International Journal of Research in Engineering, Science and Management
Volume-2, Issue-5, May-2019
www.ijresm.com | ISSN (Online): 2581-5792

A Study on Compiler Applications in Neural Networks

Riya R. Ganiga[1], Shaguftha Zuveria Kottur[2], Tallapalli Surabhi[3], A. Parkavi[4], Sini Anna Alex[5]
[1,2,3]*Student, Department of Computer Science and Engg., Ramaiah Institute of Technology, Bangalore, India*
[4,5]*Professor, Department of Computer Science and Engg., Ramaiah Institute of Technology, Bangalore, India*

***Abstract*: Compilers are used to translate between languages or representations. In neural networks, the input is usually in the form of a compute graph with tensor computations associated with nodes which needs to be translated into executable. The compiler plays an important role in this conversion, performing optimizations and lowering. It can be used as a bridge to target multiple hardware architectures from multiple frontends and hence is a major component for scalability of neural network frameworks. Also, the optimizations and conversions done by a compiler lead to reduction in time taken to train a particular network and its implementation.**

***Keywords*: compiler, neural network, scalability, frameworks, hardware architecture**

1. Introduction

Code readability, ease of construction, time and space complexity and size of the code are some of the important features of a program code. The simpler a code is for humans to read, the more difficult it is for the computer to know what it is supposed to do. Compilers translate a high-level programming code to low-level machine understandable language.

This translation is time consuming and hence needs to be optimized. Code optimization is one technique in which a code is enhanced to decrease the time and space consumption, resulting in efficiency and lower resource utilization. It is an important phase in engineering design and improves the quality of the designed product elevating the value of the product. Optimization of code can be done in the coder or by the compiler (compiler optimization). Optimization by compiler might involve several methods and techniques like machine-independent optimization, machine-dependent optimization, basic-block, loop optimization, dead-code elimination and partial redundancy. This paper analyses the application of code optimization capability of compilers into neural networks and deep learning concepts.

Neural networks are computing systems made up of layers of simple, interconnected processing elements having the capability to process information dynamical through response to external inputs. The layers of a neural network can be broadly classified into input layer, output layer and central hidden layer which in turn can have many layers. All the computational complexity comes from the advancements in the hidden layer which is the heart of the network. The hidden layers perform computations on the weighted inputs and produce net input which is then applied with activation functions to produce the actual output. Compiler optimization techniques are applied to enhance the working of the hidden layer to make the network time and resource efficient. The faster a machine is, the better, hence one would want to speeden the process a neural network would process data and optimization is exactly what is to be done. For example, development of new layers and architecture is time consuming and optimizing these developments would result in a huge cut down of processing time. Neural networks also termed as artificial neural networks (ANN) are of many types - convolution neural networks(CNN), feed-forward neural networks(FNN), recurrent neural networks(RNN), multi-layered perceptron, etc.

Deep learning is a machine learning technique that teaches the machine what to do and the machine is trained to devise a method on how to do it. Deep learning is a Neural Network consisting of a hierarchy of layers, whereby each layer transforms the input data into more abstract representations. Deep learning software demands reliability and performance. Deep neural networks are a class of models that use a system of interconnected neurons to estimate or approximate functions. The performance and integration with existing systems of a deep learning network can be boosted up by the application of compiler designer techniques on it.

2. Literature survey

A key difference between compilers for languages like C++ and compilers for deep learning frameworks is that with deep learning, the data being operated on is large and variable-sized, but highly amenable to parallelization.

In [1] Leonard Truong et al., introduce a domain specific language (DSL) called Latte which allows the user to define the deep neural network in a highly abstract form without sacrificing the efficiency and performance. Latte acts as an extension to the Julia language. In Latte, a neural network is defined as an ensemble of neurons with connections (signifying data dependencies) between them. All neurons in an ensemble are of the same type and thus use the same activation function. The Latte compiler has four phases: analysis of shared variables is done so that shared data is loaded together into a single shared

buffer for inputs which reduces memory consumption, synthesis which has three parts, dataflow, compute and distributed memory communications, optimizations are done in two phases, intra layer optimizations include library kernel pattern matching and loop tiling, and cross layer fusion after which parallelization is performed using Parallel Accelerator.jl package and the last phase of the compiler is code generation. The runtime using uses a two level hierarchical design for training to employ data parallelism called intra node level data parallelism and cluster level data parallelism. Single node evaluation and cluster evaluation was performed to demonstrate Latte's ability to outperform standard approaches in micro-benchmarks as well as on popular neural network models which can be majorly be attributed to parallelism and cross layer optimizations.

Deep learning virtual machine (DLVM) [2] is a compiler infrastructure which aims to simplify development of neural network DSLs. The basic procedure is to convert the DSL into the DLVM IR and then pass it to the DLVM which uses a mature compiler infrastructure (currently LLVM) for code generation. The DLVM IR has a virtual instruction set, control flow graph and data flow representation. The virtual instruction set includes domain specific primitive math operations and general purpose instructions. A multi stage compiler optimization strategy is employed which addressee's algorithmic differentiation, domain and general purpose optimizations and static code generation in order to target a variety of computer architectures. The reason for converting DLVM IR to LLVM IR is that being a mature compiler infrastructure, it can support multiple backends. DLVM also provides a command line interface as any industry standard compiler.

