doi:10.3772/j.issn.1006-6748.2020.02.005

NNL: a domain-specific language for neural networks[①]

Wang Bingrui(王秉睿)[②] [*], Chen Yunji[**]

([*] School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, P. R. China)

([**] Intelligent Processor Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, P. R. China)

Abstract

Recent years, neural networks (NNs) have received increasing attention from both academia and industry. So far significant diversity among existing NNs as well as their hardware platforms makes NN programming a daunting task. In this paper, a domain-specific language (DSL) for NNs, neural network language (NNL) is proposed to deliver productivity of NN programming and portable performance of NN execution on different hardware platforms. The productivity and flexibility of NN programming are enabled by abstracting NNs as a directed graph of blocks. The language describes 4 representative and widely used NNs and runs them on 3 different hardware platforms (CPU, GPU and NN accelerator). Experimental results show that NNs written with the proposed language are, on average, 14.5% better than the baseline implementations across these 3 platforms. Moreover, compared with the Caffe framework that specifically targets the GPU platform, the code can achieve similar performance.

Key words: artificial neural network (NN), domain-specific language (DSL), neural network (NN) accelerator

0 Introduction

Generally, there are two evolving trends of neural networks (NNs). The first is that the number of neural network models has increased rapidly. One of the most well-known neural networks is the multi-layer perceptron (MLPs). By increasing the number of layers, MLPs are evolving to powerful variants, such as convolutional neural networks (CNNs) that specify convolutional and sub-sampling layers to handle 2D images[1]. In the meantime, the dimensions and parameters of NNs have also grown tremendously. For instance, the runner-up in ImageNet Challenge 2014, VGGNet[2], contains 16 layers with more than 140 M parameters. Due to the wide variations in network topologies, learning algorithms, and parameters, general-purpose programming languages (e.g., C/C++, Python, and Matlab) cannot efficiently prototype and compare different NNs. Moreover, it is a challenging task to evaluate the performance of NNs on different hardware platforms. Typically, it requires deep understanding of architecture-level details, such as cache-line size, SIMD widths, and the number of cores, to optimize the performance. As an illustrative example, the performance of different implementations of CNN for classifying images on a multi-core processor can vary about 20 × between a tentative implementation and an optimized one. In addition, as today's hardware architectures are advancing towards more heterogeneous processing units (multi-cores, GPUs, FPGAs, and specialized accelerators), various architecture-specific programming models, such as OpenMP, Pthreads, and CUDA, further aggravate the burden of programming neural networks.

In this work, a domain-specific language (DSL), called neural network language (NNL), is proposed to deliver both productivity of NN programming and portable performance of NN execution across different hardware platforms. Its basic idea is to abstract the neural network as a directed graph of blocks. A block basically describes (forward/backward) operations on a specific neuron topology, and thus it mainly consists of 3 components, i.e., connection, forward operators, and

① Supported by the National Key Research and Development Program of China (No. 2017YFA0700902,2017YFB1003101), the National Natural Science Foundation of China (No. 61472396,61432016, 61473275, 61522211, 61532016, 61521092, 61502446, 61672491, 61602441, 61602446, 61732002, 61702478), the 973 Program of China (No. 2015CB358800), National Science and Technology Major Project(No. 2018ZX01031102), the Transformation and Transfer of Scientific and Technological Achievements of Chinese Academy of Sciences (No. KFJ-HGZX-013) and Strategic Priority Research Program of Chinese Academy of Sciences (No. XDBS01050200).

② To whom correspondence should be addressed. E-mail: wangbrsh@mail.ustc.edu.cn

Received on Jan. 18, 2019

backward operators. The main intuition of such block graph is that an NN can be regarded as the sequential composition of operations applied to a group of structured neurons. Based on the block-graph abstraction, the proposed NNL is enough expressive to rapidly prototype most neural networks with various sizes.

To enable portable performance of NN executions on different hardware platforms, the NNL compiler is built to transform the NNL program to an efficient low-level architecture with special designed code. More specifically, the (default/forward/backward) operations applied to structured neurons are translated to standard matrix/vector operations in order to take advantage of portable, high-performance libraries such as Basic Linear Algebra Subroutines (BLAS) and Intel Math Kernel Library (MKL). During the code generation, the compiler benefits significantly from the rich semantics offered by NNL. For instance, the compiler can easily reason about the boundary of matrices/vectors. They are explicitly specified by the connection component of a block. In short, the proposed domain-specific language not only raises the productivity level of NN programming, but also allows high performance learning/prediction of NN on various platforms.

