Matrix-DSP back-end support based on TVM compilation structure

Xiaolong Zhu[1], *Ping* Deng[1], *Haiyan* Sun[1,*], and *Yong* Li[1]

[1]College of Computer, National University of Defence Technology, 410073 Changsha, China

Abstract. The emergence of deep learning frameworks has greatly facilitated the construction of network models, but it has not solved the problem of network models deployed in different hardware backends. TVM combines hardware-independent optimization and hardware-related optimization decoupling ideas to provide excellent solutions. By analyzing the basic structure of TVM and the basic process of neural network deployment on hardware, TVM has realized the basic support of the independently developed chip Matrix-DSP,which provides a foundation for further exploring the performance of the chip and enriching the application scenarios of the chip.

1 Introduction

In recent years, with the rapid development of hardware chips and the rapid improvement of chip performance, the research on deep learning has set off a wave of enthusiasm in the world. Its accumulated theories and achievements in machine vision and artificial intelligence have been widely used in automobiles. manufacturing, manned spaceflight, medical diagnosis and many other scientific fields. In order to make deep learning quickly applicable to actual scenarios, in addition to further exploring the deep principles of neural networks and carrying out theoretical innovations, it is also necessary to solve the problems of rapid network model construction and deployment on a variety of hardware backends and network acceleration. The emergence of deep learning frameworks has greatly reduced the difficulty of building network models, but the support for a variety of back-end hardware is not complete, and it often requires a lot of unnecessary repetitive work when adding a new back-end hardware. At the same time, due to the difference in the basic structure and application scenarios of the framework, the network model defined on a certain framework cannot be well migrated to other network frameworks to complete inference.

In order to solve the problem of support for different network frameworks, and new back-end hardware support and hardware acceleration, TVM[1] used the compiler's thinking to complete the analysis of the neural network and its deployment in the back-end. The upper-level TVM undertakes networks under various frameworks, introduces intermediate representation of the code under the idea of decoupling, realizes front-end goal-independent optimization (graph optimization) and back-end related optimization

[*] Corresponding author: helen@nudt.edu.cn

(operator optimization), and then link the compiler LLVM to get Object code that can run at high speed on the target hardware. This article mainly discusses the completion of the back-end support for Matrix-DSP on TVM.

2 TVM compilation architecture and Matrix-DSP analysis

2.1 TVM overall architecture

The deep learning [10]compiler TVM is an open source project contributed and developed by the SAMPL group of the University of Washington. It draws on the ideas of traditional compilers, and achieves better decoupling by introducing intermediate representations, which greatly avoids the repetitive work required for adding new back-end hardware. Similar to traditional compilers, TVM is divided into front-end, intermediate representation and back-end. The high-level intermediate representation (relay[2] IR) resides in the front-end, and the low-level intermediate representation (tensor IR) resides in the back-end, but is relatively traditional. TVM can better obtain the overall information of the application and complete specific optimizations (such as graph optimization) for deep learning. At the same time, TVM also supports the use of mature compiler tool chains (such as LLVM[3]), so that the generation of specific hardware back-end instruction codes can be handed over to existing compilers. TVM can be dedicated to the optimization of network models and hardware-specific operator algorithm optimization. In general, the TVM front-end receives networks under different frameworks, and generates a target-independent calculation graph (relay IR) after format conversion. After performing a graph optimization algorithm suitable for deep learning on the relay IR, it is divided into a tensor expression representing the function of the operator. The back-end manually or automatically optimizes(with the assistance of AutoTVM[4]) the operator resolved into tensor IR form according to the target hardware characteristics, and finally connects the compiler LLVM to generate high-performance code specific to the target hardware. The figure 1 shows the basic compilation architecture and compilation process of TVM.

