernel component of DLPlib is the operator, which performs the computation by generating instructions and invoking the device. The process of implementing an operator is demonstrated in Fig.3. First, the operator function takes the data pointers of input neurons, optional synaptic weights, and output neurons, and descriptors of the tensors, the filter, and the operator to generate corresponding instructions. Then these instructions are sent to the device through the driver. After that, a signal is sent to the device to start the computation. While the execution is finished, the device sends a finish signal back to the CPU (host) to create an interrupt. Finally, the CPU (host) processes the interrupt and continues the execution.

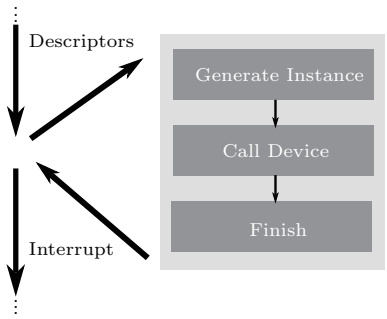


Fig.3. Runtime of an operator.

Optimization is a very important issue while implementing the library. For our deep learning processor, the performance is primitively decided by the parallel degree of computing and memory copying. The most effective and straightforward approach of improving the processing speed is to increase overlapping of computation and memory operations as much as possible. For operations with high computational intensity, such as convolution, the computation will take more time than the memory accessing and therefore become the bottleneck of optimizing. In this case, the optimal solution is that the time cost by memory accessing can be covered

entirely by the time cost by computation. And this is the main idea of optimizing the library.

5 API and Programming Model

DLPlib has a descriptor-based API where both data structures and operators are defined by the descriptors. For example, a tensor is represented by a tensor descriptor and a pointer to a block of memory.

As showed in Fig.4, the basic programming model can be concluded as three steps: 1) initialization of device, memory and operator, 2) the execution of all operators, and 3) releasing resources.

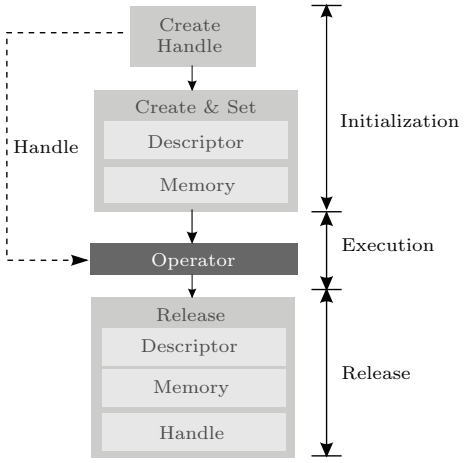


Fig.4. Programming model of DLPlib.

Initialization. There are several things that need to be initialized before calling the device. First, the device is initialized by creating a deep learning processor handle (*dlpHandle*), which is used to point to the hardware and preserves the context of the library. The handle is defined as a global variable, and the operator and memory management functions invoke it to use the processor. Second, all memories should be allocated in this phase. In DLPlib, data is stored in the tensor and filter structures (as described in Subsection 3.2). Parameters of tensors or filters are encapsulated in *descriptor* structures, which are defined for storing attributes. DLPlib provides *create, set, get, destroy* methods to manipulate the descriptors. Methods of memory management are defined as member methods of class *DDRManager*, which is contained in the global handle we previously created.

Execution. After setting up with the data and descriptors, we can simply call the operator to carry out the computation. The operators are executed by the calling sequence, in order to assure the correctness.

Releasing Resources. When the computation is finished, we need to release the resources we have used, i.e., allocated memory, descriptors and the device handle. DLPlib provides free functions for all resources, and the programmers need to invoke these functions manually to release allocated memory, thus preventing memory leakage.

Example. We demonstrate an example of performing convolutional layer by using the DLPlib API. Fig.5 shows the example code of implementing the operation. We omit the creation of descriptors and the specific parameters for simplicity, and describe the programming process by presenting the functions calling. The handle creation and memory copy happen before the computational function is called. And when the computation is finished, programmers should invoke the release functions (*dlpFree, destroyDlpHandle*) to free the memory and device resources.

```
// device handle
createDlpHandle(handle);
// create data
dlpMalloc(input);
dlpMemcpy(input);
dlpMalloc(weight);
dlpMemcpy(weight);
dlpMalloc(output);
// create descriptor for
   convolution
createConvDescriptor(desc_conv);
// execute convolution forward
dlpConvolutionForward(desc_conv,
   input, weight, output);
// free
dlpFree(input);
dlpFree(output);
dlpFree(weight);
destroyDlpHandle(handle);
```

Fig.5. Example of convolution.