This study makes the following contributions:

(1) A domain-specific language, NNL, is proposed for programming neural networks. The proposed NNL features show abundant expressiveness to efficiently describe various types of neural networks.

(2) A compiler for NNL is presented to translate the NNL to efficient architecture-specific implementations, allowing the NNL program to achieve portable performance across different architectures.

(3) 4 representative and widely-used neural networks are described with the proposed language and executed on 3 different platforms, i. e., CPU, GPU, and a neural network accelerator, DaDianNao[3]. For different types of neural networks, baseline implementations with general-purpose languages such as C++ and CUDA are provided. Caffe framework is used to build networks for the CPU and GPU platforms. It is shown that, on average, NNL code is 14.5% better than the baselines across these 3 platforms.

1 Neural network programming dilemmas

Nowadays, neural network applications can be programmed on different ways, including machine learning libraries, neural network frameworks or domain-specific languages. To name a few, Pylearn2[4] and DLPLib[5] are built on machine learning libraries. There also exists several programming frameworks targeting special neural networks such as CNNs, and Caffe[6] and Tensorflow[7] are two of those notable examples. Latte[8], OptiML[9] and Swift for TensofFlow[10] are domain specific languages for deep neural networks and general machine learning tasks. Some researchers concentrate on the intermediate representation and low-level virtual machine for machine learning DSLs, e. g., Relay[11] and TVM[12]. Although all these frameworks can be used for programming neural networks, the key drawback is that they cannot describe different neural networks in a unified and flexible manner, leading to considerable reimplementation efforts when adapting to a new type of neural networks. On the other hand, the semantics of Latte is general-purpose and doesn't take advantage of the domain knowledge to achieve productivity. Therefore, it is expected that a programming language exactly targeting the domain of general neural networks would be very helpful for NN programming.

In addition, although stated programming frameworks or languages have been widely used for neural network programming, the underlying implementations may not be compatible with a wide variety of hardware platforms, let alone to fully exploiting their computational ability. In addition to CPU and GPUs, emerging neural network accelerators, such as DianNao[13], DaDianNao, ShiDianNao[14] and Cambricon[15], are attracting more attentions. Thus, a neural network programming model or language should be able to achieve portable performance on CPUs, GPUs, and ASIC accelerators.

2 Neural network language

This section presents the syntax of the proposed neural network language and further illustrates it with examples.

2.1 Language syntax

Fig. 1 shows the language syntax of the NNL. The programming model of NNL is based on graph description, thus the syntax describes 2 primary components of graph, that is, nodes and edges among them. NN layers are defined as *blocks* and the construction of layers as *topology*. The whole program of NNL mainly comprises the above 2 parts.

Program A program in NNL consists of at most one-time declaration, a nonempty list of *blocks*, and a *blockinst*. The time declaration is an assignment to a variable named *Cycle*, which is required by neural networks with temporal behaviors, e. g., RNN and LSTM[16].

Block A block declaration consists of the keyword block, a block name, a list of variables, and the block body. The variables are used as parameters of the block body. The block body consists of a *basicBlock* or a *blockinst*. The *basicBlock* declaration includes a nonempty list of port, a *connection*, and a *calculation*. The *blockinst* consists of a nonempty list of *inst* and *topo*. Hence, the *blockinst* provides the instantiation of blocks and creation of *topo*.

Port The ports are treated as the interface of blocks, and a port of a block can be declared with its name and omissible dimension expression.

Connection The connection declaration is composed of the keyword connection, a nonempty list of loops with at most one keyword share as the constraint of the loop and a link. A loop declaration contains a loop expression that can be denoted as (*i*; *start*; *end*; *step*). The loop expression indicates that the variable *i* increases from start to end (not included) with a stride step. In addition, the keyword share means that the loop locates in the share scope. The link declaration specifies the In and Out with their shapes characterized by the dimension expression *expr*. The connection between the In and Out is denoted by the symbol <->. Apparently, the link declaration is the core of a connection declaration.

Calculation The calculation declaration, which is a part of the *basicBlock* declaration, consists of the keyword once, forward or backward with a nonempty list of statements. The statements in the forward/backward pass define the forward/backward algorithms applied to a particular part of the neural network, while the once statement is used for initialization, and it only executes once.

