doi:10.3772/j.issn.1006-6748.2019.04.006

Assembly language and assembler for deep learning accelerators①

Lan Huiying(兰慧盈)②[*,**], Wu Linyang[*,**], Han Dong[*,**], Du Zidong[*,***]
([*] Intelligent Processor Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, P. R. China)
([**] University of Chinese Academy of Sciences, Beijing 100049, P. R. China)
([***] Cambricon Technologies Corporation Limited, Beijing 100191, P. R. China)
([****] CAS Center for Excellence in Brain Science and Intelligence Technology, Shanghai 200031, P. R. China)

Abstract

Deep learning accelerators (DLAs) have been proved to be efficient computational devices for processing deep learning algorithms. Various DLA architectures are proposed and applied to different applications and tasks. However, for most DLAs, their programming interfaces are either difficult to use or not efficient enough. Most DLAs require programmers to directly write instructions, which is time-consuming and error-prone. Another prevailing programming interface for DLAs is high-performance libraries and deep learning frameworks, which are easy to be used and very friendly to users, but their high abstraction level limits their control capacity over the hardware resources thus compromises the efficiency of the accelerator. A design of the programming interface is for DLAs. First various existing DLAs and their programming methods are analyzed and a methodology for designing programming interface for DLAs is proposed, which is a high-level assembly language (called DLA-AL), assembler and runtime for DLAs. DLA-AL is composed of a low-level assembly language and a set of high-level blocks. It allows experienced experts to fully exploit the potential of DLAs and achieve near-optimal performance. Meanwhile, by using DLA-AL, end-users who have little knowledge of the hardware are able to develop deep learning algorithms on DLAs spending minimal programming efforts.

Key words: deep learning, deep learning accelerator (DLA), assembly language, programming language

0 Introduction

Deep learning algorithms become state-of-the-art techniques in a broad range of applications such as computer vision tasks and natural language processing[1,2]. Meanwhile, networks keep increasing towards deeper and larger architectures. Recently, customized accelerators have been emerging rapidly as effective alternatives to CPUs/GPUs for processing neural network algorithms because of their high efficiency in both performance and energy. On traditional devices (e.g., CPUs/GPUs), neural network programs are written with high-level frameworks (e.g., TensorFlow[3], MXNet[4], TVM[5]) or libraries (cuDNN[6]). Programming on DLAs is much more difficult than that on traditional devices due to the lack of suitable programming supports. Most DLAs require developers to write programs directly with instructions in order to achieve high efficiency. For example, DianNao[7] and ShiDianNao[8] programed the hardware through manually written instructions, which was a time-consuming approach. Deep learning frameworks are another programming interface used by DLAs, for example, TPU[9] used TensorFlow as the programming interface. The high-level abstraction of frameworks allows developers to conveniently develop deep learning algorithms, but not able to manipulate hardware resources. It is important and urgent to design a programming interface that satisfies both developing efficiency and performance so that DLAs can be used actually in practical applications.

The challenge of designing such an interface for DLAs lies in 3 aspects. 1) Computation partitioning. Due to the limited on-chip resources, a computation (e.g., a convolutional layer) needs to be partitioned into several parts of sub-computations. Reasonable partitioning can allow maximum overlapping between com-

① Supported by the National Key Research and Development Program of China (No. 2017YFA0700902, 2017YFB1003101), the 973 Program of China (No. 2015CB358800) and National Science and Technology Major Project (No. 2018ZX01031102).
② To whom correspondence should be addressed. E-mail: lanhuiying@ict.ac.cn
Received on Oct. 25, 2018

putation and memory accesses, and results in better executing efficiency. 2) As the on-chip memory of deep learning accelerators are usually divided into multiple modules for various types of data utilization, it also increases the burden of managing this memory for developers. For example, developers have to manually assign data addresses of on-chip memory for different data blocks, especially for operations that have multiple independent data blocks residing on the same on-chip memory module (e. g. , a convolutional layer needs to manage an input neuron, an output neuron and a bias in the same block of on-chip memory). 3) The partitioning strategy between segments can result in complicated scheduling. Unlike serial programs on CPUs, memory accesses and computations in deep learning programs can be highly paralleled. It is error-prone and difficult to write and debug such programs with a low-level assembly language, especially with synchronization primitives in the picture.

