Deep Neural Network Operators

Implementation Approach and Challenges

Gunjan Nandy
Machine Learning Engineer Intern
AI Tech System
Kolkata, India
gunjan.nandy1@gmail.com

Hrishikesh Murali
Machine Learning Engineer Intern
AI Tech System
Bangalore, India
hrishi98.m@gmail.com

Nalin Shani
Machine Learning Engineer Intern
AI Tech System
Delhi, India
nalinshani14@gmail.com

Nikhil Tiwari
Machine Learning Engineer Intern
AI Tech System
Bhopal, India
nikhilcbse97@gmail.com

Subham Sarkar
Machine Learning Engineer Intern
AI Tech System
Kolkata, India
kingsubham27@gmail.com

Vishal Yadav
Machine Learning Engineer Intern
AI Tech System
Indore, India
vvyy346@gmail.com

***Abstract*— Last few years have seen increasing upsurge in variety of neural network giving rise to dedicated hardware to meet their performance specification (i.e. latency and throughput). Every new dedicated hardware needs a compiler that can take a high-level specification of deep neural network and compile it into hardware specific machine code taking advantage of performance driven parallel features. Operators (aka layers) are hearts and mind of a deep neural network (DNN). In this paper we highlight design of operators as a pragmatic approach to compilation that enables the generation of highly optimized code for multiple targets.**

I. INTRODUCTION

Applications of deep learning models have been pervasive in many branches of science ranging from difficult ones like rocket science and brain surgery to easy ones like image and speech recognition. Regardless of ease and orthogonality of these skills, structures and modularity of these networks remains remarkably homogeneous. Homogeneity of neural networks is brought about by handful (less than 200) of operators. In this paper, we discuss design, features and challenges in implementation of neural network operators. These operators are part of larger open source project Deep Neural Network Compiler currently under development [4]. dnnCompiler is targeted towards devices with small formfactor like microcontrollers, which are part of all sorts of household devices: think appliances, cars, and toys. In fact, there are around 30 billion microcontroller-powered devices produced each year. They're cheap, require very little energy, and are very reliable.

In section II, we discuss overall design of dnn compiler with emphasis on operators. A small representative sample of operators was chosen for discussion. Section III after that discusses fusion, quantization, memory layout and scalability of the operators and their design. In the end, we present summary and future work of our project.

OVERVIEW

Popular deep neural networks with high accuracy have a high need for computational power for inferencing, that is generally not available on devices with low memory and low-end CPUs. Industries spend a lot of monetary resources on the hardware. Addressing this particular issue, we came up with a solution– the Deep Neural Network Compiler. The compiler turns neural networks into an executable bundled with model parameters ready to run on embedded devices such as the raspberry pi, odroid, arduino, risc-V and other controllers with small form factor. The design of the Deep Neural Network Compiler is pretty straight forward, it uses ONNX3.0 [2] as a ProtoBuf format, which is directly translated into a high-level compute graph with operators as nodes and data flowing through these nodes as tensors. This paper attempts to highlight operator design supported by ONNX3.0 format [2]. These operators are being writing in C++ for performance reasons. They are also ported to python interface for quick testing and possibility of tuning compiler in to a full framework with python interface. To achieve performance objectives, we use a third-party linear algebra library Eigen [1] available under Mozilla Public License. This is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms to perform the Neural Network operations in a fast and an efficient manner.

II. DESIGN

Main difference between neural nets' computation graph and traditional compiler IR graph is the number of high-level operators. Traditional compiler has a small and fixed set of operators, whereas neural nets' computation graph operators are over 200 and new are being added every day. This dictates our design choice of adding new operators decentralized and independent of overall compiler design [4]. Addition of new operator is likely to invariably give rise to optimization opportunities not possible older set. This requirement demands flexibility in design optimizations passes of high-level computation graphs, since addition of new nodes may demand new passes. In the next section, we discuss 3 operators as a sample of approximately 150 operators supported in ONNX3.0 [2].

Design of operators in deep neural network compiler plays an important role in performance, memory, scalability and usability of DNN compiler. Addition of new operators sometimes comes with new attributes. In the overall scheme of things registration of new attribute must not come at the expense of invalidation of supported operators set or optimization passes. Similar to Tensorflow (with OpDef) and Caffe2 (with OperatorSchema), we also plan to provide registration scheme in future versions.

Operator Instance Normalization

Operator Instance Normalization [3] is expressed as

$$y = scale.\frac{x - mean}{\sqrt{variance + epsilon}} + B$$

where mean and variance are computed per instance per channel (C).

Inputs
a [float,double]: ND tensor (Nx Cx D1xD2…Dk)
Scale: 1D vector of dimension C.
B: 1D vector of dimension C.
Attr: epsilon – float

In case variance goes to zero and to avoid division by zero.

The formula for Mean is given by:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

This can be calculated in a single pass through all the elements.

The formula for Variance is given by:

$$Var(X) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$$

According to this Mean of the elements in channel is prerequisite for Variance calculation.