Instantiation The *inst* is to instantiate the block declaration with actual parameters *expr*. Furthermore, a block instance can also be built in blocks denoted as portblock(), passblock() and streamblock().

Topology The *topo*, which describes the directed connection relationship between ports of instantiated blocks, is specified by a sequence of blocks with a port and unidirectional arrows between neighboring elements recursively. In order to describe the *topo* efficiently, the ports in a block can be omitted and two different kinds of symbols (-> and ^) are used to represent the arrows. For example, b1->b2->b3 is the abbreviation of b1(Out)->b2(In) and b2(Out)->b3(In), while b1^b2 is the abbreviation of b1(Out)->b2(Weight), where Weight is an implicit port used for blocks with connection.

Expression and statement The operators are classified into *uop* and *bop*. All the variables in NNL can be

```
program       ::= [time]{block}+{blockinat}
block         ::= block id ({var}){[basicBlock|blockinat]}
basicBlock    ::= {port}+[connection][calculation]
blockinst     ::= {inst}+{topo}+
port          ::= portname {[expr]}
connection    ::= connection {[share] loop}+link
loop          ::= (var;expr;expr;expr)
link          ::= In{[expr]}+<-> Out{[expr]}+
calculation   ::= [once:{stmt}+][forward:{stmt}+][backward:{stmt}+]
stmt          ::= var=expr
               |  if (expr) stmt [else atmt]
               |   {{stmt}}
               |  kernel stmt
inst          ::= {id}+ = portblock()|passblock()|streamblock()|id ({expr});
topo          ::= id[(portname)][<expr>]{[->|^]topo}
time          ::= Cycle=N
expr          ::= var|portname|(expr)func(expr)|uop expr|expr bop expr
uop           ::= -|/
bop           ::= +|-|*|/|%|&&|//|>|<|>=|<=
func          ::= Exp(expr)|Sum(expr)|Mean(expr)|Max(expr)|Avg(expr)
               |  Log(expr)|Abs(expr)|sin(expr)|Cos(expr)|Sinh(expr)|Cosh(expr)|tanh(expr)
portname      ::= In|Out| var
var           ::= id|t|Cycle| N
const: N
name: id
```

Fig. 1 The syntax of proposed NNL

scalar, vector or matrix. The scalars include constants, declared variables, and results of reduction functions such as *Max*() and *Sum*(). The vectors include declared vector variables, which can only be *port* name representing the data of ports. In addition, port names with the prefix *Delta* represent the gradient port with respect to the input port. Moreover, the vectors can also be the result of mathematical functions such as *Exp*() and *Log*(). The *t* is a built-in vector variable, elements of which range from 1 to *Cycle*.

2.2 Block example

As shown in Fig. 2, there are 4 types of basic blocks, i.e., *connection*, *calculation*, *conn-calc*, and *common* block.

Fig. 2(a) shows the declaration of the *connection* block. It describes connections between the neurons in a specific part of the neural network. The *size*1 and *size*2 are parameters of the block, and they are used to specify the dimension of the ports such as In and Out. The key of the connection block is the connection definition, and it is used for describing the connections between the In and Out.

Fig. 2(b) presents the *calculation* block. This block mainly contains 2 calculation passes, i.e., forward and backward pass, specified by the keyword forward and backward, respectively. The forward pass contains all the calculations that are used in the forward algorithm of the neural network, while the backward pass consists of all the calculations for the backward algorithm of the neural network.

```
block connectionblock(size1,size2, ...)
{
    In[ size1 ][ ... ];
    Out[ size2 ][ ... ];
    ...

    connection
    {
        (i;1;size1;1)
        (j;1;size2;1)
        ...

        share
            ...

            In[ i ][ ... ]<->Out[ j ][ ... ]
    }
}
```

(a) *connection* block

```
block calculationblock( p1,p2, ... )
{
    ...

    forward:
        ...

        Out = In*p1
        ...

    backward:
        ...

        Delta_In = Out/p2

        ...
}
```

(b) *calculation* block

```
block conncalcblock(size1,size2, ...)
{
    In[ size1 ][ ... ];
    Out[ size2 ][ ... ];
    ...
    connection
    {
        (i;1;size1;1)
        (j;1;size2;1)
        ...

        share
            ...

            In[ i ][ ... ]<->Out[ j ][ ... ]
    }
    forward:
        ...

        Out = In
        ...

    backward:
        ...

        Delta_In= Out-In

        ...
}
```

