DLIR: An Intermediate Representation for Deep Learning Processors

Huiying Lan, Zidong Du

DLIR: An Intermediate Representation for Deep Learning Processors

Huiying Lan[1 2 3] and Zidong Du[1 3]

[1] Intelligent Processor Research Center, Institute of Computing Technology (ICT), CAS, China
[2] University of Chinese Academy of Sciences (UCAS), China
[3] Cambricon Tech. Ltd.

Abstract. The Deep learning processor (DLP), especially ASIC-based accelerators, have been proved to be a promising device for accelerating the computation of deep learning algorithms. However, the learning cost of mastering these DLPs is high as they use different programming interfaces. On the other hand, many deep learning frameworks are proposed to ease the burden of developing deep learning algorithms, but few of them support DLPs. Due to the special features in DLPs, it is hard to integrate a DLP into existed frameworks.
In this paper, we propose an intermediate representation (called DLIR) to bridge the gap between DL frameworks and DLPs. DLIR is a tensor-based language with built-in tensor intrinsics that can be directly mapped to hardware primitives. We show that DLIR allows better developing efficiency and is able to generate efficient code.

Keywords: deep learning processor, intermediate representation, deep learning framework, deep learning

1 Introduction

Deep learning processors (DLPs) have become powerful devices for processing large scale neural networks, especially ASIC-based DLPs [1–6]. However, DLPs are still not fully accepted by DL participants due to the lack of programming supports. On the other hand, many DL programming frameworks [7–10] have been proposed to ease the burden of developing DL algorithms but often only on traditional devices (e.g., CPUs and GPUs). Primitives on such devices are basically scalar computations and they use cache in their system. Therefore, frameworks designed for such devices are often lower operators to fine-grained operations and completely ignore the management of on-chip memories. For example, TVM [11] is a software stack for deep learning, which leverages Halide IR to present computation loops and extracts several useful scheduling primitives to allow users to manually optimize the computation. However, TVM require the user to describe the computation through scalar operations and use *tensor_intrinsics* scheduling primitive to map the tensor operation to instructions in the backend DLP (which is VTA in the case of TVM). This complicates the

programming of the DLP as the code describing the computation of the tensor intrinsics is completely unnecessary. XLA is a recent proposed backend embedded in TensorFlow to provide subgraph optimizations. It proposes an High-level optimizer (HLO) and also with an IR to represent the computation graph received from the TensorFlow frontend. Although XLA provides tensor semantics that in a way match DLP primitives, operators in XLA is very high-level and does not provide hardware-specific operations such as memory copying between main memory and on-chip scratchpad memory which is extensively used in DLPs. Such frameworks lack necessary components to seamlessly support a DLP. Therefore, an indirection layer that is specifically designed for DLPs is on demand to bridge the gap between frameworks and DLPs.

Our solution is an indirection layer composed of an intermediate representation (called DLIR), a compiler and runtime. DLIR is a tensor-based IR, inherently support tensor types (neurons and synapses) and tensor intrinsics (e.g., convolution, pooling, matmul) that can be directly mapped to hardware primitives. By leveraging such structures, DLIR compiler is able to generate highly efficient code that is comparable to hand-optimized instructions.

2 Intermediate Representation Language

In this section, we introduce the intermediate representation language, i.e., DLIR, which can be interpreted into operations supported by DLPs. In order to reduce the learning costs, DLIR is designed to be embedded in C++ as a library. It can be directly called by front-ends functions and generate instructions for backend.

2.1 Data Structure

DVIR defines two N-dimensional (N-D) tensor data types, *Neuron* and *Synapse* to encapsulate data and be used as operands of HLIR operators (see Section 2.2). Both types are defined using a built-in data structure, *Dimension*, which helps specify the tiling of a dimension. Due to the limited on-chip resources, an N-D array often needs to be partitioned into several segments to fit into on-chip buffers. Computation partitioning on DLPs is complicated as there are multiple dimensions for a N-D tensor. A dimension with size d can be tiled as $d = n \times s + r$, which requires at least three variables to describe the partitioning. *Dimension* is introduced to encapsulate these variables. By iterating through combinations of segments of different dimentions of a tensor, we are able to traverse all possible segments in the tiled tensor. In addition, to enbale explicit memory copy between the main memory and on-chip buffers, we provide two data structures, i.e., *NeuronBuffer* and *SynapseBuffer*, to represent allocated data on on-chip buffers. A segment in *Neuron* will be transferred to a corresponding *NeuronBuffer*.

