A Deep Learning Compiler for Vector Processor

Pingping Pan[1], Jun Wu[2(✉)], Songyuan Zhao[1], Haoqi Ren[1], and Zhifeng Zhang[1]

[1] Department of Computer Science, Tongji University, Shanghai, China
pppgary@163.com,
{1830835, renhaoqi, zhangzf}@tongji.edu.cn
[2] School of Computer Science, Fudan University, Shanghai, China
wujun@fudan.edu.cn

Abstract. The technical route of machine learning compiler generally refers to the application of automatic or semi-automatic code generation in the optimization process instead of hand-optimization. This paper presents a deep learning compiler (DLCS) for target vector processor based on LLVM framework, which lowers deep learning (DL) models to an intermediate representation (IR) of two levels. The high-level IR realizes target-independent optimizations including kernel fusion, data replacement and data simplification, while the low-level IR allows the compiler to perform target-dependent optimizations, such as Eight-Slots VLIW and special intrinsic function. The proposed compiler customizes the architecture description of target vector processor to achieve a high-quality automatic code generation. We evaluate the performance comparison between DLCS and hand-optimization when deploying ResNet-18 model and MobileNet model to the target vector processor. Experimental results show that DLCS offers Multi-slot parallel performance for target vector processor and achieves speedups ranging from 1.5× to 3.0× over existing frameworks backed by hand-optimized libraries.

Keywords: Deep learning compiler · Target optimization · Code generation · Vector processor

1 Introduction

Artificial Intelligence (AI) [1] is undoubtedly a hot topic at present, most of which focus on algorithm. However, engineering is crucial when we put AI into practical application.

A DL algorithm needs to go through two steps from theoretical model to practical application: training [2] and inference [3]. Training refers to the process of guiding DL models through the algorithm to train the data and making it available. At present, training technology has been quite sophisticated, where current mainstream DL

H. Gao et al. (Eds.): ChinaCom 2020, LNICST 352, pp. 670–687, 2021.
https://doi.org/10.1007/978-3-030-67720-6_46

frameworks include TensorFlow [4, 5], PyTorch [6], MXNet [7], Caffe [8], etc. Inference is the process of compiling and deploying DL models to a target hardware. As shown in Fig. 1, inference framework or DL compiler [9] works as a bridge to connect top frameworks and target hardware, which convert the flow graph of DL

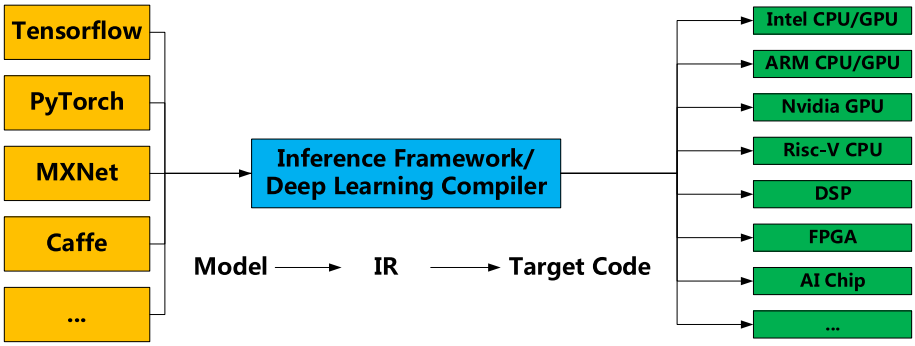

Fig. 1. Inference framework

models into a graph IR [10] and performs optimization operations, such as cycle scheduling [11] and operator fusion [12], and generates machine codes of each target device as required.

On the one hand, the market situation of DL compiler is diverse but not uniform. DL models are usually deployed to a variety of target devices including embedded CPUs, GPUs, DSPs, FPGAs, and ASICs. It is difficult to maintain an efficient inference performance in all target devices. Major hardware vendors have launched their own inference framework, such as Intel OpenVINO [13], ARM NN [14], NV TensorRT [15], but the major equipment manufacturers do not have a framework of generality. For example, the operator functions generated by various DL frameworks are not fully supported, especially the TensorFlow operators, where a DL model can run in one target device but not another.

On the other hand, most DL algorithms are computationally and memory-intensive, which require tens of megabytes of parameter storage and hundreds of millions of operations. It is difficult to compile and deploy DL models to embedded or battery-limited systems such as smartphones and smart glasses. However, DSP has strong computing power, which is more suitable for processing computation-intensive tasks comparing with other embedded processors. Target vector processor [16, 17] is an embedded high performance digital signal processor, which not only has the special convolution instruction based on large-scale matrix multiplication, but also realizes the eight-slots parallel technology of Vary Long Instruction Word (VLIW) [18] and the 2560 bits operations of Single Instruction and Multiple Data (SIMD) [19].

This paper presented the design of DLCS, a deep learning compiler for the independently designed target vector processor, which perform target optimizations through lowering DL models into a high-level IR and a low-level IR. DLCS takes advantage of model pruning to get lightweight model and customize the architecture description of target vector processor to achieve a high-quality automatic code generation.

2 Relate Works

TVM [9, 20] is an automated end-to-end optimizing compiler for deep learning that exposes both graphic-level and operation-level optimizations to provide performance portability to deep learning workloads across different hardware back-ends. TVM addresses optimization challenges for deep learning, such as advanced operator fusion, mapping to arbitrary hardware primitives, and memory latency hiding [21]. TVM automates low-level program optimization into hardware features by using a new, learning-based cost modeling approach to rapidly explore code optimization. TVM serves primarily as a bridge between the top framework and the target device. A trained model can generate optimized operators and inference code for different target platforms through TVM, thus realizing the deployment of multiple target platforms for a single model. The automatically generated operators can be further optimized by hand. In addition, one of the attractions of TVM is that its automatically generated code is comparable to those of hand-optimized acceleration libraries.

