Liu L, Ma X, Liu HX *et al.* FlexPDA: A flexible programming framework for deep learning accelerators. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 37(5): 1200–1220 Sept. 2022. DOI 10.1007/s11390-021-1406-9

FlexPDA: A Flexible Programming Framework for Deep Learning Accelerators

Lei Liu[1,2] (刘　磊), Xiu Ma[1,2] (马　秀), *Student Member*, *IEEE*, Hua-Xiao Liu[1,2,*] (刘华虓), *Member*, *CCF*
Guang-Li Li[3,4] (李广力), *Student Member*, *CCF*, *ACM*, *IEEE*, and Lei Liu[3] (刘　雷)

[1] *College of Computer Science and Technology, Jilin University, Changchun 130012, China*
[2] *Key Laboratory of Symbolic Computation and Knowledge Engineering of Ministry of Education, Jilin University Changchun 130012, China*
[3] *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences Beijing 100190, China*
[4] *University of Chinese Academy of Sciences, Beijing 100049, China*

E-mail: liulei@jlu.edu.cn; maxiu18@mails.jlu.edu.cn; liuhuaxiao@jlu.edu.cn; {liguangli, liulei}@ict.ac.cn

Received March 1, 2021; accepted September 18, 2021.

Abstract There are a wide variety of intelligence accelerators with promising performance and energy efficiency, deployed in a broad range of applications such as computer vision and speech recognition. However, programming productivity hinders the deployment of deep learning accelerators. The low-level library invoked in the high-level deep learning framework which supports the end-to-end execution with a given model, is designed to reduce the programming burden on the intelligence accelerators. Unfortunately, it is inflexible for developers to build a network model for every deep learning application, which probably brings unnecessary repetitive implementation. In this paper, a flexible and efficient programming framework for deep learning accelerators, FlexPDA, is proposed, which provides more optimization opportunities than the low-level library and realizes quick transplantation of applications to intelligence accelerators for fast upgrades. We evaluate FlexPDA by using 10 representative operators selected from deep learning algorithms and an end-to-end network. The experimental results validate the effectiveness of FlexPDA, which achieves an end-to-end performance improvement of 1.620x over the low-level library.

Keywords deep learning accelerator, programming framework, domain-specific language

1 Introduction

In recent years, deep learning (DL) has emerged as the state of the art across a broad range of applications such as image classification [1–3], speech recognition [4, 5], natural language processing [6, 7], automatic driving [8, 9], and cancer detection [10, 11]. Traditionally, DL applications are executed on general-purpose platforms such as CPUs which are usually inefficient because general-purpose processors put in excessive hardware resources to support various workloads flexibly. Therefore, a large variety of DL accelerators as efficient alternatives have emerged.

Along with the rapid increase in the performance of deep learning accelerators, programming productivity gradually hinders their deployments. A traditional DL accelerator often has many heterogeneous parallel components. It is notoriously difficult to program heterogeneous systems and parallel systems. The low-level library which provides a series of efficient and versatile programming interfaces for accelerating various deep learning algorithms, is developed to reduce the programming burden on the intelligence accelerators. A DL framework, such as Tensorflow [12], Caffe [13], and Theano [14], is a unified abstraction of the data and operations involved in the training and inference process

Regular Paper

This work was supported by the National Key Research and Development Program of China under Grant No. 2017YFB1003103, the Natural Science Research Foundation of Jilin Province of China under Grant No. 20190201193JC, and the Fundamental Research Funds for the Central Universities, JLU.

*Corresponding Author

of the neural network, which improves the efficiency of development. The DL framework embeds the low-level library to support the end-to-end execution with a given model. Unfortunately, it is inflexible for developers to build a network model for every deep learning application, which probably brings unnecessary repeating implementation. Generally, there are some traditional modules such as reading input, pre-processing data, and some intelligent modules containing neural networks such as classification and object detection in an intelligence application. As shown in Fig.1, an intelligent system implemented by C language contains three modules A, B, C, in which A and C are traditional modules and B is an intelligent module. Currently, the DL framework is the unique choice when developers want to accelerate module B with deep learning accelerators. The network model is implemented by a DL framework, such as Tensorflow [12] and Caffe [13]. The DL framework employs deep learning accelerators to accelerate application modules through the low-level library. However, it is not flexible or even efficient enough for developers in some application scenarios. For example, when developers want to speed up a matrix multiplication calculation, it is very unfriendly to build a network model that probably needs to be tuned. In addition, in order to improve the programming productivity on GPGPU (General Purpose Computing on Graphics Processing Unit) accelerators, both CUDNN (NVIDIA CUDA Deep Neural Network library)[①] developed for various DL frameworks and CUDA (Compute Unified Device Architecture)[②] language developed for users are aimed to promote the development of GPGPU-accelerated applications. In a nutshell, a programming framework, which is similar to CUDA, is necessary to support users for the development of more flexible and efficient deep learning applications.

In this paper, a flexible and efficient programming framework for DL accelerators, FlexPDA (Flexible Programming On DL Accelerators), is proposed. FlexPDA is composed of a domain-specific language (frontend) and a code generator (backend). DL accelerators programming with sufficient flexibility and efficiency can be performed through FlexPDA so that intelligence applications can be transplanted easily to the DL platform for fast upgrades. Specifically, abstractions of applications in DL fields and architectures similar to DaDianNao [15] are summarized firstly. DaDianNao [15] is a representative DL accelerator that supports various vector operations, which has high performance and low energy consumption. Then, based on the aforementioned abstractions, a domain-specific language for deep learning computing, FlexPDA C, is proposed as an extension of C language, which provides a series of high-performance DL calculation interfaces for programming. What is more, a backend code generator is presented as a fully functional and high-performance module based on LLVM [16] and Clang[③].

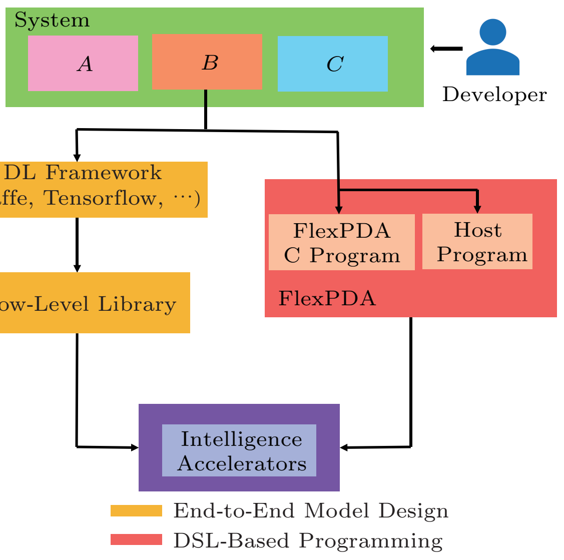


Fig.1. Example of the system with DL accelerators.

In summary, our main contributions are the followings.

• We give abstractions of the applications in DL fields and architectures similar to DaDianNao, to guide the language design.

• We present a programming framework, FlexPDA, including an easy-to-use domain-specific language, FlexPDA C, and a corresponding backend code generator, which can automatically generate the high-performance machine code for different DL operations.

• We evaluate FlexPDA by using 10 representative operators selected from deep learning algorithms and an end-to-end network. The experimental results show that FlexPDA can achieve an end-to-end performance improvement of 1.620x over the low-level library.

The rest of the paper is organized as follows. In Section 2, abstractions of applications in DL fields and architectures similar to DaDianNao are presented. Section 3 describes the overall language design as well as

①https://developer.nvidia.com/cuDNN, Sept. 2021.

②https://developer.nvidia.com/cuda-toolkit, Sept. 2021.

③http://clang.llvm.org, Sept. 2021.

example programs that use FlexPDA. The implementation and optimization for device and host code generators are depicted in Section 4. Experimental evaluations are shown in Section 5, and the discussion is given in Section 6. The related work is presented in Section 7. Conclusions are presented in Section 8.

2 Abstractions of Applications and Architectures

In this section, abstractions of intelligence applications and DL accelerators similar to DaDianNao[15] are given. The abstractions of applications that are used to acquire the characteristics of data and operations of DL algorithms, can guide the design of a language so that users are enabled to develop applications flexibly and efficiently. The abstractions of architectures that analyze the parallel structure and the storage model of accelerators can be used to guide the backend code generator to generate high-performance machine code.

2.1 Abstractions of Applications

Recent trends in technology scaling, the availability of large amounts of data, and novel algorithmic breakthroughs have spurred the adoption of intelligence accelerators. In this subsection, applications from the perspective of data structure and operations are abstracted.

2.1.1 Data Structure

In various deep learning algorithms that are executed on intelligence accelerators, data is typically in different computational nodes with the form of vectors or multi-dimensional arrays. In the field of natural language processing, an n-dimensional vector is often used to characterize a word, and then a model is trained to learn a word embedding matrix to perform tasks such as text similarity and sentiment analysis. In addition, input data, filters, extracted feature maps, etc. are usually represented by a multi-dimensional array when performing tasks in the area of computer vision such as image classification and object detection. Moreover, in the field of speech recognition, a matrix is usually required to represent the acoustic signals of a speech; thereby acoustic and linguistic models are established which can convert a speech into a piece of text. Multi-dimensional arrays are kept in a sequence of memory. The index is used to query elements and traverse the array, which is convenient and fast. Furthermore, multidimensional arrays can improve computational efficiency and provide opportunities for optimization. Therefore, each data structure is abstracted into a tensor type, an advanced data structure of an n-dimensional array, in the DL applications.

2.1.2 Operations

It can be observed that frequently used operations consist of basic operations such as addition, multiplication by constant, and DL typical operations such as convolution, pooling, and fully-connected operations. From the perspective of computational patterns, these operations are abstracted into region operations and elementwise operations.

Region Operations. Each operation such as a convolution operation and a fully-connected operation, is based on a region and produces a value. Region operations are usually used to extract image features, compress image sizes, etc. They are often used in fields such as image classification and object detection.

Elementwise Operations. Elementwise operations are the most widely used in plentiful deep learning scenarios, in which each input produces a result. This kind of operations contains not only basic operations such as vector addition and vector multiplication, but also activation operations such as sigmoid and relu operations.

2.2 Abstractions of Architectures

In this subsection, abstractions of architectures are summarized, including characteristics for multi-chip organization and storage model.

2.2.1 Multi-Chip Organization

The multi-chip system is commonly used in the DL accelerators similar to DaDianNao[15, 17], as shown in Fig.2(a). Fig.2(a) is a 4-node system. Every four nodes are connected to a DDR controller which is represented as $*$. A node adopts a tile-based organization which consists of 16 leaf tiles and one central tile. A tile mainly contains a neural functional unit (NFU) and a cache bank, as shown in Fig.2(b). NFU is largely a pipelined version of the typical computations required to evaluate an output. The cache bank is used to cache data and instructions. All the tiles are connected through an on-chip network which serves to broadcast the input values to each tile and to collect the output values from each tile.