In order to address the above challenges, a novel high-level assembly language and an assembler are designed, which is designed to achieve the following 2 objectives:

Performance efficiency Performance efficiency should be achieved by professional and amateur developers. For deep learning accelerators, the proposed assembly language should allow developers to fully leverage the potential of the hardware. For developers who do not understand the hardware details, the language should help them to construct neural networks with acceptable performance. Experienced developers, through our language, should have full access to the hardware and thus be able to explore the hardware potential.

Developing efficiency Developing efficiency refers to the efficiency of writing codes for neural network algorithms. Our assembly language should allow developers to construct a neural network easily (e. g. , less code) for achieving near-the-best performance.

The following major contributions are made:

- A domain-specific assembly language designed for deep learning accelerators is proposed, which is composed of specialized data structures and a set of high-level operators.
- An assembler and runtime to support all features of the proposed assembly language are designed.
- The assembly language and the assembler on ten benchmarks with Cambricon as the backend are evaluated, that shows that the language can significantly alleviate the developing burden and the assembler is able to produce highly efficient executable code.

1 Assembly language

In this section, newly designed DLA-AL, a domain-specific assembly language for DLAs is presented. First the overall view of DLA-AL is illustrated and then the data type, basic statements, macro instructions, blocks, and cross-layer fusion are introduced.

1.1 Overview

Fig. 1 shows the overall architecture of DLA-AL while implementing deep learning algorithms. To build a network, developers are expected to mostly use the block that is predefined as optimized for high-level abstractions of neural network operations, e. g. , convolutional (Conv.), pooling (Pool.), or fully-connected (FC). Thus, developers can easily specify a network by using the built-in blocks. Additionally, developers are allowed to build their own blocks with basic statements and macros. Macro is a meaningful sequence of basic statement that achieves a certain purpose, e. g. , computation or IO. Note that because of partitioning, a block usually contains segments that execute several times.

In Fig. 2 and Fig. 3, an example code is showed for a 2-layer FC network written in DLA-AL. Similar to that of a traditional assembly language, a DLA-AL source program can be divided into several meaningful sections: code, static _ ro, and static _ rw. The code section is used to store DLA-AL statements, which will be translated into executable machine codes. Section static _ ro and section static _ rw are used for declaring statically allocated data, where ro and rw represent read-only and read-write, respectively, which is introduced for address checking. Code and static data are defined separately as they will be placed in different memory segments and processed differently during assembly.

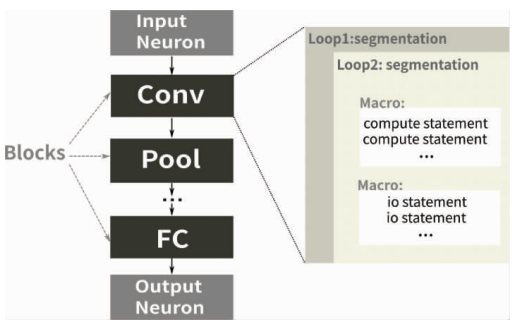

Fig. 1 Overview of DLA-AL

```
;; FC1 (1024->256)
;; FC2 (256->128)
;; Both layers uses 128 as the segment size
.code
  smove $R1, #1
  smove $R2, #1
  smove $R3, #256
  @block_fc_fp fc1_out, fc1_inp, fc1_weight, fc1_bias
  @block_fc_fp fc2_out, fc1_out, fc2_weight, fc2_bias

.static_rw
  @neuron fc1_out, 1, 1, 1, 256, 1, 1, 1, 128
  @neuron fc2_out, 1, 1, 1, 128, 1, 1, 1, 128

.static_ro
  @neuron fc1_inp, 1, 1, 1, 1024, 1, 1, 1, 128
  @synapse fc1_weight, 256, 1, 1, 1024,
                       128, 1, 1, 128
  @synapse fc2_weight, 128, 1, 1, 256,
                       128, 1, 1, 128,
```

Fig. 2 Code example of a 2-FC layer network

```
;; in_seg_size: input segment size
;; in_seg_n: input segment number
;; out_seg_size: output segment size
;; out_seg_n: output segment number

smove $reg_out_seg, $out_seg_n
out_seg:
  smove $reg_in_seg, $in_seg_n
  in_seg:
    @macro_load_input  $in_seg_size
    @macro_load_weight $w_seg_size
    @macro_compute_fc  $in_seg_size,
                       $out_seg_size
    ssub seg_in_seg, #1
    CB $reg_in_seg, $in_seg_size

  @macro_store_output $out_seg_size
  ssub, out_seg, #1
  CB $reg_out_seg, $out_seg_size
```

Fig. 3 Code example of the block of FC

1.2 Datatype

There are 3 types of data in DLA-AL: neuron, synapse, and parameter. These 3 data types can cover most data structures in neural networks, and also can be mapped to different functional units and on chip memory in Cambricon architecture.