6 Integration with Caffe

Although CNN has been adopted to solve many tasks, the programming is still a barrier. The deep learning framework has been proposed as a solution to reduce the cost of building a deep network model. As a low-level library for a specific back-end device, DLPlib is designed to be able to be integrated into existing frameworks simply and elegantly.

Caffe is an extensively used deep learning framework. In this section, we present the Caffe integration with our library. For the high modularity of Caffe, it is convenient to append new implementations to the framework while retaining the core unaltered. The modifications are mostly additive.

6.1 Framework Structure

Fig.6 shows a basic process of the inference execution of Caffe, which follows a two-step manner: preparation and computation. In the first step, it builds the whole neural network and prepares data and attributes of each layer for the next step. In the second step, the computations will be executed. The behavior of step 2 is determined by step 1. We elaborate the integration as follows.

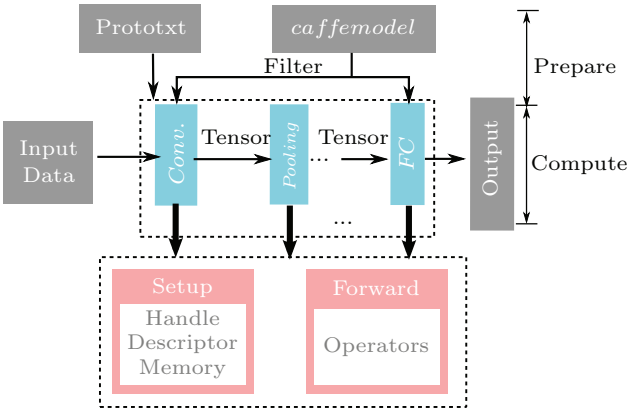


Fig.6. Integrating DLPlib into Caffe.

Preparation. Caffe uses a prototxt file to describe the topology structure of the network, and a *caffemodel* file to encapsulate the trainable parameters. The Neural network model is built according to the prototxt file. For networks with branches, such as GoogLeNet, the model will be parsed as a sequence of layers, which satisfies topological ordering. Caffe uses a structure named *Blob* as the interface to control storage and communication between the host and the device. A *Blob* holds dimension parameters (N, C, H, W) and pointers referring to memory blocks of different back-ends. Trainable parameters from *caffemodel* will be loaded into the *Blob* structure of corresponding layers. Moreover, attributes of each layer, such as the output dimensions, will also be completed in this phase. According to these, Caffe will allocate spaces on the host and the device (if needed) for the input and the output neurons.

Computation. At the beginning of this step, the whole model should have been completely prepared for every aspect that might affect the computation. In this step, Caffe will call the forward functions to perform the computation and acquire the final results.

6.2 Integration

To integrate DLPlib into Caffe, we need to modify the *proto* file, expand the corresponding cases (e.g., layer factory) to alter the control flow, and add new layers.

Prototxt. Proto file declares data structures that hold attributes of all layers. DLPlib is added as a new value of the enumeration type *Engin*, and goes along with other engines, i.e., *CUDNN* and *CAFFE*.

Blob. Both tensors and filter data are encapsulated in the *Blob* structure. We modify the *Blob* class by adding a new pointer field referring to the data for DLP and extending the memory copy function to support data transfer between DLPlib devices and hosts.

Layer. Caffe encapsulates neural network operations, such as *Conv.*, *Pooling*, in *Layer*. To add a new implementation, we define new inherited layers, and override the *setup*, *Reshape* and *Forward* functions.

As showed in Fig.2, in the *setup* function, we create descriptors that would be used later, and in the *Reshape* function, set them up. For a convolution layer, five descriptors will be needed, which are for the input tensor, output tensor, filter, bias tensor and convolution operator. There are two approaches to computing output dimensions, through the helper function provided by DLPlib or using Caffe's built-in function *computeOutputDim*(), and either is workable for the DLPlib.

In function *Forward*, we utilize operators provided in DLPlib to perform the algorithm. For instance, to implement a convolution layer, we adopt the convolution operator to get the final results. Since all data and parameters are already set in previous steps, we can simply feed the convolution operator with data and descriptors, and invoke the function. For layers that do not have a corresponding operator, we use combinations of existing operators to achieve the same results.