(c) *conn-calc* block

```
block commonblock(size1, ...)
{
    IPort[ size1 ][ ... ];
    OPort1[ size1 ][ ... ];
    OPort2[ size1 ][ ... ];

    ...

    Once:
        ...
        IPort = Exp(2)

        ...
    forward:
        ...

        OPort1 = IPort+OPort2
        ...

    backward:
        ...

        Delta_IPort = OPort1-OPort2

        ...
}
```

(d) *commom* block

Fig. 2 Basic block construct

Fig. 2(c) is the *conn-calc* block that contains both the connection and the calculation, while Fig. 2(d) is an example of the *common* block. Actually, the *calculation* block is a special case of the *common* block, that is, *common* block with only 2 ports, one for input and the other for output.

2.3 CNN example

Fig. 3 shows the code example for building a typical convolution neural network. As shown above, the programmers only need to provide three more block declarations, i.e., convolution, pooling, and lrn. The convolution is connected by the active operation ac1, while the pooling is connected by the active operation ac1 and lrn. The lrn is further connected with the active operation ac2 and ac3.

3 Implementation

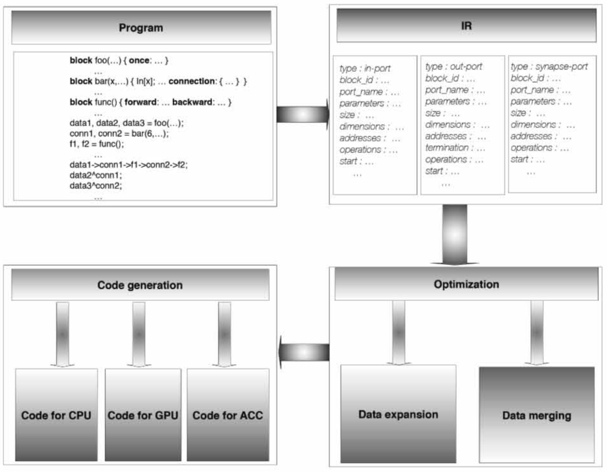
This section introduces the language implementation and describes the techniques for compiling the NNL programs to high-performance platform-specific code. NNL uses domain-specific knowledge to reason about programs at a higher level than a general purpose language or a library, which allows to achieve portable performance on various hardware platforms.

```
block data( x,y,z) { once : ... }           // define data block
block parameter( x,y,z1,z2) { forward : ... backward : ... } // define parameter
block mlp( x1,x2) { In[x1] Out[x2] ... connection { ... } } // define connection
block active() { forward : ... }            // define active operation
block loss() { forward : ... backward : ... } // define loss operation
block Convolution( xi,yi,fi,xo,yo,fo,...) //define convolution connection
{ In[xi][yi][fi]; Out[xo][yo][fo]; connection { ... } }
block Pooling( xi,yi,fi,xo,yo,fo,...)       // define pooling connection & calculation
{ In[xi][yi][fi]; Out[xo][yo][fo] connection { ... } forward : ... backward : ... }
block Lrn( xi,yi,fi,xo,yo,fo,...)           // define lrn connection & calculation
{ In[xi][yi][fi]; Out[xo][yo][fo] connection { ... } forward : kernel ... backward : ... }
synapse1 = parameter(8,8,3,8);              // parameters of convolution
bias1 = parameter(8,0,0,0) ;                // parameters of convolution
synapse2 = parameter(32,10,0,0);            // parameters of mlp
bias2 = parameter(8,0,0,0);                 // parameters of mlp
input = data(8,8,3); label = data(10,0,0); // input and label
conv = Convolution(8,8,3,4,4,8,...);        // convolution connection
pool = Pooling(4,4,8,2,2,8,...);            // pooling connection & calculation
lrn = Lrn(2,2,8,2,2,8,...);                 // lrn connection & calculation
full = mlp(32,10);                          // To generate mlp connection
ac1, ac2, ac3, ac4 = active();              // To generate active
sft_loss = loss();                          // To generate loss
input->conv->ac1->pool->ac2->lrn->ac3->full->ac4->sft_loss // neuron forward
synapse1^conv;                              // synapse connection
bias1->ac1;                                 // bias
synapse2^full;                              // synapse
bias2->ac4;                                 // bias
label->sft_loss;                            // label
```