Accelerated Linear Algebra (XLA) [22] is a compiler that optimizes the computation of TensorFlow. The input language for XLA is called High Level Optimizer (HLO) [23] which is equivalent to the compiler's IR. The layering criteria for TVM IR are computational graphs and operators, while the layering criteria for HLO IR is device independent and device dependent. XLA improves server and mobile platform speed, memory usage, and portability. XLA performs target hardware-independent optimization and analysis [24], such as the common subexpression elimination (CSE) [25], process fusion of operators independent of the hardware back-end, and buffer analysis for memory allocation at run time for computation. XLA performs target hardware-dependent optimization and analysis, which will be targeted at the specific information and requirements of the hardware target, such as operator fusion on the XLA GPU back-end, and determining how to divide the calculation into streams. In addition, the backend can pattern match certain operators or combinations of them to the optimized library call. Finally, XLA does code generation for specific target hardware. The CPU, including the GPU used by XLA, uses LLVM [26] to generate low-level IR, optimization, and code generation. The hardware back-end emits the LLVM IR [27] required to represent the XLA HLO calculation in an efficient manner, and then calls LLVM to emit native code from the LLVM IR. The GPU back-end currently supports NVIDIA GPU through the LLVM NVPTX back-end, while the CPU back-end supports multiple CPU instruction set architecture (ISA).

Graph lowering compiler (Glow) [28] is a heterogeneous hardware-oriented machine learning compiler. It provides a practical compilation method that generates highly optimized code for multiple targets. Glow reduces the traditional neural network data flow diagram to an intermediate representation of a two-phase strongly-type [29]. The advanced intermediate representation allows the optimizer to perform domain-specific optimizations. Lower level instructions-based addressing only intermediate representations allow the optimizer to perform memory-related optimizations such as instruction scheduling [30], static memory allocation [31], and copy elimination [32]. The intermediate representation at the lowest level allows the optimizer to perform machine-specific code generation to take advantage of specialized hardware features. Glow features a lowering phase in which the compiler can support a large number of input operators and a large number of hardware targets by eliminating the need to implement all operators on all targets. The purpose of the lowering phase is to reduce the input space and allow the new hardware back end to focus on a small number of linear algebra primitives.

Deep Learning Virtual Machine (DLVM) [33] is a compiler infrastructure designed for modern deep learning systems. DLVM applies a multi-stage compiler optimization strategy to high-level linear algebra and low-level parallelism, performs domain-specific transformations, reduces the overhead of front-end languages, and acts as a host for the research and development of neural network domain-specific languages (DSLs). DLVM IR is the core language of the system, which uses static single assignment (SSA) forms, control flow diagrams, advanced types (including first-order tensor types), and a set of linear algebraic operators combined with a generic instruction set. The system supports a variety of domain-specific analyses and transformations, such as inverse pattern algorithmic differentiation [34], algorithmic differential checkpoints, algebraic simplification [35], and linear algebraic fusion. The complete DLVM software stack, including sample front-end neural network DSLs, and DLVM Core contains essential components for an optimizing compiler: intermediate representation and passes.

3 DLCS Core

As shown in Fig. 2, DLCS IR adopts a two-layer optimized structure, including high level IR and low level IR. Low level IR refers to the LLVM IR, which is mainly used for memory-related optimization such as instruction scheduling, static memory allocation, and copy elimination. High level IR refers to Graph IR, which is mainly used for the optimization of computational graphs, such as kernel fusion, data displacement compilation optimization, data layout transformation.

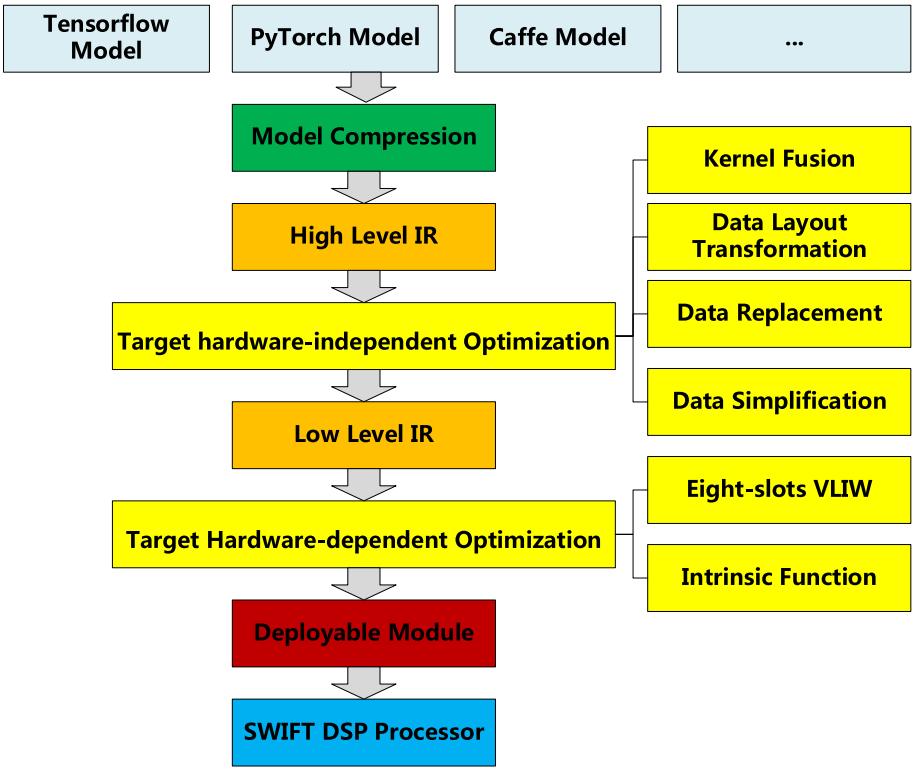

Fig. 2. DLCS framework

3.1 Target Optimization

Target Hardware-Independent Optimization

Data Simplification and Data Layout Transformation. Data simplification mainly includes node pruning and constant folding. Node pruning refers to the removal of identity nodes related to the training phase in the deployment phase, which simplify the computational graph and saving the operation cost. Constant folding optimization is to replace the nodes in the computational graph whose output values can be determined with constants in advance. On the one hand, the output of shape, rank and size in the computational graph is only related to the shape of the output tensor, which has nothing to do with the specific data input and can be calculated in advance. On the other hand, the output of a node whose input is all constant can also be calculated in advance.