Neuron Neuron is the basic type in DLA-AL, designed as a 4-dimensional data structure. It provides abstractions for neuron data for the inference process and the gradient data of the back-propagating process, as well as bias data and its corresponding gradient. To declare a neuron, the developer must specify the batch, height, width, channel, and segment size, respectively. The declaration of a neuron is placed in data sections (including the static _ ro and static _ rw sections) of the source file with the following form:

@ neuron[var _ name]
[B],[H],[W],[C],[BS],[HS],[WS],[CS];

As partitioning is inevitable while performing large neural networks on modern accelerators with limited on-chip storage, the neuron data type also provides parameters for developers to flexibly set partitioning sizes, i. e. , the *BS*, *HS*, *WS*, and *CS*. Fig. 4 shows the partitioning of a neuron, where the batch dimensional is set to one for brevity. For each computational operation, only a slice of data (with a size of $HS \times WS \times CS$) will be loaded into the on-chip memory to compute a partial sum, which will be accumulated to obtain final result.

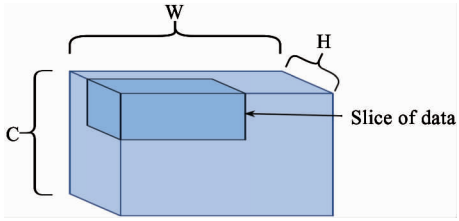

Fig. 4 Data partitioning

Synapse The synapse is a basic data type in DLA-AL, which is designed as a four-dimensional data structure to declare synaptic weights in neural networks. The declaration of a synapse is similar to that of a neuron:

@ synapse [var _ name] CO],[KH],
[KW],[CI],[COS], [KHS], [KWS],[CIS];

CO, KH, KW, and CI represent the 4 dimensions of the synapse data, and *COS*, *KHS*, *KWS*, and *CIS* are the corresponding segments. The reason that the synapse and neuron are considered as 2 independent data types is twofold. First, from the algorithm aspect, neuron and synapse represent different entities; the former can be activated and passed to the next layer, and the latter is a group of connections that determine the neurons. Second, with considerations of the architecture of Cambricon-ACC, execution, data of neuron and synapse will be loaded to different types of on-chip memory, i. e. , neuron will be loaded to the vector scratchpad, and synapse will be loaded to the matrix scratchpad.

Parameter The parameter type represents scalars used to specify configuration parameters in a neural network such as the kernel size and stride of convolutional layers. Parameters are also stored in the data sections.

1.3 Basic statements

Basic statements in DLA-AL correspond to executable instructions that can be translated into one-to-one instructions. These statements are included as the atomic operations by which blocks and macrostatements are built. Through basic statements, DLA-AL can provide enough flexibility. Basic statements of DLA-AL are grouped into 4 categories, i. e. on-chip memory management statements, computational statements, logicalstatements, and control statements.

1.4 Macro statements

In DLA-AL, a macro statement is defined as the fundamental scheduling unit. In traditional assembly languages, the macro instruction is a widely used technique that can provide abstractions by simple substitution. Similarly, in DLA-AL, macro instructions are used to specify a sequence of serially executed basic statements that can create computationalor memory access instruction fragments. Each macrostatement should include only one type of instruction: computational (including computational statements and logical statements) or memory access. The following is the declaration of a macro:

```
C- Macro c _ macro var1
; comments
; a string of computational or
; logical statements
statement MYMvar1
…
mend
IO - Macro io _ macro var2
; a string of load / store statements
statement MYMvar2
…
mend
; call macro with r1 as its parameter
@ c _ macro r1
@ io _ macro r2
```

The C-macro and IO-macro are words reserved for declaring computational macros and memory access-macros, C-macro and IO-macro are the names of the-macro statement instances, and *var*1 and *var*2 are the parameter lists. Macro is closed at the mend reserved word. The reason why providing macro in DLA-AL is twofold. First, a macro can provide higher abstractions and code reused for developers. It is difficult for developers to perform a complex NN algorithm by basic statements that include only low-level vector/matrix operations.

Macros can help developers to organize a program in a more readable and structural fashion while improving code reuse in the assembly language. Second, by grouping statements into separate types of macros, it will be easier for the developers to predict the parallel execution behaviors by the types of the macros, thus making the manual scheduling process easier.