Fig. 3 Using NNL for constructing a CNN example

3.1 Compiler overview

Fig. 4 shows the compiling framework of NNL. The compiler mainly contains 4 main phases: the front-end, intermediate representation (IR) generation, IR analysis and optimization, and platform-specific code generation.



```
block foo(...) { once: ... }
    ...
block bar(x,...) { In[x]; ... connection: { ... } }
    ...
block func() { forward: ... backward: ... }
    ...
data1, data2, data3 = foo(...);
conn1, conn2 = bar(6,...);
f1, f2 = func();
    ...
data1->conn1->f1->conn2->f2;
data2^conn1;
data3^conn2;
    ...
```

```
type : in-port        type : out-port       type : synapse-port
block_id : ...        block_id : ...        block_id : ...
port_name : ...       port_name : ...       port_name : ...
parameters : ...      parameters : ...      parameters : ...
size : ...            size : ...            size : ...
dimensions : ...      dimensions : ...      dimensions : ...
addresses : ...       addresses : ...       addresses : ...
operations : ...      termination : ...     operations : ...
start : ...           operations : ...      start : ...
    ...               start : ...               ...
                          ...
```

Fig. 4 Compiling framework for NNL

To reduce the efforts required to construct the front-end, NNL is implemented as an embedded DSL within C + + language, so that it can reuse the front-end of C + +. Moreover, the object-oriented features of C + + , such as composition, inheritance, and operator overloading, allow the NNL to have a flexible syntax. Nevertheless, NNL is not limited to C + + , and other general-purpose languages (e. g. , Scala and Python) with such features can be used as NNL's host languages as well.

In contrast to traditional embedded DSLs that are constrained to the backends of host languages, NNL employs an intermediate representation (IR), i. e. , the directed graph of blocks or topology, in order to conduct IR-based optimizations and target heterogeneous architectures. By analyzing the IR, compiler can generate different tables for facilitating platform-specific code generation. During the generation of tables, based on the connection between different blocks, several domain-specific optimizations are conducted to reduce the memory consumption and improve the execution efficiency. Details on topology generation, analysis and optimization are discussed below.

The final stage is platform-specific code generation. NNL provide separate backends for different platforms: C code for CPUs, CUDA/OpenCL code for GPUs, and assemble instructions for the accelerator. Since the basic operations of neural networks are matrix (vector)-related computations, the key of code generation is to produce executable codes for different platforms directly.

3.2 Topology generation

The graph topology is generated from the block instances. More specifically, the block declaration first provides the structural description of the neural network components. Then, multiple block instances can be created from the block declaration, sharing the same parameters. Once all block instances are built, the topology of different block instances can be intuitively generated according to the symbol -> and ^. The topology generation does not rely on any platform, and the back-end will generate platform-specific code from the topology-based IR.

3.3 Topology analysis

During the topology analysis, the compiler generates the port table, including the in-port and out-port table. Such tables are generated from the in-ports and out-ports (which are used for data exchange between connected nodes) of the block declaration. Both the in-port and out-port tables store the data information such as address, size, and dimension, etc. Also, for the sink block of the topology graph, the termination status is added to the out-port tables. Once the termination status is set as false, the neural network needs to undergo back propagation.

In NNL, there are 2 types of connection between blocks, that is, -> for neuron transfer and ^ for synapse transfer. If a block is connected to another one with ->, the compiler would update the related entries in the in-port table. The compiler would also add the out-ports information of the block into the out-port table. Therefore, the in-port and out-port table would have overlapped data. Moreover, if more than one blocks are connected to the same block with ->, additional information (i. e. , the default addition operation applied to these out-ports data and in-port data) should be attached to the corresponding entries in the in-port and out-port table.

If two blocks are connected by ^, the synapse table, which stores the corresponding synapses (or weights), would be created or updated. The synapse table is only used for generating code for a basic block containing connection declaration. This will be elaborated in the following code generation section.

In summary, during the topology analysis, the in-port table, out-port table and synapse table are generated for facilitating platform-specific code generation. To initiate code generation, a start status is set in all the tables to indicate whether the code generation should start. Among these tables, only in-port and out-port table are necessary for arbitrary network, while the synapse table is needed for some neural networks such as MLP and CNN.