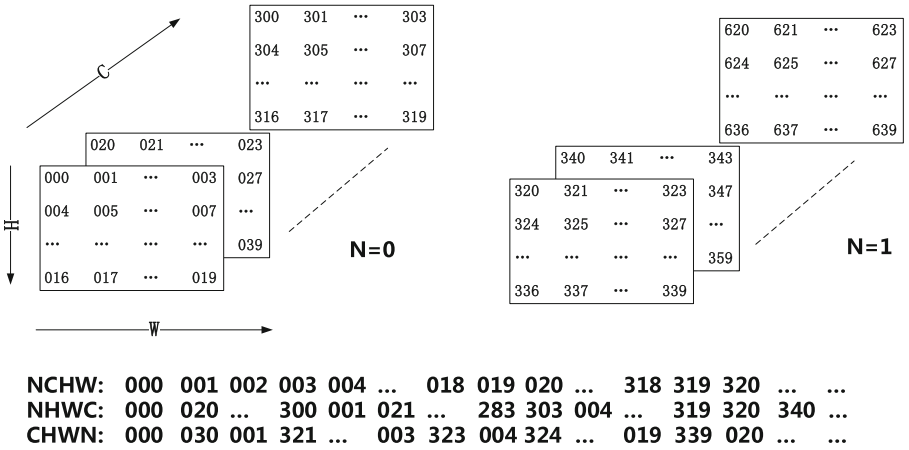

Fig. 3. Data layout: NCHW, NHWC, CHWN

The data layout transformation optimization mainly refers to the storage mode of transforming tensors to adapt to the target hardware architecture. The target hardware of DLCS is the target vector processor, and the optimal storage format is NCHW [23], where N represents the number of images in a batch, H represents the number of pixels in the direction of vertical height, W represents the number of pixels in the direction of horizontal width, and C represents the number of channels, as shown in the Fig. 3. The data layout optimization done by DLCS is to convert various data layout (NHWC [23] and CHWN [23]) to NCHW storage format. The data layout order of NCHW is [W-H-C-N], where the first element is 000 and the second element is along the W direction from 001 to 003. The second direction is along the H direction from 004 to 019. Next direction is along the C direction from 020 to 319. The last direction is along the N direction from 320 to 639.

Kernel Fusion. Kernel fusion, also known as operator fusion, is to fuse multiple operators in the computational graph into a single kernel. For example, (1) and (2) can be fused into (3), therefor, the entire data flow diagram can be completed with only one kernel. Moreover, by properly designing the placement of input and output data of different kernel functions, such as using shared memory or registers on GPU, the data transmission efficiency and the overall computing performance can be greatly improved. General operations can be divided into three types: one-to-one mapping, complex fusion and infusion. Multiple injective operators can be merged into another injective operator. We can fuse element-level operators into the output of operators which are complex and fusible, like conv2d operator.

$$Z = \text{op1}\ (X,\ Y) \tag{1}$$

$$T = \text{op2}\ (Z) \tag{2}$$

$$T = op2\ (op1\ (X,\ Y)) = op3\ (X,\ Y) \tag{3}$$

Data Replacement. The deployable objects generated by DLCS mainly include parameters and computational graphs, which are serialized by binary to a final output file. As shown in Fig. 4, it's common to simply put the data into an unsigned char array and compile the corresponding file. However, if a large amount of model data is trained, the deployment and compilation phase will take a lot of time. The optimization proposed in this section is to replace the serialized data by placing the flag information in the unsigned char array. This flag information is found in the resulting file and replaced with the original binary serialized data. Finally, we can directly deploy the resulting file to the target vector processor to achieve the deployment of DL models.

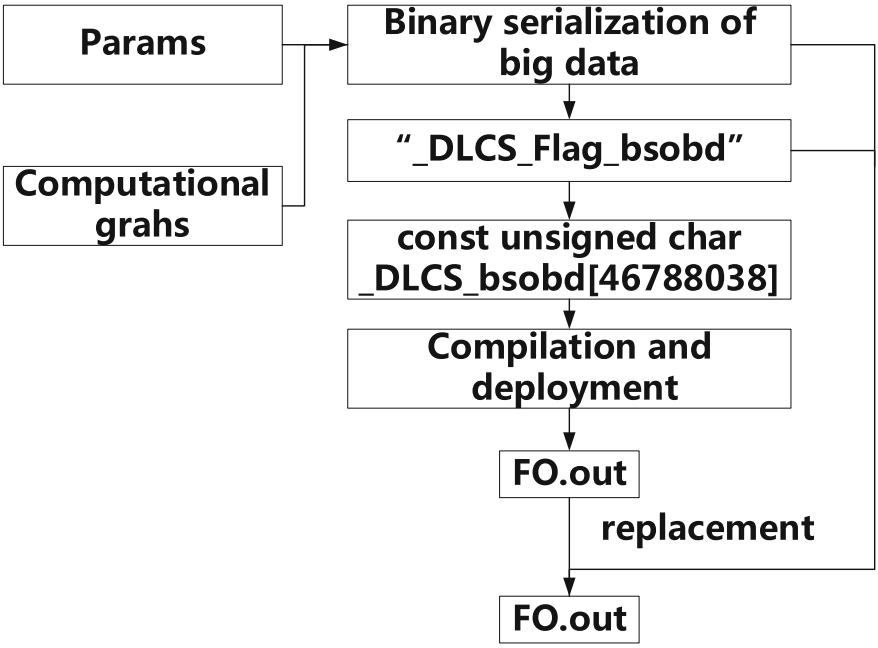

Fig. 4. Data replacement

Target Hardware-dependent Optimization

Eight-slots VLIW. Eight-slots VLIW mainly includes VLIW detection and VLIW coding. VLIW detection refers to the determination of which instructions can form a VLIW package. The judgment of detection is based on whether the empty slots can be allocated and whether the instructions have a dependency with each other. VLIW encoding refers to assigning a value to the slot encoding bit of an instruction in the same VLIW package.

