CCF Transactions on High Performance Computing (2020) 2:332–347
https://doi.org/10.1007/s42514-020-00044-7

REGULAR PAPER


# Survey and design of paleozoic: a high-performance compiler tool chain for deep learning inference accelerator

Zihan Liu[1] · Jingwen Leng[1] · Guandong Lu[1] · Chenhui Wang[1] · Quan Chen[1] · Minyi Guo[1]

Received: 20 March 2020 / Accepted: 17 July 2020 / Published online: 6 October 2020


**Abstract**
Specialized hardware accelerators for deep learning are widely introduced by many hardware vendors because of their high performance and efficiency. However, different vendors adopt different accelerator architectures, making it challenging for the compiler tool-chain to generate and optimize high-performance codes. Moreover, the current tool-chains provided by the vendors are either highly abstract, which makes it hard to optimize or contain too many hardware-related details, which makes it inconvenient to program. So, in this paper, we propose a middle layer compiler tool-chain for Cambricon MLU-100 to fill the gap between high-level runtime library and low operator-level SDK. Our tool-chain is based on the operator level SDK but abstracts away its redundant initialization and allocation statement. We also expose the interface of major optimization knobs compared to the existing runtime, thus enabling a considerable optimization space. We evaluate our work by several state-of-the-art neural networks and choose the line of code and optimization knobs as evaluation metrics. We also compare the performance against state-of-the-art tool-chain TensorRT applying simple optimization strategy and find that our work has great potential in optimization. Our work can guarantee the user a vast optimization space with only around 20% amount of the codes that hides the redundant initialization and allocation statements from users.

**Keywords** Deep learning accelerator · Compiler tool-chain · Hardware-related optimization

## 1 Introduction

### 1.1 Deep learning accelerator

With the evolution of computing power, computation intense deep learning has been increasingly applied in the key application domains, including computer vision, natural language processing, etc. Nowadays, conventional general-purpose processors like CPU/GPU can hardly meet the growing need in computation power. On the other hand, the computation patterns in deep learning are good candidates for hardware specialization. There exist a few kinds of patterns in a deep neural network, including convolution, pooling, activation, batch normalization, and fully connected layers. These calculations are mostly based on linear calculation, with conjunctions of linear transformations, matrix decomposition, etc. The general-purpose CPUs that adopt deep and complex pipelines are highly inefficient in this scenario. Since the linear calculation deals with a huge amount of data, the optimal memory hierarchy is also different. So, increasing vendors are releasing their own specialized accelerators (Jouppi et al. 2017; Zhang et al. 2016; Marchisio et al. 2019), and these accelerators have superior performance and energy efficiency in deep learning tasks, these specialized accelerators also have simpler and more diverse architectures than the general-purpose processor, as well as different memory subsystems. Meanwhile, in addition to these specialized-designed accelerators, increasing researchers focus on accelerator architecture with better universality for

✉ Jingwen Leng
leng-jw@cs.sjtu.edu.cn

✉ Minyi Guo
guo-my@cs.sjtu.edu.cn

Zihan Liu
altair.liu@sjtu.edu.cn

Guandong Lu
lugu0525@sjtu.edu.cn

Chenhui Wang
wang-chen-hui@sjtu.edu.cn

Quan Chen
chen-quan@cs.sjtu.edu.cn

[1] Shanghai Jiao Tong University, Shanghai 200240, China

scalar, vector, matrix and tensor computation instead of only focusing on convolution (Guo et al. 2020), which also bring challenges to compiler design.

### 1.2 Compiler tool-chain

The significant difference in the architecture and memory hierarchy leads to great challenges in code scheduling and generation, instruction selection, memory accessing on the specialized hardware. Currently, most of the compiler tool-chains are designed for CPU/GPU, causing difficulties in achieving the maximal efficiency of the specialized hardware. Though researchers also try to extend the existing tool-chain for better task scheduling on heterogeneous architecture, including Laius (Zhang et al. 2019) and Ebird (Cui et al. 2019), the overhead of these methods is higher than compiler level optimization. On the other hand, the existing neural network accelerators are based on different architectures, ranging from systolic array (Quinton 1994) to dot-product unit (Chen et al. 2014, 2014; Liu et al. 2015), bringing significant challenges to the compiler tool-chain design. Currently, the hardware vendors provide their own compiler tool-chains, including TensorRT (NVIDIA Corp 2020) of NVIDIA Corp., Cambricon Neuware (Cambricon Technologies 2019b) from Cambricon Technologies, etc. However, how to deal with diverse hardware architectures remains an open question. Moreover, when it comes to the optimization of deep neural networks, very few vendors give the source code and algorithms to the users, making the optimization process a black box. Given that some vendors may provide some low-level SDK of their hardware, these tools are highly hardware-related, making it hard for users with little hardware knowledge.

### 1.3 Optimization space

The challenges in compiler tool-chain also reflect on the optimization of neural network tasks. There are two stages in a deep neural network application: training and inference, where the optimization methodology is different too. In this research, we mainly focus on the inference process. In the inference process, we mainly want the higher throughput (FPS) and lower power consumption. To satisfy the demand in throughput, researchers propose several optimization knobs, including sparsity, precision-accuracy trade-off, operation fusion/concatenation, etc. We mainly introduce the mentioned factors.

For sparsity, from the introduction of AlexNet (Krizhevsky et al. 2012), the networks apply the DropOut technology to eliminate in-active neurons in a network to simplify the structure of the network. This technology is aiming to relieve the accuracy drop caused by over-fitting, and sparsity is the side-product of this technology. The sparsity of the network significantly reduces the demands in data transferring and computation power. Currently, some sparse version state-of-the-art neural networks cost only 10%–20% computation count comparing to the original version. The sparsity mentioned previously is called static sparsity, and most of them are sparsity of weight/bias matrix, which can be determined before the network running. Another sparsity is called dynamic sparsity (Zhou et al. 2018), which is introduced by the wide application of `ReLU` activation function, which generates many zeros when the network is running, and the position of 0 depends on every specific input, which can not be determined in advance. The sparsity of the network brings the problem of irregularity, which means the accelerators have difficulty in determining the exact position of 0 to skip. If not well processed, to find the 0 and skip may even introduce extra overhead results in an overall performance drop. For compiler tool-chain, how to deal with the irregularity brought by sparsity has yet to be solved.

For accuracy, in the training process, there exist many no-linear calculations related to the gradient. Since the gradient may be very small or big when training the network, low precision may result in a great decrease in classification accuracy. However, in the inference process, since all the weight/bias are fixed, accuracy drop brought by proper precision cutting may be acceptable for only 1% or 2%. Actually, there are several methods to decrease the precision, including simply casting FP32 to FP16, quantizing to INT8/INT4 (Dong et al. 2019), etc. The problem is how to determine the proper precision that will not result in unacceptable accuracy drop, which is a trade-off between precision and final classification accuracy.

The operator fusion and concatenation are widely used in the current compiler tool-chain provided by various vendors (NVIDIA Corp 2020; Cambricon Technologies 2019b). Given 2 layers with data dependency, fusion means omitting the intermediate output of prior layers, thus reducing the data movement of intermediate results (Wang et al. 2010; Filipovic 2015). However, the absence of intermediate results inevitably introduces redundant calculation. For example, there is an overlapping area in 2D sliding window convolution that will be computed multiple times (Ragan-Kelley et al. 2013; Alwani et al. 2016). So, operator fusion is also a trade-off between reduced memory access and redundant computation. On the other hand, operator concatenation is the optimization between 2 layers/kernel without data dependency. By concatenation, the overhead of kernel launch can be reduced considerably, and the computational intensity can be increased too.

### 1.4 Contributions

In this work, we perform a survey on current specialized deep learning accelerators and their compiler tool-chain. We

find that currently, it is difficult for users to customize their inference sessions to achieve better performance on target platforms. The vendors either abstract all the details making it unavailable for users to research or expose too many redundant operations, making it inconvenient to code and optimize. So, choosing Cambricon MLU-100 as our target platform, we introduce another abstract layer that hides the redundant operation like initialization, memory allocation, but exposes the optimization interface. Out abstract layer exposes a huge optimization space with only 20% amount of code, and this abstract layer has a tiny influence on the performance compared to the original tool-chain.

## 2 How to program deep learning applications

In this section, we introduce current frameworks for deploying a deep learning service on various platforms. It should be noted that what the programmers do in deep-learning training and deep-learning inference is quite different, and we mainly focus on deep-learning inference.

### 2.1 Training vs inference

Training in supervised machine learning means adjusting the human-constructed mathematical model by feeding the pair of input and correct output data, which enable the model to match the correct mathematical distribution as close as possible. In detail, the initialized model calculates the output of current input, compare the output with correct output, and adjust the model with mathematical methods. Inference in machine learning means calculating the output of an unknown input given the trained mathematical model. Obviously, the training process includes the inference process, and there is a significant difference in computational characteristics. Most of the computation in the inference process is the linear calculation, but a huge amount of non-linear calculation related to gradient exists in the training process.