3.4 Domain-specific optimization

During the generation of tables from the graph topology, the compiler also conducts two domain-specific optimizations, including the data merging and data expansion. In fact, such optimization can be easily implemented with the help of various tables generated from the topology-based IR.

Data merging In the so-called data merging optimization, duplicated data can be merged together to reduce the memory consumption. There are a lot of data reuse due to the special structure of neural networks. The number of data reuse increases significantly with the size of CNN kernel, and the percentage of reused data also increases with feature map size. One more illustrative example is that, for two interconnect blocks, the out-port data of the source block might be the in-port data of the destination block. For all those data reuses, the compiler only allocates memory spaces for the reused data once. Therefore, the total memory consumption can be significantly reduced. For instance, for the workload LeNet-5[17], the memory consumption can be reduced by 12% by using the data merging technique.

Data expansion The basic idea of data expansion optimization is to expand scalar/vector/matrix as matrices, so as to take advantage of highly efficient implementations for matrix operations, which might be compute-bound operations that can better exploit the computation capacity of modern multi-/many-core architectures[18] on different platforms. More specifically, the data expansion optimization is enforced on the blocks connected to the original variables or other blocks with the symbol ^. In other words, data expansion only works for the blocks with synapses.

The compiler leverages rich semantics provided by the NNL to efficiently conduct data expansion. In more detail, as synapses only appear in the blocks with the connection, the compiler can easily expand the data to matrices by exploiting the structure specified by the connection declaration and the synapse table generated after the topology analysis. Apparently, during the data expansion, more memory space is required to gain better data parallelism. Our compiler carefully controls the tradeoff between the data merging and data expansion to balance the achieved performance and consumed memory.

3.5 Executable codes generation

Based on the tables generated from the topology analysis, the platform-specific code can be generated. Moreover, for a neural network, the generated code contains 2 parts, one for the prediction process, and the other for the training process.

For the prediction process, the compiler should handle the data in port tables and the operations in the forward pass. For the data in the inport table, the compiler allocates corresponding memory space based on the stored data information (e. g. , data size). For the operations in the forward pass, the compiler can infer what operations (i. e. , scalar operation or vector operation) should be conducted according to the calculation expression. Also, there is a default matrix-vector operation applied to the connection phase for data from the in-port and synapse table. Once all these operations are processed, the outport table should be updated. In this process, the compiler can dynamically optimize the memory usage. Once the compiler completes the table analysis, it determines whether a back propagation should be conducted based on the termination status. If the termination status is true, the compiler finishes the code generation. Otherwise, the back-propagation process is conducted for neural network training.

For the training process, the compiler also analyzes various tables for error propagation and parameter updates. More specifically, the operations defined in the backward pass of a block are applied to the parameters. Once the start status is detected, the compiler terminates the backward code generation. In this case, both the codes for prediction and training process are generated through the IR.

For both the prediction and training process, the compiler efficiently translates the (forward/backward) operations to function calls of platform-specific libraries, that is, BLAS for CPUs, cuBLAS for GPUs, and the built-in library for the accelerator.

4 Experiments

This section evaluates a set of artificial neural networks written in NNL on different platforms, and compares them to baseline implementations with existing programming frameworks.

4.1 Platforms

The configurations of evaluated platforms (i. e. , CPU, GPU and the accelerator) are listed as follows.

CPU The CPU is an x86-CPU with 256-bit SIMD support (Intel Xeon E5-2620, 2.10 GHz, 64 GB memory). GCC v4.7.2 is used to compile all benchmarks with options (-O2 -lm -march = native) to enable SIMD instructions.

GPU The GPU is a modern GPU card (NVIDIA K40M, 12 GB GDDR5, 4.29 TFlops peak at a 28 nm process).

Accelerator The accelerator for evaluation is DaDianNao, a state-of-the-art NN accelerator exhibiting remarkable energy-efficiency improvement over a GPU.

4.2 Benchmarks

Four representative and widely used NN techniques, including MLP, CNN, LSTM, and LRCN, are chosen as our benchmarks. The detailed configurations of these NN are listed in Table 1. These benchmarks represent state-of-the-art neural networks that are widely used in both academics and industry.