A bit of mathematics reveals that mean is not required for Variance they can be calculated simultaneously.

$$\begin{aligned} Var(X) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2 &= \sum_{i=1}^{n}(x_i^2 - 2\mu x_i + \mu^2) \\ &= \frac{1}{n}\sum_{i=1}^{n} x_i^2 - \frac{2\mu}{n}\sum_{i=1}^{n} x_i + \frac{n\mu^2}{n} \\ &= \left(\frac{1}{n}\sum_{i=1}^{n} x_i^2\right) - \mu^2 \end{aligned}$$

And this formulation became part of dnn compiler operator implementation. The operator is O(n) where n = Number of elements in the tensor = $N*C*D_1\ldots*D_k$.

Algorithm

*a.reshape([N,C,D_1*D_2…..*D_k]);*
channel_size = ND_1*D_2…..*D_k*
sum=sq_sum=0
For channel in channels
  For element in channel
    sum=sum+element
    *sq_sum=sq_sum+element*element*
  end
  mean = sum/channel_size
  *var = sq_sum - mean*mean*
  For element in channel:
    element = scale[channel](element - mean)/sqrt(var+epsilon)+B[channel]*
  end
end

Operator Gemm

Gemm is called **Ge**neral **M**atrix **M**ultiplication, is given as

$$Y = \alpha AB + \beta C$$

Where A and B can optionally be transposed or hermitian-conjugated inside the routine and all three matrices may be strided. The ordinary matrix multiplication A B can be performed by setting α to one and C to an all-zeros matrix of the appropriate size.

Inputs

A of shape(M,K) if transA=0 or (K,M) if transA!=0

B of shape(K,N) if transB=0 or (N,K) if transB!=0

C of shape (M,N)

Attributes

transA (int) : defaut 0

transB (int) : defaut 0

alpha (float) : defaut 1.0

beta (float) : defaut 1.0

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mp} \end{pmatrix}$$

The matrix product C = AB (denoted without multiplication signs or dots) is defined to be the n × p matrix

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{np} \end{pmatrix}$$

Such that

$$c_{ij} = a_{i1}b_{1j} + \cdots + a_{im}b_{mj} = \sum_{k=1}^{m} a_{ik}b_{kj},$$

Gemm is general matrix multiplication, which takes 4 attributes, alpha and beta of type float, transA and transB of type int. transA and transB shows if matrix A and matrix B needs transposing before the operation. We have used Eigen matrix multiplication to achieve this operator. All tensor ranks should be of rank 2. Now apply this formula to compute the output Y

*Y = alpha * A' * B' + beta * C]*

where A' = transpose(A) if transA else A

B' = transpose(B) if transB else B

Outputs

matrix Y of shape (M,N), the same shape of the input tensor.

Algorithm

IF transA == 1
  a = a.T
IF transB == 1
  b = b.T
*Y = alpha * (a * b) + beta * c*

Time complexity is less than $O(n^3)$ because Gemm is very data parallel as each of the n^2 output elements is independent of the rest.

Operator LpNormalization

LpNormalization is a matrix normalization technique that takes axis and a constant p as an attribute which are of type int respectively. Axis defines in which particular axis of matrix we have to apply the normalization of the matrix and value of p provides the information of norm to be used. Only 1 or 2 is supported as value for p.

Inputs

ND matrix (type-float tensor, double tensor)

Attributes

p (int): default 2 (onnx supports only 1 or 2)

axis (int): default -1 (last axis i.e. 1 since axis of int type)

For Normalization of matrix as per the p value taken as attribute will require computation of L1 norm (p=1) or L2 norm (p=2) and formula to compute both is given by:

L1 norm:

$$||x||_1 = \sum_i |x_i| = |x_1| + |x_2| + \ldots + |x_i|$$

L2 norm:

$$||x||_2 = \sqrt{\left(\sum_i x_i^2\right)} = \sqrt{x_1^2 + x_2^2 + \ldots + x_i^2}$$

Normalization helps in scaling of the input data and features.

Algorithm:

For elements along the axis 0 or 1:
If p==1
For element in matrix:
sum=0
sum=sum+abs(element)
End
For element in matrix:
element=element/sum
End
Else If p==2:
For element in matrix:
sum=0
*sum=element*element*
End
For element in matrix:
element=element/sqrt(sum)
End
End
Outputs

ND matrix (type-float tensor,double tensor) after applying LpNormalization.

Time complexity of the algorithm is $O(n^2)$ where n is no. of elements in the input matrix.

Operator SoftMax

SoftMax is a function that takes as input a vector of *K* real numbers, and normalizes it into a probability distribution consisting of *K* probabilities proportional to the exponentials of the input numbers. That is, prior to applying SoftMax , some vector components could be negative, or greater than one; and might not sum to 1; but after applying SoftMax , each component will be in the interval {0,1}, and the components will add up to 1, so that they can be interpreted as probabilities. Furthermore, the larger input components will correspond to larger probabilities. SoftMax is often used in neural networks, to map the non-normalized output of a network to a probability distribution over predicted output classes.