Moreover, the programming model of training and inference is quite different. When programming the training session, programmers mainly care about how to construct a model with higher accuracy, which means they may adjust the structure of the model repeatedly, introduce the new type of computation, change the hyper-parameter setting, etc. They care less about the model optimization, execution on ending hardware, and actual deployment. They can choose whatever system with strong computational power and train the network. So, they need the programming model with high-level hardware abstract, but convenient to invoke the operator and adjust the parameter settings. If possible, a higher flexibility for newly introduced operators is preferred. However, when programming the inference session, the structures of the model are fixed, programmers are mostly not allowed to adjust the model, and generally, the target platform is fixed in advance. So, the programmers should care about the actual executing time, power consumption, throughput, stability, etc., on actual hardware, which means the optimization is highly hardware-related, as shown in Fig. 1. This means they need the programming model to give enough hardware specifications, characteristics, in other words, enough optimization space so that they can optimize the trained model for specific hardware and reach the peak speed, stability on the target platform.

In this paper, we mainly focus on the technologies in the inference stage.

### 2.2 Problem of network format

Currently, there are many kinds of deep learning frameworks for training and inferencing, including TensorFlow, PyTorch, MXNet, Caffe (Jain et al. 2019; MXNet 2020; Jia et al. 2014), etc. For training, researchers choose the frameworks to fit their coding habit, and different frameworks produce network files in different formats, as listed in Table 1.

However, things get complicated in the inference (deploying) stage. For example, the frameworks based on Python like TensorFlow, PyTorch are quite convenient for coding but inferior in executing speed. The frameworks based on C++ like Caffe have great executing performance but hard to configure, coding. This introduces a great gap between academia and industry. Typically, the company should reconstruct a state-of-the-art network originally written in Python to C++, and this is a time costing process. The Open Neural Network eXchange (ONNX) format addresses this problem. Currently, the providers of the frameworks integrate conversion APIs for the users, supporting them to convert network specification files into ONNX file, making it easier for deployment. Currently, various frameworks, including PyTorch, TensorFlow, Caffe, MXNet, CNTK, Chainer, PaddlePaddle, support converting between ONNX format (ONNX 2020). So, in this research, we directly use ONNX as our standard, bypassing the problem of the diversity of frameworks.

### 2.3 Programming interface in inference

As mentioned previously, the inference is strongly related to actual hardware. So, the hardware vendors provide their own interface for users. For example, Google provided a series of commands for users to access their cloud TPU, supporting users to upload the models and get the output of TPU, as shown in Fig. 2. NVIDIA also provided TensorRT with C++ API for users to load the models and execute the inference session, as shown in Fig. 3. However, these front-ends of existing tool-chain for inference sessions do not support

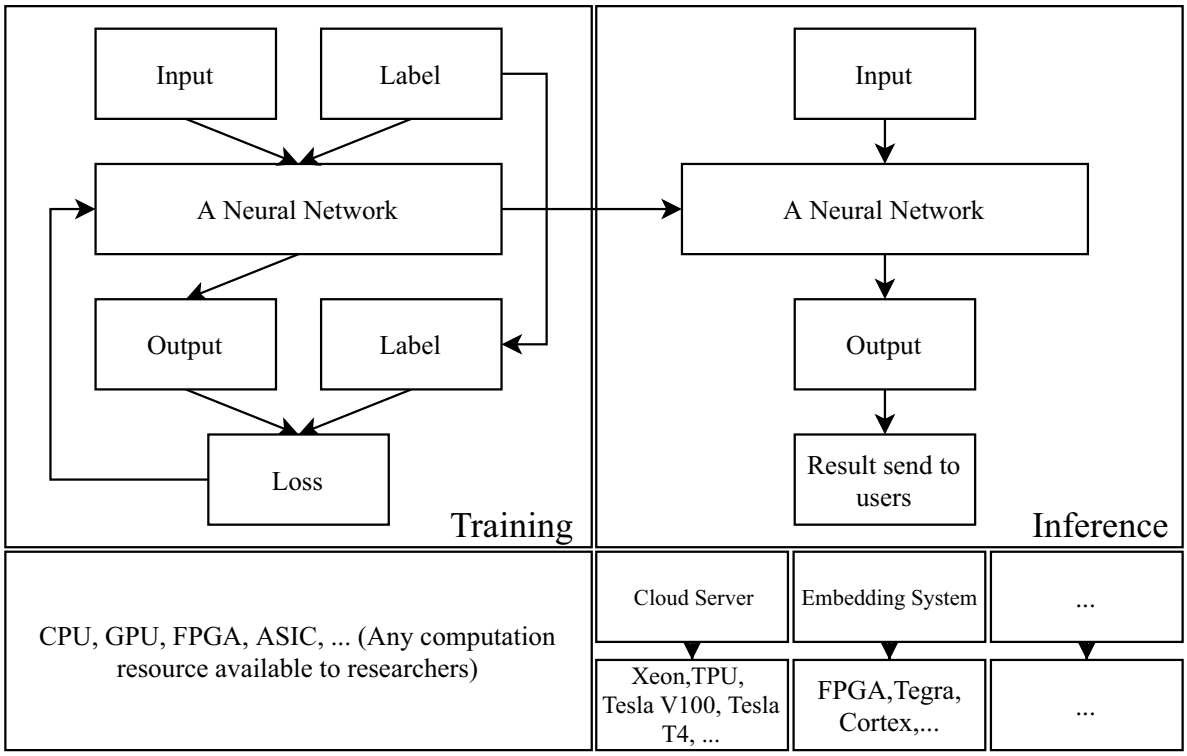



**Fig. 1** Training vs inference (Copeland 2020)

**Table 1** Difference of frameworks

| Framework | Language | File Format |
|---|---|---|
| TensorFlow | Python | .pb |
| PyTorch | Python | .pkl |
| Caffe | C++ | .caffemodel |
| MXNet | Python | .json .params |
| CNTK | Python | .model |

many hardware-related optimization knobs. They simply load a network specification file, parse it and execute it, and users get the output directly from their input, and this is convenient for end-users but challenging for optimization.

## 2.4 Optimizer in inference

Given the fact that end-user can not efficiently optimize the performance, the power consumption of the inference session run on the specific hardware, these hardware vendors provide their own built-in optimizer to leverage the advantage of the hardware. For example, the TensorRT conduct tensor and layer fusion/concatenation, kernel auto-tuning, memory allocation, etc. by their own algorithm (NVIDIA Corp 2020), as shown in Fig. 4. Among these optimizations, there are hardware-related parts, including memory allocation, tensor layout transformation. However, most of

```
# Configurations
> export MODEL_DIR=...
> export DATA_DIR=...
...

# Launch a Cloud TPU with Compute Engine VM
> ctpu up --zone=... --tf-version=... [...]
> gcloud compute ssh ...
> export TPU_NAME=...
...

# Run the model and get the output
> python3 model.py  --tpu=$TPU_NAME \
                    --model_dir=$MODEL_DIR \
                    --data_dir=$DATA_DIR \
                    --distribution_strategy=tpu\
                    --download
                    ...

# Clean
> exit
> ctpu delete --zone=...
...
```

**Fig. 2** Cloud TPU usage

```
// Initialization
auto builder = Session<nvinfer1::IBuilder>(...);
auto network = Session<nvinder1::INetworkDefinition>(...);
auto config  = Session<nvinfer1::IBuilderConfig>(...);
auto parser  = Session<nvonnxparser::IParser>(...);
std::shared_ptr<nvinfer1::ICudaEngine> engine;
engine = builder->buildEngineWithConfig(*network, *config);
...

// Read and Construct the Network, Create the Inference Session
auto model = parser->parseFromFile("ONNX file name", ...);
builder->set...;
config ->set...;
auto context = Session<nvinfer1::IExecutionContext>(.../* Specific Engine and Model */);
BufferManager buffer{engine, ...};

// Run the Session
buffer.copyInputToDevice();
context->execute(buffer.getDeviceBindings().data());
buffer.copyOutputToHost();
```

**Fig. 3** TensorRT usage

the vendors do not expose their optimization algorithms or strategies, a lot of libraries, SDKs with high performance are well-tuned by a lot of human experts manually with considerable human input parameters. The details are not available to end-users. Besides, the vendors conduct deep optimization toward their own products with different algorithms, which brings significant challenges in universality to the tool-chain design.

### 2.5 Cambricon MLU-100 and Software

In this paper, we use Cambricon-MLU100 and corresponding tool-chain, SDK, to study for a better way to program and optimize the inference session on specialized hardware. Cambricon MLU-100 is a dedicated deep learning accelerator designed by Cambricon Technologies, the chip support inference task. The hardware specifications are listed in Table 2.