As can be found, each node of intelligence accelerators (IAC) is relatively “heavy” and fully functional. The calculation and the data access are independent

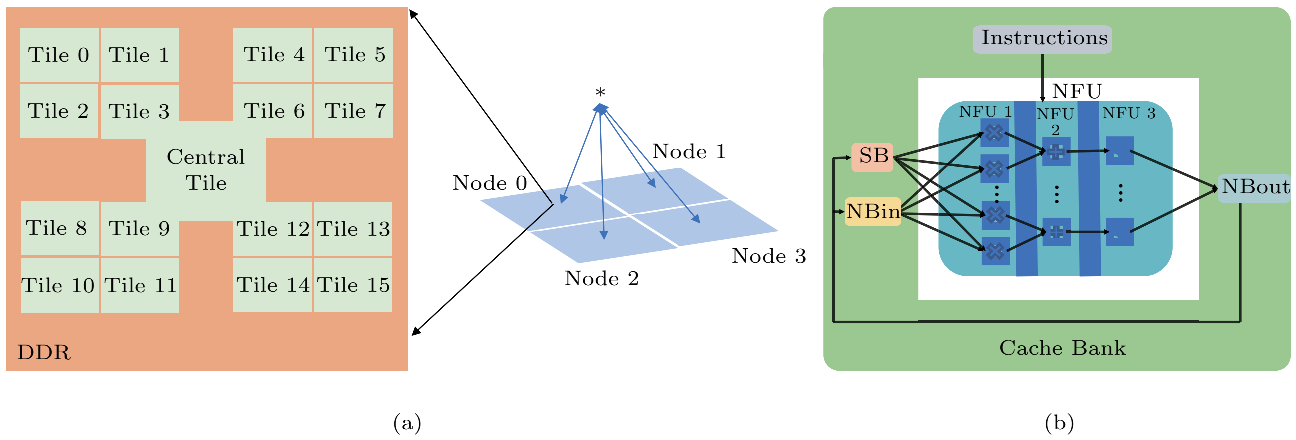


Fig.2. Architecture of DL accelerators. (a) Multi-chip organization. (b) Storage model of each tile.

between nodes. In this article, a larger granularity, the job is used to represent the computations on the nodes of accelerators. When the multi-node mode is enabled, the intelligence accelerator splits the computation into multiple jobs. Each node performs a batch of jobs, and each job contains a set of calculations. For example, when acting a multiplication of a matrix of 32×16 and a matrix of 16×8, and simultaneously specifying that four nodes are used for parallel execution, the jobs on each node may be assigned as a multiplication of an 8×16 matrix and a 16×8 matrix. The jobs between nodes are independent of each other and can communicate with each other.

To this end, a two-tuple

$$IAC_{\text{parallel}} = (job\ scale, job\ parallelism)$$

is employed as the programming abstraction of parallel of multi-chip. Among them, *job scale* represents the amount of computation, and *job parallelism* indicates how many nodes the jobs want to be decomposed to execute on.

2.2.2 Storage Model

It can be seen from Fig.2(b) that each tile has a local on-chip memory, and all tiles can access off-chip DRAM as shown in Fig. 2(a). As shown in Fig. 2(b), on-chip memory is typically split into three structures: an input buffer (NBin), an output buffer (NBout), and a synaptic weights buffer (SB). The splitting structure can tailor the appropriate read/write width of the SRAMs and avoid conflicts that probably occur in a cache. The on-chip memory that caches instructions is not visible to the user; therefore it will not be mentioned here.

In this paper, $M = \{m_1, m_2, ...\}$ is used as the storage model, where M is the on-chip memory and m_i represents the buffers with different functions in the on-chip memory. The on-chip memory of the architectures mentioned above is divided into three types. Therefore, $IAC_{\text{memory}} = \{M_{\mathtt{I}}, M_{\mathtt{O}}, M_{\mathtt{W}}\}$ is used to represent the hierarchical memory model of intelligence accelerators similar to DaDianNao. $M_{\mathtt{I}}$, $M_{\mathtt{O}}$, and $M_{\mathtt{W}}$ are used to store input data, output data, and weight parameters, corresponding to NBin, NBout and SB, respectively.

3 Frontend Language Design

3.1 Device Programming

FlexPDA C, a domain-specific language, is developed for intelligence accelerators programming. Based on the characteristics of DL applications and accelerators, FlexPDA C is implemented as an extension of C programming language. The aim of the language is to allow developers to program DL accelerators with sufficient flexibility. In this subsection, the C language is extended from three aspects which are data structure and operations, hierarchical memory model, and multi-chip parallelism.

3.1.1 Data Structure and Operations

The words in the natural language processing, filters in the computer vision, and acoustic signals in the speech recognition are all represented by n-dimensional arrays. Based on the characteristic, a tensor data structure, *DLCollect*, is introduced to help users express their DL programs.

Fig. 3 shows the grammar rules for the new data structure, *DLCollect*, and some of the operations associated with *DLCollect* in FlexPDA C, including the declaration, access, and calculation of *DLCollect* variables. *DLCollect* encapsulates the dimension and type

```
digit ::= [0-9]
letter ::= [a-zA-Z]
DLCollect-id ::= (letter | [_]) (letter | digit | [_$.])*
DLCollect-id-list ::= DLCollect-id | DLCollect-id, DLCollect-id-list decimal-literal ::= digit+
float-type ::= 'f16' | 'f32'
interger-type ::= 'i8' | 'i16' | 'i32'
DLCollect-element-type ::= float-type | interger-type
static-dimention-list ::= (',' decimal-literal)+
DLCollect-type ::= 'DLCollect' '<' DLCollect-element-type static-dimention-list '>'
MLCollec-def ::= DLCollect-type DLCollect-id-list
DLCollect-decl ::= DLCollect-def ';' | DLCollect-def ';' DLCollect-decl
DLCollect-operator ::= operator-elementwise | operator-region
DLCollect-expr ::= DLCollect-operator | DLCollect-id | DLCollect-slice
DLCollect-index ::= DLCollect-id ('[' letter ']')+     // elementwise access
DLCollect-slice ::= DLCollect-id ('[' letter ':'  digit ':'  digit ']')+     // region access
DLCollect-operator-params ::= (DLCollect-expr)+
// elementwise operators such as add, sub and mul
DLCollect-operator-elementwise ::= 'flexpda::add(' DLCollect-operator-params ')'
                                | 'flexpda::sub(' DLCollect-operator-params ')'
                                | 'flexpda::mul(' DLCollect-operator-params ')'
// region operators such as conv and maxpool
DLCollect-operator-region ::= 'flexpda::conv(' DLCollect-operator-params ')'
                           | 'flexpda::maxpool(' DLCollect-operator-params ')'
```

Fig.3. Grammar rules of the new extended data type within FlexPDA C.

for data, which is similar to ndarray in numpy[4]. When declaring *DLCollect* variables, the length of each dimension needs to be explicitly specified.

In terms of data access, the access to individual elements through subscripts for elementwise operations and the access to multiple elements of adjacent regions by hash expression for region operations are offered. For example, $map[x][y]$ represents the access to the element at coordinates (x, y), and the hash expression $map[x : 3 : 1][y : 3 : 1]$ represents a region with the coordinates (x, y) as the starting point, a length of 3 in the X/Y direction, and an adjacent element interval of 1.

The elementwise and region operations of *DLCollect* are implemented by built-in functions. For example, Fig.4 lists the interface of convolution operation of *DLCollect*. The input data *input* and convolution kernels *filter* with the *DLCollect* type, and the length of the vertical or horizontal translation after each convolution *stride_height* and *stride_width*, are transferred to the *flexpda::conv* interface. The convolution results with the *DLCollect* type will be kept in *output*.

3.1.2 Memory Hierarchy

Unlike CPUs which present their memory as a uniformly accessible address space, it can be found that the storage model of intelligence accelerators, IAC_{memory}, consists of three types of memory hierarchy: $M_{\mathtt{I}}$, $M_{\mathtt{O}}$, $M_{\mathtt{W}}$, from Section 2. Fatahalian *et al.*[18] demonstrated that we can benefit from the design of exposing the notion of the hierarchical memory to language. In this paper, the division of the memory hierarchy is also taken into consideration in the design of FlexPDA C.

FlexPDA C presents a variety of memory keywords that allow the user to explicitly control the allocation of data. The code generator is aware of all of these mem-

void flexpda::conv(DLCollect output, DLCollect input, DLCollect filter, int stride_height, int stride_width)

A convolutional operation, which identify characteristic elements of the input data

output: The result of convolution with $< datatype, C_{\text{o}}, H_{\text{o}}, W_{\text{o}} >$ shape.

input: The input of convolution with $< datatype, C_{\text{i}}, H_{\text{i}}, W_{\text{i}} >$ shape.

filter: The weight parameters of convolution with $< datatype, C_{\text{o}}, C_{\text{i}}, H_{\text{f}}, W_{\text{f}} >$ shape.

stride_height: Length of translation in the vertical direction after each convolution operation.

stride_width: Length of translation in the horizontal direction after each convolution operation.

Fig.4. Convolution operation of *DLCollect*.

[4]http://www.numpy.org, Sept. 2021.

ory hierarchies and is able to automatically optimize each of them. Table 1 is a list of memory hierarchies in the language. Variables with "*__icache__*", "*__ocache__*" and "*__wcache__*" represent the creation of the statically sized space on $M_{\mathtt{I}}$, $M_{\mathtt{O}}$, and $M_{\mathtt{W}}$, respectively.

Table 1. Memory Hierarchies of FlexPDA C

Memory Hierarchy	Description
__icache__	FlexPDA C memory hierarchy, corresponding to $M_{\mathtt{I}}$
__ocache__	FlexPDA C memory hierarchy, corresponding to $M_{\mathtt{O}}$
__wcache__	FlexPDA C memory hierarchy, corresponding to $M_{\mathtt{W}}$

$M_{\mathtt{I}}$, $M_{\mathtt{O}}$, and $M_{\mathtt{W}}$ memories are always allocated by using on-chip resources of the accelerators. By default, they are not accessible by the host. Each memory hierarchy is guaranteed to appear coherently to the developer. The resources used to implement each memory hierarchy are restricted. The developers need to ensure that the data stored at each memory hierarchy cannot exceed the size allocated in on-chip.

The design of memory hierarchy gives users the right to manage the memory of accelerators so that the access latency can be decreased by exactly controlling the memory hierarchy of data. Moreover, the backend code generator will perform architecture-related optimizations based on the information of the memory hierarchy passed by FlexPDA C.

3.1.3 Parallelism

The architectures of DL accelerators similar to DaDianNao[15] are based on the multi-chip organization. A two-tuple is employed as the abstraction of parallel programming (refer to Section 2). When performing programs in multi-node mode, IAC_{parallel} must be specified on the host. When the computation is implemented with one node, the job parallelism and the job scale are set to 1 respectively.

For example, $IAC_{\text{parallel}} = (64, 8)$ indicates that the job scale is 64, and eight nodes are selected to execute jobs parallelly. In the program, *chipId* is used to index the node where the current job is located. It should be noted that the job parallelism cannot exceed the number of nodes of accelerators. It can be seen that the parallelism of job granularity makes the parallel management hierarchy more distinct, and easier for users to carry out parallel computing.

3.2 Host Programming

In the FlexPDA programming framework, the corresponding host interface needs to be called if the application developer wants to speed up a certain part of the system with DL accelerators. The parameters required by the kernel function, the function pointer of the kernel function, and the parallel parameters will be passed to the host interface. Fig.5 shows the interface specification of *flexpda::executeKernel*. As can be found, the programming on the host is efficient for users.

```
flexpda::executeKernel(param1,
                       param2,
                       ...
                       the function pointer of
                       kernel,
                       IACP);
```

Fig.5. Host interface.

3.3 Flexibility of FlexPDA

FlexPDA is a flexible programming framework for DL accelerators. DL accelerators programming with sufficient flexibility and efficiency can be performed through FlexPDA so that intelligence applications can be transplanted easily to the DL platform for fast upgrades. In this subsection, we discuss the flexibility of FlexPDA from two aspects.