1.5 Blocks

Block is a critical structure in DLA-AL. It is composed of a set of predefined optimized code that implements high-level neural network operations with arbitrary input and output data scales. DLA-AL provides a set of built-in blocks. With blocks, developers can conveniently construct a neural network. Blocks in DLA-AL are defined with the same calling routine as macros:

```
@ block _ conv n1, w, n2, #c _ str _ x ,#c _ str _ y
@ block _ relu n1, n1
@ block _ pool n2, n3, #p _ str _ x, #p _ str _ y
```

Block conv, block relu, and block pool are predefined blocks for Conv., ReLU, and pooling layers, respectively; *n*1, *n*2, and *n*3 are neuron data, and *w* is synapse data. This program computes the convolution of strides *str _ x* and *str _ y*, followed by the pool operation of the *p _ str _ x*, and *p _ str _ y* strides. Thus, by repeatedly calling the predefined blocks, it can easily define an existing neural network. DLA-AL includes a set of built-in blocks dedicated to existing common neural network algorithms for both inference and training phases. The supported algorithms are listed in Table 1.

Table 1 Built-in blocks in DLA-AL, both inference and training are supported for each algorithm

Domain	Blocks
Neural network	Conv, MaxPool, AvgPool, LRN, BN, FC, Active(Sigmoid, ReLU, Tanh)
Math	matrix multiply vector, vector multiply matrix, matrix/vector add/sub/mul/div, vector power

Cross-layer fusion is a useful technique for improving performance. The output data of the previous block can be computed immediately by the next block before it is stored to the main memory, thus avoiding storage of the previous block and loading of the next block. It could be very effective for blocks with extremely large input/output neurons. Fig. 5 shows an example of a 3-layer (Conv-Pool-FC) fusion, the input and output neurons of the three layers are partitioned into segments to fit in the on-chip memory. For a fused block (e. g. , ReLU), the loading and storage of the intermediate data (e. g. , the output of Conv/inputof ReLU) are saved. Only the operation loading the

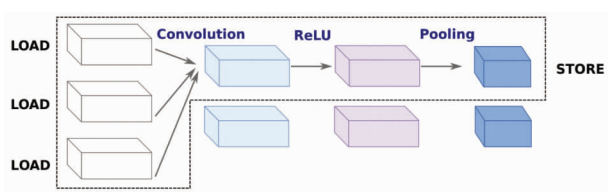

Fig. 5 Layer fusion

first layer input, and the operation storing the last layer output are performed in this process.

2 Assembler

An assembly and runtime are also provided to support DLA-AL. In this section, DLA-AL assembler architecture is outlined and the design of the assembler from 3 perspectives: preprocessor, address, and storage allocation and relocation are discussed. Then, the implementation of the built-in block and cross-block scheduling are introduced.

2.1 Assembly process

In Fig. 6, the architecture of DLA-AL assembler is shown, which produces executable code composed of code for both the host and the accelerator.

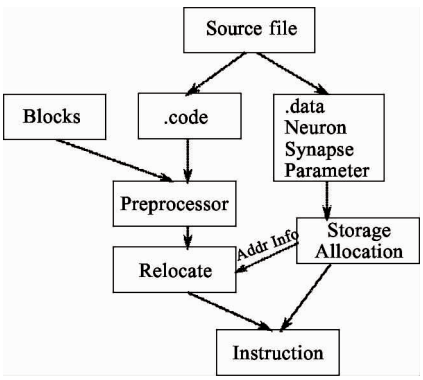

Fig. 6 Assembly process

The assembler first preprocesses the source file, dividing the code into separate sections according to the section directives (i. e., .code,. static _ rw and static _ ro). The preprocessor fetches the code section in an assembly program for the substitution of built-in blocks and macro statements. DLA-AL assembler further fetches the data section that defines the neurons, synapses, and parameters for data processing. The assembler will further allocate addresses for static data. At last, the assembler will be relocated for the program and produces the executable program.

2.2 Built-in blocks

The set of built-in blocks is a substantial component of DLA-AL, which ensures performance and development efficiency by providing predefined high-level operations. In this section, scheduling inside a block is introduced to demonstrate techniques for obtaining high performance while programming on Cambricon architecture.