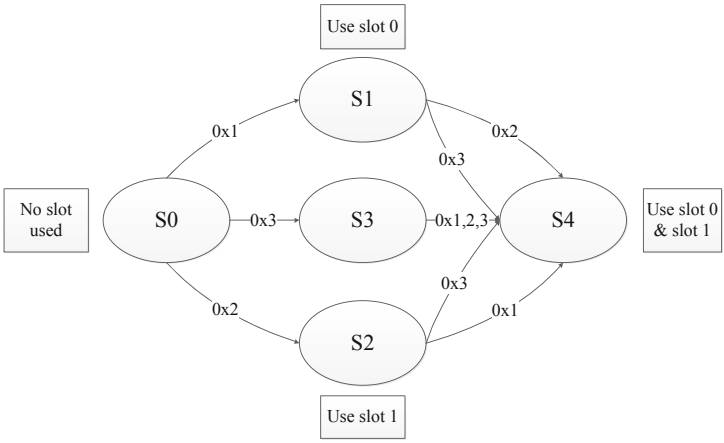

Fig. 5. DFA state transition diagram of two-slots VLIW

We use Deterministic Finite Automaton (DFA) to describe the first detection condition. Given an input instruction sequence in a two-slot state transition diagram, as shown in the Fig. 5, the automatic machine performs a state-to-state transition according to the state transition equation. If an instruction in the input instruction sequence causes the automaton to stop in an unacceptable state, the automaton rejects the instruction. The DFA is reset to start a new round of judgment after the previously accepted instructions forming a VLIW package. As shown in the Table 1, the slot allocation of each instruction would be judged with the VLIW constraint.

Table 1. Instruction slot constraint of target vector processor

Instruction type	Slot 0	Slot 1	Slot 2	Slot 3	Slot 4	Slot 5	Slot 6	Slot 7
Control operations	√	√						
Scalar operation	√						√	
Data transfer operations	√		√		√	√	√	√
Vector operation			√	√				
Loading operations					√	√		
Storage operations							√	√

After completing the first condition of the VLIW detection, we enable a separate compile pass to complete the dependency determination of the detection. The data dependence that we need to judge mainly includes true dependence and output dependence, as shown in the Fig. 6. If one of the two detection conditions is not satisfied, the detection will be exited, and then all the instructions that the previous detection satisfies will be arranged in the same VLIW package.

The input of VLIW encoding is the VLIW package sequence obtained after VLIW detection. We iterate over each VLIW package and assign values to the slot encoding

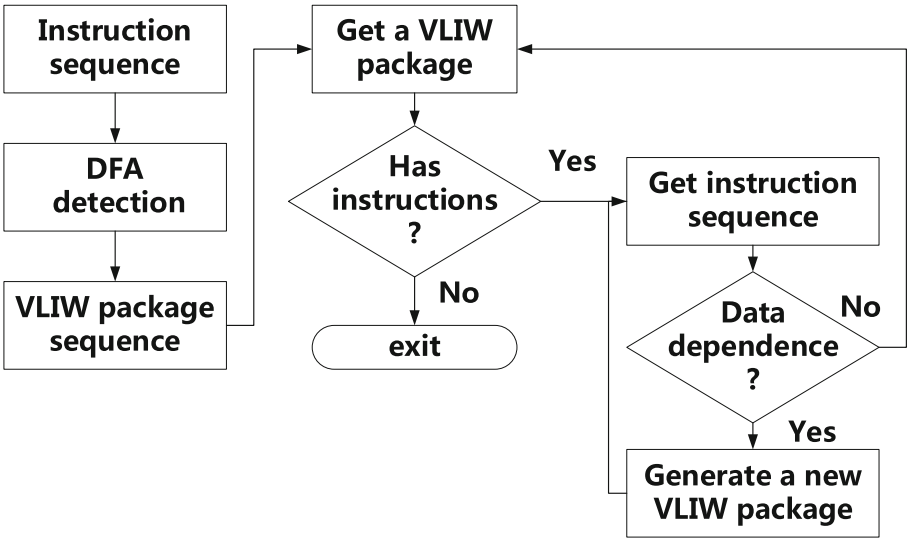

Fig. 6. VLIW detection

bits of all instructions inside. The instruction set of target vector processor is 32-bits encoding mode, where InstBit{31:29} is the slot encoding bits. InstBit{31} indicates whether the instruction is the last instruction in the VLIW packet, while InstBit{30:29} means instruction slot allocation, where InstBit{31:29} = {00} means that slot 0, which has priority, and slot 1 can be allocated to this instruction; InstBit {31:29} = {01} means that slot 2, which has priority, and slot 3 can be allocated to this instruction; InstBit{31:29} = {10} means that slot 4, which has priority, and slot 5 can be allocated to this instruction; InstBit{31:29} = {11} means that slot 6, which has priority, and slot 7 can be allocated to this instruction. Table 2 is an example of VLIW encoding where a detected eight-slot VLIW packet contains five instructions of ABCDE, and their slot encoding bits are 000,001,011,010,111 respectively.

Table 2. An example of VLIW encoding

	Slot 0	Slot 1	Slot 2	Slot 3	Slot 4	Slot 5	Slot 6	Slot 7	Slot allocated	Slot encoding bit
A	√	√							Slot 0	000
B	√	√							Slot 1	001
C			√	√					Slot 3	011
D			√						Slot 2	010
E	√	√					√	√	Slot 6	111

Intrinsic Function and SIMD. Intrinsic function is a special source-language-related function that is not implemented by a call instruction with an offset address, but the compiler. We need to support the call to intrinsic function in DLCS Lower IR. For some special instructions of the target vector processor, as shown in the Fig. 7, we can directly use intrinsic function to map functions from source code to DLCS Lower IR.

```
//#include "vector_support.h"
typedef int      v4i      __attribute__((vector_size(16)));
typedef short  v8s      __attribute__((vector_size(16)));
typedef char   v16c     __attribute__((vector_size(16)));
v4i  v4i_a = {1, 2, 3, 4},   v4i_b = {3, 3, 3, 3};
v8s  v8s_a = { 1, 2, 3, 4, 5, 6, 7, 8 },    v8s_b = { 1, 2, 3, 4, 5, 6, 7, 8 };
v16c v16c_a = {  1,  2,  3,  4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 };
v16c v16c_b = { 13, 14, 15, 16, 5, 6, 7, 8, 9, 10, 11, 12,  1,  2,  3,  4 };
int main(int argc, char const *argv[]){
        ...
        v4i v4i_dst = __builtin_swift_dsp_vmax_i(v4i_a, v4i_b);
        v8s v8s_dst = __builtin_swift_dsp_vmax_s(v8s_a, v8s_b);
        v16c v16c_dst = __builtin_Swift_dsp_vmax_c(v16c_a, v16c_b);
        ...
}
```

Fig. 7. The example of the intrinsic function of target vector processor in the C language

As shown in Fig. 8, on the front-end of DLCS, we mainly define three types of vectors, the properties of the intrinsic function and the corresponding function nodes. The total bit width of the three vector types is 128 bits, including v4i (i128 = 4xi32), v8s (i128 = 8xi16) and v16c (i128 = 16xi8). The intrinsic function node mainly defines the types of parameters and return values.