Performing Low-Level Optimizations. The low-level library such as DLPlib[19] is commonly used in deep learning frameworks such as Caffe[5], Tensorflow[6]. The computational-graph level optimizations such as channel pruning, operator fusion can be performed, but optimizations of code generation are challenging for developers because the low-level information of deep learning accelerators is hidden in the deep learning frameworks. FlexPDA, which exposes the notion of hierarchical memory to programming language and provides a parallel mechanism for users, is complement with these deep learning frameworks. The developers can decrease the access latency of algorithms through exactly controlling the memory hierarchy of variables. Besides, low-level optimizations can be performed according to the computation pattern of a specific algorithm. For example, the data transmission can be covered by the elaborate-designed double buffer. The register overflow can be trimmed down by reducing the

[5]https://github.com/BVLC/caffe, Aug. 2022.

[6]https://github.com/tensorflow/tensorflow, Aug. 2022.

use of local variables. Moreover, it is convenient to vectorize scalar calculations and design efficient parallel methods with FlexPDA. The low-level optimization of FlexPDA greatly contributes to the performance improvement of deep learning algorithms. The experimental results demonstrate the effectiveness of low-level optimization (refer to Subsection 5.3 and Subsection 5.4).

Customizing Functional Module. With the outbreak of the artificial intelligence revolution again, various efficient neural network models for different computational complexity and memory access budgets have sprung up. The new network architectures reduce computational and memory overhead while maintaining the accuracy by introducing new operations. However, the infrastructures of deep learning accelerators such as high-performance libraries are ignorant to users, which makes it difficult to add new modules of specific functions to the framework. For example, if the library of a dedicated accelerator does not provide Convolution-Depthwise operation, then many advanced networks such as ShuffleNet[20] cannot directly use the accelerator to enhance performance. In this case, users have to implement the operation by themselves. In FlexPDA, the data structure of *DLCollect* and the operation of *flexpda::conv* can be used to customize the depthwise separable convolution. FlexPDA can help users to add specific functional layers for new network architectures.

3.4 Usage and Samples

In this subsection, FlexPDA C is discussed with two samples. One sample displays the realization of convolution, and the other sample presents the application of multi-chip parallelism.

Fig. 6 and Fig. 7 show how the convolution is implemented within FlexPDA C on device and host respectively. As shown in Fig.6, on the device, the entry function *ConvKernel* can only be called by the host program which is identified by *__global__*. In *ConvKernel*, first of all, three variables with *DLCollect* type which are initialized by the input of the entry function (*output* is initialized to 0) are constructed. Then, the *flexpda::conv* interface is called to make the convolution operation. Finally, the first address of the result of convolution *output* is passed back to *out_data*. The interfaces such as *flexpda::add* and *flexpda::conv* which are provided by FlexPDA C, can help users flexibly and efficiently implement various deep learning algorithms. As shown in Fig. 7, on the host, the parallel parameter *IACP* (IAC_{parallel}) is set to $(1, 1)$ since the multi-chip parallelism is not used. Then, the interface *flexpda::executeKernel* is called.

Fig. 8 and Fig. 9 display how to achieve multi-chip matrix multiplication within FlexPDA C on device and host respectively. As shown in Fig.8, on the device, in the *MultiNodeMatrixMUL* entry function, firstly, three *DLCollect* instances *input*1, *input*2, *output* are constructed and initialized with the data of the entry function (*output* is initialized to 0). Then, all nodes are synchronized. Secondly, the nested loop is employed to implement the multiplication of an $M \times P$ matrix and a $P \times N$ matrix with multi-chip mode. Finally, the address of the calculation result *output* is passed back to *dst*. As shown in Fig. 9, on the host, *IACP* (IAC_{parallel}) is set to $(M, 8)$. The multiplication calcu-

```
__global__ void ConvKernel(half *out_data,
               half *in_data,
               half *filter_data) {
// Construct and initialize DLCollect instances
__icache__ DLCollect<half,
          IN_CHANNEL,
          IN_HEIGHT,
          IN_WIDH> input(in_data);
__wcache__ DLCollect<half,
          OUT_CHANNEL,
          IN_CHANNEL,
          FILTER_HEIGHT,
          FILTER_WIDH> fileter(filter_data);
__ocache__ DLCollect<half,
          OUT_CHANNEL,
          OUT_HEIGHT,
          OUT_WIDH> output(0);
// Call the interface to perform convolution calculation
flexpda::conv(output, input, filter,
       STRIDE_HEIGHT, STRIDE_WIDTH);
// Store output
out_data = output.data();
}
```

Fig.6. Implementation of convolution on device within FlexPDA C.

lation is divided into eight jobs, and each node performs the multiplication of a matrix of $\frac{M}{8} \times P$ and a matrix of $P \times N$. Then, the interface *flexpda::executeKernel* is called.

```
flexpda::parallel IACP(1,1);
flexpda::executeKernel(output, input, weight,
                       &ConvKernel, IACP);
```

Fig.7. Implementation of convolution on host within FlexPDA C.

```
__global__ void MultiNodeMatrixMUL(half *dst,
                   half *src1,
                   half *src2) {
// Construct and initialize DLCollect instances
__icache__ DLCollect<half, M, P> input1(src1);
__icache__ DLCollect<half, P, N> input2(src2);
__ocache__ DLCollect<half, M, N> output(0);
// Synchronize all nodes
flexpda::sync_chips();
// Perform matrix multiplication calculation with
   multi-chip parallel
int everyChipJob = ceil(M/chipDim);
for (int i = everyChipJob * chipId;
    i < min(M, everyChipJob * (chipId + 1));
    i++) {
 for (int j = 0; j < N; j++) {
  for (int k = 0; k < P; k++) {
    output[i][j] += input1[i][k] * input2[k][j];
  }
 }
}
// Store output
dst = output.data();
}
```

Fig.8. Implementation of multi-chip matrix multiplication on device within FlexPDA C.

```
flexpda::parallel IACP(M,8);
flexpda::executeKernel(result, src1, src2,
                       &MultiNodeMatrixMUL, IACP);
```

Fig.9. Implementation of multi-chip matrix multiplication on host within FlexPDA C.

4 Backend Code Generator

In this section, the backend code generator of FlexPDA based on LLVM compiler infrastructure[16] is presented, as shown in Fig.10. The input contains the followings: 1) a kernel program in which an algorithm is implemented by users using FlexPDA C, whose name is suffixed with .iac, for example, kernel.iac; 2) a host program that runs on the CPU, whose name is suffixed with .cpp, for example, host.cpp.

On the device, the features of syntax including various memory hierarchies, built-in parallel variables, and the implementation of various DL interfaces, will be supported by the device code generator. Next, the frontend of FlexPDA will perform memory management such as the mapping of memory hierarchies and the data transmission between different memory hierarchies. Moreover, the device code generator will achieve pointer address space inference based on a data flow analysis, and data layout optimization based on pointer address space inference. The code generator generates object files for the device program.

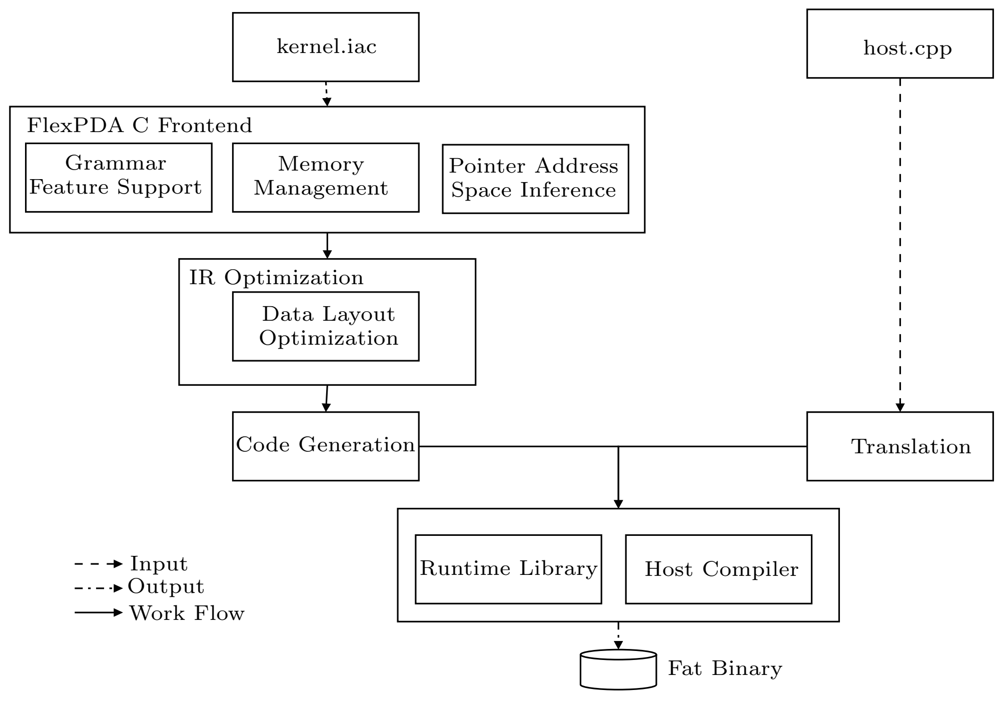


Fig.10. Overview of the backend code generator of FlexPDA based on LLVM compiler infrastructure[16].

On the host, the host program is translated firstly. Secondly the translated host program is compiled using the host compiler. The runtime library of DL accelerators provides a set of APIs for the communications between the host and the device. Finally, the executable file, fat binary, is built by the host compiler through linking the object files generated by the device and the host, and the runtime library.

4.1 Device Code Generator

4.1.1 *Grammar Feature Support*

FlexPDA C extends the C language in terms of the data structure and operations, hierarchical memory model, and multi-chip parallelism. In this subsection, the grammar features are introduced from the following perspectives.

Tensor Data Structure. There are two solutions for implementing the tensor data structure: 1) an external library with tensor type and various relevant operations; 2) a built-in type in which operations are fulfilled as built-in functions. The hardware characteristics of accelerators, such as the pattern of memory access and SIMD instructions, cannot be utilized by external libraries effectively. FlexPDA adopts the second solution which has built-in *DLCollect* type, and all the operations have corresponding built-in functions.

Multi-Chip Parallel. Multi-chip parallel in which each node processes a subset of task has shown promising results for computationally intensive algorithms [21]. The parallel parameters corresponding to the job scale and the job parallelism are built into the code generator of FlexPDA. It is flexible and simple to manipulate programs in parallel through built-in variables. When the multi-chip mode is enabled to achieve operations in the algorithms, the device code generator will control the synchronization of the multi-chip by inserting the synchronization instructions to ensure data consistency.

Architecture Distinction. New language features will be added by different architectures of accelerators, such as new operations and new data types. A distinction between architectures is required. The option of the command line, –flexpda-arch, is put to the frontend of the FlexPDA by the code generator to specify the specific architecture version. The parameters of the command line can be used for checking the new features at the frontend and lowering instructions at the backend.

4.1.2 *Memory Management*

The device code generator of FlexPDA provides memory management for IAC_{memory}. In this way, on-chip resources can be efficiently utilized, which greatly reduces the number of off-chip memory accesses and improves throughput. The device code generator of FlexPDA assigns an identifier for each memory hierarchy and binds it to the corresponding storage qualifier. When a variable is declared with a storage keyword, the device code generator will bind the attribute of memory to the variable type in the declaration. Depending on the type, the optimal access IR associated with the variable will be generated, so that the access latency can be decreased and resource utilization can be improved.