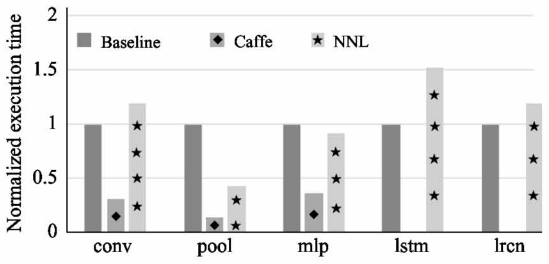
For all these benchmarks, the baseline is implemented by using a general-purpose language, e. g. , C++ for the CPU platform and CUDA for the GPU platform. Due to the limitation of the framework, only optimized implementations of the MLP and CNN for both the CPU and GPU platforms can be obtained by applying Caffe. As a comparison, all the NN benchmarks are programmed with NNL and compiled targeting CPU, GPU, and the NN accelerator.

Table 1 The NN benchmarks for evaluation (H stands for hidden layer, C stands for convolutional layer, K stands for kernel, P stands for pooling layer, F stands for classifier)

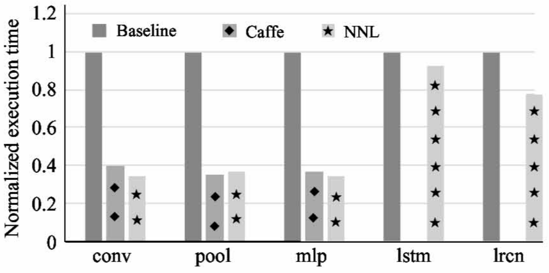
Techniques	Network structure
MLP	input(64) - H1(150) - H2(150) - Output(14)
CNN	input(1@32x32)-C1(6@28x28, K: 6@5x5)-S1(6@14x14, K: 2x2)-C2(16@10x10, K:16 @5x5)-S2 (16 @ 5x5, K: 2x2)-F (120)-F(84)-output(10)
LSTM	input(26) - H(93) - output(61)
LRCN	input(1@32x32)-C1(6@28x28, K: 6@5x5)-S1(6@14x14, K: 2x2)-C2(16@10x10, K:16 @5x5)-S2 (16 @ 5x5, K: 2x2)-F (120)-F1(84)-F2(10)-H(93)-output(61)

Fig. 5 shows the performance comparison of NNL, Caffe and C + + code on the CPU, GPU and NN accelerator. The experiment result shows that, on average, the executable program compiled from NNL code is 14.5% better than the baselines across these 3 platforms.

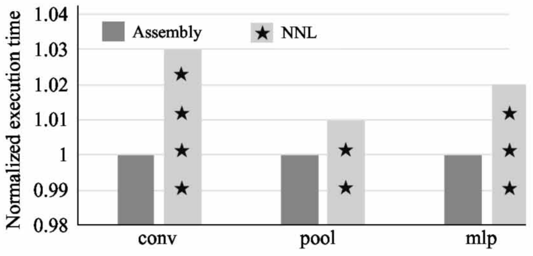
Fig. 5(a) shows the performance comparison of NNL, Caffe and C ++ code on the CPU platforms. All these 3 implementations eventually invoke the native BLAS libraries. The first observation is that the Caffe code perform significantly better than the CPU baseline, since the Caffe framework is highly tuned for the CNN and MLP algorithm. The second observation is that the NNL code performs much better than the CPU baseline for the pool and MLP network. However, on the rest 3 scenarios, i. e., conv, lstm, and lrcn, the NNL code cannot outperform the CPU baseline. The potential reason is that data expansion results in too much off-chip memory access. The detailed analysis on the aggressiveness of data expansion should be left as our future work. Fig. 5(b) compares the execution performance of NNL code, Caffe code and CUDA code on the GPU platform. It shows that both NNL and Caffe significantly outperform the GPU baseline. Because Caffe is a highly tuned library, it is natural that Caffe obtains much better results compared with the baseline for the conv, pool, and MLP networks. Moreover, since NNL expands data to generate large matrix-matrix operations to fully take advantage of the high parallelism of the GPU platform, the performance of the NNL code is comparable to that of the Caffe implementation. In fact, even on the lstm and lrcn network, where the Caffe cannot handle flexibly, the NNL code performs better than the baseline code.



(a) CPU



(b) GPU



(c) NN accelerator

Fig. 5 Execution time of NNL on different platforms

Experiments for the neural network accelerator are also conducted and the results in Fig. 5(c) show that NNL code does not outperform the assemble instructions of the accelerator. This is because the accelerator is specifically designed for CNN. It is not enough to expand the data as a large matrix to benefit the hardware for NNL, while the assemble instructions is highly tied with the hardware to take full advantage of the accelerator.