The setup step prepares memory and attributes for the next step while the forward step carries out the computation. This two-step schema can be mapped to DLP's programming model. For DLPlib, descriptor configuring and memory allocating can naturally fit into the preparing step, and the operators are called in the *Forward* function. Currently the library only supports inference process. Hence we will bypass the *Backward* function in this paper.

7 Experiment

In this section, we evaluate the performance of DLPlib. We describe the comparative baselines and the benchmarks, and then report the performance results.

7.1 Experimental Methodology

Baseline. We evaluate the performance of DLPlib by comparing the performance before and after using DLPlib. In this experiment, we consider only the speed to measure the performance of the library. Both DLPlib and assembly instructions are executed on the same back-end. The assembly instructions are written, obeying the principle of optimization introduced in Section 4, and the overlapping between computation and memory accessing is fully considered while implementing.

Benchmark. As shown in Table 4, seven single-layer benchmarks and two entire-network benchmarks are selected to evaluate the performance and also demonstrate the flexibility of our library. The single-layer benchmarks are extracted from existing network models[2,24], and two entire networks are included, AlexNet and VGG16, which are representative for their heavy computational workloads and widely utilization by various scenarios.

Back-End. In this experiment, we use an architecture similar to Cambrian-X[11], which is implemented in Verilog, compiled and simulated by Synopsys Verilog Compiler Simulator (VCS). The accelerator includes 16 processing elements (PEs), each of which has 16 multipliers, one 16-in adder-tree, and a synapse buffer (SB) of 2 KB. It also includes two on-chip neural buffers (NBin and NBout), each with 8 KB of memory. And we achieve a frequency of 1 GHz.

7.2 Experimental Results

Results. We compare our library against the hand-written assembly instructions on all networks and layers listed in Table 4. In Fig.7, all performance data is normalized to that of DLPlib. As we can see, the library achieves a speedup of 0.79x on average compared with the hand-written assembly instructions. Regarding single layers, the library achieves a speedup of 0.77x∼0.93x, 0.89x∼0.9x, 0.56x∼0.81x, and 0.78x respectively on convolution, fully-connected, pooling, and LRN respectively. Regarding entire networks, the library achieves a speedup of 0.78x and 0.75x on AlexNet and VGG respectively. And According to [11], the hardware achieves a speedup of 5.2x compared with Caffe-GPU. Therefore, the library is 4.1x faster than the GPU.

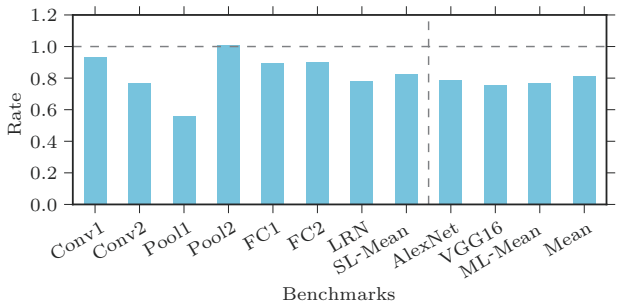


Fig.7. Speedup before and after using DLPlib.

Single-Layer Benchmarks. As we have discussed in Section 4, the performance is mainly affected by how much the computation is overlapped with the memory accessing. Overlapping happens not only between but also inside operations. For operations with large input, output or weight size, e.g., *Conv*2 in Table 4, there is no enough on-chip memory to load all neurons and synaptic weights into the processor, and the computation needs to be split into several smaller computational segments and therefore creates more overlapping issues.

The performance differences of the library and hand-written assembly instructions are caused by the different parallel degrees of computation and memory accessing. For hand-written assembly instructions, it is rather easy to process such overlapping between different segments; thus, they achieve better performance than the library.

Table 4. Benchmarks

Operation	In (Size@FM)	Out (Size@FM)	Kernel	Stride	Pad	AlexNet	VGG16
*Conv*1	112@96	112@384	3	1	1	conv(96,11,4)-lrn-pool-	*conv*(64,3)-conv(64,3)-pool-conv(128,3)-
*Conv*2	112@96	112@256	3	1	1	conv(256,5)-lrn-pool-	*conv*(128,3)-pool-conv(256,3)-conv(256,3)-
*Pool*1	55@96	27@96	3	2	0	conv(384,3)-conv(384,3)-	conv(256,3)-conv(256,3)-
*Pool*2	224@64	112@64	2	2	0	conv(256,3)-pool-	*conv*(256,3)-pool-conv(512,3)-conv(512,3)-
LRN	55@96	55@96	-	-	-	fc(4096)-fc(4096)-	-conv(512,3)-pool-conv(512,3)
*FC*1	1@4096	1@4096	-	-	-	fc(1000)	-conv(512,3)-conv(512,3)-
*FC*2	1@4096	1@1000	-	-	-		pool-fc(4096)-fc(4096)-fc(1000)