2.2.1 An example of FC

Scheduling the computational and memory access operations is a tricky process for deep learning accelerator. As the parallel mechanism is hardware-specific, without a comprehensive understanding of the hardware execution process, one can hardly predict the results of a program. An FC layer is used as a driving example to demonstrate how the instruction order can affect the execution process. In Fig. 7 (Code #1), an intuitive implementation of an FC layer written with instructions from the Cambricon ISA[10] is presented. The developer who wrote this code might expect that the computational instructions, i. e., VMM and VADD, and the memory access instructions, i. e., VLOAD and MLOAD, will be executed in parallel. However, because of the limitation of the depth of issue queue in Cambricon[10] architecture (i. e., 2), this process is more serious than expected. Code #2 in Fig. 7 shows the program after switching the order of the computational instructions (i. e., VMM and VADD) and the memory access instructions (i. e., VLOAD and MLOAD) for optimization. Fig. 8 shows the pipelining

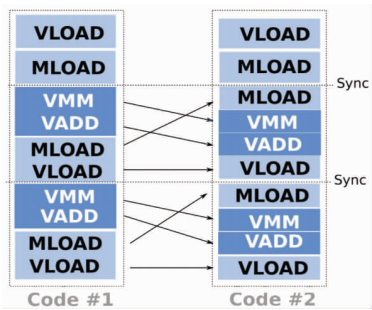

Fig. 7 Codes implementing a two-FC network with Cambricon-ISA, Code #1 shows a naive version, and Code#2 is an optimized version based on Code#1

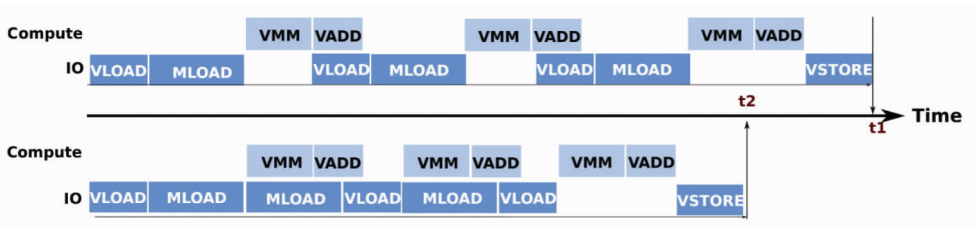

Fig. 8 A pipeline of different implementations, top: Implementation of Code#1, bottom: implementation of Code#2

of Code #1 and Code #2 in Fig. 7 is shown to further clarify such optimization. It can be observed that parallel execution happened only between VADD and VLOAD instructions and that no memory access instruction can be overlapped during the execution of the VMM instruction and MLOAD instruction (see top Figure in Fig. 8). However, as an optimized version of Code #1, Code #2 improves the parallelism greatly. It is observed that the two more instructions, i. e., VMM are overlapped with memory access instructions (see the bottom Figure in Fig. 8).

3 Runtime

The runtime of DLAs is similar to that of a GPU device, which is basically handled by the host, as the accelerators are not able to support functionalities like storage management.

The process of executing a program is as follows. First, the program is executed on CPU. All types of data are allocated and initialized by the host (i. e., CPU). Then, CPU sends the ACC program generated by the assembler to the accelerator and invokes the device by sending an extra signal. As only static data allocation is consided, the ACC program will be executed until all instructions are finished, and then a finished signal is sent back to the host. The host will take over and perform the rest of the operations (e. g., copying result data from device space to host space).

Data layout is a critical issue that has not been discussed thoroughly in the programming of deep learning accelerators. Under the circumstance of segmentation, the data loading order might not be the same as the storage order of the original data. An intuitive way of loading non-continuous data is to break one loading operation into multiple loading operations, and each of which loads a part of the data block. This solution is flexible and easy to implement, but it has 2 shortcomings. First, the latency of memory access is inevitably enlarged with the increasing number of instructions. Second, this could lead to a drop in parallelism because the issue queue in the accelerator might be stalled by a long sequence of memory access instructions. Our solution to this problem is to deploy a very specific data layout so that for each computation instruction sequence the required data can be loaded by only one loading instruction. In DLA-AL, neurons are stored in the order of CWHB (column-width-height-batch). Fig. 9(a) shows the layout of a neuron with the batch set to one, where the number in the square represents the order of storage before segmentation. In this case, the neuron is partitioned into 3 segments from the feature map direction, which leads to discontinuous data access during execution. To avoid such nonconsecutive data access, the data is managed in a computational-order-first layout, where the data resides on the device memory according to the computational order, as shown in Fig. 9(b). This rearrangement process is handled by CPU when the data are copied from the host memory to the device memory. During execution, the required sequence of data can be treated as an entire unit and loaded onto the on-chip memory, in order to reduce memory latency and increase parallelism.