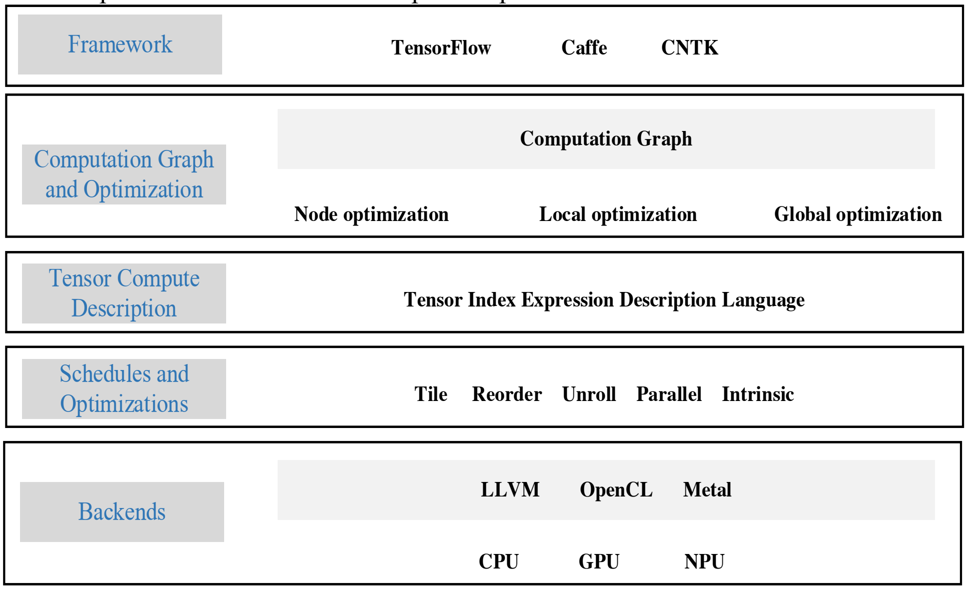

Fig.1. The basic compilation architecture and compilation process of TVM.

2.2 TVM front-end

The front end first implements support for networks under various frameworks such as pytorch[5] and tensorflow[6]. After the unified analysis of the network model is completed, the network is expressed as a calculation graph in the form of relay IR, and then the graph optimization algorithm is used to achieve the goal-independent optimization of the calculation graph . The calculation graph mainly looks at the entire calculation process from a global perspective, without the need for meticulous implementation of each operator. In the calculation graph, nodes represent tensor operations or input tensors, and edges represent dependencies between nodes. TVM performs a large number of graph optimization operationson the front end. These optimizations can be roughly divided into three categories[9]: node level, local level and global level. Table 1 shows the basic optimization methods.

Table 1. The basic optimization methods.

Node optimization	Node elimination: eliminate unnecessary operation nodes, such as X and Y are zero-dimensional tensors and constant tensors, respectively, constant node Y can be used to replace Y-X operations.
Local optimization	Algebraic identity: the calculation of x×1 can be changed to x.
	Constant folding: such as replacing the expression x×2×4 with the final value x×8.
	Operator fusion: reduce the number of nodes in the calculation graph to better share calculations.
Global optimization	Common sub-expression elimination (CSE): if A is a common sub-expression, the value of A only needs to be calculated once, and there is no need to repeat calculations where the expression A is used later.
	Layout conversion: Convert the tensor into the best data layout on the hardware to facilitate fast calculations by the hardware.

2.3 TVM back-end

After the front-end relay IR has undergone basic optimization operations such as graph optimization, the operators in the calculation graph will be further parsed into operator expressions described in tensor language. The optimization primitives provided by TVM can be used to implement target hardware-specific operators optimization. The construction of TVM's tensor language and the implementation of tensor IR borrow the characteristics of the separation of calculation and scheduling of the Halide[7] language, so that the nodes in the calculation graph do not need to be bound to their specific implementation on the backend,only need to know the corresponding type of its node to complete global optimization. After identifying the back-end target string target, the operator can be mapped to the specific back-end implementation and optimization. Regarding back-end-specific operator optimization TVM provides a large number of optimization primitives. The corresponding logic code after the operator optimization can be printed out through tvm.lower( ) for debugging and viewing. It is found that the functional composition of the operator is actually equivalent to a multi-level nested for loop.