Along with the hardware, the vendor also provides us with 2 sets of tool-chain. One is a high-level runtime library, and another is the operator-level SDK. The programming interface of the high-level runtime library is similar to TPU/GPU mentioned before, as shown in Fig. 5 (The format of network file in `cnrtLoadModel` should be `.cambricon`, which can be converted from other format like `.pb/.caffemodel/.pkl` by provided script).

**Table 2** The MLU100 hardware specification

| Item | Descriptions |
|---|---|
| Core freq. | 1GHz |
| Float perf. (FP16) | 64 TFLOPS |
| Integer perf. (INT8) | 128 TOPS |
| Memory size | 8 GB |
| Memory bandwidth | 102.4 GB/s |
| Memory bit width | 256-bit |
| Host Interface | PCIe 3.0x16 |
| TDP | 110 W |
| ECC Enabled | Yes |

However, the operator-level SDK supports common operators such as convolution, ReLU, vector scaling, Batch Normalization, etc., and in this research, we will mainly use operator-level to implement and optimize our tool-chain.

## 3 Why our tool-chain necessary?

Although there are already a runtime library and an operator-level SDK to program the Cambricon MLU-100, many problems still exist in this model from actual performance to optimization, making it necessary to develop a tool-chain with acceptable performance, flexibility to optimize and convenience to program.

### 3.1 Performance

Currently, the provided runtime library called Cambricon Neuware RunTime supports the networks generated by various frameworks. By receiving the network specification file in various formats as input, CNRT packs them into .cambricon file, load, and extract the interface function in the model and conduct the inference. However, as the selected standard of our research, network in ONNX format experience a significant performance gap against networks in other formats, as shown in Fig. 6.

The inference performance of network files in ONNX file is lower than the performance of other frameworks,

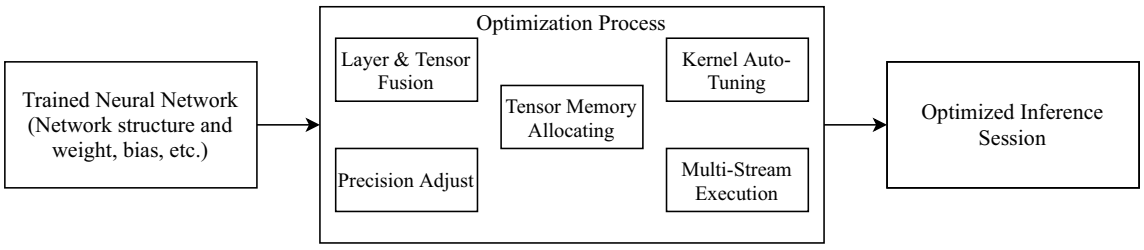



**Fig. 4** TensorRT workflow (NVIDIA Corp 2020)

```
1 // Initialization
2 cnrtInit(0);
3 cnrtModel_t model;
4 cnrtFunction_t function;
5 cnrtDev_t device;
6
7 // Load Model and Extract Function
8 cnrtLoadModel(&model, "Network File Name");
9 cnrtExtractFunction(&function, model, "Function Name");
10
11 // Configure Device, I/O, ...
12 cnrtGetDeviceHandle(&dev, 0);
13 cnrtSetCurrentDevice(dev);
14 ...
15
16 // Allocating and Initailizing Memory
17 cnrtMallocBatchByDescArray(...);
18 cnrtInitFunctionMemory_V2(function, ...);
19 ...
20
21 // Run the Inference Session on Device
22 cnrtMemcpy(..., Host_to_Device);
23 cnrtInvokeFunction(function, ...);
24 cnrtMemcpy(..., Device_to_Host);
```

**Fig. 5** CNRT usage

and since the runtime library abstracts all the detailed implementation, it's hard to analyze the reason for the performance gap and optimize it. As our assumption, since all the networks are packed into `.cambricon` format in advance, the gap between ONNX and other framework results from different translation and scheduling process. With the ability to access lower-level SDK, we should make up this gap.

### 3.2 Programming interface

As mentioned before, the provided runtime library is highly abstracted, leaving almost no optimization space to the user. Though with the provided operator-level SDK, user can control the behavior of the hardware at operator level and adjust the hyper settings include sparse mode, fusion scheme, core to use, etc. when constructing a network, this SDK contains many fragmented codes for configuring the device, allocating the memory, initializing the stream, setting the timer, etc., as shown in Fig. 7 which account for executing a convolution operator. Most of the code account for prerunning work, only line 28-30 account for actual inference. This annoys the programmer much when they want to build a network since many of the work is redundant. The work done from line 1–25 can be abstracted to a more convenient interface. It should be noted that currently, we are not able to program new operators given the existing SDKs.

Under this circumstance, we want to develop a set of toolchain with the proper balance between high-level abstract and low-level hardware specification, giving enough optimization space to the user with the convenience of programming guaranteed.

### 3.3 Optimization

To optimize the neural network inference, programmers mainly devote their effort in reducing the kernel launch, off-chip memory access to increase the FPS, and decrease the power consumption. The provided runtime library and operator-level SDK also provide some optimization knobs. However, they are not very convenient and flexible to use.

The optimization knobs provided by runtime library only contain binary selection between fusion/not fusion, sparse/

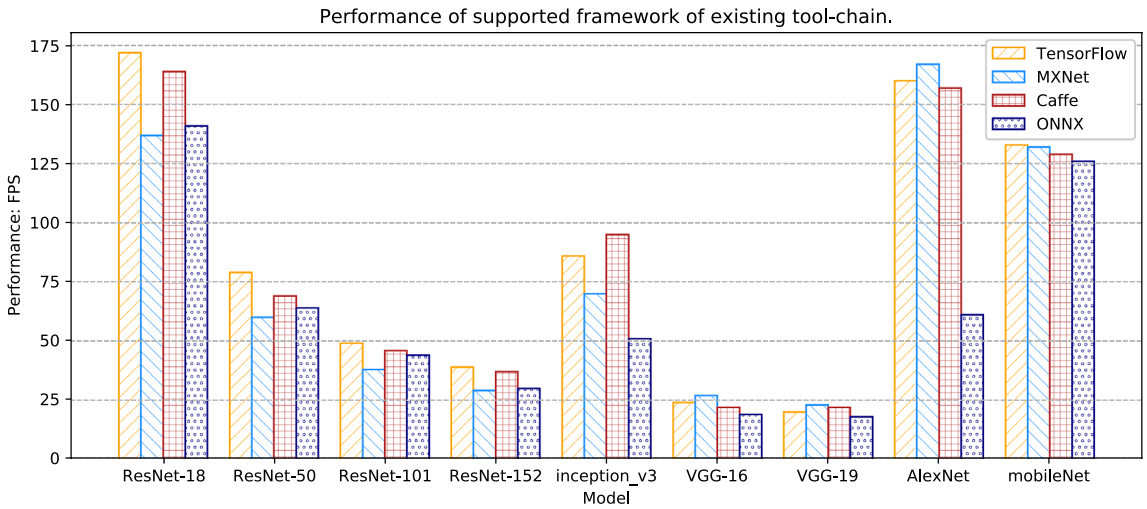



**Fig. 6** Performance of various supported frameworks

```
// Global Configurations
...

// Tensor Allocating
cnmlTensor_t input, output, filter, bias;
cnmlCpuTensor_t filter_cpu, bias_cpu;
cnmlCreateTensor(&input, ...);
cnmlCreateTensor(&output, ...);
cnmlCreateTensor(&filter, ...);
cnmlCreateTensor(&bias, ...);
cnmlCreateCpuTensor(&filter_cpu, ...);
cnmlCreateCpuTensor(&bias_cpu, ...);

// Loading filter/bias Data
cnmlLoadTensorFromFile("filter.params", filter_cpu, ...);
cnmlLoadTensorFromFile("bias.params", bias_cpu, ...);
cnmlBindConstData(filter, filter_cpu, ...);
cnmlBindConstData(bias, bias_cpu, ...);

// Create Convolution Operator
cnmlConvOpParam_t conv_param;
cnmlBaseOp_t conv_op;
cnmlCreateConvOpParam(&conv_param, ... /* Stride, Kernel, ... */);
cnmlCreateConvOp(&conv_op, conv_param, ... /* Input, Output */);
cnmlCompileBaseOp(conv_op, ...);

// Copy the Input, Run and Copy the Output
cnmlMemcpyBatchTensorToDevice(&in_data, input, ...);
cnmlComputeConvOpForward(op, input, output, ...);
cnmlMemcpyBatchTensorToHost(&out_data, output, ...);
```