In addition, the device code generator manages the transmission of the data between different memory hierarchies. For example, when a variable with the storage qualifier "*__icache__*" is initialized by the data in DDR, a copy of data from DDR to M_{I} will be performed. For various vector operations, if the attribute of memory of output is not given, the code generator of device will transfer the result to DDR. Otherwise, the code generator of device will copy the operation result to the corresponding memory hierarchy.

The device code generator of FlexPDA which supports the memory hierarchy, allows the user to explicitly control the storage without considering the transmission of data. The device code generator will help the user transfer data according to the attributes of memory of variables, which trims down the burden on the user and improves the performance of the operations.

4.1.3 *Pointer Address Space Inference*

The device code generator of FlexPDA makes use of storage keywords to specify the memory space in which the variable is located. Based on the storage qualifier, the code generator will generate faster ld/st instructions. It is well known that accessing data from on-chip is faster than from DDR. However, the variable with the pointer type has no storage qualifier, and its attribute of memory is related to the attribute of memory of the space it points to. For example, pointer p is an M_{I} pointer when p points to an M_{I} space in line 7 of Fig.11, and p is a DDR pointer when p points to the DDR space in line 10 of Fig.11. The memory space of the pointer expression needs to be determined so that the optimal ld/st instruction can be selected for the subsequent access to the pointer. To this end, an optimization of pointer address space inference that analyzes the attributes of memory of pointers is proposed.

```
 1 half *p;
 2 half vi;
 3 // Apply for a 256-byte on-chip buffer
 4 __icache__ half array_icache[128];
 5 // Apply for a 256-byte off-chip buffer
 6 half array[128];
 7 p = array_icache;
 8 // Load data from on-chip buffer
 9 vi = *p;
10 p = array;
11 // Load data from off-chip buffer
12 vi = *p;
```

Fig.11. FlexPDA C example of address space inference.

The pointer address space inference is implemented through a data flow analysis based on a recursive traversal syntax tree (Algorithm 1). The device code generator performs the optimization of pointer address space inference on each function. The algorithm recursively traverses subexpressions for each pointer expression of a function. For a pointer expression of the declarative reference type such as the variable p in statement "*half *p = array*", if the defined expression is a non-pointer expression, then PASI obtains the address space of the defined expression as the address space of the pointer; otherwise the definition expression is traversed recursively. For a unary or conversion expression, its subexpressions will be recursively traversed. For a binary or conditional expression, its left or right child will be recursively traversed. Finally, a non-pointer expression that is initialized or assigned is found. The address space of the non-pointer expression is the address space of the pointer.

As shown in Fig.11, a general instruction *ld.lddr.f16* will be generated in line 12. The faster instruction *ld.icache.f16* will be generated in line 9.

4.1.4 Data Layout Optimization

The DL accelerators usually provide corresponding vector instructions for region operations and elementwise operations in the DL. However, the operands of these vector instructions must be stored in on-chip, and are restricted in terms of data precision, data storage, and data alignment. For example, in order to improve the efficiency of memory access, data needs to be stored in a low-precision type. When the input data or weight parameters of the operation are stored in the DDR, the accelerators perform a series of scalar instructions to complete the operation by accessing the off-chip data. When the data layout of operands stored in on-chip does not meet the requirements of memory access of vector instructions, the scalar instructions will be employed to implement calculations.

Algorithm 1. Pointer Address Space Inference (PASI)

```
Input: a function $F$
Output: a set of pointers with $AS$
$GAS \leftarrow \emptyset$;
for pointer $EP$ used in $F$ do
    if $EP$ is a declarative reference then
        if $EP.getDefineExpr()$ without a pointer type then
            $AS \leftarrow getAddressSpace(EP.getDefineExpr())$;
        else
            $AS \leftarrow PASI(EP.getDefineExpr())$;
        end
    else if $EP$ is a unary operator or a conversion operator then
        // e.g., &a, p++, ++p;
        // e.g., convert an array type to a pointer type;
        $AS \leftarrow PASI(EP.getSubExpr())$;
    else if $EP$ is a binary operator or a conditional operator then
        if the left operand of $EP$ without void type then
            $AS \leftarrow PASI(EP.getLHS())$;
        else
            $AS \leftarrow PASI(EP.getRHS())$;
        end
    else
        $AS \leftarrow$ AddressSpace::DDR;
    end
    $GAS \leftarrow GAS \cup (EP, AS)$
end
return $GAS$;
```

To take full advantage of the DL accelerators, when the DDR data is passed to the interfaces in FlexPDA C, the device code generator will lower LLVM-IR to a series of scalar instructions to complete the operation. When the interface parameters are stored in on-chip, the code generator will deal with the data layout problem, and generate corresponding legal vector instructions to achieve the operation.

The optimization of the data layout is performed based on the LLVM-IR instructions (Algorithm 2). This optimization depends on the address space inference algorithm mentioned above. For each IR instruction in each function, the code generator of device will check whether the operation belongs to the set of vector instructions supported by the DL accelerators. The legality of memory hierarchies of operands will also be checked based on the address space inference algorithm. For example, the input data must be stored in $M_{\texttt{I}}$, and the weight parameters must be stored in $M_{\texttt{W}}$ for convolution and fully-connected instructions. Then, a series of conversions consisting of data precision conversions, storage layout conversions and data alignment operations are performed on the operands of legal IR instructions. For example, the precision conversions from f32 to f16 type are performed firstly for weight parameters. In addition, it is necessary to transpose the weight matrix from the row-first storage to the column-first storage. Last but not least, the size of operands has aligned constraints on DL accelerators. Therefore, padding operations are performed by the code generator of the device to satisfy the requirements of alignment of operands.

Algorithm 2. Data Layout Optimization (DLO)

```
Input: a function $F$ and basic operation set $BOS$
Output: a function $F'$ with DLO
$GAS \leftarrow \emptyset$;
for IR instruction $I$ in $F$ do
    if $I \in BOS$ and $checkOperandsMemoryHierarchy(I)$ then
        for operand $O$ in $getOperands(I)$ do
            $O1 \leftarrow dataPrecisionConversion(O)$;
            $O2 \leftarrow transposeOperation(O1)$;
            $O3 \leftarrow alignOperation(O2)$;
            $replaceOperands(I, O, O3)$;
        end
        $insertInstruction(F', I)$;
    else
        $insertInstruction(F', I)$
    end
end
```

4.1.5 Code Generation

The object code generation is a vital part in the device code generator. The resources used to implement each memory hierarchy are restricted; therefore the border checks on access to different memory hierarchies will be performed during the generation of the object code. Besides, synchronization instructions are inserted in front of the data transfer instructions to ensure the consistency of data.

4.2 Host Code Generator

In this paper, we use the API of DLPlib[19] to achieve the management of memory and the call of kernel of computation on the host. DLPlib, a low-level library, provides a series of efficient and versatile programming interfaces for accelerating various deep learning algorithms on the DL accelerators similar to DaDianNao. Specifically, the invoked host interface *flexpda::executeKernel* is translated into a series of host-called API of DLPlib. For example, the interface *dlpMalloc* allocates space to parameters on the device side, the interface *dlpMemcpy* completes the data transmission between the host and the device, the interface *dlpConvolutionForward* performs the kernel of convolution, and the interface *dlpFree* releases the memory of device allocated for parameters.

4.3 Runtime Library

The runtime library of DL accelerators provides a set of APIs for communications between the host and the device. Services such as the management of devices, memory, and execution contexts are offered by APIs. The interfaces of the management of devices are provided by device management, such as the initialization of device and the designation of device. The interfaces of management of memory are provided by memory management, such as memory allocation and memory release. The execution contexts are responsible for the asynchronism, synchronization, and scheduling of task queues. In addition, interfaces are provided by the runtime library to enable the reuse of instructions and data for offline models that support the separation of instruction data.

5 Evaluation

Hardware Platform. In this paper, we use the DaDianNao[15] architecture to evaluate our proposed framework. The Verilog[22] and VCS (Synopsys Verilog Compiler Simulator) are used to implement, compile, and simulate the DaDianNao architecture. The architecture has 16 PEs. Each PE has 16 multipliers and one 16-in adder tree, which are used for the vector inner product of 16 half-precision floating-point numbers. In addition, in the on-chip memory, the SB of 2 KB is sustained in each PE, and the NBout of 8 KB and the NBin of 8 KB are shared by all PEs. Besides, we achieve the frequency of 1 GHz for the simulator. The CPU is Intel Xeon CPU with 3 GHz, and the operating system is Ubuntu Linux (version 16.04.4 LTS).

Benchmarks. In order to evaluate the system of FlexPDA, we select two categories of benchmarks based on the application scenarios: one is the low-level elementwise operations such as addition commonly used in various general algorithms, and the other is representative region operators such as convolution typically used in deep learning applications. In addition, we select a representative DNN model, AlexNet[23], to evaluate the end-to-end performance. Three low-level elementwise operators, which contain ADDITION, MULTIPLICATION, and MULCONST, have 2M input. Besides, seven deep learning region operators are described in Table 2. Moreover, it is found that the performances of the ADDITION, MULTIPLICATION, and MULCONST operators are very similar; therefore their average is taken in the experiment. Furthermore, we com-

Table 2. Configurations of Seven Deep Learning Region Operators

Operator	O_c	I_c	I_h	I_w	Kernel	Stride	Pad	O_h	O_w	Layer	Neural Network
Conv1	256	128	14	14	3	1	1	14	14	res4a_2a	ResNet-18
Conv2	64	192	28	28	1	1	0	28	28	conv4_3	MobileNet_v2
Conv3	128	64	15	15	3	1	1	15	15	conv2.1	ConvNet
Pool1	128	128	15	15	3	2	-	8	8	pool2	ConvNet
Pool2	128	128	112	112	2	2	-	56	56	pool2	Vgg-16
FC1	1 000	4 096	-	-	-	-	-	-	-	fc8	Vgg-16
FC2	4 096	4 096	-	-	-	-	-	-	-	fc7	Vgg-16

Note: O_c/I_c: input channel/output channel; I_h/I_w: input height/input width; O_h/O_w: output height/output width; Kernel: kernel height/width; Stride: stride height/width; Pad: padding height/width.

pare FlexPDA over low-level library using AlexNet [23] as the benchmark.

In this section, the performance improvements of data layout optimization are evaluated firstly through the representative 10 DL operators. Secondly, the performances of serial C and FlexPDA are compared by the representative 10 DL operators. We also compare the performances of FlexPDA and parallel C which employs OpenMP [24]. Finally, we present the end-to-end performance speedup of FlexPDA over the low-level library.

5.1 Data Layout Optimization Evaluation

In order to utilize the characteristics of the architectures of DL accelerators, the on-chip operands of operations in FlexPDA are adjusted in the data layout optimization in terms of data precision, data storage, and data alignment. In this subsection, the performance benefits of data layout optimization are evaluated through the selected 10 DL operators.

The storage space of each memory hierarchy is restricted on DL accelerators. The performance improvements of three elementwise operators with data layout optimization are evaluated, and the tiling sizes are 128 B, 256 B, 512 B, 1 KB, 2 KB, 4 KB, 8 KB, 16 KB, 32 KB, 64 KB, respectively. The average speedups of the three elementwise operators with data layout optimization under different tiling sizes are shown in Fig.12. On average, the FlexPDA with data layout optimization is able to achieve a speedup of 1.790x, 3.481x, 6.305x, 8.333x, 11.479x, 12.377x, 16.992x, 23.054x, 27.496x, 33.048x for the tiling size of 128 B, 256 B, 512 B, 1 KB, 2 KB, 4 KB, 8 KB, 16 KB, 32 KB, 64 KB, respectively. The experimental results show that the overheads of data transmission decrease, so that better performance is realized as the tiling size gradually increases.