The optimization process of the operator is actually the processing process of the tensor IR syntax tree. Since the basic structure of the syntax tree corresponds to the structure of the multi-layer for loop, its functional representation can be realized without an overly complicated syntax tree. After completing the basic tensor ir syntax tree construction and optimization corresponding to the tree, TVM will connect the back-end compiler LLVM to complete the code generation of the target hardware, in order to avoid converting the tensor ir syntax tree into a program in a certain language (Such as C language), and then connect the compiler LLVM to parse the unnecessary work brought by the program. TVM directly uses LLVM to complete the analysis of the tensor IR syntax tree, and calls the LLVM library function to traverse the nodes, and directly generate a module containing LLVM IR while processing. The module containing LLVM IR will further complete IR optimization, instruction selection and other operations through the LLVM backend, and finally generate target code that can run at high speed on the target platform. To achieve this process, support for Matrix-DSP is added to LLVM. Including the basic information describing the hardware, the description of the instruction set, and the regulation of instruction matching.

2.4 Matrix-DSP analysis

Matrix-DSP is a high-performance DSP with SIMD+VLIW characteristics independently developed by the National University of Defense Technology. Figure 2 shows the main structure of its core. The instruction dispatch component extracts the instruction to be executed from the received instruction packet, and sends it to the scalar component and the vector component for execution. The scalar processing unit (SPU) in the scalar component is not only responsible for scalar data operation, instruction flow control, and execution of serial tasks, but also for the control of the vector component. The scalar storage unit (SM) is mainly used for scalar data access, and the first level Cache (L1D) realizes the caching of scalar data. The vector processing unit (VPU) in the vector component is mainly responsible for performing computationally intensive tasks. The VPU is composed of 16 homogeneous vector computing engines (VPE), and the VPE contains 3 floating-point multipliers MAC. The on-chip array memory (AM) can realize 16-channel SIMD-wide vector data access, supports two vector storage instructions and DMA parallel access operations, and provides a higher memory access bandwidth for the VPU.

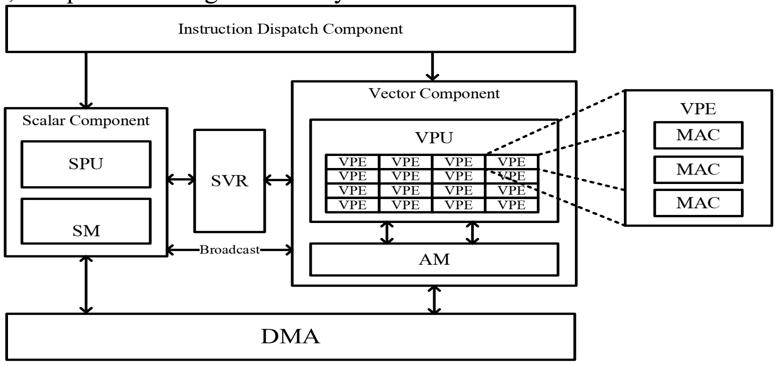

Fig. 2. The main structure of Matrix-DSP.

3 Matrix-DSP back-end support implementation

Through the analysis of the TVM compilation architecture in Section 2, the main function of the TVM front-end is to analyze the network model and the optimization of the calculation graph that has nothing to do with the processing target. The TVM back-end is an important part of completing the support for Matrix-DSP. The support for Matrix-DSP in the backend mainly includes the following types of tasks: complete operator programming and hardware structure-specific operator optimization, supplement the hardware-related interfaces in the compilation process, and call LLVM library functions to implement LLVM IR generate.

3.1 Operator optimization method

In order to optimize the operator according to the hardware characteristics, TVM provides a wealth of optimization primitives, including loop-specific optimization primitives such as tile, split, reorder, unroll and parallel, and intrinsic primitives for embedding the internal mapping of the hardware, etc. When optimizing primitives on the specified hardware, we need to pay special attention to the following hardware information: the data storage method in the memory (row or column first), the register size and cache size on the hardware, and the vector length supported by the hardware. Using optimization primitives to optimize operators is actually by changing the order of data access, reducing the number of cache misses, and using vector acceleration components to improve operator performance. We can refer to the theory in the article and combine the characteristics of the Matrix-DSP structure to complete the optimization of the operator .Due to space limitations, the implementation of specific operator optimizations will not be given. Table 2 gives the functional description of some primitives.