2.2 Operators

We classify the programming supports of current DLPs according to whether they require programmers to manually write tiling and computation partitioning

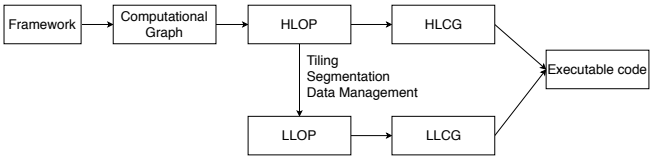

Fig. 1. Compilation process of DLIR

within a layer, i.e., the ability to process arbitrary scale of computation. We call the code generator (CG) provided by DLPs as high-level code generator (HLCG) and low-level code generator (LLCG). HLCG refers to CGs that can process arbitrary scale of computation, e.g., CG of DianNao and ShiDianNao. LLCG refers to CG that can compile programs at the level of assembly or ISA, e.g., Cambricon ISA. Accordingly, we provides two levels of operator that can map to these two CGs, i.e., high-level operators (HLOP) and low-level operators (LLOP). Therefore, both types of CG can be integrated into DLIR.

These two levels of OP are also corresponding to the data structures. HLOP takes *Neuron* or *Synapse* as input and output parameters, and LLOP takes segments in a *Neuron* or *Synapse* as input and output parameters. Both HLOP and LLOP can be translated directly into hardware-specific assembly languages or instructions by invoking HLCG and LLCG.

In addition to directly invoke vendor-provided CG to generate code, HLOPs can also be first interpreted to LLOPs, and then translated to instructions. With such transformation, DLPs with LLCGs can also use HLOPs as the official programming interface which is typical for DL frameworks.

3 Compilation

The compilation process is shown in Figure 1. Operations in the computational graph can be mapped to HLOPs. For DLPs using HLCGs, DLIR passes the parameters to HLCGs to generate executable code. For those using LLCGs, DLIR will invoke the HLOP defined with LLOPs and memory operations to generate LLOP sequences, which will then be compiled by the LLCGs. In the function that defines HLOPs by LLOPs, users need to specify loop tiling, data segmentations and the use of on-chip buffers in such functions. In addition, as DLPs have strict restriction on data layout, the compiler will rearrange a tensor according to the dimension informations so that the required data can be sequentially fetched.

4 Evaluation

We use Caffe as the front-end as it is a commonly used DL framework. We reimplement *Setup*, *Forward* and *Backward* functions in the layers in Caffe. Each call of these functions will invoke the DLIR compiler to generate an instruction sequence that will be transferred to our backend and executed. We use Cambricon as the backend, as it is a state-of-the-art ISA and architecture proposed for NN algorithms, and it involves many representative features of DLPs.

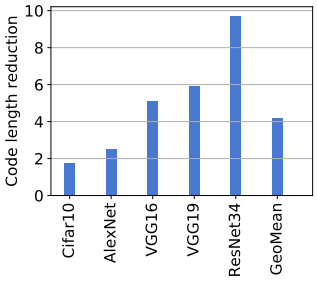

Fig. 2. Code length reduction of using DLIR compared to hand-written code.

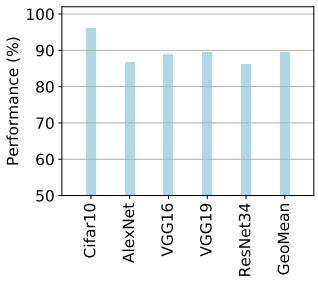

Fig. 3. Performance of DLIR compared to hand-optimized code.