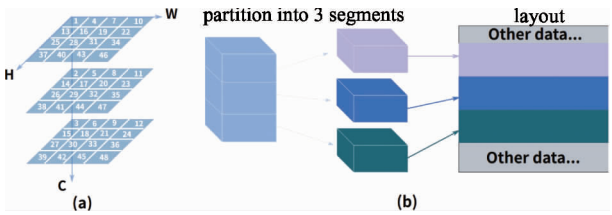

Fig. 9 Data layout

4 Evaluation

In this section, the performance of DLA-AL is evaluated by comparing its performance to 3 baselines: 1) hand-optimized instruction sequence, which is considered as the optimum performance, running on Cambricon-ACC[10] (same as DLA-AL programs). 2) hand-written instruction sequence, which is implemented according to the natural computational order of the neural network algorithms. 3) GPU. Caffe[11] is used, the high-performance deep learning framework, run on an advanced modern GPU card (NVIDIA K40M, 12 GB GDDR5, 4.29 TF lops peak at a 28 nm process) with the back-end of cuDNN, which is a high-performance library. Only a NVIDIA GPU is used instead of both NVIDIA and AMD GPUs to evaluate the performance for 2 reasons. First, NVIDIA GPU is the most pervasive and widely accepted backend for executing DL algorithms and the DL framework used does not support AMD. Second, the experiment is done without an AMD card and it is believed that the NVIDIA card is enough to show the performance advantage of DLA-AL over the GPU.

Both DLA-AL programs and hand-written instructions run on a carefully implemented Cambricon-ACC simulator. All specifications are set according to the published paper. Our evaluation will demonstrate: 1) DLA-AL can produce high performance executable code. 2) DLA-AL provides a significantly programming interface more productively than programming directly with ISA.

4.1 Performance

DLA-AL is evaluated on 10 representative benchmarks, as listed in Table 2, which are extracted from realistic networks. The speedups of GPU, hand-optimized instructions, and hand-written instructions over DLA-AL are presented in Fig. 10. DLA-AL achieves a speedup of 2.90 × at the forwarding propagation and 3.57 × at the backward propagation compared to that of the GPU baseline on average. Compared with the hand-optimized instructions, DLA-AL achieves a speedup of 0.95 × and 0.96 × for forward and backward propagation on average. Compared with hand-written instructions, DLA-AL achieves a speedup of 1.20 × and 1.14 × for forward and backward propagation on average, respectively. DLA-AL accomplishes the highest speedup of 0.99 × on FC forward benchmarks, and the conv3-fp benchmark has a poor result with only 0.74 × compared with hand-optimized instructions. The main reason is the imbalance between two load operations that the load time depends on input data size and thus it is impossible for DLA-AL to dynamically schedule ahead with optimum. However, it still achieves a speedup of 2.76 × and 1.25 × compared with the GPU and the hand-written baseline, respectively. DLA-AL is further evaluated on 2 realistic entire networks, VGG16 and AlexNet, compared with handwritten instructions to demonstrate the performance improvement obtained by DLA-AL. DLA-AL achieves 1.16 × improvements for AlexNet-inference and 1.09 × improvements for VGG16-inference. Since a deep neural network can be viewed as the composition of a sequence of layers, and the performance of single-layer is evaluated and hand-optimized baseline for this evaluation is not performed. To demonstrate the benefits one can get it from cross-block fusion, and select a representative 3-layer benchmark extracted from VGG16: Conv-ReLU-Pool. Fusion of the 3-layer benchmark achieves 1.02 × improvements over non-fusion DLA-AL. In addition, 10.16% of memory access time has been reduced by saving a convolution output storing, a ReLU input loading, a ReLU output storing and a pooling input loading operation.

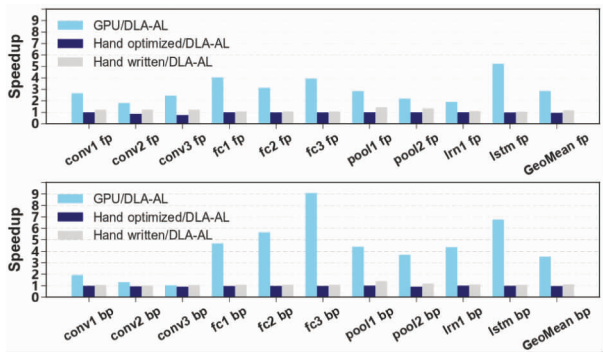

Fig. 10 Speedup over DLA-AL

Table 2 Benchmarks

Name	Layer	InS#F	OutS#F	K	S	Source
conv1	Conv	7#512	7#2048	1	1	ResNet
conv2	Conv	14#512	14#512	3	1	VGG
conv3	Conv	13#256	13#384	3	1	AlexNet
pool1	Pool	28#512	14#512	2	2	VGG
pool2	Pool	56#256	28#256	2	2	VGG
fc1	FC	1#9216	1#4096	-	-	AlexNet
fc2	FC	1#4096	1#4096	-	-	AlexNet
fc3	FC	1#4096	1#1024	-	-	AlexNet
lrn	LRN	27#256	27#256	-	-	AlexNet
lstm	LSTM	Input(3)-hidden(400)-output(121)				