**Fig. 7** Code segment of running a convolution operator with CNML

```
// Initalizing, Configuring, Creating Tensor (I/O, Filter, Bias, ...)
cnmlBaseOp_t op1, op2;
cnmlOpParam_t op1_param, op2_param;
cnmlTensor_t op1_input, op1_output;
cnmlTensor_t op2_input, op2_output;
cnmlCreateTensor(&op1_input);
cnmlCreateTensor(&op1_output);
cnmlCreateTensor(&op2_input);
cnmlCreateTensor(&op2_output);
...

// Creating Op
cnmlCreateOpParam(&op1_param, ... /* Operator 1 Specifications */);
cnmlCreateOpParam(&op2_param, ... /* Operator 2 Specifications */);
cnmlCreateOp(&op1, &op1_param, I/O, ...);
cnmlCreateOp(&op2, &op2_param, I/O, ...);

// Initalizing Fusion Op, Configuring
cnmlFusionOp_t fusion_op;
cnmlCreateFusionOp(&fusion_op);
...

// Fusing Base Op and Setting I/O, Compiling
cnmlFuseOp(&op1, &fusion_op);
cnmlFuseOp(&op2, &fusion_op);
cnmlSetFusionInput(&fusion_op, op1_input, ...);
cnmlSetFusionOutput(&fusion_op, op2_output, ...);
cnmlCompileFusion(&fusion_op, ...);

// MemCpy and Executing
cnmlMemcpyTensorToDevice(&CPU_DATA_INPUT, op1_input, ...)
cnmlComputeFusionOpForward(fusion_op, ...);
cnmlMemcpyTensorToHost(&CPU_DATA_OUTPUT, op2_output, ...);
```

**Fig. 8** Code segment of running a fusion operator with CNML

not sparse. Moreover, once the choice has been made, the runtime library packs the original network specification file in other formats into `.cambricon` format with the selected setting, which means this static selection can not be changed once the network file is packed, which is not flexible enough. The algorithm of fusion/sparse optimization is also unavailable, making it barely impossible to do further optimization on the provided runtime library.

On the other hand, the optimization knobs provided by the operator-level SDK include detailed fusion schemes, selection of core to use (`Model_Parallelism`), sparsity, how to fuse layers (Fusion Scheme), etc., some of them are hardware-related optimization knob. Fusion scheme and `Model_Parallelism` are focused on in our implemented tool-chain. However, to program and optimize on the original operator-level is also not very convenient because of many redundant statements as Figure 8 illustrated, line 19-28 are responsible for configurations of fusion operator, which can be abstracted.

So, it is necessary to develop a tool-chain with a proper balance between software abstract and hardware specification, by which programmer can optimize the inference session more convenient. Moreover, the principle for optimization is also necessary to guide the user on how to optimize the inference session on the hardware.

## 4 Our approach: Paleozoic

In this research, we develop the tool-chain named **Paleozoic**, stands for the lower but wider level of **Cambrian** in geological time. Our work is based on higher-level IR (ONNX) and lower-level IR (operator description), the tool-chain will generate C++ code based on using operator-level SDK directly from a network specification file in ONNX format, by which an inference session can be compiled by `g++`. Moreover, we integrate some optimization knobs supporting the user to conveniently optimize the inference session using their own findings, algorithm, etc. Figure 9 illustrate the structure of our research.

### 4.1 Front-end

Our front-end is mainly responsible for parsing the network specification file into low lever operator descriptions in which it can match the interface of provided operator-level SDK.

First, our front-end only receive the network specification file in ONNX format, and the user can convert network specification files in other framework using the built-in API. Currently, most of the frameworks support

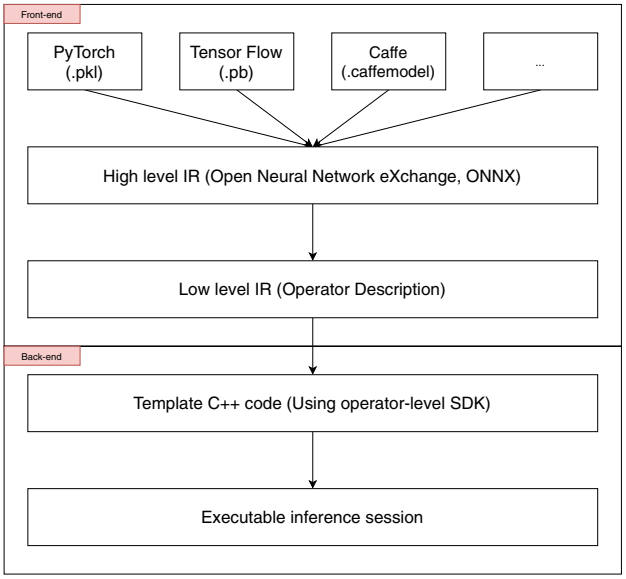



**Fig. 9** Workflow of our tool-chain

converting between ONNX and their own format. Part of the normative specification of the semantics of ONNX IR is shown in Table 3, to match the programming model of provided SDK, we design our own lower-level IR, also shown in Table 3.

In our IR, we expose several optimization knobs related to the hardware. It should be noted that there already exists a build-in ONNX library in Python supporting the parsing of network specification file in ONNX, but the support of operators is incomplete, including the lack of dilation in convolution. For end-users, they can easily optimize the model on **F-Block** level of fusion schemes and part of hyper-parameters. Besides, we reserve the optimization interface of some other hyper-parameters on **Layer** level.

In this work, we only use the front-end part of the TVM (TVM.Relay) to parse the network specification file to lower level operator descriptors and network specifications. The reason for not integrating Cambricon MLU-100 to existing TVM backend is listed below:

- TVM backend applied GA to every nested loop in the network to find an optimal configuration, thus schedule the code with high efficiency. However, the provided SDKs of Cambricon MLU-100 does not support such a low level.
- We want to get a template C++ file that is convenient to debug and profiling, which is not very similar with the existing programming model of TVM.

With these reasons, we only use the TVM front-end as parser.

First, out front-end scan from the beginning of the network specification file, acquiring the settings of type, parameters layer by layer. Also, the shape of tensors should be acquired for scheduler using. Since the shape of tensors running in the network depend on the shape of the input tensor and parameters of layers, and the network specification file does not contain information about the shape of tensors. So, after processing one layer, our front-end calculates the shape of the output tensor of the layer. Finally, the layers description of the network and shape of corresponding tensors are pushed into a queue to be scheduled. There is also an optimization pipeline before scheduling, which will be mentioned in the following part (Fig. 10).

## 4.2 Back-end

The back-end is the main part of our design. As mentioned in Sect. 3, existing tool-chain has inconvenience in both programming interface and optimization knob. Moreover, it will be better if the gap between ONNX and other frameworks can be filled. So, we design the back-end model as following.

First, we decide to generate a template C++ file using the CNML SDK, which will be compiled to executable inference session later instead of directly execute the inference

**Table 3** ONNX IR and our IR

| Type | Abstract | Description | Important fields |
|---|---|---|---|
| ONNX IR | Model | Top-level construct associating metadata with a graph | *ir_version, opset, graph* |
| | Op Set | Explicitly naming the operator sets a model relies on | *opset_version, ops* [] |
| | Graph | Describing the dataflow of executing the model | *nodes* []. *input, output* |
| | Node | Describing the behaviour of a layer in the neural network | *op, attribute, in, out* |
| | Operator | Explicitly declaring the used operators | *op_version, op_type* |
| Our IR | Model | Describing the execution flow of the model | *ir_version*, ***f_blocks***[] |
| | Op Set | Explicitly naming the operator sets a model relies on | *opset_version, ops*[] |
| | **F-Block** | One fusion block fusing multiple layers together | ***layers***[], ***MP***, *in, out* |
| | **Layer** | One single regular layer in the model | *op*, ***MP, Sparsity, ...***, *in, out* |
| | Operator | Describing the detail parameters of the operator | *op_type, attribute* |

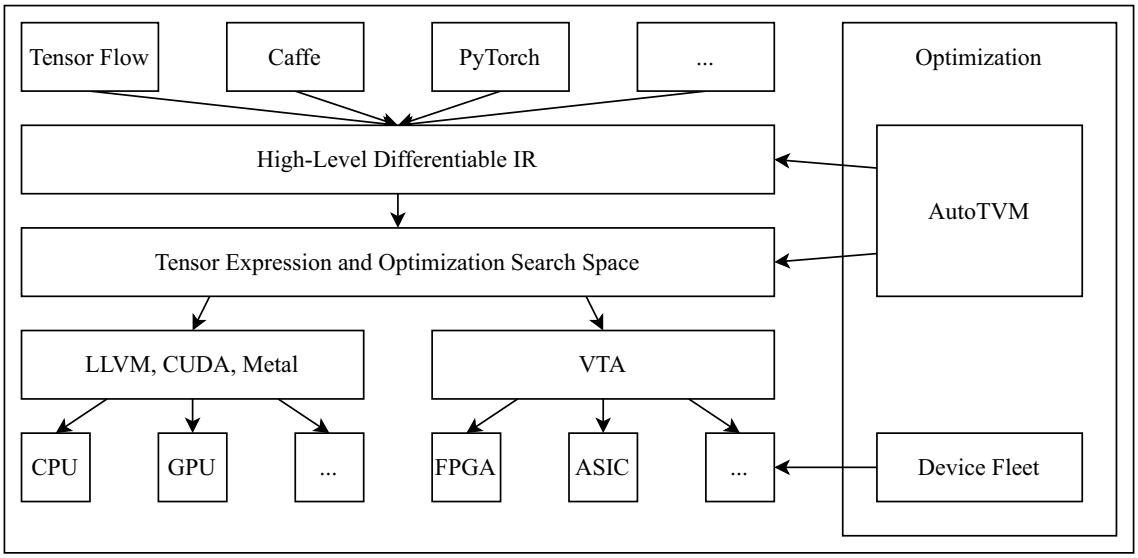



**Fig. 10** TVM workflow (Chen et al. 2018)

session from input network description files. In this way, the user can still conveniently run the inference session by simply compile the generated C++ file, or they can also adjust, debug, and conduct their own optimization on the C++ file. So, to make it convenient for users to operate on the template C++ file, we design a middle layer interface to invoke the CNML. Comparing to origin CNML SDK, we abstract the allocating of tensors into the constructor of one operator (layer) and configuring of parameters into a single function. The design of part of our middle layer interface is shown in Fig. 12 in UML format, and we are keeping supporting other common operators currently, aiming to support state-of-the-art neural networks in CV, NLP.