Entire Network Benchmarks. As we can observe, the library performs worse on entire networks than single-layer benchmarks, i.e., 0.79x of single-layer and 0.77x of multi-layer on average. This is caused by the overlapping of computation and memory access between different layers. For multi-layer networks, handwritten instructions can hide the latency of memory writing and reading between adjacent layers. For example, while computing the last output of a convolutional layer, synaptic weights or neurons of next layer can be loaded into the memories at the same time, therefore increasing the degree of parallel of computing and memory accessing. However, for the library, memory loading will not start until the computation of the previous layer finishes.

8 Conclusions

We described DLPlib, a library designed for bridging the gap between the deep learning processor and the end-users. We also provided a programming model and the Caffe integration of DLPlib. The library supports the inference operations of mainstream neural network techniques and a set of basic matrix/vector operations. Through them, end-users are able to construct and perform their neural networks on the deep learning processor. In addition, DLPlib can also be easily integrated into deep learning frameworks, e.g., Caffe, thus enabling end-users to employ the deep learning processor by calling the Caffe interface. Our future work includes improving the efficiency of our library by further increasing the parallel degree, and providing more operations to support more new networks.

References

[1] Zhang S J, Du Z D, Zhang L, Lan H Y, Liu S L, Li L, Guo Q, Chen T S, Chen Y. Cambricon-X: An accelerator for sparse neural networks. In *Proc. the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2016.

[2] Krizhevsky A, Sutskever I, Hinton G E. Imagenet classification with deep convolutional neural networks. In *Proc. the 26th Annual Conference on Neural Information Processing Systems*, Dec. 2012, pp.1106-1114.

[3] Sun Y, Liang D, Wang X G, Tang X O. DeepID3: Face recognition with very deep neural networks. arXiv:1502.00873, 2015. http://arxiv.org/abs/1502.00873, Feb. 2017.

[4] Karpathy A, Li F F. Deep visual-semantic alignments for generating image descriptions. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition*, Jun. 2015, pp.3128-3137.

[5] Eriguchi A, Hashimoto K, Tsuruoka Y. Tree-to-sequence attentional neural machine translation. In *Proc. the 54th Annual Meeting of the Association for Computational Linguistics*, Aug. 2016.

[6] Ren S Q, He K M, Girshick R B, Sun J. Faster R-CNN: Towards real-time object detection with region proposal networks. In *Proc. Annual Conference on Neural Information Processing Systems*, Dec. 2015, pp.91-99.

[7] Farabet C, Poulet C, Han J Y, LeCun Y. CNP: An FPGA-based processor for convolutional networks. In *Proc. the 19th International Conference on Field Programmable Logic and Applications*, Aug.31-Sept.2, 2009, pp.32-37.

[8] Zhang C, Li P, Sun G Y, Guan Y J, Xiao B J, Cong J. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proc. the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Feb. 2015, pp.161-170.

[9] Chen T S, Du Z D, Sun N H *et al.* DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proc. the 19th ACM Int. Conf. Languages and Operating Systems*, Mar. 2014, pp.269-284.

[10] Chen Y, Luo T, Liu S *et al.* DaDianNao: A machine-learning supercomputer. In *Proc. the 47th Annual IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2014, pp.609-622.

[11] Liu S L, Du Z D, Tao J H *et al.* Cambricon: An instruction set architecture for neural networks. In *Proc. the 43rd ACM/IEEE Annual Int. Symp. Computer Architecture* (*ISCA*), Jun. 2016, pp.393-405.

[12] Chakradhar S T, Sankaradass M, Jakkula V, Cadambi S. A dynamically configurable coprocessor for convolutional neural networks. In *Proc. the 37th International Symposium on Computer Architecture*, Jun. 2010, pp.247-257.

[13] Chi P, Li S C, Xu C *et al.* PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. In *Proc. the 43rd ACM/IEEE Annual Int. Symp. Computer Architecture* (*ISCA*), Jun. 2016, pp.27-39.