It is observed that Conv. layer has the poorest performance and speedup compared with the other benchmarks. The ideal situation of implementing computational-intense algorithms like convolution is that all memory access operations can be executed overlapping with computational operations. By inspecting the execution process, it is found that a portion of convolution computation operations are not overlapped by memory loading and storing. The reason is that for each segment in Conv., it needs at least 2 loading instructions to load the neurons and the synaptic weights, one of them could be stalled by the other one and leads to a non-parallel window that causes the performance loss. This observation does not appear in the FC layer evaluation, since for IO-intense algorithms like FC, computation can be fully overlapped with the synaptic weights

loading instruction.

The worst case scenario is further analyzed with five more convolutional layers with various scales (see Table 3), and similar scenarios are observed. The result is shown in Fig. 11, where convolutional layers achieve 89.8% performance compared with that of the hand-optimized implementation. The non-overlapping window in these layers is caused by the limited depth of issue queue of Cambricon-ACC. As only 2 instructions can be issued simultaneously, later instructions are blocked until these 2 instructions are finished. In the case of convolutional layers, the 1st two instructions (i. e., load neurons and synapses) block the later computational instructions until the first load instruction is finished. Therefore, only the 2nd load instruction is executed in parallel with later computational instructions. This overhead can be saved in hand-optimized code by manually splitting the computation into 2 instructions, each processes a part of the computation, and the 2 load instructions can be overlapped with the 2 partial computations.

Table 3 Benchmarks for evaluating the worst-case scenario

Name	Layer	InS#FM	OutS#FM	K	S
Conv1	Conv	227#3	55#96	11	4
Conv2	Conv	27#96	25#256	5	1
Conv3	Conv	56#128	56#256	3	1
Conv4	Conv	56#256	56#256	3	1
Conv5	Conv	28#512	28#512	3	1

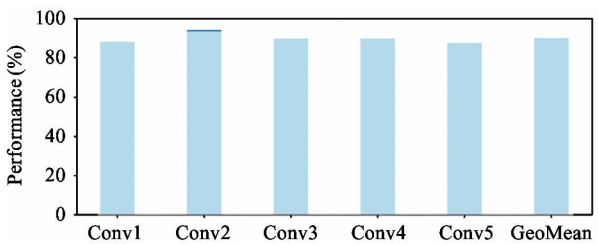

Fig. 11 Test cases for worst-case scenario analysis

4.2 Developing efficiency

As a high-level assembly language, significantly DLA-AL is more friendly and efficient than those program via hand-written instructions, which can be reflected in 3 respects. First, the length of the source program is vastly reduced. By DLA-AL, only 15 lines of code is required to implement a 2-layer FC layer, which is 3.33 × shorter than the code produced using original programming interface introduced in Ref. [10]. Second, DLA-AL assembler provides automatic data address allocation and relocation that can release the developers from calculating the addresses by themselves. Third, DLA-AL assembler provides automatic data deployment to make sure that the data can be loaded sequentially and in the correct layout. Without this mechanism, developers have to manage and arrange the data manually, which is a very time-consuming and error-prone approach.

5 Related work

Many DLAs are proposed in recent years as they are more efficient in both performance and energy compared with traditional devices. DianNao family[7-8,10,12-14] is a serial of ASIC accelerators that leverage data locality in deep learning algorithms to improve performance. In addition, FPGA-based accelerators are also proposed. In addition, many accelerators leverage the sparsity in neural networks to further reduce workloads of both computation and memory accessing[12,15,16]. Although these accelerators are able to provide impressive performance, their usability is limited by the lack of suitable programming supports.

As a new technique, studies about programming supports for DLAs are still insufficient. Most accelerators use the instruction set as the programming interface, which is error-prone. Deep learning frameworks are another option for accelerators, for example, TPU[9] uses TensorFlow as the programming interface. The shortcoming of directly using a high-level framework to program is that the program efficiency entirely relies on the implementation of the framework, and they do not have the opportunity to optimize the program by themselves.