The parameters of the layers in the network will be filled into the constructor and configuring function according to the descriptions of the layers in the queue pushed by the front-end. It should be noted that the provided CNML SDK does not originally support some operators so that we should find an equivalent implementation to guarantee the correctness of the network. For example, given the means $\mu$ and varance $\sigma^2$, originally supported `Batch_Normalization` compute the result as equation 1.

$$\hat{x} \leftarrow \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \tag{1}$$

However, the `Batch_Normalization` in ResNet need to scale and shift its distribution after origin normalization, as equation 2 added after equation 1 which is called `Weighted_Batch_Normalization` ans not supported originally by CNML SDK, so we combine a `BN` layer with a `Scaling` layer.

$$y \leftarrow \omega\hat{x} + \beta \tag{2}$$

In addition to `Weighted_Batch_Normalization`, there are some other operators should be substituded with equivalent operators including `GEMM`, `LSTM` cell, etc.

In terms of the programming interface, according to Figure 7, an operator should be compiled before it can run. Before actually compile the operator, there are several parameters can be optimized, including the core to be used (`Model_Parallelism`, in the following parts we use `MP` for short) and how to fuse the layer (layer fusion scheme). So, our designed middle interface reserves the programming interface for these two optimizations making it very convenient for users to optimize the network using their own algorithm. Figure 11 illustrates the usage of our programming interface.

### 4.3 Optimizer

Once we get the operator description and network specification (low-level IR) from high-level IR, we conduct the optimizing procedure. In this stage, the users can custom their own optimization algorithm with given optimization knobs, by which an optimized low-level IR can be achieved. Then, the template C++ code will be scheduled according to the optimized low-level IR. Once the scheduling parameters are determined in the optimization procedure, they will not change dynamically when executing.

As mentioned before, the highly abstracted runtime library is very convenient, but with very limited space to optimize. In contrast, the low-level SDK has a much larger optimization space that may contain better schedule

```
// 0. Global configuration (Static Code)

// 1. Allocating Memory for I/O
float *input  = new float[...];
float *weight = new float[...];
float *bias   = new float[...];
float *output = new float[...];

// 2. Constructing anc Configuring layers
ConvLayer layer0{kernel, stride, padding, .../* operator specifications */};
layer0.configLayer(weight, bias, ... /* Other computing parameters */);

// 2.5 Optimization Pipeline, Users Custom Algorithm
//     Optimization of Model_Parallelism and Fusion Scheme mainly.

// 3. Compiling and Executing
layer0.compileOp();
output = layer0.run(input);
```

**Fig. 11** Usage of our back-end

configuration, but with increasing complexity in both programming and searching. However, with efficiency search algorithms proposed, including Reinforced-GA in Google REGAL (Paliwal et al. 2020), parallel simulated annealing algorithm in TVM (Chen et al. 2018) and even Google CP-SAT, an approximation solver for NP-hard constraint programming problem (Google 2020), the optimization space that human can process increase dramatically. So, our work is trying to expose the optimization space as large as possible with the convenience in programming reserved. Our back-end offers several optimizing knobs, and we will show them by actual code.

The first knob is called `MP`, representing the core to be used by every layer in an inference session. The Cambricon MLU-100 have 32 physical cores (Cambricon Technologies 2019a), which means `MP` can be selected from 1 to 32.

The second knob is Fusion Scheme representing how to fuse the layers in the network to make use of data locality, by which the off-chip memory access can be reduced so that the inference throughput can increase, however, fusion may also introduce redundant calculation (Ragan-Kelley et al. 2013), making it a trade-off between reduced memory access and redundant calculation. And the trade-off is designed by the users.

A fusion layer can also select its optimal `MP`, so that the joint optimization between `MP` and Fusion Scheme can be done. The actual code segment in Figure 13 shows that it is convenient to conduct optimization in `MP` and layer fusion in our back-end.

Since arbitry 2 layers can be fused into a fusion block, and every fusion block can be set a `MP`, the total optimization space is very huge, as shown in equation 3, where $\frac{\prod_{x=1}^{i}(n-x)}{i!}$ represent the selection of fusion scheme and $32^{i+1}$ represent the selection of `MP` of every fusion block.

$$Space(n) = \sum_{i=1}^{n-1}\left(32^{i+1} \times \frac{\prod_{x=1}^{i}(n-x)}{i!}\right) \tag{3}$$

This optimization space is too big for users to conduct the brute search, for example, given a network with 32 layers, there are $3.79 \times 10^{38}$ potential combination of the settings. So, user-customized searching algorithms are necessary.

## 5 Evaluation

To evaluate our work, we mainly consider the convenience of programming and space for optimization, and our work will be compared with runtime library CNRT and operator level SDK CNML. The performance of optimization strategies will not be evaluated. It should be noted that although there are many optimization knobs in the configuration file of CNRT, it only selects the network file matches human input settings. Once a network file is packed, users can not make any changes. We select raw performance, amount of code, and optimization knobs as our evaluation indices.

First, we evaluate the optimization knobs of various toolchain, as shown in Table 4. We are keeping supporting the operator, networks, and optimization knobs. Apart from Load Balance Mode, we reserve enough interface for sparsity and operator concatenation. They can respectively be configured in `configLayer()` stage and be implemented by `ConcatLayer` derived from `Layer` in Fig. 12. Comparing to runtime library CNRT, our backend offer much larger optimization space to search. However, larger space will result in increasing in complexity when programming. So, it is necessary to evaluate the convenience of programming.

We evaluate the convenience of programming by counting the line of code when programming various state-of-the-art neural networks in the CV field. Together with the amount of code, we also evaluate the optimization space by counting the potential optimization combination. The calculation of the amount of code ignores the code responsible for model loading. Table 5 shows the line of code of the file generated from the network specification file. Figure 14 shows the ratio of the amount of code and corresponding optimization space. Our work reduces the line of code considerably and thus reduces complexity when programming. In the meantime, we provide users with a huge optimization space.

An increased abstract layer may increase the overhead when calling and invoking, resulting in a decrease in performance. So, we also evaluate the performance of our back-end with no optimization and compare the gap between directly using operator level SDK CNML. Figure 15 illustrated the performance comparison between