Table 2. The functional description of some primitives.

Tile	Divide the tensor into blocks, which can enhance the locality of memory access and reduce the number of cache misses
Reorder	Reorder the cycle axis to optimize the order of data access
Unroll	Loop unrolling can reduce data dependence and facilitate the acceleration of instruction pipeline
Parallel	Multithreading can be used to accelerate program execution
Intrinsic	A certain part of the calculation in the operator is directly replaced with a highly optimized program

3.2 Add call interface

The function tvm.build( ) in TVM is the core function to generate the target machine code. Knowing its basic workflow is beneficial to the realization of code addition. The input of this function contains the parameter S that represents operator scheduling (operator optimization), and the string target that indicates the target hardware name. After obtaining the parameter S, the function build( ) in the file build_module.py will realize the optimization after the corresponding scheduling through the function lower( ), and convert it into a LoweredFunc in the form of Tensor IR. After the generation of LoweredFunc, it will jump to codegen.cc. In the function build( ), this function is the real entry point for generating target hardware code from Tensor IR. It will complete the analysis of the target string for the first time, and prepare for the backend to generate the module on LLVM. After continuing to jump to the function init( ) of the file llvmmodule.cc, it starts to traverse all LoweredFunc and parses the target string again to call the codegen class of the specified hardware to complete the translation of the syntax tree corresponding to LoweredFunc. The following shows part of the code added by the interface.

```
class CodeGenMatrix-DSP final : public CodeGenCPU {
 public:
  void InitTarget(llvm::TargetMachine* tm) final {
     native_vector_bits_ = 16 * 8;
     CodeGenCPU::InitTarget(tm);
  }......};
```

```
TVM_REGISTER_GLOBAL("tvm.codegen.llvm.target_Matrix-DSP")
   .set_body([](const TVMArgs& targs, TVMRetValue* rv) {
      CodeGenLLVM* cg = new CodeGenMatrix-DSP( );
      *rv = static_cast<void*>(cg);});
```

3.3 LLVM IR generation

The LoweredFunc generated after the tensor expression is processed by the function lower( ) actually exists in the form of a syntax tree in TVM. The syntax tree needs to include necessary nodes such as basic operation nodes Add node, Sub node and Max node, including judgments nodes with loop information such as IfThenElse node and For node, nodes used for vector optimization such as ramp node and other nodes. In TVM, VisitExpr_( ) or VisitStmt_( ) is used to complete the translation of nodes. These functions will directly call LLVM library functions to generate LLVM IR sentence by sentence. The following shows examples of functions used to translate nodes.

```
VisitExpr_(const MinNode* op) {
   llvm::Value* a = MakeValue(op->a);
   llvm::Value* b = MakeValue(op->b);
   return builder_->CreateSelect(CreateLT(op->a.dtype(), a, b), a, b);
}
```

```
VisitStmt_(const IfThenElseNode* op) {
   using llvm::BasicBlock;
   llvm::Value* cond = MakeValue(op->condition);
   BasicBlock* then_block = BasicBlock::Create(*ctx_, "if_then", function_);
   BasicBlock* end_block = BasicBlock::Create(*ctx_, "if_end", function_);
……
}
```