In Fig.13, the speedups of the seven region operators with data layout optimization are presented. As can be seen, FlexPDA can achieve a speedup of 43.173x–66.131x, 42.971x–46.186x, and 63.284x–77.634x for con-

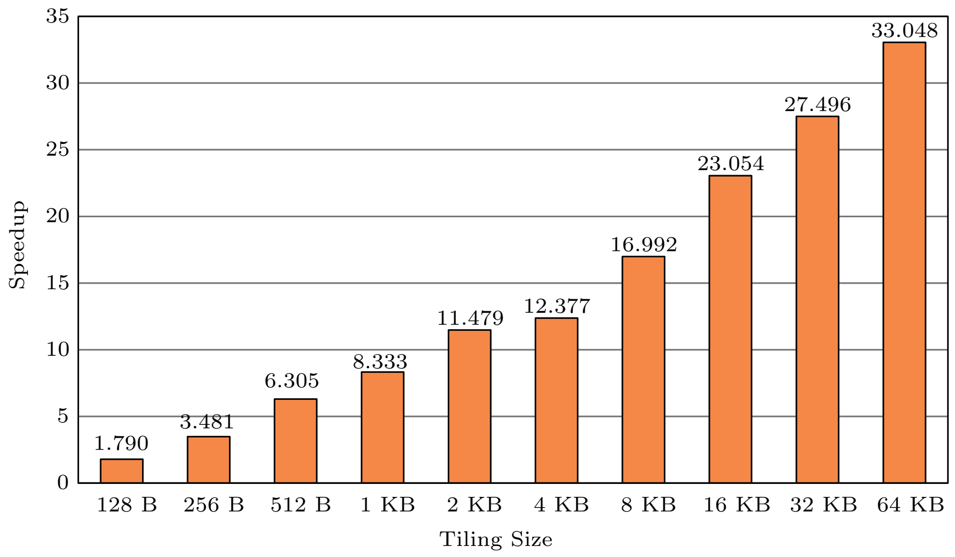


Fig.12. Average speedups of three elementwise operators with data layout optimization under different tiling sizes.

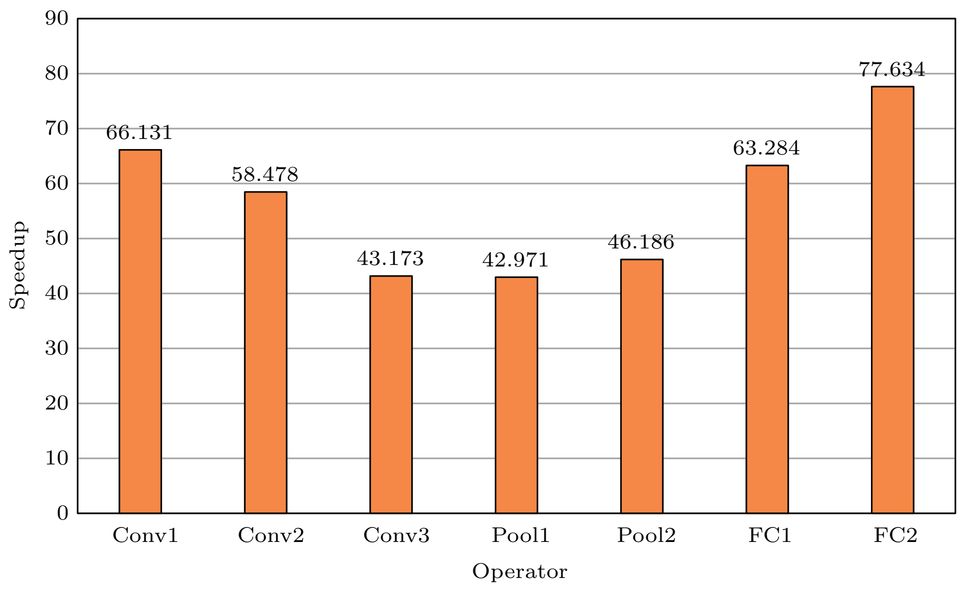


Fig.13. Speedups of seven region operators with data layout optimization.

volution, pooling, and fully-connected operations, respectively.

It can be seen that the performance of 10 DL operators is significantly improved after data layout optimization. In addition, the computing-intensive operations can achieve amazing performance improvement, such as convolution and fully-connected operators.

5.2 Performance Comparison with Serial C

Nested loops are commonly used in C to implement region and elementwise operations in DL. FlexPDA can implement these operations with just one statement. In addition, developers using C can quickly get started with FlexPDA and use DL accelerators to speed up their algorithms. In this subsection, the performances of serial C and FlexPDA are compared. We use "serial C" to represent the serial implementation of the program using C language running on the CPU platform.

When the operands of operators are in on-chip, FlexPDA will perform data layout optimization, so that the vector instructions can be used to perform the operations. In this subsection, the performance improvements of FlexPDA with on-chip input over serial C implementation are presented for the 10 DL operators.

Fig. 14 shows the average performance benefits of three elementwise operators with on-chip input under different tiling sizes, compared with serial C implementation. On average, compared with serial C program, FlexPDA with tiling sizes of 128 B, 256 B, 512 B, 1 KB, 2 KB, 4 KB, 8 KB, 16 KB, 32 KB, 64 KB can achieve a speedup of 0.923x, 1.869x, 3.409x, 5.898x, 10.964x, 20.436x, 34.982x, 55.369x, 76.981x, 95.441x, respectively.

It can be observed that when the tiling size is 128 B, the performance of the FlexPDA is not so good as serial C, because the overheads of data transmission are greater than the computational benefits on DL accelerators. The performance benefits of the FlexPDA are better as the tiling size becomes larger.

Fig.15 shows the performance improvements of the seven region operators. As can be seen, FlexPDA can achieve a speedup of 157.392x–208.686x, 60.669x–66.755x, and 106.110x–125.465x for convolution, pooling, and fully-connected operations, respectively. For operations with large input, output or weight, there is no enough on-chip memory to load all data used by operators. The data blocking scheme is used to complete the operations. The operations with higher computational intensity, such as convolution, will overlap more overheads of memory access which can achieve better performance improvements. Overall, the larger the data scale, the greater the overhead of memory access, and the lower the performance, when the on-chip resources are insufficient.

5.3 Performance Comparison with OpenMP

OpenMP [24] is an efficient programming framework on general-purpose processors. It can employ C to implement parallel applications of the shared memory, which takes full advantage of the characteristic of flex-

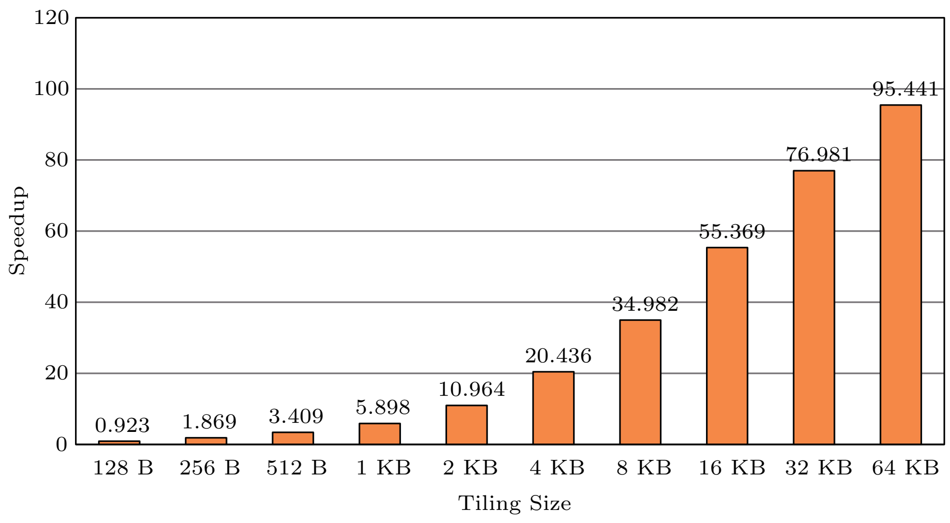


Fig.14. Average speedups of three elementwise operators with on-chip input under different tiling sizes (FlexPDA vs serial C).

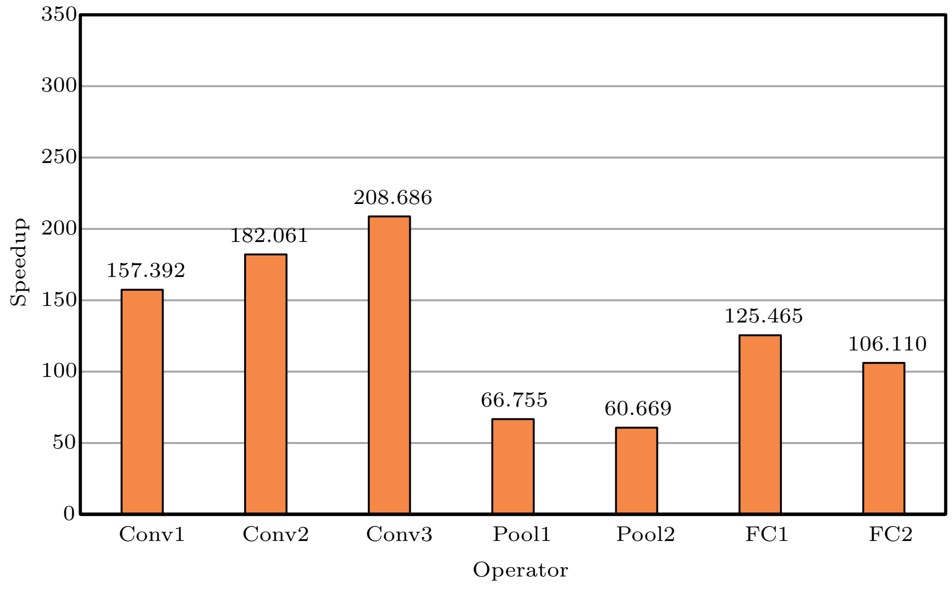


Fig.15. Speedups of seven region operators with on-chip input (FlexPDA vs serial C).

ible support of general-purpose processors for various workloads. The proposed approach in this paper, FlexPDA, is a programming framework for dedicated deep learning accelerators such as DaDianNao. It is implemented as an extension of C, which makes full use of the instruction set and storage characteristics of the deep learning accelerators. In this subsection, we compare FlexPDA with OpenMP using serial C as a baseline. OpenMP implementations are running on the Intel Xeon CPU. Besides, for the OpenMP, the performances of OpenMP programs with different numbers of threads are shown. For FlexPDA, performances in Fig.15 and the performance of the 64 KB tiling size in Fig. 14, are selected as the performances of operators with on-chip input.

Fig. 16 shows the performance improvements of 10 operators implemented by FlexPDA and OpenMP, compared with serial C. The elementwise speedup is the average speedup of ADDITION, MULTIPLICATION, and MULCONST operators. OpenMP can achieve a speedup of 1.37x–12.13x, 0.12x–1.01x, 0.98x–1.14x, and 0.91x–4.03x over serial C for convolution, pooling, fully-connected and elementwise operators respectively. However, FlexPDA can improve the performances by 157.39x–208.69x, 60.67x–66.76x, 106.11x–125.47x, and 95.44x over serial C for convolution, pooling, fully-connected and elementwise operators respectively, as shown in Fig.16.