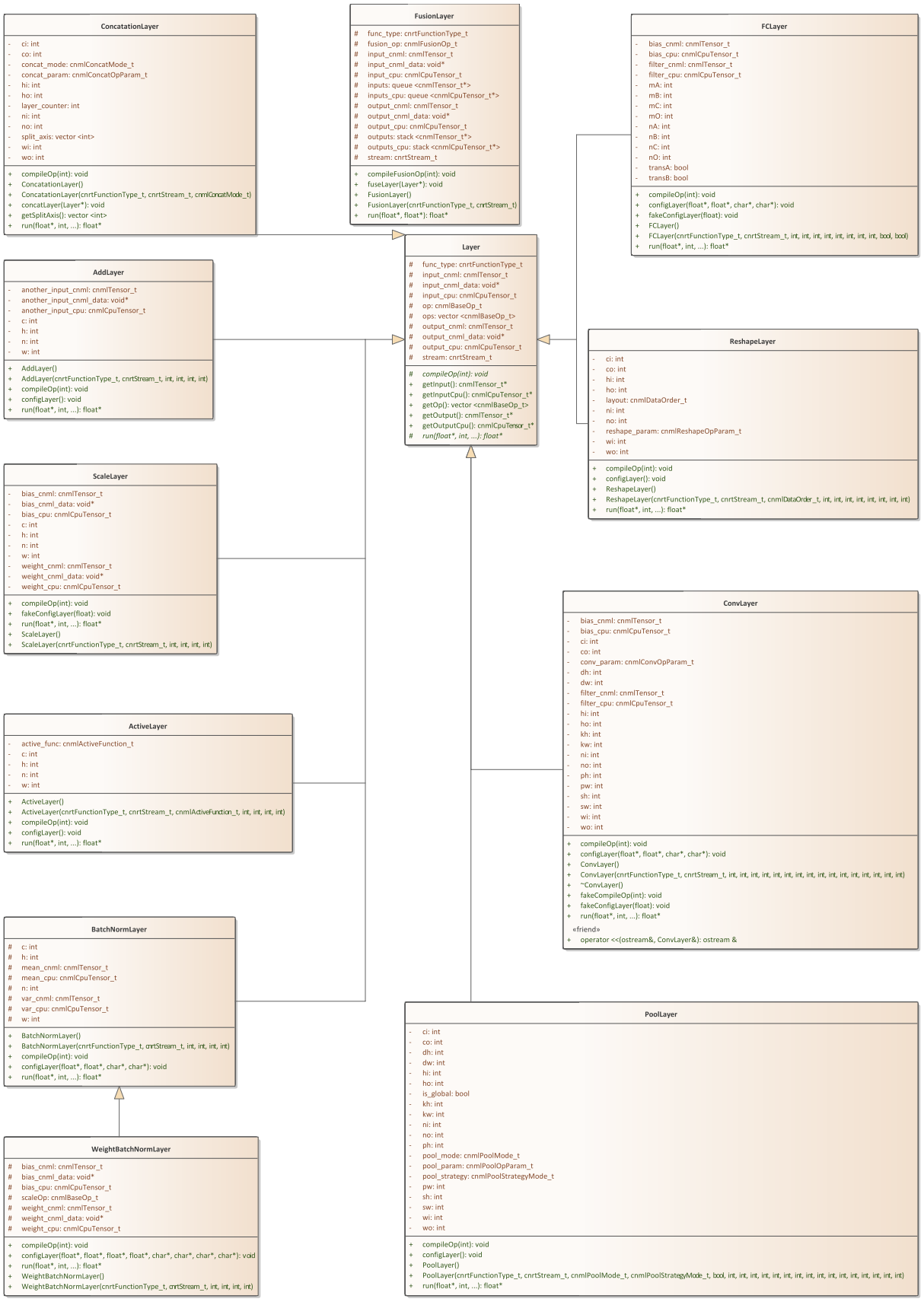



**Fig. 12** Part of interface design

```
// 0. Global configuration (Static Code)
// 1. Allocating Memory and I/O

// 2. Constructing and Configuring Layers
if (FUSION == true) FusionLayer f{...};
ConvLayer layer0{kernel, stride, padding, ...};
PoolLayer layer1{kernel, stride, padding, Avg./Max., ...};
layer0.configLayer(weight, bias, ...);
layer1.configLayer(...);

// 3. Optimization Pipeline, User Custom Algorithm
//    Selection of optimal parameters and fusion scheme

// 4. Compiling
if (FUSION == true) {
    f.fuseLayer(layer0);
    f.fuseLayer(layer1);
    f.compileFusionOp(/* Model_Parallelism = */ opt_mp_fusion);
} else {
    layer0.compileOp(/* Model_Parallelism = */ opt_mp_for_0);
    layer1.compileOp(/* Model_Parallelism = */ opt_mp_for_1);
}

// 5. Executing
if (FUSION == true) {
    output = f.run(input);
} else {
    intermediate = layer0.run(input);
    output = layer1.run(intermediate);
}
```

**Fig. 13** Code sample for setting `MP` and layer fusion of our back-end

**Table 4** Supported optimization knobs

| | CNRT | CNML | Our Work |
|---|---|---|---|
| Batch Size | ✓ | ✓ | ✓ |
| Load Balance Mode | ✓ | ✓ | × |
| Accuracy | × | ✓ | ✓ |
| Sparsity | × | ✓ | Δ |
| Model Parallelism | × | ✓ | ✓ |
| Fusion | × | ✓ | ✓ |
| Concatation | × | ✓ | Δ |

* Δ: Currently private, but interface reserved (Concat layer in Figure 12 and sparsity in `configLayer()`)

**Table 5** Line of code

| | CNRT | CNML | Our Work |
|---|---|---|---|
| ResNet-18 | / | 2491 | 485 |
| ResNet-50 | / | 6225 | 1180 |
| VGG-16 | / | 1387 | 260 |
| VGG-19 | / | 1603 | 296 |
| mobileNet | / | 5579 | 1053 |
| AlexNet | / | 730 | 156 |

our work and operator level SDK CNML, where the gap is tiny, which means our abstract layer will not have a great negative influence on the performance.

It should be noted that any further optimization that will greatly influence the performance will be done by the users. Here we only dispel the misgivings of whether our back-end abstract will influence the performance and ignore the performance of runtime library CNRT since the optimization strategies inside CNRT are unavailable.

Finally, we evaluate the performance of our work by comparing it with NVIDIA RTX2080TI with TensorRT (NVIDIA Corp 2020, 2018), and hardware specification of RTX 2080TI is listed in Table 6.

Table 7 provides the detailed description of deep neural network models used in our evaluation.

We also apply a straightforward optimization strategy to improve the performance of codes generated by our compiler tool-chain. According to Fig. 16, with TensorRT, RTX 2080TI has much better performance against Cambricon MLU-100 with the existing tool-chain, with even half of the theoretical performance in single/half-precision floating computation.

By applying a simple strategy of **Fusing all the layers into one block, and set a maximal** `MP` , the performance of various models on MLU100 improves significantly, and our work has great potential one applying user-customized optimization strategies.

Combining Table 7 and Fig. 16, we have several findings with our simple optimization strategy:

- For networks with low operation count per layer (e.g., ResNet and AlexNet), they can benefit a lot from layer fusion provided by our work (Comparing to the rules-based fusion in TensorRT, which means one fusion block contain only one `CONV` at most).
- For mobileNet, although it has low operation count per layer, it uses depthwise separable convolution with different access patterns with low data reuse (Shao et al. 2019; Sandler et al. 2018) compared to ordinary convolution, which need manually tunning of even lower-level code than provided SDK. So, the fine-tunned TensorRT outperforms our work on mobileNet.
- For networks with less layer but much higher operation count per level (VGG-19), the negative influence of redundant operation brought by fusion is greater than the positive influence of locality, resulting in a lower performance compared to TensorRT.

Currently, we are studying a better optimization strategy for the Cambricon MLU100 accelerator.

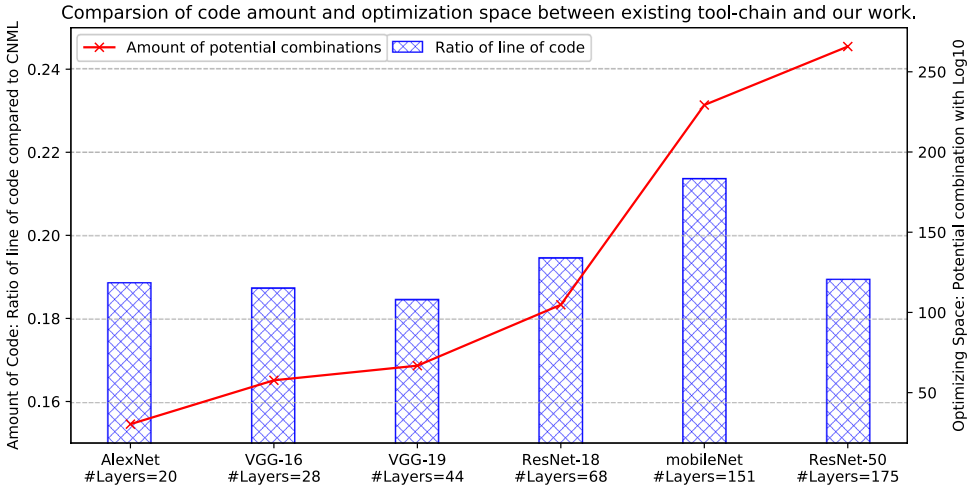



**Fig. 14** Comparsion of code amount and optimization space between existing tool-chain and our work

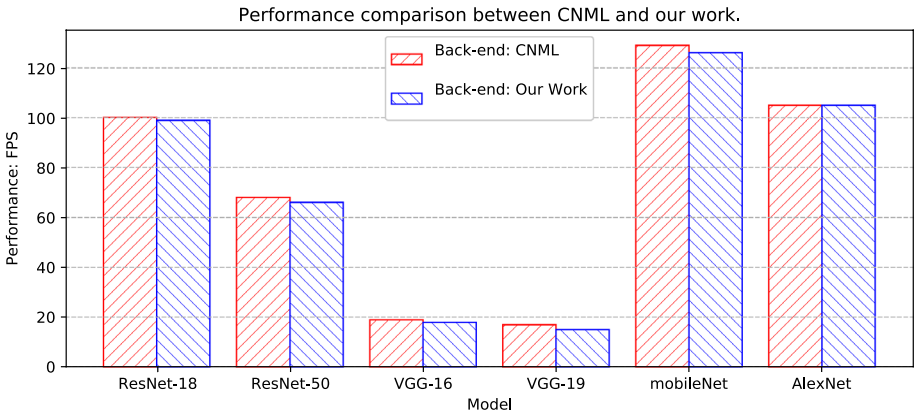



**Fig. 15** Performance comparison between CNML and our work

**Table 6** The RTX 2080TI hardware specification (NVIDIA Corp 2019)

| Item | Descriptions |
|---|---|
| Core freq. | 1.35/1.55 GHz |
| Perf.(FP32) | 13.4 TFLOPS |
| Perf.(FP64) | 420 GFLOPS |
| Perf.(Tensor@INT4) | 440 TOPS |
| Memory size | 11 GB |
| Memory bit width | 352 bit |
| Memory bandwidth | 616 Gib/s |
| Host Interface | PCIe 3.0x16 |
| TDP | 250 W |