On the backend of DLCS, we mainly implement the intrinsic function of the instruction template definition, vector register definition, the mapping and legalization of intrinsic instruction from front-end to back-end. The instruction template definition process mainly makes use of the architecture description language (ADL) to describe the various attributes of the instruction, such as instruction encoding, slot assignment, operand type, matching pattern, node mapping and assembly string. The detailed code of how to define the instruction template will be introduced in the code generation section. Each property declaration of vector register in back-end matches the three vector types of the front-end.

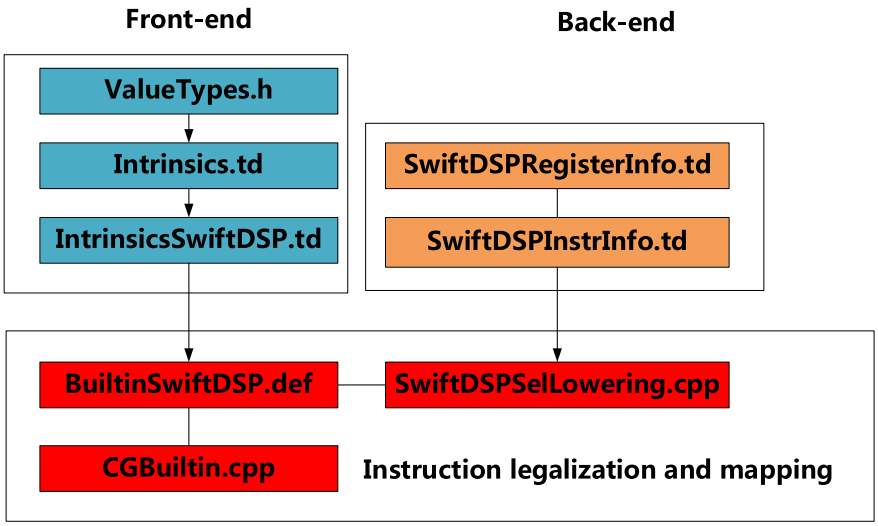

Fig. 8. The files involved in implementing the intrinsic function on front-end and back-end

3.2 Code Generation

Code generation for embedded processors mainly includes operation legalization, instruction selection, instruction scheduling, register allocation, optimization, target code generation. DLCS is based on the LLVM framework, which has already realized the generic parts of code generation. Therefor this section mainly explains the code generation for three special cases that DLCS Lower IR and instruction set of target vector processor are not compatible.

Architecture Description of Target Vector Processor

Description of Register. Before we discuss operational incompatibilities, we use the TableGen tool provided by LLVM to realize architecture description of target vector processor, including the description of instructions and registers. We use the register base class provided by LLVM to describe register properties. As shown in Table 3, the register of target vector processor mainly contains 32 scalar registers and 16 vector registers.

Table 3. The description of register heap for target vector processor

Tag	Register	Bit wide	Counts	Lowest limit of addressing	Highest limit of addressing
GR	General	32	32	000000	011111
VR	Vector	160	16	100000	101111
OFF	Offset	32	4	110100	110111
BAR	Base address	32	4	111000	111011
MR	Modular	32	4	111100	111111

Figure 9 depicts three parts: the definition of the general register, the allocation of the general register, and the calling convention for the register. The general register definition needs to specify information such as the namespace and encoding bit width. Since each register is an instance of a register class, we instantiate each register. We use multiple inheritance to inherit the base class of the scalar register or vector register to generate the definition register. We define registers (V0, V1, A0–A7, VR0–VR15) and classify them into general register class, vector register class and special register class. In the calling convention of register, we stipulate i32 as scalar standard type, and extend i8 and i16 types to i32 for unified processing. We specify that i32 are stored in the general register class, while v4i, v8s, and v16c are stored in the vector register class (VR0–VR15). We specify that the alignment of the function call stack is 4 bytes (i32) and 16 bytes (v4i, v8s and v16c).

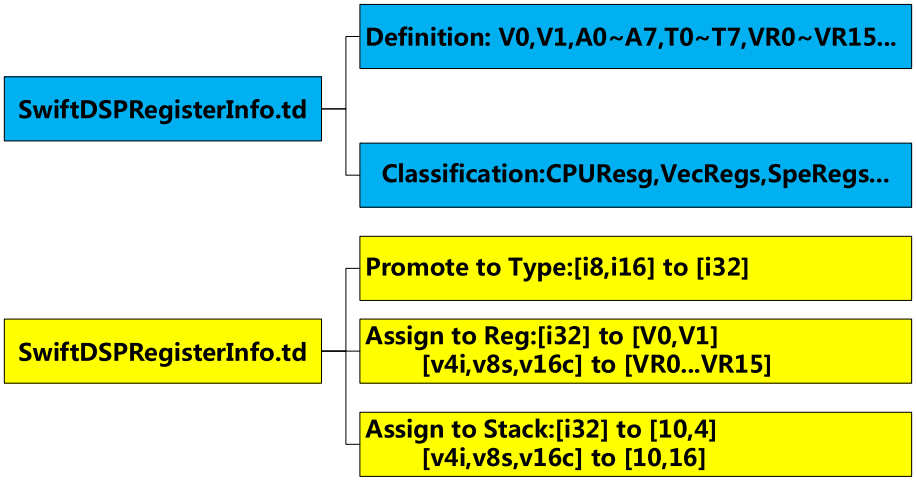

Fig. 9. The description of registers with LLVM ADL

Description of Instructions. We extend the instruction template based on the instruction base class template provided by LLVM. According to the complexity of the instruction set of SWFIT DSP, the instruction template can be divided into four levels of the instruction template, as shown in Fig. 10, where the definition of each level of the instruction template is inherited from the previous level of the template. The level 1 instruction base class template mainly declares parameters common to all instructions and some flag bit settings related to the target hardware. The second level instruction base class template is divided into four common types: single register class, double registers class, three registers class and special registers class. The third level instruction base class template is divided according to the instruction operation code bit width. The level 4 instruction base class template gives a template for one-to-one relationships with each instruction, as shown in Fig. 10.