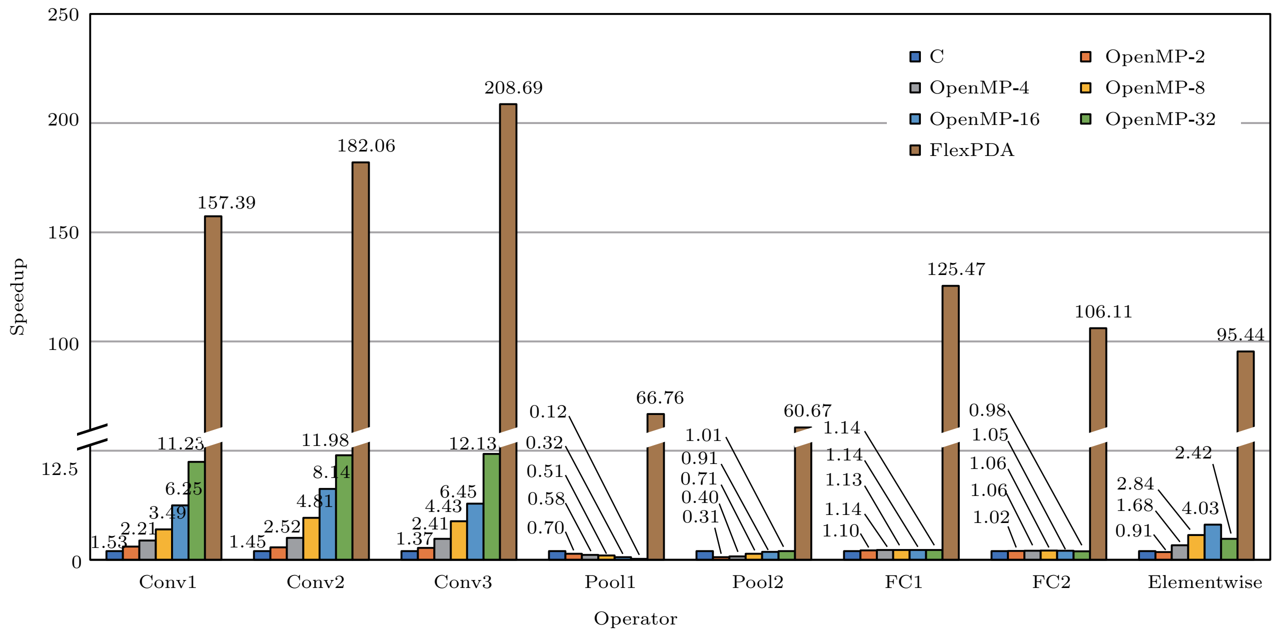


Fig.16. Speedups of 10 operators. Serial C is the baseline (speedup = 1x). OpenMP-2, OpenMP-4, OpenMP-8, OpenMP-16, and OpenMP-32 indicate that the performance speedups of the OpenMP program with 2, 4, 8, 16, and 32 threads compared with serial C, respectively. FlexPDA represents the performance speedups of the FlexPDA program with on-chip input compared with serial C.

The experimental results show that the computing and storage resources of the deep learning accelerator can be utilized by FlexPDA. In addition, as can be observed, as the number of threads increases, the performance of convolution operators implemented by OpenMP improves, the OpenMP performance of the Pool1 operator with a smaller input scale decreases significantly, the OpenMP performance of the Pool2 operator with a larger input scale is slightly improved, the OpenMP performance of fully-connected operators does not fluctuate much, and the performance of OpenMP of elementwise operators improves first and then decreases with 16 threads as the demarcation point. It can be seen that the performance benefits obtained by enabling multi-thread of OpenMP are unstable with the impact of the specific algorithm and the input scale of the algorithm, and there are additional overheads when using OpenMP, such as the overhead of creation of threads.

5.4 Performance Comparison with Low-Level Library

We compare FlexPDA with DLPlib[19] using AlexNet[23] as a benchmark in this subsection. DLPlib is commonly used in deep learning frameworks such as Caffe, Tensorflow. Therefore, we use Caffe[13], a representative open-source deep learning framework, to evaluate the performance of DLPlib. An important observation about deep neural networks (DNNs) is that the convolution layers and fully-connected layers occupy most of the time during the inference of the entire network. Therefore, we replace the convolution and fully-connected layers achieved by DLPlib with convolution and fully-connected layers achieved by FlexPDA to evaluate FlexPDA and DLPlib.

Fig.17 presents the end-to-end performance speedup of FlexPDA over DLPlib through AlexNet. The net contains eight learned layers: five convolutional and three fully-connected layers. The convolution and the fully-connected layers account for 67% of the execution time of AlexNet with DLPlib. FlexPDA achieves a performance improvement of 1.620x over DLPlib after replacement. Moreover, the library of DLPlib is 4.1x faster than GPU according to [19]. Therefore, FlexPDA achieves a performance speedup of 6.6x over GPU. The convolution and the fully-connected layers performed by FlexPDA achieve a performance improvement of 4.152x and 1.727x on average over DLPlib respectively. In addition, the performance of the conv1 of AlexNet is improved significantly. However, FlexPDA performs worse on FC8 of AlexNet than DLPlib. On the whole, the convolution layers contribute more to the overall performance gain of the network than fully-connected layers.

As we can observe, FlexPDA is comparable to the DLPlib library on the entire network. The performance

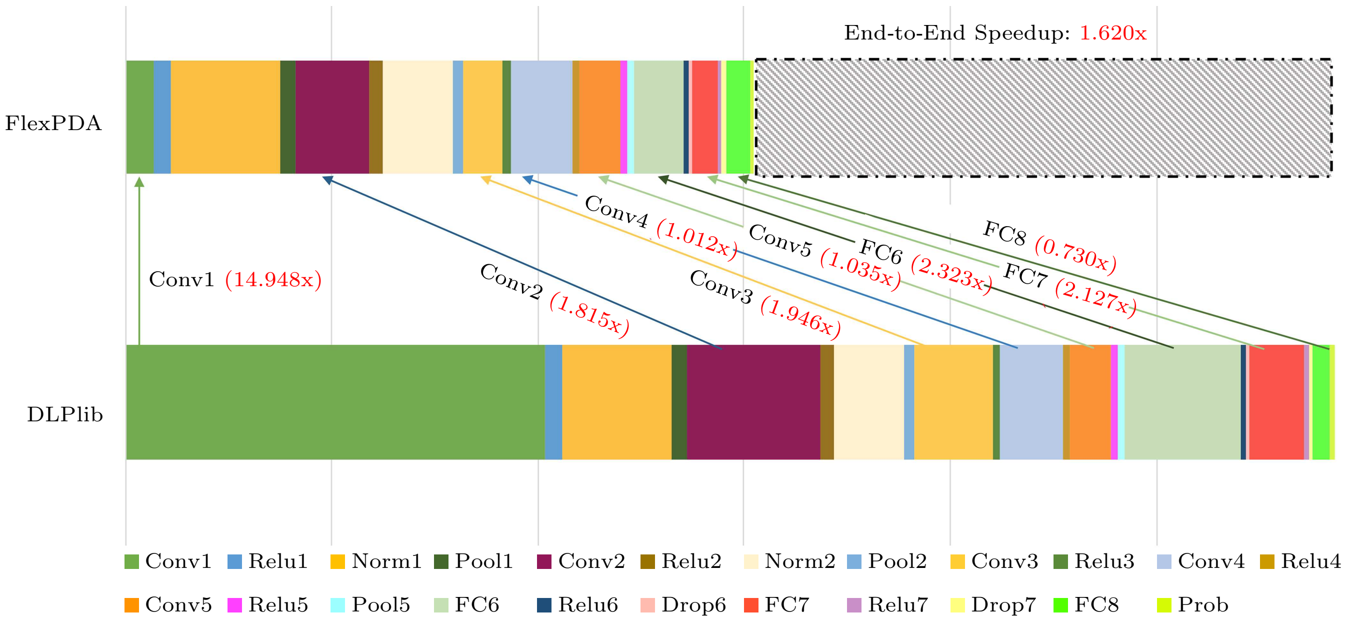


Fig.17. Layerwise performance speedups of AlexNet with FlexPDA over DLPlib.

improvement substantially comes from the optimization of pointer address space inference and data layout, and the low-level tuning in the implementation of operators and the detailed explanations are presented in Subsection 5.5.

5.5 Performance Analysis

In this subsection, we conduct an in-depth analysis of the performance of FlexPDA. The experimental evaluations show that FlexPDA is comparable to OpenMP and DLPlib. The performance improvement substantially comes from the following three aspects.

Optimization of Pointer Address Space Inference (*PASI*). Deep learning accelerators are characterized by multiple memory hierarchies for operands. PASI determines the memory hierarchy for each operand through a data flow analysis based on a recursive traversal syntax tree. According to the attributes of memory of operands, the optimal ld/st instructions can be conducted for the requirement of low memory latency. As shown in Fig.11, the pointer p is an M_{I} pointer when p points to an M_{I} space in line 7, and p is a DDR pointer when p points to the DDR space in line 10. After the optimization of PASI, a general instruction *ld.lddr.f16* will be generated in line 12. The faster instruction *ld.icache.f16* will be generated in line 9.

Data Layout Optimization (*DLO*). Deep learning accelerators also are characterized by vector instructions for operations. However, the operands of vector instructions must be stored in on-chip, and are restricted in terms of data precision, data storage, and data alignment based on the hardware characteristics. As shown in Fig.18, according to the specific computation pattern of the operator, the precision conversions from float 32 to float 16 are performed for weight parameters. Besides, the weight parameters are transposed from the row-first storage to the column-first storage, and they are tiled and placed according to the distri-

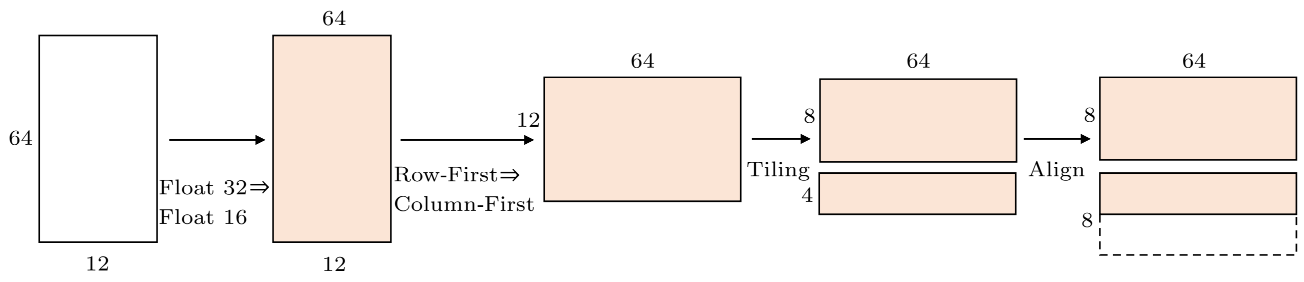


Fig.18. Weight parameters with data layout optimization.

bution characteristics of $M_{\mathtt{W}}$. Furthermore, in keeping with the calculation characteristics of the hardware functional components, it is necessary to align the operand size of operators by padding. Finally, the instruction *conv.icache.f16* will be performed to complete the convolution operations after data layout optimization. The experimental results show the data layout optimization can achieve a speedup of 77.634x (refer to Subsection 5.1).

Low-Level Tuning. Based on the computation pattern of a specific algorithm, the developers can perform low-level optimizations with FlexPDA. For example, an elaborately designed double buffer can cover the time cost of data transmission by the time cost of computation. Besides, we can lessen the pressure on registers by reducing the use of local variables. Moreover, more optimization opportunities, such as parallel computation and vectorizing scalar computation, can be found in FlexPDA. The program can benefit from these low-level optimizations.