## 6 Related work

To address the difficulties in programming the accelerator and optimize the code generated, researchers propose domain-specific language (DSL) to schedule the hardware code with high efficiency. The representative DSL include Halide (Ragan-Kelley et al. 2013), Tensor Comprehension (Vasilache et al. 2018). Originally, Halide is designed for image processing pipeline on NVIDIA GPU, similar to Diesel (Elango et al. 2018), and Tensor Comprehension is designed for machine learning applications. DSLs will abstract complex control logic comparing to traditional language like C/C++/Java. For example, Tensor Comprehension will infer the loop bound automatically, making users free from deciding the

**Table 7** Networks description (Op is in the unit of TOPs)

| Network | Total Op | Avg. Op | No. of CONV |
|---|---|---|---|
| ResNet-18/50 (He et al. 2016) | 3.38/7.61 | 0.169/0.144 | 20/53 |
| VGG-19 (Simonyan and Zisserman 2015) | 36.34 | 2.27 | 16 |
| AlexNet (Krizhevsky et al. 2012) | 1.22 | 0.244 | 5 |
| mobileNet* (Sandler et al. 2018) | 10.33 | 0.199 | 52 |

* With depthwise separable convolution

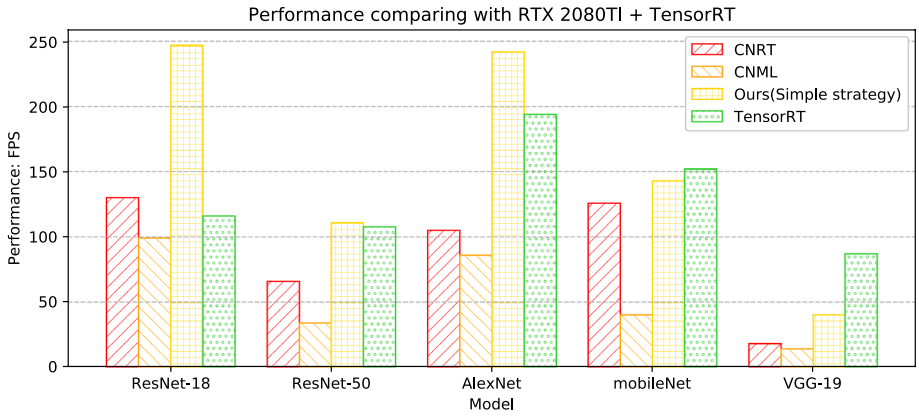



**Fig. 16** Performance comparison between Cambricon MLU100 and NVIDIA RTX 2080Ti (TensorRT)

loop bound. Since neural network tasks have high similarity with image processing pipelines, currently, many programming frameworks of neural networks are based on Halide IR, like TVM (Chen et al. 2018). TVM is proposed to address the problem of universality. To generate more efficient hardware code, there are also several newly introduced schedule frameworks including FlexTensor (Zheng et al. 2020) as a back-end optimizer and TASO (Jia et al. 2019) as graph level optimizer. In this research, we are proposing a DSL for Cambricon MLU100 with lower complexity and higher flexibility comparing to existing tool-chain. In addition to the availability of service to high performance, the robustness of the service is gaining increasing attention, including path-extract based adversarial defend (Qiu et al. 2019), task-level error recovery system in asymmetric architecture (Leng et al. 2020). To integrate these features into existing accelerators with high efficiency, the support of the compiler is necessary.

When it comes to optimization, many proposed hardware architecture designs are considering sparsity (Zhu et al 2019; Albericio et al. 2016), which should be combined with the sparse algorithm. Fusion are another general applied optimization method including loop fusion (Qiao et al. 2019), kernel fusion (Wang et al. 2010). These fusion are mainly focusing on CPU/GPU code. For specialized accelerators, few vendors expose their algorithms. In this work, we reserve the optimization interface of sparsity, fusion, and some other knobs, including tensor layout transformation (Kim et al. 2019), accuracy adjusting to enable the users to apply their own optimization strategies.

## 7 Conclusion

In this paper, we propose a compiler tool-chain design for supporting the inference session of DNN models on the specialized accelerator Cambricon-MLU100. Our tool-chain includes a network parser, a code generator, and an optimization interface. Combined together, it is able to take in ONNX-based network specification files and generate the corresponding C++ codes that wraps over the low-level operator library. Our tool-chain maintains the programmer-friendly APIs that are similar to the high-level runtime library. Meanwhile, it also exposes the programming interface for low-level optimization knobs such that the users can easily customize their own optimization strategies.

**Acknowledgements** We thank the anonymous reviewers for their constructive feedback. This work was supported by National Key R&D Program of China (2019YFF0302600) and the National Natural Science Foundation of China (NSFC) Grant (61702328 and 61832006). Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors

## References

Albericio, J et al.: Cnvlutin: Ineffectual-Neuron- Free Deep Neural Network Computing. In: 43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18–22, 2016, pp. 1–13 (2016). https://doi.org/10.1109/ISCA.2016.11

Alwani, M., et al.: Fused-layer CNN accelerators. In: 49th Annual IEEE/ACM International Symposium on Microarchitecture. (2016)

Chen, T. et al.: TVM: an automated end-to- end optimizing compiler for deep learning. In: 13th USENIX Symposium on Operating Systems Design and Implementation. (2018)

Chen, T. et al.: DianNao: a small-footprint high-throughput accelerator for ubiquitous machinelearning. In: Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1–5, 2014. Ed. by Rajeev Balasubramonian, Al Davis, and Sarita V. Adve, pp. 269–284 (2014). https://doi.org/10.1145/2541940.2541967

Chen, Yunji et al.: DaDianNao: a machine-learning supercomputer. In: 47th Annual IEEE/ACM international symposium on microarchitecture, MI- CRO 2014, Cambridge, United Kingdom, December 13–17. pp. 609–622 (2014). https://doi.org/10.1109/MICRO.2014.58

Copeland, M.: What's the difference between deep learning training and inference? https://blogs.nvidia.com/blog/2016/08/22/difference-deep-learning-training-inference-ai/. Accessed Feb. 20 (2020)

Cui, W. et al.: Ebird: Elastic batch for improving responsiveness and throughput of deep learning services. In: 37th IEEE International Conference on Computer Design, ICCD 2019, Abu Dhabi, United Arab Emirates, November 17–20, 2019.