4.1 Developing Efficiency

We evaluate the developing efficiency of DLVM on five large realistic networks, i.e., Cifar10, AlexNet, VGG16, VGG19 and ResNet34, covering five representative algorithms (convolution, pooling, fully-connected, batch normalization, and local respond normalization) used in popular deep learning networks. Figure 2 shows that by using DLIR, we can reduce the source code by 4.19 × on average. The highest reduction comes from ResNet34 (i.e, 9.72×), and the lowest reduction comes from Cifar10 (i.e., 1.75×). The more layers a network composed of, the higher the reduction ratio is. Because the code reduction is primarily gained from eliminating redundant implementations of the same algorithm with different scales.

4.2 Performance

We evaluate the performance of DLVM on the mentioned networks to show that DLIR is able to generate efficient code. The performance is demonstrated in Figure 3. DLIR achieves 89.37% performance compared to that of hand-optimized code on average. The performance loss primarily comes from the missing overlapping between computations and memory accesses especially between layers and the memory accesses saved by layer fusion. However, the hand-optimized code could takes seasoned programmers days to maximize the optimize. In DLIR, we mostly concern about usability instead of performance, therefore this performance loss is acceptable for us.

5 Conclusion

In this paper, we propose an intermediate representation (DLIR) to bridge the gap between high-level DL frameworks and DLPs. DLIR is composed of an intermediate representation language with special designed data structures (i.e., *Dimension*, *Neuron* and *Synapse*), hierarchic operators and memory operations. By leveraging DLIR, we are able to shorten the code by 4.19 × on five large networks on average. In addition, the compiler is able to generate code with up to 89.37% performance compared to hand-optimized code using Cambricon as the backend.

6 Acknowledgement

This work is partially supported by the National Key Research and Development Program of China (under Grant 2017YFA07009022017YFB1003101), the NSF of China (under Grants 6147239661432016, 61473275, 61522211, 61532016, 61521092, 61502446, 61672491, 61602441, 61602446, 61732002, and 61702478), the 973 Program of China (under Grant 2015CB358800), National Science and Technology Major Project (2018ZX01031102) and Strategic Priority Research Program of Chinese Academy of Sciences (XDBS01050200).

References

1. Chen, T., Du, Z., Sun, N., Wang, J., Wu, C.: DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In: Proceedings of the 19th international conference on Architectural support for programming languages and operating systems (ASPLOS), Salt Lake City, UT, USA (2014) 269–284
2. Chen, Y., Luo, T., Liu, S., Zhang, S., He, L., Wang, J., Li, L., Chen, T., Xu, Z., Sun, N., Temam, O.: DaDianNao: A Machine-Learning Supercomputer. In: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47). (2015) 609–622
3. Zhang, S., Du, Z., Zhang, L., Lan, H., Liu, S., Li, L., Guo, Q., Chen, T., Chen, Y.: Cambricon-X : An Accelerator for Sparse Neural Networks. In: Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49). (2016)
4. Liu, D., Chen, T., Liu, S., Zhou, J., Zhou, S., Temam, O., Feng, X., Zhou, X., Chen, Y.: Pudiannao: A polyvalent machine learning accelerator. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015. (2015) 369–381
5. Du, Z., Fasthuber, R., Chen, T., Ienne, P., Li, L., Luo, T., Feng, X., Chen, Y., Temam, O.: Shidiannao: shifting vision processing closer to the sensor. In: Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015. (2015) 92–104
6. Liu, S., Du, Z., Tao, J., Han, D., Luo, T., Xie, Y., Chen, Y., Chen, T.: Cambricon: An instruction set architecture for neural networks. In: 43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016. (2016) 393–405
7. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: A system for large-scale machine learning. (2016) 18
8. Collobert, R., Kavukcuoglu, K., Farabet, C.: Torch7: A matlab-like environment for machine learning
9. System., N.: github.com/nervanasystems/neon. (2016)
10. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: Convolutional architecture for fast feature embedding. arXiv preprint arXiv:1408.5093 (2014)
11. Chen, T., Moreau, T., Jiang, Z., Shen, H., Yan, E.Q., Wang, L., Hu, Y., Ceze, L., Guestrin, C., Krishnamurthy, A.: TVM: end-to-end optimization stack for deep learning. CoRR **abs/1802.04799** (2018)