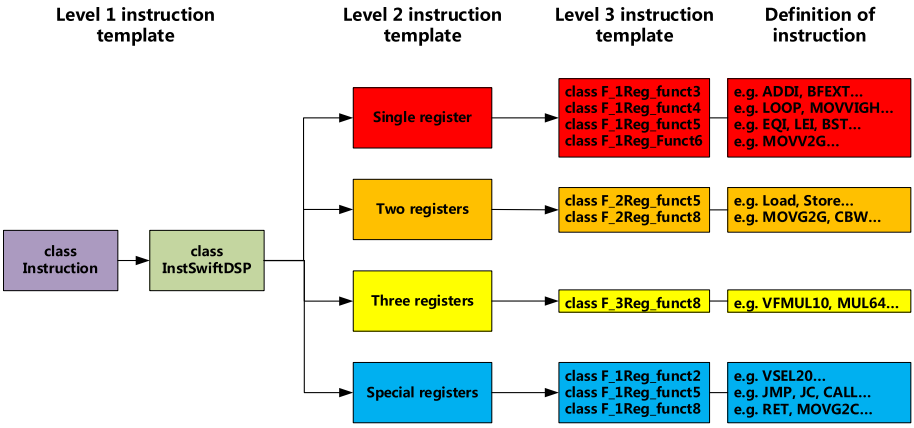

Fig. 10. Inheritance diagram of instruction class

Figure 11 shows the template inheritance diagram of the "VMAX40" instruction, and the corresponding detailed description code is shown in Fig. 12. The level 3 instruction base class template defines the instruction encoding format, assembly strings, matching patterns, and other parameters. We need to define an additional DAG node corresponding to the instruction in the process of defining the instruction. A DAG node represents an instruction. The realization of DAG node and instruction mapping is determined by matching pattern. One of the important steps in target code generation is the mapping of DAG nodes to assembly instructions for the target hardware.

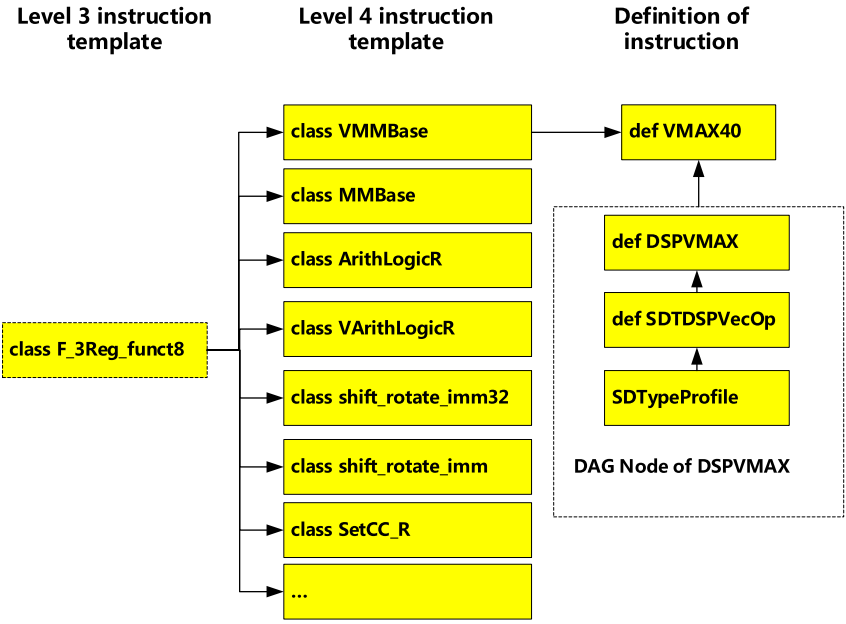

Fig. 11. Inheritance diagram for instruction "VMAX40"

```
//SwiftDSPInstrFormats.td
...
class F_3Reg_funct8<bits<3> type,bits<3> op, bits<8> funct,          dag outs, dag ins, string asmstr,
      list<dag> pattern, InstrItinClass itin>:InstDSP<outs, ins, asmstr, pattern, itin>{
            bits<6>  ra;
            bits<6>  rb;
            bits<6>  rc;
            let Inst{31-29} = type;
            let Inst{28-26} = op;
            let Inst{25-20} = ra;
            let Inst{19-14} = rb;
            let Inst{13-8} = rc;
            let Inst{7-0}  = funct;
}
...
//SwiftDSPInstrInfo.td
...
def SDTDSPVecOp : SDTypeProfile<1,2,[SDTCisSameAs<0,1>,SDTCisSameAs<0,2>,SDTCisVec<0>]>;
def DSPVMAX :      SDNode<"DSPISD::VMAX",SDTDSPVecOp,[SDNPCommutative]>;
...
class VMMBase<bits<3> typeop,bits<3> op, bits<8> inner_op,       string instr_asm, SDNode OpNode,
                        InstrItinClass          itin, RegisterClass RC>:F_3Reg_funct8<typeop,op,inner_op,
                        (outs RC:$ra),          (ins RC:$rb, RC:$rt),!strconcat(instr_asm, "\t$ra, $rb, $rt"),
                        [(set RC:$ra, (OpNode RC:$rb, RC:$rt))],itin>{
                        let isCommutable = 1;
}
def VMAX40 :VMMBase<5,1,0b01001001,"vmax40",DSPVMAX,ALU32_V_SLOT23,VecRegs>;
...
```

Fig. 12. Description definition code for instruction "VMAX40"