6 Discussion

Dynamic Memory Allocation. Some DL applications do not naturally hold a fixed length data structure, such as image captioning, in which the outputs are sentences consisting of different numbers of words. It is not suitable for the DLCollect structure to deal with these DL applications. The dynamic allocation of memory is selected when we do not know how much amount of memory would be needed for the program beforehand. Allocating memory dynamically is flexible for programming and efficient for resource utilization. Saini and Simon[25] eliminated the undesirable effects of paging-in empty data arrays from the service nodes to the compute nodes by the dynamic allocation of memory. Udayakumaran and Barua[26] presented a highly predictable, low-overhead, and dynamic memory allocation strategy for embedded systems with scratch-pad memory. We implement static memory allocation for variables. The dynamic allocation of memory will be supported in the future to allow DL applications such as picture captioning.

Compatibility of Optimizations. We mainly focus on the optimization of program parallelism and memory alias analysis, which are compatible with other program optimizations. Compilation optimizations based on the polyhedron model[27] are an effective method to solve the automatic parallel of programs on multi-core architectures. The polyhedron model can deal with loop transformation such as loop tiling and distribution, which greatly improves the parallelism of the program. In addition, the dependency analysis in the polyhedron model provides the basis for the analysis and optimization of other parallel models. For example, Pellegrini *et al.*[28] used the analysis of the dependence of the polyhedron model to optimize the communication of MPI programs. More importantly, loop tiling and data compression in the polyhedron model play a key role in the research of data locality. Besides, the optimizations such as instruction scheduling, redundant expression deletion, and so on are inseparable from memory alias analysis. Wu *et al.*[29] performed the memory-space alias analysis for GPGPU. It is exciting to improve the performance of FlexPDA by combining these optimizations into FlexPDA; which is our future work.

New Operations. The basic operations of DNNs such as convolution and fully-connected operations are implemented in the form of interface in FlexPDA. Some new operations cannot be directly allowed on DL accelerators currently such as Top-K. We can achieve these operations through basic operations, which greatly reduce the burden of programming for users. For example, the implementation of Top-K is as follows: 1) the *flexpda::max* interface is called to find the maximum value of n-dimension vector *src* and the index of maximum value in *src*; 2) the found maximum value is kept in the result vector *dst*; 3) the value of according position in *src* is set to an infinitely small value; 4) the previous three steps are repeated until the first K maximum value *dst* is derived. It is effortless for users to achieve Top-K by the *flexpda::max* interface. Further work includes providing more programming interfaces for deep learning algorithms.

Application to Other Accelerators. The architectures of DianNao[17], DaDianNao[15], ShiDianNao[30], and Cambricon-X[31] all belong to the DianNao family and are based on the same design concept as shown in Fig.2. For the memory, the on-chip memory is divided into NBin, NBout, and SB according to functions to store input data, output data, and weights respectively. For the computation module, point multipliers are used to implement operations such as convolution and matrix multiplication in deep learning. The computation unit NFU is divided into three stages: NFU-1 for multiplication, NFU-2 for accumulation, and NFU-3 for activation. FlexPDA presents an easy-to-use domain-specific language and the corresponding backend code generator based on the abstraction of the architecture of the DianNao family. The deep learning accelerators are characterized by multiple memory hierarchies for operands; therefore we perform the optimization of pointer address space inference to generate the faster memory access instructions. In addition, the deep learning accelerators also are characterized by vector instructions for operations; thus the data layout optimization is conducted to fully utilize the resource of computation. In summary, FlexPDA provides a flexible programming framework for users and can automatically generate the high-performance machine code for different deep learning algorithms, which has good adaptability to be easily migrated to other accelerators including DianNao, ShiDianNao, and Cambricon-X.

7 Related Work

7.1 Abstractions of Architectures

The abstractions of architectures have proved to be valuable for increasing portability and simplifying the development of applications by hiding the hardware intricacies. Each abstraction has different focuses due to different goals. Fahmy and Holt [32, 33] modeled the architecture as a graph. Graph rewriting is used to transform the architectures in a variety of situations. Moriconi *et al.* [34] modeled the architecture as mathematical theories using predicates. Chen *et al.* [35] proposed a general model which consists of series of function units and interconnected data transfer paths for neural network accelerators. Mishra *et al.* [36] proposed a functional abstraction based on the design space exploration methodology which is capable of capturing a wide variety of programmable architectures. Peterson and Athanas [37] introduced resource pools as an abstraction of general computing devices which provides a homogeneous description of FPGAs, ASICs, CPUs, or even an entire network of workstations. Handziski *et al.* [38] provided a powerful set of abstractions that enable timing, alarms, communication, sampling, storage, and low power operation across different hardware platforms. Our abstractions of architectures are based on DL accelerators similar to DaDianNao [15], which guide the design and optimization of FlexPDA.

7.2 Neural Network Programming

Du *et al.* proposed ZhuQue [39], a neural network programming model based on the labeled data layout for Cambricon-X [31] hardware platform. Song *et al.* [40] proposed a novel programming style called stage level parallel (SLP) with layer fusion optimization and intralayer pipelining optimization for neural network algorithms on DianNao [17], which takes advantage of the parallel execution of instructions on different types of on-chip resources. Chen *et al.* [41] proposed TVM, an end-to-end compiler stack, which can deploy deep learning workloads across diversiform hardware backends. TVM automatically generates optimized codes of diversiform hardware backends for the models trained by different front-end deep learning frameworks. FlexPDA is a programming model similar to CUDA, which helps users implement the high-performance code on deep learning accelerators. FlexPDA can be connected to TVM as a back-end. The users can combine the optimization techniques of TVM to generate high-performance FlexPDA code for accelerators. Truong *et al.* [42] presented Latte, which contains a domain-specific language that provides a high-level abstraction for describing new layers, and a compiler with general optimization for deep learning networks on CPU. Vasilache *et al.* [43] presented a domain-specific language, named Tensor Comprehensions (TC), and an end-to-end compilation flow for engines of computation graph on GPU, which can generate highly-optimized kernels for tensor expressions. RainBuilder⑦ is an end-to-end toolchain for CAISA architecture, which provides the rapid deployment of DL algorithms on FPGA-based accelerators and supports most DL frameworks such as Caffe, Tensorflow. Mind Studio⑧ is a full-stack development toolchain based on the IntelliJ framework for Huawei Ascend DL processors, which provides development, debugging, tuning of operators, and porting, optimization, analysis of networks for users. BANG C⑨ is a low-level programming language for MLU (machine learning unit) hardware. In this paper, we proposed FlexPDA, a domain-specific language for intelligence accelerators similar to DaDianNao. FlexPDA gives abstractions of the applications in DL fields and abstractions of architectures. The abstractions of applications, which are used to acquire the characteristics of data and operations of DL algorithms, guide the design of the data types and interfaces, thereby helping users program applications flexibly and efficiently. The architectures are abstracted from the parallel structure and the storage model of accelerators. The abstractions

⑦http://www.corerain.com/RainBuilder/en, Sept. 2021.

⑧https://support.huaweicloud.com/usermanual-mindstudioc32/atlasonh_02_c32_0004.html, Sept. 2021.

⑨http://www.cambricon.com/docs/bangc/developer_guide_html, Sept. 2021.

of architectures bring the opportunity of the optimization of data layout and the flexible parallel mechanism of tasks, thereby leading the backend code generator to generate high-performance machine code.

7.3 Optimizations for DL Accelerators

The optimization approaches such as layer fusion and data reuse are commonly used in the inference of DNNs on intelligent accelerators. Song *et al.*[41] introduced a series of layer-based compile optimizations for DL accelerators. Du *et al.*[39] added the optimization of data layout to the neural network development kit (NDK) for neural network accelerators. Kim *et al.*[44] proposed an automated optimization framework including a flexible buffer structure, effective dataflow for fusing operations, and programmable data-access control for DL accelerators. In addition, Li *et al.*[45] presented an optimization and inference engine, namely XDN, for accelerating deep neural networks on MLUs. Liu *et al.*[46] proposed an auto-tuning algorithm to jointly optimize the model parallelism and layer fusion scheme on MLUs for a given DNN model. Zhao and Di[47] designed a novel composition of tiling and fusion in the polyhedral optimizers to maximize the utilization of the memory hierarchy on DL accelerators like Huawei Ascend 910. Zheng *et al.*[48] leveraged the polyhedral model to eliminate unnecessary data movements in the workload and maximize the utilization of on-chip memory by maintaining data locality in the scratchpad for DL accelerators such as AWS Inferentia. FlexPDA presents an optimization of the data layout to use the powerful compute units and limited on-chip memory of DL accelerators like DaDianNao[15].

8 Conclusions

In this paper, a flexible and efficient programming framework on DL accelerators, FlexPDA, was proposed. FlexPDA utilizes pointer address space inference, data layout optimization and low-level tuning to accelerate the performance of deep learning algorithms. We evaluated FlexPDA by using 10 representative operators selected from deep learning algorithms and an end-to-end network. The experimental results validated the effectiveness of FlexPDA, which achieves an end-to-end performance improvement of 1.620x over the low-level library. In future work, it is exciting to study more compilation optimization techniques, such as polyhedron models.

References

[1] Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition. arXiv:1409.1556, 2014. http://arxiv.org/abs/1409.1556, Sept. 2021.

[2] He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. In *Proc. the 2016 IEEE Conference on Computer Vision and Pattern Recognition*, June 2016, pp.770-778. DOI: 10.1109/CVPR.2016.90.

[3] LiKamWa R, Hou Y, Gao J, Polansky M, Zhong L. RedEye: Analog convnet image sensor architecture for continuous mobile vision. *ACM SIGARCH Comput. Archit. News*, 2016, 44(3): 255-266. DOI: 10.1145/3007787.3001164.

[4] Qian Y, Woodland P C. Very deep convolutional neural networks for robust speech recognition. In *Proc. the 2016 IEEE Spoken Language Technology Workshop*, Dec. 2016, pp.481-488. DOI: 10.1109/SLT.2016.7846307.

[5] Abdel-Hamid O, Mohamed A, Jiang H, Deng L, Penn G, Yu D. Convolutional neural networks for speech recognition. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 2014, 22(10): 1533-1545. DOI: 10.1109/TASLP.2014.2339736.

[6] Eriguchi A, Hashimoto K, Tsuruoka Y. Tree-to-sequence attentional neural machine translation. arXiv:1603.06075, 2016. http://arxiv.org/abs/1409.1556, Sept. 2021.

[7] Deng L, He X, Gao J. Deep stacking networks for information retrieval. In *Proc. the 2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, May 2013, pp.3153-3157. DOI: 10.1109/ICASSP.2013.6638239.

[8] Chen X, Ma H, Wan J, Li B, Xia T. Multi-view 3D object detection network for autonomous driving. In *Proc. the 2017 IEEE Conference on Computer Vision and Pattern Recognition*, July 2017, pp.1907-1915. DOI: 10.1109/CVPR.2017.691.

[9] Maqueda A I, Loquercio A, Gallego G, García N, Scaramuzza D. Event-based vision meets deep learning on steering prediction for self-driving cars. In *Proc. the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, June 2018, pp.5419-5427. DOI: 10.1109/CVPR.2018.00568.

[10] Cireşan D C, Giusti A, Gambardella L M, Schmidhuber J. Mitosis detection in breast cancer histology images with deep neural networks. In *Proc. the International Conference on Medical Image Computing and Computer-Assisted Intervention*, Sept. 2013, pp.411-418. DOI: 10.1007/978-3-642-40763-5_51.