IEEE, pp. 497–505 (2019). https://doi.org/10.1109/ICCD46524.2019.00075
Dong, Z. et al.: HAWQ: Hessian aware quantization of neural networks with mixed-precision. In: 2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27–November 2. pp. 293–302 (2019). https://doi.org/10.1109/ICCV.2019.00038
Elango, V. et al.: Diesel: DSL for linear algebra and neural net computations on GPUs. In: Proceedings of the 2nd ACM SIGPLAN international workshop on machine learning and programming languages, MAPL@PLDI 2018, Philadelphia, PA, USA, June 18–22, 2018. Ed. by Justin Gottschlich and Alvin Cheung, pp. 42–51 (2018).https://doi.org/10.1145/3211346.3211354
Filipovic, J., et al.: Optimizing CUDA code by kernel fusion: application on BLAS. J. Supercomput. **71**(10), 3934–3957 (2015)
Google. Route. Schedule. Plan. Assign. Pack. Solve. OR-Tools is fast and portable software for com- binatorial optimization. https://developers.google.com/optimization. Accessed May 20, (2020)
Guo C et al.: Flexibility for DNN Acceleration via Temporal GPUSystolic Array Integration. In: CoRR abs/2002.08326 (2020). url: arXiv:2002.08326
He, K. et al.: Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition. (2016)
Jain, A. et al.: Performance Characterization of DNN Training using TensorFlow and PyTorch on Modern Clusters. In: 2019 IEEE International Conference on Cluster Computing, CLUSTER 2019, Albuquerque, NM, USA, September 23–26. pp. 1–11 (2019). https://doi.org/10.1109/CLUSTER.2019.8891042
Jia, Y. et al.: Caffe: convolutional architecture for fast feature embedding. In: Proceedings of the ACM International Conference on Multimedia, MM '14, Orlando, FL, USA, November 03–07, 2014. Ed. by Kien A. Hua et al. pp. 675–678 (2014). https://doi.org/10.1145/2647868.2654889
Jia, Z. et al.: TASO: optimizing deep learning computation with automatic generation of graph substitutions. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles. (2019)
Jouppi, N.P. et al.: In-datacenter performance analysis of a tensor processing unit. In: Proceedings of the 44th Annual International Symposium on Computer Architecture. (2017)
Kim, J. et al.: A code generator for high-performance tensor contractions on GPUs. In: IEEE/ACM International Symposium on Code Generation and Optimization. (2019)
Krizhevsky, A., et al.: ImageNet classification with deep convolutional neural networks. In: Advanced in Neural Information Processing Systems. (2012)
Leng, Jingwen et al.: Asymmetric Resilience: Exploiting Task-Level Idempotency for Transient Error Recovery in Accelerator-Based Systems. In: IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22–26, 2020. IEEE, pp. 44–57 (2020). https://doi.org/10.1109/HPCA47549.2020.00014
Liu, D-F et al.: PuDianNao: a polyvalent machine learning accelerator. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems. (2015)
Marchisio, A., Hanif, M.A., Shafique, M.: CapsAcc: An Efficient Hardware Accelerator for CapsuleNets with Data Reuse. In: Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019. Ed. by Jürgen Teich and Franco Fummi. pp. 964–967 (2019). https://doi.org/10.23919/DATE.2019.8714922
MXNet. A flexible and efficient library for deep learning. A truly open source deep learning framework suited for exible research prototyping and production. https://mxnet.apache.org/. Accessed Feb. 20 (2020)
NVIDIA Corp. Geforce RTX 2080Ti. User Guide. (2019)
NVIDIA Corp. NVIDIA AI INFERENCE PLAT- FORM. Giant Leaps in Performance and Efficiency for AI Services, from the Data Center to the Network's Edge. (2018)
NVIDIA Corp. NVIDIA TensorRT. Programmable Inference Accelerator. (2020)
ONNX. Open Neural Network Exchange. The open standard for machine learning interoperability. http://onnx.ai. Accessed Feb. 20 (2020)
Paliwal, A. et al.: Reinforced genetic algorithm learning for optimizing computation graphs. In: 8th International Conference on Learning Rep- resentations, ICLR 2020, Addis Ababa, Ethiopia, April 26–30 (2020)
Qiao, B. et al.: From loop fusion to kernel fusion: a domain-specific approach to locality optimization. In: IEEE/ACM International Symposium on Code Generation and Optimization. (2019)
Qiu, Yuxian et al.: Adversarial Defense Through Network Profiling Based Path Extraction. In: IEEE Conference on Computer Vision and Pat- tern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20. pp. 4777-4786 (2019). https://doi.org/10.1109/CVPR.2019.00491
Quinton, P.: Systolic arrays: why and how? In: Parcella 1994, VI. International Workshop on Parallel Processing by Cellular Automata and Arrays, Potsdam, Germany, September 21–23, 1994. Proceedings. Ed. by Chris R. Jesshope, Vesselin Jossifov, and Wolfgang Wilhelmi. Vol. 81. Mathematical Research. pp. 39–50 (1994)
Ragan-Kelley. J., et al.: Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In: Conference on Programming Language Design and Implementation. (2013)
Sandler, M., et al.: MobileNetV2: Inverted Residuals and Linear Bottlenecks. In: Conference on Computer Vision and Pattern Recognition. (2018)
Shao, Y.S. et al.: Simba: Scaling Deep- Learning Inference with Multi-Chip-Module-Based Architecture. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12–16, 2019. ACM, pp. 14–27 (2019). https://doi.org/10.1145/3352460.3358302
Simonyan, K., Zisserman, A.: VVGGery Deep Convolutional Networks for Large-Scale Image Recognition. In: 3rd International Conference on Learning Representations. (2015)
Cambricon Technologies. Cambricon MLU100 Datasheet. Aug. (2019)
Cambricon Technologies. Cambricon Neuware Whitesheet. Aug. (2019)
Vasilache, N., et al.: Tensor comprehensions: framework-agnostic high-performance machine learning abstractions. In: CoRR 1802.04730 (2018)
Wang, G., Lin, Y., Yi, W.: Kernel fusion: an effective method for better power efficiency on multithreaded GPU. In: 2010 IEEE/ACM Int'l Conference on Green Computing and Communications. (2010)
Zhang, S. et al.: Cambricon-X: An accelerator for sparse neural networks. In: 49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15–19, 20:1–20:12 (2016). https://doi.org/10.1109/MICRO.2016.7783723
Zhang, W et al.: Laius: towards latency awareness and improved utilization of spatial multitasking accelerators in datacenters. In: Proceed- ings of the ACM International Conference on Supercomputing, ICS 2019, Phoenix, AZ, USA, June 26-28, 2019. Ed. by Rudolf Eigenmann, Chen Ding, and Sally A. McKee. ACM, pp. 58–68 (2019). https://doi.org/10.1145/3330345.3330351
Zheng, S. et al.: FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System. In: ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland,

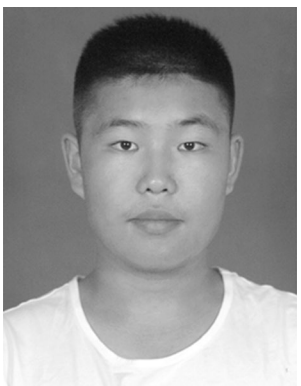
March 16-20, 2020 [ASPLOS 2020 was canceled because of COVID-19]. Ed. by James R. Larus, Luis Ceze, and Karin Strauss. pp. 859-873 (2020). https://doi.org/10.1145/3373376.3378508

Zhou, X et al.: Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach. In: 51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20–24. pp. 15–28 (2018). https://doi.org/10.1109/MICRO.2018.00011

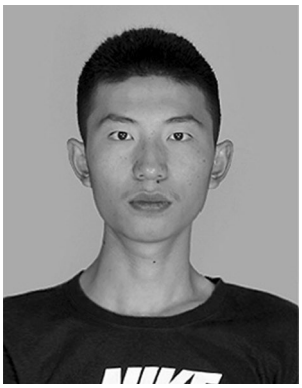
Zhu, M. et al.: Sparse Tensor Core: Algorithm and Hardware Co-Design for Vector-wise Sparse Neural Networks on Modern GPUs. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, pp. 359-371 (2019). https://doi.org/10.1145/3352460.3358269

**Zihan Liu** received the B.Sc. degree from East China Normal University, China. He is currently toward the M.Sc degree in the field of computer science under supervision of Dr. Jingwen Leng with Department of Computer Engineering Faculty, Shanghai Jiao Tong University, China. His research interests include computer system architecture and deep learning compiler.

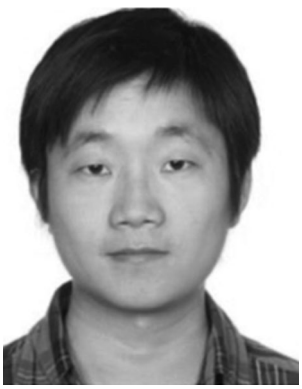
**Jingwen Leng** received the Ph.D. degree from the University of Texas at Austin, where he focused on improving the efficiency and resiliency of general-purpose GPUs. He is a Tenure-Track Associate Professor with the John Hopcroft Computer Science Center, Computer Science Department, Shanghai Jiao Tong University. He is currently interested at taking a holistic approach to optimizing the performance, efficiency, and reliability for heterogeneous computing systems.



**Guandong Lu** received the B.Sc. degree from Shanghai Jiao Tong University, China. He is currently toward the M.Sc degree in the field of computer science under supervision of Dr. Jingwen Leng with Department of Computer Engineering Faculty, Shanghai Jiao Tong University, China. His research interests include resilient computing on machine learning system and computer architecture.



**Chenhui Wang** is currently toward the B.Sc. degree in automation specialty (AI) with Department of Computer Engineering Faculty, Shanghai Jiao Tong University, China. His research interest include high performance computing and parallel optimization.



**Quan Chen** received the PhD degree from the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China, June 2014. He is a tenure-track associate professor with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. His research interests include high performance computing, task scheduling in various architectures, resource management in datacenter, runtime system and operating system.

**Minyi Guo** received the PhD degree in computer science from the University of Tsukuba, Japan. He is currently Zhiyuan chair professor and head of the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. His present research interests include parallel/distributed computing, compiler optimizations, embedded systems, pervasive computing, big data, and cloud computing. He is currently on the editorial board of the IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Cloud Computing, and Journal of Parallel and Distributed Computing. He is a fellow of CCF.