Incompatible Instruction. As shown in Table 4, the first incompatibility is that DLCS Lower IR supports operation X but target vector processor does not. We generally divide X into several combinations of operations supported by target vector processor, such as (4) where target vector processor supports operations of Y, X1 and X2. The second incompatibility is that DLCS Lower IR does not support operation X but target vector processor does. We solved this problem by intrinsic functions in target hardware-dependent optimization. The third incompatibility refers to the case where the bit width of the same operation X, supported by DLCS Lower IR and target vector processor, is different, which is called operation legalization problem. DLCS uses a compile pass to legalize operations. On the basis of DLCS Lower IR and target vector processor instruction set, we analyze which instruction is illegal operation, and then describe the legalization process through a set of interfaces as (5).

$$\mathrm{X} = \mathrm{Y}\ (\mathrm{X1},\ \mathrm{X2}) \tag{4}$$

$$\mathrm{SetOperationAction}\ (\mathrm{Operation},\ \mathrm{Type},\ \mathrm{LegalizaAction}) \tag{5}$$

"Operation" is an instruction of DLCS Lower IR, and "Type" is the operand Type of the instruction, and "LegalizeAction" is the processing when legalizing the instruction. Corresponding to the previous situation that the instruction bit width does not match, we designed three handling methods: "Promote" represents this operation should be put in a larger type; "Expand" is to try to expand this to other operations, otherwise use a libcall; "Custom" is to use legitimate functions to implement custom lowering.

Table 4. Incompatibility cases related to operation X

	Case 1	Case 2	Case 3
DLCS Lower IR	Support	No	Support
Instruction set of Target Vector Processor	No	Support	Support
The bit width of the instruction			Different

4 Evaluation

We compare the effects of various target optimization operations proposed in this paper and hand-optimization operations on the compilation speed of the DL model, and we take the performance value of hand-optimization as the benchmark. As shown in Fig. 13, the compiled object of the comparison experiment are conventional convolution and pooling with input size (16, 28, 28) and convolution kernel (32, 16, 3, 3), and deep convolution with input size (128, 14, 14) and convolution kernel (256, 128, 3, 3). Experimental results show that all target optimization operations can improve the performance of execution speed, in which SIMD optimization brings the smallest acceleration ratio about 1.1×. Data simplification and data layout transformation can lead to significant improvements in execution speeds up to 2.7×.

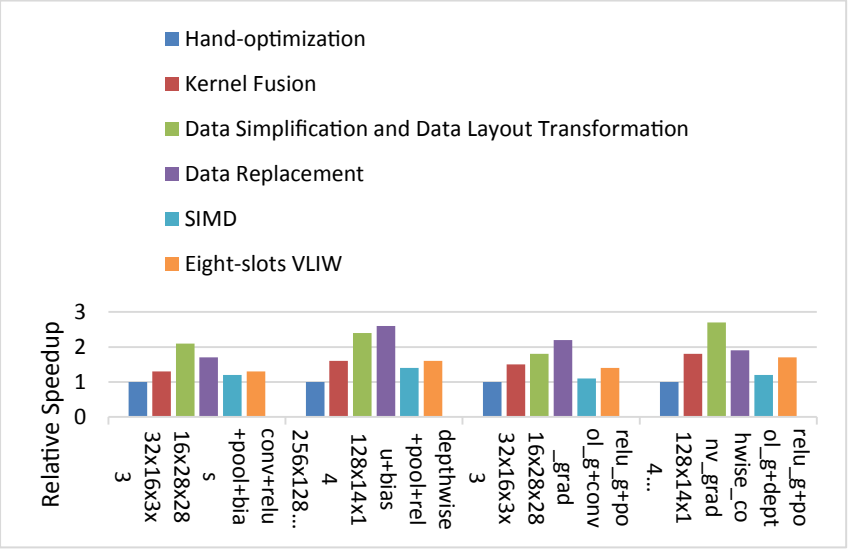

Fig. 13. Performance comparison between hand-optimization and target optimizations

Finally, we evaluate the whole compilation system, including target optimization and code generation. ResNet-18 and MobileNet are used as DL models in the comparison experiment in R1–R5 phase and M1–M5 phase. The performance index is the relative speedup of the two models on the target vector processor. Figure 14 shows a

comparison of the performance of the two models deployed on the target vector processor with DLCS and hand-optimization. Experimental results show that DLCS offers Multi-slot parallel performance for target vector processor and achieves speedups ranging from 1.5× in R5 phase to 3.0× in M4 phase over existing frameworks backed by hand-optimized libraries, where the compilation performance improvement of DLCS is generally over 2.0× during most phases of the two DL model compilation.

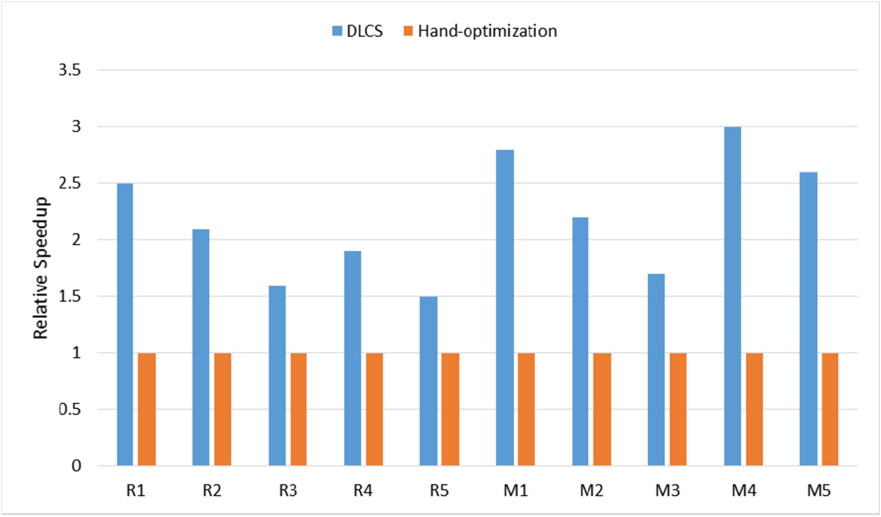

Fig. 14. Performance comparison between DLCS and hand-optimization

5 Conclusion

This paper presented the design of a DL compiler (DLCS) for the independently designed target vector processor. DLCS lowers the deep learning model to an intermediate representation of two levels and realizes the target hardware-independent optimization and target hardware-dependent optimization, so that the performance of the model finally deployed on the target vector processor is far superior to that of hand-optimization. We hope that this work will encourage more research on DL compilation methods for self-developed embedded chips and open-up new opportunities for self-developed embedded system hardware/software co-design technologies.