4 Overall system test

In order to verify the correctness of the implementation support on TVM, a vector addition written using tensor expressions is selected for testing, and the function get_source( ) provided by TVM is used to print out the LLVM IR and assembly code corresponding to Matrix-DSP. Examples and results are shown in the following.

```
Test case:
tgt="llvm -mtriple=Matrix-DSP"
n=1024
A = te.placeholder((n,), name='a')
B = te.placeholder((n,), name='b')
C = te.compute(A.shape, lambda i: A[i] + B[i], name='c')
s = te.create_schedule(C.op)
outer, inner = s[C].split(C.op.axis[0], factor=16)
s[C].vectorize(inner)
func = tvm.build(s, [A, B], target=tgt, name='add')
print(func.get_source('ll'))
print(func.get_source("asm"))
```

```
IR output result：
……
%12 = load <16 x float>, <16 x float>* %11, align 64, !tbaa !82
%13 = getelementptr inbounds float, float* %7, i64 %9
%14 = bitcast float* %13 to <16 x float>*
%15 = load <16 x float>, <16 x float>* %14, align 64, !tbaa !85
%16 = fadd <16 x float> %12, %15
%17 = getelementptr inbounds float, float* %8, i64 %9
%18 = bitcast float* %17 to <16 x float>*
store <16 x float> %16, <16 x float>* %18, align 64, !tbaa !88
……
```

```
Assembly output result：
……
vldh.L         *+ar10[0],vr51
snop        6
sfints32.M1         r2, r53
snop        2
svbcast.M1 r53, vr52
vfmulas32.M1     vr51, vr52,vr50,vr50
snop        3
sadd.M1     r12,r51, r53
sadd.M1     -1,r52, r52
smvaga.M1         r53, ar10
snop        1
vsth.L       vr50,*+ar10[0]
……
```

Combining the results of the above tests, we can see that according to the process description of the back-end support in Section 3, after adding the corresponding hardware interface and the basic function for parsing the syntax tree node, the operator can be correctly generated through the TVM to correspond to the Matrix-DSP LLVM IR and assembly code.

5 Summary

This paper introduces TVM by analyzing the multiple back-end deployment problems faced by the deep learning field, and according to the basic composition of the TVM compilation architecture, discusses the basic process of TVM deploying the network model on the hardware back-end from both front-end and back-end aspects. According to this process, TVM's support for Matrix-DSP is completed, and test cases are used to demonstrate the correctness of the implementation. Supporting Matrix-DSP on TVM is of great significance to broadening the application scenarios of the chip.

Reference

1. Tianqi Chen, Thierry M, Ziheng Jiang,et al . TVM: an automated end-to-end optimizing compiler for deep learing[C] // In Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI'18). USENIX Association, USA, 2018:579–594.
2. Jared Roesch, Steven Lyubomirsky, Logan Weber, et al . Relay: a new IR for machine learning frameworks. In Proceedings of the 2nd ACM SIGPLAN International

Workshop on Machine Learning and Programming Languages (MAPL 2018). Association for Computing Machinery, New York, NY, USA, 2018:58–68.

3. Chris Lattner，Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization. IEEE Computer Society, 2004:75.
4. Tianqi Chen, Lianmin Zheng, Eddie Yan,et al. Learning to optimize tensor programs. In Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18). Curran Associates Inc., Red Hook, NY, USA, 2018: 3393–3404
5. Adam Paszke, Sam Gross, Francisco Massa,et al. PyTorch: An imperative style, high-performance deep learning library. In Advances in Neural Information Processing Systems,2019: 8024–8035.
6. Martín Abadi, Paul Barham, Jianmin Chen, et al. Tensorflow: A system for large-scale machine learning. In 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16),2016:265–283.
7. Jonathan R K, Connelly B, Andrew A, et al. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines[J]. SIGPLAN Not. 48, 2013:519–530.
8. D N Parikh, J Huang, M E Myers, et al. Learning from Optimizing Matrix-Matrix Multiplication[C] // 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Vancouver, BC, 2018.
9. Li Mingzhen, Liu Yi, Liu Xiaoyan, et al. The Deep Learning Compiler: A Comprehensive Survey[J]. 2020.
10. Sihang Zhou, Xinwang Liu, Miaomiao Li, et al. Multiple Kernel Clustering With Neighbor-Kernel Subspace Segmentation. IEEE Trans Neural Netw Learn Syst. 2020 Apr;31(4):1351-1362. doi: 10.1109/TNNLS.2019.2919900. Epub 2019 Jun 28. PMID: 31265409.