Field programmable gate arrays (FPGA) are integrated circuits which are customizable by the consumers and hence can be used to implement convolutional neural networks. Customizing FPGA design for a specific application is time consuming.

[3] presents a software programmable hardware overlay called Deep Learning Accelerator(DLA). Their design uses lightweight very long instruction word(VLIW) instructions to reprogram the overlay to support various CNN models. The VLIW network consists of kernels connected to a ring network via which the instructions are distributed to them by the VLIW reader that continuously fetches instructions from external memory. The graph compiler used first slices the neural network graph into subgraphs (list of functions that do not require writing to the buffer except at the end) which is enhanced by slicing multiple sequential slices together to reduce number of external memory spill points. The stream buffer is used as a double buffer in order to reduce fragmentation and allocation pass is done to calculate the read and write addresses for each slice. Next subgraph scheduling is done using priority queue approach where cost of execution of a node is the ratio of its output size to effective input size.

A library based CNN RTL compiler is used in [4] to reduce the effort required to design a custom FPGA. the input to the compiler are the dimensions and connections of CNN layers and pertained weights which are then transformed into a layer by layer schedule. The schedule also determines the read and write orders of kernel weights stored in memory which are then sorted to control access of external memory. The RTL library consists of modules designed for different types of layers which can be configured with CNN parameters. These modules are hand coded Verilog templates built on optimized CNN acceleration strategy. Thus, the layer by layer execution, hardware computing architecture and memory transactions can be customized for different CNN algorithms.

Due to the emergence of many novel topologies, frameworks require a lot of changes to be compatible with the new backends. [5] introduces the open source nGraph library which acts as a middle layer between the frameworks and backends. The nGraph IR is a DAG with each node capable of having multiple inputs and outputs. these nodes operate on tensors and can have additional attributes which can alter their behaviour. It includes framework bridges which convert the graph used by the specific frameworks into the nGraph IR which is then transformed into code which can be run on a specific backend using the corresponding transformer.

Another compiler technique which performs a similar functionality of targeting various back ends from various frontends is Glow (Graph-lowering) [6]. The aim is to encourage research on efficient hardware accelerators and using Glow for automation of compilation tasks. Glow uses multiple IRs for graph lowering. The high level IR is optimized using domain independent optimizations, then differentiated and lowered to allow additional optimizations. IRGen then converts it into instructions on which low level IR optimizations are performed using LLVM, utilizing its existing capability to target various hardware architectures, similar to [2]. To decrease memory usage, Glow performs quantization by using tensors like Int8. Also, profile guided quantizations are applied in two phases: first is attaching special profiling nodes to record activation ranges in the network, optimize the network and then run interference, second is to obtain quantized form by recompiling the network which allows static optimizations.

All of these different frameworks are primarily based on compilers which can provide the abstraction required for different languages and the compatibility to specific hardware. Compiler optimization is altering features of executable code while considering the output of the compiler. Ballal et at in [7] discuss a genetic algorithm approach to determine the combination of compiler(GNU) flags, to control optimization, which could be used to increase time efficiency of the executable. The compiler flags are considered as genes and a combination of these is a chromosome. An inverse function of sum of execution time and compilation time is taken as the fitness function and is different for every chromosome. Selection, crossover and mutation over generations on this

initial population of chromosomes would result in a set of flags that are optimum in terms of time. At each generation, the chromosomes corresponding to a maximum fitness are selected for cross-over for the next generation. The procedure repeats until the best set of genes i.e., an nearly optimal set of flags that boost the time efficiency of compilation is achieved. This approach of genetic algorithm turns out to be helpful in case of a large solution space. Also, the larger the gene pool, higher are the chances of achieving the optimal/idle fitness for the program.

James et al in [8] describes Theano, a mathematical expression in Python which combines NumPy syntax with the speed of an optimized machine language. In the first phase, Theano optimizes the choice of expressions which is then translated to C++ (or CUDA for GPU), later it compiles them into dynamically loaded Python modules. It is applied to multi-layer perceptron (MLP) and a convolutional network. Compilation function of Theano involves several stages like canonicalization, stabilization, specialization, optional GPU transfer and code generation. It was found to be 1.6x to 7.5x faster than C/C++, NumPy/SciPy and MATLAB when compiled for the CPU and 6.5x to 44x faster when compiled for the GPU when implemented to machine learning algorithms. Though it is much faster it is not optimal. It can handle graphs with only thousand nodes and does not cover all functionalities of NumPy and few features of SciPy. [9] shows how theano can be used by taking the example of logistic regression and describes the content and scope of the deep Learning Tutorials, showing how Theano can be used for deep learning by making them fast, elegant and compact.