[14] Shafiee A, Nag A, Muralimanohar N *et al.* ISAAC: A convolutional neural network accelerator with In-Situ analog arithmetic in crossbars. In *Proc. the 43rd ACM/IEEE Annual International Symposium on Computer Architecture*, Jun. 2016, pp.14-26.

[15] Du Z D, Fasthuber R, Chen T S *et al.* ShiDianNao: Shifting vision processing closer to the sensor. In *Proc. the 42nd Annual Int. Symp. Computer Architecture*, Jun. 2015, pp.92-104.

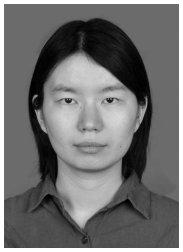
[16] Chetlur S, Woolley C, Vandermersch P, Cohen J, Tran J, Catanzaro B, Shelhamer E. cuDNN: Efficient primitives for deep learning. arXin: 1410.0759, 2014. http://arxiv.org/abs/1410.0759, Feb. 2017.

[17] Abadi M, Barham P, Chen J *et al.* Tensorflow: A system for large-scale machine learning. In *Proc. the 12th USENIX Symp. Operating Systems Design and Implementation*, Nov. 2016, pp.265-283.

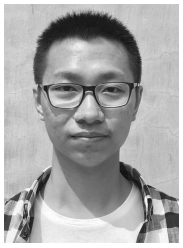
[18] Lecun Y, Bottou L, Bengio Y, Haffner P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998, 86(11): 2278-2324.

[19] Szegedy C, Liu W, Jia Y Q *et al.* Going deeper with convolutions. In *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, Jun. 2015.

[20] Krizhevsky A. Cuda-convnet: High-performance C++/CUDA implementation of convolutional neural networks. https://code.google.com/p/cuda-convnet, Feb. 2017.
[21] Ioffe S, Szegedy C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proc. the 32nd Int. Conf. Machine Learning*, Jul. 2015, pp.448-456.
[22] He K M, Zhang X Y, Ren S Q, Sun J. Deep residual learning for image recognition. In *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, Jun. 2016, pp.770-778.
[23] Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition. arXiv: 1409.1556, 2014. http://arxin.org/abs/1409.1556, Feb. 2017.



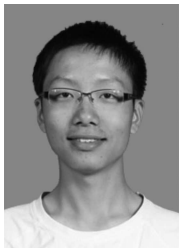
Hui-Ying Lan received her B.E. degree in software engineering from Wuhan University, Wuhan, in 2012. She received her Master's degree from School of Software and Microelectronics, Peking University, Beijing, in 2015. She is currently a Ph.D. student at Institute of Computing Technology, Chinese Academy of Sciences, Beijing. Her research interests include computer architecture and computational intelligence.



Lin-Yang Wu received his B.S. degree in computer science from University of Science and Technology of China, Hefei, in 2014. He is currently a Ph.D. student at Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His research interests include computer architecture and computational intelligence.

Xiao Zhang received his B.S. degree in communication engineering from University of Science and Technology of China, Hefei, in 2014. He is currently a Ph.D. student at Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His research interests include computer architecture and computational intelligence.

Jin-Hua Tao received his B.S. degree in statistics from University of Science and Technology of China, Hefei, in 2013. He is currently a Ph.D. student at Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His research interests include computer architecture and computational intelligence.



Xun-Yu Chen received his B.S. degree in computer science from University of Science and Technology Beijing (USTB), Beijing, in 2015. He is currently a graduate student at Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing. His research interests include computer architecture and computational intelligence.

Bing-Rui Wang received his B.S. degree in atomic and molecular physics from School for the Gifted Young, University of Science and Technology of China (USTC), Hefei, in 2015. He is currently a Ph.D. student at Department of Computer Science of USTC, Hefei. His research interests include computer architecture and computational intelligence.

Yu-Qing Wang graduated from University of Science and Technology of China, Hefei, 2015. He is currently a Ph.D. student at Department of Computer Science of USTC, Hefei. His research interests include computer architecture and computational intelligence.

Qi Guo received his B.E. degree in computer science from Tongji University, Shanghai, in 2007, and his Ph.D. degree in computer science from Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, in 2012. He currently is an associate professor at ICT, Beijing. His research interests include computer architecture, VLSI design and verification.

Yun-Ji Chen graduated from the Special Class for the Gifted Young, University of Science and Technology of China, Hefei, in 2002. He received his Ph.D. degree in computer science from Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, in 2007. He is currently a professor at ICT, Beijing. His research interests include parallel computing, microarchitecture, hardware verification, and computational intelligence.