5 Conclusion

As the sizes and types of neural network models continue to grow, how to efficiently build neural network models for different platforms is very challenging. In this paper, a neural network domain-specific language, NNL, is proposed specifically targeting the NN programming domain. NNL raises the productivity level of NN programming by abstracting an NN as a directed graph of blocks. Moreover, the NNL compiler enables portable performance of NN execution across different hardware platforms (e.g., CPUs, GPU, and ASIC). Experiments are conducted to show that NNL is efficient and flexible for fast prototyping of various NNs, and the execution performance is comparable to state-of-the-art NN programming frameworks.

References

[1] Krizhevsky A, Sutskever I, Hinton G E. ImageNet classification with deep convolutional neural networks[C] // Advances In Neural Information Processing Systems, Harrahs and Harveys, USA, 2012: 1097-1105

[2] Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition [EB/OL]. http://arxiv.org/abs/1409.1556: arXiv, 2015

[3] Chen Y, Luo T, Liu S et al. DaDianNao: a machine-learning supercomputer[C] // Proceedings of the Annual International Symposium on Microarchitecture, Cambridge, UK, 2015: 609-622

[4] Goodfellow I J, Warde-Farley D, Lamblin P, et al. Pylearn2: a machine learning research library[EB/OL]. http://arxiv.org/abs/1308.4214: arXiv, 2013

[5] Lan H Y, Wu LY, Zhang X, et al. DLPlib: a library for deep learning processor[J]. *Journal of Computer Science and Technology*, 2017, 32(2): 286-296

[6] Jia Y, Shelhamer E, Donahue J, et al. Caffe: convolutional architecture for fast feature embedding[EB/OL]. http://arxiv.org/abs/1408.5093: arXiv, 2014

[7] Abadi M, Agarwal A, Barham P, et al. TensorFlow: large-scale machine learning on heterogeneous systems [C] // Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, Savannah, USA, 2016: 265-283

[8] Truong L, Barik R, Totoni E, et al. Latte: a language, compiler, and runtime for elegant and efficient deep neural networks[C] // Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, Santa Barbara, USA, 2016: 209-223

[9] Sujeeth A K, Lee H, Brown K J, et al. OptiML: an implicitly parallel domain-specific language for machine learning[C] // Proceedings of the 28th International Conference on Machine Learning, Bellevue, USA, 2011: 609-616

[10] Lattner C. Swift for TensorFlow [EB/OL]. https://github.com/tensorflow/swift: Google, 2018

[11] Roesch J, Lyubomirsky S, Weber L, et al. Relay: a new IR for machine learning frameworks[C] // Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, Philadelphia, USA, 2018: 58-68

[12] Chen T, Moreau T, Jiang Z, et al. TVM: an automated end-to-end optimizing compiler for deep learning[C] // Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, Berkeley, USA, 2018: 579-594

[13] Chen T, Du Z, Sun N, et al. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning[C] // Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, Salt Lake City, USA, 2014: 269-284

[14] Du Z, Fasthuber R, Chen T, et al. ShiDianNao: shifting vision processing closer to the sensor[C] // Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, USA, 2015: 92-104

[15] Liu S, Du Z, Tao J, et al. Cambricon: an instruction set architecture for neural networks[C] // Proceedings of the 43rd International Symposium on Computer Architecture, Seoul, Korea, 2016: 393-405

[16] Graves A, Schmidhuber J. Framewise phoneme classification with bidirectional LSTM networks[C] // Proceedings of the International Joint Conference on Neural Networks, Montreal, Canada, 2005: 2047-2052

[17] LeCun Y, Bottou L, Bengio Y, et al. Gradient-based learning applied to document recognition[J]. *Proceedings of the IEEE*, 1998, 86(11): 2278-2324

[18] You Y, Song S L, Fu H, et al. MIC-SVM: designing a highly efficient support vector machine for advanced modern multi-core and many-core architectures [C] // Proceedings of the International Parallel and Distributed Processing Symposium, Phoenix, USA, 2014: 809-818

Wang Bingrui, born in 1994. He received his B. S. degree in physics from University of Science and Technology of China in 2015. He is currently a master candidate at School of Computer Science and Technology, University of Science and Technology of China. His research interests include computer architecture and programming language.