Rudy et al in [10] have formulated a framework which considers only the correctness of targeted input distributions and adapts programs to increase its efficiency. The model being specified includes a controller and a machine. The controller takes charge of what should be executed and machine follows the commands of the controller. The models aims at finding the best set of weights in order to perform a correct input to output mapping. Metrics considered are correctness, halting, confidence and efficiency. The model uses addition of softmax layer for reformulation of a constrained problem to an unconstrained one. The input program is being converted to an intermediate representation and then to a weight matrix which is used to do the mapping. The model turned out to be good in finding efficient solutions for simple programs, but found only close to optimal solutions for complex programs. The model also carries forward the drawbacks of gradient descent method (which it uses for optimisation) i.e., it performs only local transformations.

While handling advanced software and hardware an automatic and accurate optimization heuristics are highly required. These heuristics can be learnt by the model by machine learning technique by training it with the help of features/parameters that are hard coded by experts by their knowledge, trial and error methods. Thus the accuracy of the model depends on the features that are chosen for optimisation. In [11] the authors proposed an advanced method for formulating the optimised heuristics over raw data, without using features. Neural networks are used for analysis the code and to find the optimal heuristics where there is no human intervention. During compilation/execution of code, decisions are made based on hand coded heuristics thus the performance and accuracy of the output depends on the heuristic that are hard coded.

Deep tune is an end to end machine learning approach for optimising the heuristics with deep neural networks. It has following phases,

Source Rewriter: Source normalizing transformations are applied on the raw code to generate a consistent code structure.

Sequence Encoder: Source code is encoded with integers which is an index of predetermined hybrid vocabulary (character and token)

Embedding: From sequence encoding the relation between tokens cannot be interpreted so we translate the semantically related tokens into a lower dimensional vector space.

Sequence Characterization: This phase is equivalent to manual feature extraction. Long Short Term Memory (LSTM) is a two-layer network, inputs the embedded vectors and outputs a single output vector characterizing the given sequence.

Heuristic model: The former layer has 32 neurons and the latter one has a single neuron for each heuristic decision. Finally, whichever neuron has the more activation is considered to the optimal heuristic.

3. Conclusion

Deep learning implementations require a high level of abstraction to separate developers from low level design. This abstraction is provided by highly efficient compilers which either work with specific DSLs or can be customized for any DSL. Since the structure of neural networks can be represented by graphs, compilers need to translate these graphs representing tensor computations into executable which is usually done by generating multiple IR, where on each IR optimizations and lowering can be done. This paper explored various ways in which compilers are used in the deep learning community from implementing specific DSLs to use in frameworks to simplifying the implementation by mapping onto FPGA. For the former it resolves scalability issues by acting as an intermediate between frontends and backends and for the latter, it aids in faster training and implementation of neural networks to make use of the efficiency and reprogrammability of field programmable gate arrays. Certain compiler optimizations and neural network frameworks were also reviewed to understand the current approaches used.

References

[1] Truong, L., Barik, R., Totoni, E., Liu, H., Markley, C., Fox, A. and Shpeisman, T., 2016. Latte: a language, compiler, and runtime for elegant

and efficient deep neural networks. *ACM SIGPLAN Notices*, *51*(6), pp.209-223.

[2] Wei, R., Schwartz, L. and Adve, V., 2017. DLVM: A modern compiler infrastructure for deep learning systems.

[3] Abdelfattah, M.S., Han, D., Bitar, A., DiCecco, R., O'Connell, S., Shanker, N., Chu, J., Prins, I., Fender, J., Ling, A.C. and Chiu, G.R., 2018, August. DLA: Compiler and FPGA Overlay for Neural Network Inference Acceleration. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)* (pp. 411-4117). IEEE.

[4] Ma, Y., Cao, Y., Vrudhula, S. and Seo, J.S., 2017, September. An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)* (pp. 1-8). IEEE.

[5] Cyphers, S., Bansal, A.K., Bhiwandiwalla, A., Bobba, J., Brookhart, M., Chakraborty, A., Constable, W., Convey, C., Cook, L., Kanawi, O. and Kimball, R., 2018. Intel nGraph: An intermediate representation, compiler, and executor for deep learning.

[6] Rotem, N., Fix, J., Abdulrasool, S., Deng, S., Dzhabarov, R., Hegeman, J., Levenstein, R., Maher, B., Nadathur, S., Olesen, J. and Park, J., 2018. Glow: Graph lowering compiler techniques for neural networks.

[7] Ballal, P.A., Sarojadevi, H. and Harsha, P.S., 2015. Compiler optimization: A genetic algorithm approach. *International Journal of Computer Applications*, *112*(10).

[8] Bergstra, James & Breuleux, Olivier & Bastien, Frederic & Lamblin, Pascal & Pascanu, Razvan & Desjardins, Guillaume & Turian, Joseph & Warde-Farley, David & Bengio, Y. (2010). Theano: A CPU and GPU math compiler in Python. Proc. Of The 9th Python in Science Conf.

[9] Bergstra, James, et al. "Theano: Deep learning on gpus with python." *NIPS 2011, BigLearning Workshop, Granada, Spain*. vol. 3. Microtome Publishing., 2011.

[10] Bunel, Rudy R., et al. "Adaptive neural compilation." *Advances in Neural Information Processing Systems*. 2016.

[11] Cummins, Chris, et al. "End-to-end deep learning of optimization heuristics." 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT). IEEE, 2017.