Acknowledgment. The authors thank the editors and the anonymous reviewers for their invaluable comments to help to improve the quality of this paper. This work was supported in part by the National Natural Science Foundation of China under Grant Nos. 61831018 and 61631017, and Guangdong Province Key Research and Development Program Major Science and Technology Projects under Grant 2018B010115002.

References

1. Suchman, L.A., Trigg, R.H.: Artificial intelligence as craftwork (1993)
2. Moore, R.C., Lewis, W.: Intelligent selection of language model training data. In: Proceedings of the ACL 2010 Conference Short Papers. Association for Computational Linguistics, pp. 220–224 (2010)
3. Yao, L., Mimno, D., McCallum, A.: Efficient methods for topic model inference on streaming document collections. In: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 937–946 (2009)
4. Abadi, M., Barham, P., Chen, J., et al.: TensorFlow: a system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016), pp. 265–283 (2016)
5. Abadi, M., Agarwal, A., Barham, P., et al.: TensorFlow: large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467 (2016)
6. Paszke, A., Gross, S., Massa, F., et al.: PyTorch: an imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems, pp. 8024–8035 (2019)
7. Chen, T., Li, M., Li, Y., et al.: MXNet: a flexible and efficient machine learning library for heterogeneous distributed systems. arXiv preprint arXiv:1512.01274 (2015)
8. Jia, Y., Shelhamer, E., Donahue, J., et al.: Caffe: convolutional architecture for fast feature embedding. In: Proceedings of the 22nd ACM International Conference on Multimedia, pp. 675–678 (2014)
9. Chen, T., Moreau, T., Jiang, Z., et al.: TVM: an automated end-to-end optimizing compiler for deep learning. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018), pp. 578–594 (2018)
10. Adachi, Y., Kumano, T., Ogino, K.: Intermediate representation for stiff virtual objects. In: Proceedings Virtual Reality Annual International Symposium 1995. IEEE, pp. 203–210 (1995)
11. Yoo, H., Shim, M., Kim, D.: Dynamic duty-cycle scheduling schemes for energy-harvesting wireless sensor networks. IEEE Commun. Lett. **16**(2), 202–204 (2011)
12. Chan, C.H., Tahir, M.A., Kittler, J., et al.: Multiscale local phase quantization for robust component-based face recognition using kernel fusion of multiple descriptors. IEEE Trans. Pattern Anal. Mach. Intell. **35**(5), 1164–1177 (2012)
13. Kustikova, V., et al.: Intel distribution of OpenVINO toolkit: a case study of semantic segmentation. In: van der Aalst, W.M.P., et al. (eds.) AIST 2019. LNCS, vol. 11832, pp. 11–23. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-37334-4_2
14. Arm, N.N.: SDK. https://developer.arm.com/products/processors/machine-learning/arm-nn
15. Vanholder, H.: Efficient Inference with TensorRT (2016)
16. Ren, H., Zhang, Z., Wu, J.: SWIFT: a computationally-intensive DSP architecture for communication applications. Mobile Netw. Appl. **21**(6), 974–982 (2016)
17. Wang, W.: Functional verification for SWIFT DSP based on software-based simulation and FPGA. In: International Conference on Wireless Communications, Networking and Applications
18. Ellis, J.R.: Bulldog: a compiler for VLIW architectures. Yale Univ., New Haven, CT (USA) (1985)
19. Nassimi, D., Sahni, S.: Data broadcasting in SIMD computers. IEEE Trans. Comput. **100**(2), 101–107 (1981)
20. Chen, T., Moreau, T., Jiang, Z., et al.: TVM: end-to-end optimization stack for deep learning. arXiv preprint arXiv:1802.04799 (2018)

21. Maiyuran, S., Garg, V., Abdallah, M.A., et al.: Memory access latency hiding with hint buffer: U.S. Patent 6,718,440, 6 April 2004
22. Leary, C., Wang, T.: XLA: TensorFlow, compiled. TensorFlow Dev Summit (2017)
23. Larsen, R.M., Shpeisman, T.: TensorFlow Graph Optimizations (2019)
24. Chadha, P., Siddagangaiah, T.: Performance analysis of accelerated linear algebra compiler for TensorFlow
25. Yao, C.Y., Chen, H.H., Lin, T.F., et al.: A novel common-subexpression-elimination method for synthesizing fixed-point FIR filters. IEEE Trans. Circuits Syst. I Regul. Pap. **51**(11), 2215–2221 (2004)
26. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, CGO 2004. IEEE, pp. 75–86 (2004)
27. Lattner, C.A.: LLVM: an infrastructure for multi-stage optimization. University of Illinois at Urbana-Champaign (2002)
28. Rotem, N., Fix, J., Abdulrasool, S., et al.: Glow: graph lowering compiler techniques for neural networks. arXiv preprint arXiv:1805.00907 (2018)
29. Sivalingam, K., Mujkanovic, N.: Graph compilers for AI training and inference
30. Gibbons, P.B., Muchnick, S.S.: Efficient instruction scheduling for a pipelined architecture. In: Proceedings of the 1986 SIGPLAN symposium on Compiler construction, pp. 11–16 (1986)
31. Kogure, M.: Static memory allocation system: U.S. Patent 5,247,674, 21 September 1993
32. Gopinath, K., Hennessy, J.L.: Copy elimination in functional languages. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 303–314 (1989)
33. Wei, R., Schwartz, L., Adve, V.: DLVM: a modern compiler infrastructure for deep learning systems. arXiv preprint arXiv:1711.03016 (2017)
34. Griewank, A., Walther, A.: Evaluating derivatives: principles and techniques of algorithmic differentiation. SIAM (2008)
35. Buchberger, B., Loos, R.: Algebraic Simplification. Computer Algebra, pp. 11–43. Springer, Vienna (1982). https://doi.org/10.1007/978-3-7091-7551-4_2