[11] Ma M, Shi Y, Li W, Gao Y, Xu J. A novel two-stage deep method for mitosis detection in breast cancer histology images. In *Proc. the 24th International Conference on Pattern Recognition*, Aug. 2018, pp.3892-3897. DOI: 10.1109/ICPR.2018.8546192.

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

[13] Jia Y, Shelhamer E, Donahue J, Karayev S, Long J, Girshick R, Guadarrama S, Darrell T. Caffe: Convolutional architecture for fast feature embedding. In *Proc. the 22nd ACM International Conference on Multimedia*, Nov. 2014, pp.675-678. DOI: 10.1145/2647868.2654889.

[14] Al-Rfou R, Alain G, Almahairi A *et al.* Theano: A Python framework for fast computation of mathematical expressions. arXiv:1605.02688, 2016. https://arxiv.org/abs/1605.02688, Sept. 2021.

[15] Chen Y, Luo T, Liu S *et al.* DaDianNao: A machine-learning supercomputer. In *Proc. the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2014, pp.609-622. DOI: 10.1109/MICRO.2014.58.

[16] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. the International Symposium on Code Generation and Optimization*, Mar. 2004, pp.75-86. DOI: 10.1109/CGO.2004.1281665.

[17] Chen T, Du Z, Sun N, Wang J, Wu C, Chen Y, Temam O. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *SIGARCH Comput. Archit. News*, 2014, 42(1): 269-284. DOI: 10.1145/2654822.2541967.

[18] Fatahalian K, Knight T J, Houston M *et al.* Sequoia: Programming the memory hierarchy. In *Proc. the 2006 ACM/IEEE Conference on Supercomputing*, Nov. 2006, Article No. 4. DOI: 10.1109/SC.2006.55.

[19] Lan H Y, Wu L Y, Zhang X, Tao J H, Chen X Y, Wang B R, Wang Y Q, Guo Q, Chen Y J. DLPlib: A library for deep learning processor. *Journal of Computer Science and Technology*, 2017, 32(2): 286-296. DOI: 10.1007/s11390-017-1722-2.

[20] Zhang X, Zhou X, Lin M, Sun J. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proc. the IEEE Conference on Computer Vision and Pattern Recognition*, June 2018, pp.6848-6856.

[21] Li J, Jiang Z, Liu F, Dong X, Li G, Wang X, Cao W, Liu L, Wang Y, Li T, Feng X. Characterizing the I/O pipeline in the deployment of CNNs on commercial accelerators. In *Proc. the 2020 IEEE Int. Conf. Parallel Distributed Processing with Applications, Big Data Cloud Computing, Sustainable Computing Communications, Social Computing Networking*, Dec. 2020, pp.137-144. DOI: 10.1109/ISPA-BDCloud-SocialCom-SustainCom51426.2020.00043.

[22] Thomas D, Moorby P. The Verilog® Hardware Description Language. Springer Science & Business Media, 2008.

[23] Krizhevsky A, Sutskever I, Hinton G E. ImageNet classification with deep convolutional neural networks. *Commun. ACM*, 2017, 60(6): 84-90. DOI: 10.1145/3065386.

[24] Dagum L, Menon R. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 1998, 5(1): 46-55. DOI: 10.1109/99.660313.

[25] Saini S, Simon H. Enhancing applications performance on Intel Paragon through dynamic memory allocation. In *Proc. the Scalable Parallel Libraries Conference*, Oct. 1993, pp.232-239. DOI: 10.1109/SPLC.1993.365561.

[26] Udayakumaran S, Barua R. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proc. the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, Oct. 2003, pp.276-286. DOI: 10.1145/951710.951747.

[27] Feautrier P, Lengauer C. Polyhedron model. In *Encyclopedia of Parallel Computing*, Padua D (ed.), Springer, 2011. DOI: 10.1007/978-0-387-09766-4_502.

[28] Pellegrini S, Hoefler T, Fahringer T. Exact dependence analysis for increased communication overlap. In *Proc. the European MPI Users' Group Meeting*, Sept. 2012, pp.89-99. DOI: 10.1007/978-3-642-33518-1_14.

[29] Wu J, Belevich A, Bendersky E, Heffernan M, Leary C, Pienaar J, Roune B, Springer R, Weng X, Hundt R. GPUCC: An open-source GPGPU compiler. In *Proc. the 2016 International Symposium on Code Generation and Optimization*, March 2016, pp.105-116.

[30] Du Z, Fasthuber R, Chen T, Ienne P, Li L, Luo T, Feng X, Chen Y, Temam O. ShiDianNao: Shifting vision processing closer to the sensor. In *Proc. the 42nd Annual International Symposium on Computer Architecture*, June 2015, pp.92-104.

[31] Zhang S, Du Z, Zhang L, Lan H, Liu S, Li L, Guo Q, Chen T, Chen Y. Cambricon-X: An accelerator for sparse neural networks. In *Proc. the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2016, Article No. 20. DOI: 10.1109/MICRO.2016.7783723.

[32] Fahmy H, Holt R C. Software architecture transformations. In *Proc. the 2000 International Conference on Software Maintenance*, Oct. 2000, pp.88-96. DOI: 10.1109/ICSM.2000.883020.

[33] Fahmy H, Holt R C. Using graph rewriting to specify software architectural transformations. In *Proc. the 15th IEEE International Conference on Automated Software Engineering*, Sept. 2000, pp.187-196. DOI: 10.1109/ASE.2000.873663.

[34] Moriconi M, Qian X, Riemenschneider R A. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 1995, 21(4): 356-372. DOI: 10.1109/32.385972.

[35] Chen X, Peng S, Jin L, Zhuang Y, Song J, Du W, Liu S, Zhi T. Partition and scheduling algorithms for neural network accelerators. In *Proc. the 13th International Symposium on Advanced Parallel Processing Technologies*, Aug. 2019, pp.55-67. DOI: 10.1007/978-3-030-29611-7_5.

[36] Mishra P, Dutt N, Nicolau A. Functional abstraction driven design space exploration of heterogeneous programmable architectures. In *Proc. the 14th International Symposium on Systems Synthesis*, September 30-October 3, 2001, pp.256-261. DOI: 10.1145/500001.500061.

[37] Peterson J B, Athanas P M. Resource pools: An abstraction for configurable computing codesign. *Proceedings of the SPIE*, 1996, 2914: 218-224. DOI: 10.1117/12.255819.

[38] Handziski V, Polastre J, Hauer J H, Sharp C, Wolisz A, Culler D. Flexible hardware abstraction for wireless sensor networks. In *Proc. the 2nd European Workshop on Wireless Sensor Networks*, Feb. 2005, pp.145-157. DOI: 10.1109/EWSN.2005.1462006.

[39] Du W, Wu L, Chen X, Zhuang Y, Zhi T. ZhuQue: A neural network programming model based on labeled data layout. In *Proc. the 13th International Symposium on Advanced Parallel Processing Technologies*, Aug. 2019, pp.27-39. DOI: 10.1007/978-3-030-29611-7_3.

[40] Song J, Zhuang Y, Chen X, Zhi T, Liu S. Compiling optimization for neural network accelerators. In *Proc. the 13th International Symposium on Advanced Parallel Processing Technologies*, August 2019, pp.15-26. DOI: 10.1007/978-3-030-29611-7_2.

[41] Chen T, Moreau T, Jiang Z *et al.* TVM: An automated end-to-end optimizing compiler for deep learning. In *Proc. the 13th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2018, pp.578-594.

[42] Truong L, Barik R, Totoni E, Liu H, Markley C, Fox A, Shpeisman T. Latte: A language, compiler, and runtime for elegant and efficient deep neural networks. *ACM SIGPLAN Notice*, 2016, 51(6): 209-223. DOI: 10.1145/2908080.2908105.

[43] Vasilache N, Zinenko O, Theodoridis T *et al.* Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. arXiv:1802.04730, 2018. https://arxiv.org/abs/1802.04730, Sept. 2021.

[44] Kim H, Lyuh C G, Kwon Y. Automated optimization for memory-efficient high-performance deep neural network accelerators. *ETRI Journal*, 2020, 42(4): 505-517. DOI: 10.4218/etrij.2020-0125.

[45] Li G, Wang X, Ma X, Liu L, Feng X. XDN: Towards efficient inference of residual neural networks on Cambricon chips. In *Proc. the 2nd Bench Council International Symposium on Benchmarking, Measuring and Optimization*, Nov. 2019, pp.51-56. DOI: 10.1007/978-3-030-49556-5_4.

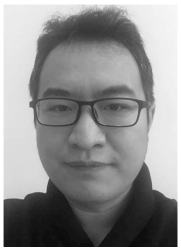
[46] Liu Z, Leng J, Chen Q, Li C, Zheng W, Li L, Guo M. DLFusion: An auto-tuning compiler for layer fusion on deep neural network accelerator. arXiv:2011.05630, 2020. https://arxiv.org/abs/2011.05630, Sept. 2021.

[47] Zhao J, Di P. Optimizing the memory hierarchy by compositing automatic transformations on computations and data. In *Proc. the 53rd Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2020, pp.427-441. DOI: 10.1109/MICRO50266.2020.00044.

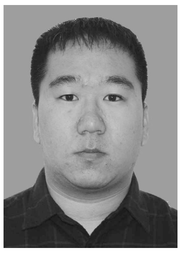
[48] Zheng H, Oh S, Wang H, Briggs P, Gai J, Jain A, Liu Y, Heaton R, Huang R, Wang Y. Optimizing memory-access patterns for deep learning accelerators. arXiv:2002.12798, 2020. https://arxiv.org/abs/2002.12798, Sept. 2021.

Lei Liu is a doctoral supervisor in College of Computer Science and Technology, Jilin University, Changchun. He received his M.S. degree in computer science from Jilin University, Changchun, in 1985. The central themes of his research are programming language and its realization technology, software security and cloud computing, the semantic web and ontology engineering, knowledge representation and reasoning, etc.

Xiu Ma received her B.S. degree in network and information security from the College of Computer Science and Technology, Jilin University, Changchun, in 2016. She is currently a Ph.D. student in computer software and theory of College of Computer Science and Technology, Jilin University, Changchun. Her research interests include programming systems and computational intelligence. She is a student member of IEEE.



Hua-Xiao Liu is an assistant professor in College of Computer Science and Technology, Jilin University, Changchun. He received his Ph.D. degree in computer science from Jilin University, Changchun, in 2013. The central theme of his research is improving software quality, and his recent research concerns the software requirements engineering, software cybernetics and formal methods of software development. More specifically, he develops techniques to verify aspect-oriented requirements model based on ontology.



Guang-Li Li is a Ph.D. student at State Key Laboratory of Computer Architecture, Institute of Computing Technology of the Chinese Academy of Sciences, Beijing. His research interests include programming systems and machine learning. He has authored/co-authored 26 publications in these areas. He is a student member of CCF, ACM, IEEE and CAAI.

Lei Liu received his B.S. degree in computer science from Changchun University of Science and Technology, Changchun, in 2001, his M.S. degree in computer science from Jilin University, Changchun, in 2004, and his Ph.D. degree in computer architecture from Institute of Computing Technology (ICT) , Chinese Academy of Sciences (CAS), Beijing, in 2010. He participated in the Advanced Compiler Technology Laboratory (ACT) of ICT, CAS, in 2010, and is now an assistant professor of ICT, CAS, Beijing. His research interests include programming language and compiler optimization.