6 Conclusion

This paper proposes DLA-AL, a domain-specific assembly language and an assembler for DLAs to solve the issue of development efficiency and also provide high execution performance. The language is composed of 3 major components, basic statements, macro statements, and blocks, each of which has a higher abstraction level than that of the previous one. Also an assembler is proposed that supports macro processing, static address allocation, and relocation. This paper evaluates DLA-ALon 10 representative benchmarks, achieves a speedup of 2.90 × and 3.57 × on forward and backward propagation over the GPU, a speedup of 0.95 × and 0.96 × on forward and backward propagation over the hand-optimized baseline, and a speedup of 1.20 × and 1.14 × on forward and backward propagation over the intuitive hand-written instruction baseline. It is observed that the worst-case scenario appears

in convolutional layers, as multiple loading operations stalling later computational instructions, and they cannot execute in parallel. The worst case of DLA-AL achieves 74% performance compared with that of the hand-optimized implementation.

References

[1] He K, Zhang X, Ren S, et al. Deep residual learning for image recognition[C]//Proceedings of Computer Vision and Pattern Recognition, Las Vegas, USA, 2016: 770-778

[2] Devlin J, Chang M, Lee K, et al. BERT: Pre-training of deep bidirectional transformers for language understanding [J]. *arXiv*:1810.04805, 2019

[3] Abadi M, Barham P, Chen J, et al. TensorFlow: a system for large-scale machine learning[J]. *arXiv*: 1605.08695, 2016

[4] Chen T, Li M, Li Y, et al. MxNet: A flexible and flexible and efficient machine learning library for heterogeneous distributed systems[J]. *arXiv*:1512.01274, 2015

[5] Chen T, Moreau T, Jiang Z, et al. TVM: an automated end-to-end optimizing compiler for deep learning[J]. *arXiv*:1802.04799, 2018

[6] Chetlur S, Woolley C, Vandermersch P, et al. cuDNN: efficient primitives for deep learning[J]. *arXiv*: 1410.0759, 2014

[7] Chen T, Du Z, Sun N, et al. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning[C]//Architectural Support for Programming Languages and Operating Systems, Salt Lake City, USA, 2014: 269-284

[8] Du Z, Fasthuber R, Chen T, et al. ShiDianNao: shifting vision processing closer to the sensor[C]//International Symposium on Computer Architecture, Portland, OR, USA, 2015, 43(3): 92-104

[9] Jouppi N P, Young C S, Patil N, et al. In-Datacenter Performance Analysis of a Tensor Processing Unit[C]// International Symposium on Computer Architecture, Portland, USA, 2017: 1-12

[10] Liu S, Du Z, Tao J, et al. Cambricon: an instruction set architecture for neural networks[C]//International Symposium on Computer Architecture, Seoul, Korea, 2016: 393-405

[11] Jia Y, Shelhamer E, Donahue J, et al. Caffe: convolutional architecture for fast feature embedding[C]//ACM Multimedia, Orlando, USA, 2014: 675-678

[12] Zhang S, Du Z, Zhang L, et al. Cambricon-X: an accelerator for sparse neural networks[C]//International Symposium on Microarchitecture, Taipei, China, 2016: 1-12

[13] Chen Y, Luo T, Liu S, et al. DaDianNao: a machine-learning supercomputer[C]//International Symposium on Microarchitecture, Cambridge, UK, 2014: 609-622

[14] Liu D, Chen T, Liu S, et al. PuDianNao: a polyvalent machine learning accelerator[C]// Architectural Support for Programming Languages and Operating Systems, Istanbul, Turkey, 2015: 369-381

[15] Han S, Liu X, Mao H, et al. EIE: efficient inference engine on compressed deep neural network[C]//International Symposium on Computer Architecture, Seoul, Korea, 2016: 243-254

[16] Albericio J, Judd P, Hetherington T, et al. Cnvlutin: ineffectual-neuron-free deep neural network computing[C]// International Symposium on Computer Architecture, Seoul, Korea, 2016: 1-13

Lan Huiying, born in 1990. She is currently a Ph. D student at Institute of Computing Technology, Chinese Academy of Sciences, Beijing. She received her B. S. degree in software engineering from Wuhan University, Wuhan, in 2012. She received her M. S. degree from School of Software and Microelectronics, Peking University, Beijing, in 2015. Her research interests include computer architecture, programming language and computational intelligence.