Inputs -

The input tensor that's coerced into a 2D matrix of size (ND)

Output:

The output values with the same shape as input tensor.

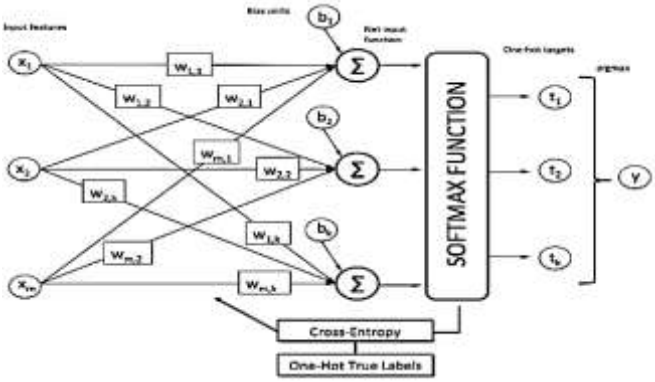
Attributes:

axis(int):default is 1

Formula:

$$softmax(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Diagrammatically:



Algorithm:

We apply the standard exponential function to each element Zi of the input vector Z and normalize these values by dividing by the sum of all these exponentials; this normalization ensures that the sum of the components of the output vector f(Z) is 1. (where f=sigmoid function).

If axis==1:
For row in matrix:
sum=0
For col in matrix:
sum=sum+exp(matrix[row][col])
End
For col in matrix:
matrix[row][col] =(exp(matrix[row][col]))/sum
End
End
If axis==0:
For col in matrix:
sum=0
For row in matrix:
sum=sum+exp(matrix[col][row])
End
For row in matrix:
matrix[row][col] =(exp(matrix[col][row]))/sum
End
End

III. FEATURES

Operator Fusion

Operator fusion [7] is a popular technique to minimize the off-chip data movement between layers by re-organizing two or more adjacent operators into one. DNNC employs loosly typed typed base operator with generic type information in the compute graph. This design makes it easy to fuse two or more operators painlessly with trivial operator type, tensor shape and rank-checking.

A. *Quantization*

Quantization, as introduced in [6] is a popular technique for deep learning models to help reduce power consumption, lower memory bandwidth, lower storage and higher performance. Low precision (normal precision uses FP32) can be used here, which uses FP16, INT8 and so on. Mixed precision utilizes both FP32 and FP16 model. And mixed precision is used as not all parameters or operators can be formatted in FP16, to maintain accuracy. No matter the data type, the DNNC can easily convert them, or we can change them manually also, as DNNC uses C++ template as the building block of its operators.

B. *Memory Layout*

There are many ways to represent the memory of a given tensor in the computational graph. The most common data layout choices are column major and row major [1]. Performance of DNN largely depends on memory operations like, fetch, store, transportation and others. As the old saying goes, a chain is as strong as its weakest link, memory or data tiling or alignment of a network dominates the performance of DNNs as measured by latency and throughput. We rely on Eigen [1] to improve tensorization, reduce temporaries, memory accesses and cache misses using latency hiding, scheduling and other optimizations.

C. *Scalability*

The operators are implemented with Eigen Library in C++, for performance reasons and memory optimization across different devices. The interface is done with swig, which helps us to build a C++ backend with python front end, for easier testing and wider usage. And it is fairly easy to add new operators as required. Add the operator in C++ which can be extended from baseOperator package and connect it with swig, which then can be accessed with python.

SUMMARY

By bringing deep learning models to tiny microcontrollers, we can boost the intelligence of billions of devices that we use in our lives, without relying on expensive hardware or reliable internet connections. Imagine smart appliances that can adapt to your daily routine, intelligent industrial sensors that understand the difference between problems and normal operation, and magical toys that can help kids learn in fun and delightful ways.

ACKNOWLEDGMENT

We acknowledge the efforts by predecessor projects like Git, GitHub, Eigen, ONNX, LLVM and many others to make DNNC project a reality.

REFERENCES

[1] Guennebaud, G., Jacob, B., et al.: Eigen v3 (2010). http://eigen.tuxfamily.org

[2] Bai, Junjie and Lu, Fang and Zhang, Ke and others, ONNX 3.0: Open Neural Network Exchange, 2019, https://github.com/onnx/onnx

[3] D. Ulyanov, A. Vedaldi, and V. Lempitsky. Instance normalization: The missing ingredient for fast stylization. arXiv preprint arXiv:1607.08022, 2017.

[4] Rohit Sharma et. Al, DNNC: Deep Neural Network Compiler, 2019 https://github.com/ai-techsystems/dnnCompiler

[5] Gemm https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms#Level_3

[6] E. Fiesler, A. Choudry, and H. J. Caulfield, "Weight discretization paradigm for optical neural networks," in Optical Interconnections and Networks, H. Bartelt, Ed. SPIE, Aug 1990: https://doi.org/10.1117/12.20700

[7] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer, Efficient Processing of Deep Neural Networks: A Tutorial and Survey, Proceedings of the IEEE. 2017