Optimization of Tensor Operation in Compiler

Chenguang Qiu[1](✉), Jun Wu[2], Haoqi Ren[1], and Zhifeng Zhang[1]

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

Abstract. This paper proposes an AI compiler architecture, which can compile the trained model and deploy it on DSP chip. The biggest difficulty in deploying the reasoning model on DSP is the multiplication between tensors. Tensor multiplication is the main operation and the most time-consuming operation in the process of model reasoning. Therefore, the operation efficiency of tensor multiplication directly restricts the performance of reasoning. However, there is no matrix computing unit in DSP chip, instead of vector computing unit. We define a new dialect in MLIR(Multi-Level Intermediate Representation) to efficiently compile AI models, especially GEMM and conv operations. The dialect is based on the basic features of mhlo, so this new dialect can make full use of the existing optimized pass of mhlo. Moreover, we have added some functions to support architecture related optimization, mainly the lower algorithm of operation, such as GEMM and conv. we finally map dialect to LLVM dialect and convert it into LLVM IR(immediate representation). The advantage of converting to LLVM IR is that more detailed instruction scheduling can be carried out at the backend of the compiler. We compare the efficiency of a speech model in the code generated by the traditional compiler clang and the code generated by our compiler. The experimental results show that this conversion method has greatly improved the efficiency.

Keywords: MLIR · Deep learning · Compiler · Vector processor

1 Introduction

AI development tools were designed for the data center infrastructure behind applications like internet queries, voice search, and online facial recognition. But as AI technology advances, so does the desire to leverage it in all sorts of use cases – including those that run on small, resource-constrained, MCU-based platforms at the edge. So instead of focusing solely on high-end hardware accelerators running cloud-based recommendation systems, for example, tools like compilers must also be able to optimize AI data and algorithms for smaller footprint devices.

High performance digital signal processor (DSP) is a high-performance processor composed of VLSI chips for signal processing. It is mainly used to realize various

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2023
Published by Springer Nature Switzerland AG 2023. All Rights Reserved
F. Gao et al. (Eds.): ChinaCom 2022, LNICST 500, pp. 207–219, 2023.
https://doi.org/10.1007/978-3-031-34790-0_16

digital signal processing algorithms in real time and quickly, especially various audio and video processing algorithms. DSP generally has a multi-bus structure, the data storage space is separated from the program space, and has an independent data bus and address bus. It can fetch instructions and read data at the same time. DSP has an efficient hardware multiplier, which can complete the multiplication instructions in a short period, and speed up the multiplication operations such as FFT, matrix operation, convolution, digital filtering and so on. Therefore, DSP chip is very suitable for the reasoning of audio processing model. DSP chips are widely used in all kinds of smart phones. In addition, the scene of smart home needs frequent voice interaction with users. With the help of the trained effective AI model and the powerful computing power of DSP, speech recognition can be carried out more accurately and timely.

Matrix multiplication calculation is the core of many architectures based on transformer (such as BERT). It is the key factor restricting the speed of model training and reasoning. The operation efficiency of matrix multiplication can be used as an important index to measure the compilation efficiency. However, compared with some hardware designed for AI operation, DSP lacks matrix operation unit. Compared with other embedded processors, DSP has vector operation unit, that is, SIMD instruction. The training of the algorithm and the implementation of the reasoning algorithm are the two most important parts, which also need the most computing resources. Compared with training, the hardware resources required for model reasoning are much smaller. In addition, the tensor dimension used by models such as speech recognition is small. These factors improve the feasibility of deploying the model reasoning process on DSP chip.

At present, various high-level IR optimizations focus on optimizing the combination form of the kernel, rather than the code generation of a single kernel. And most of the implementation of kernel is still based on handwritten assembly library. This method can make great use of the ability of hardware, but it also loses the opportunity to generate better code to a certain extent, because the compiler is difficult to optimize deeply from a global perspective.

MLIR is a general-purpose IR that also supports hardware specific operations. Many hardware targets can use this infrastructure and will benefit from it. MLIR is more suitable for high-level compilation optimization, but not for generating final machine code. Moreover, LLVM IR does not provide good support for matrix data types and GEMM, which is consistent with the SIMD operation characteristics of DSP [1–3].

Therefore, this paper proposes a method to convert the kernel in AI model into DSP supported operations through the MLIR framework. This process converts the matrix multiplication operation lower into the LLVM IR representation, and generates machine code with the help of the mature compiler back-end LLC. The front end is mainly responsible for converting the trained model into DSP dialect, and doing some operator level optimization on the intermediate code of DSP dialect, and generating LLVM IR code with the help of the infrastructure of MLIR. Finally, the LLVM IR is handed over to the LLVM backend for more architecture related instruction level optimization, and finally binary machine code is generated. Compared with traditional methods, the proposed method can convert operators into corresponding machine opcode at the IR level, and make use of various optimization passes provided by the LLVM backend.

2 Relate Work

MLIR is a basic compilation framework, which is mainly used in AI compilation. MLIR is committed to building reusable, scalable and powerful compilers, and can be combined with LLVM to take advantage of the existing back-end compilation framework [4, 5]. Thanks to its hybrid IR structure, MLIR can support a variety of different needs under the condition of providing a unified infrastructure. MLIR can represent data flow graph, including dynamic shape, user extensible OP ecosystem, tensorflow variable, etc. In addition, MLIR can optimize loops in high-performance computing across cores, such as fusion, loop switching, tiling and so on. MLIR can also convert the memory layout of data to suit different hardware architectures. MLIR can complete the "reduced" conversion of code generation, such as DMA insertion, explicit cache management, memory tiling, and vectorization of one-dimensional and two-dimensional register architecture. MLIR can decompose some patterns into more fine-grained combinations of small local patterns through pass, and supports rewriting specific patterns at the granularity of a single operation [17, 23].

TVM / NNVM mainly hopes to reduce the gap between the deep learning framework and the underlying hardware[6]. However, on different hardware, there are inevitable differences in various hardware resources, such as memory, L1 / L2 cache, bandwidth, etc. Therefore, TVM / NNVM adopts the philosophy of separating calculation from schedule. All hardware platforms share the compute attribute to ensure the consistency of the final results; Different hardware platforms enjoy the schedule attribute exclusively according to their own characteristics to ensure the efficiency of their execution [24–27]. TVM is mainly responsible for how to compile the operators in the calculation diagram into code that can be executed efficiently on different hardware. This can be abstracted into an optimization problem. Therefore, in this process, TVM will also use the method of deep learning to optimize different hardware platforms, which is called autotvm. Using automation methods such as deep learning to optimize different hardware platforms can be well applicable to the diversity of end-to-end devices in the future [10–12].

XLA (accelerated linear algebra) is a compiler for tensorflow launched by Google in 2017. XLA uses JIT compilation technology to analyze tensorflow diagrams created by users at runtime, converts tensorflow OP into HLO (high level optimizer) intermediate representation [7], and completes various graph optimization including OP fusion on the HLO layer. Finally, it completes the automatic generation of CPU / GPU and other machine codes based on LLVM. XLA adopts a relatively simple technical path [8, 9, 17]. For computationally intensive operators with high requirements for automatic CodeGen, such as matmul / convolution, like tensorflow, they will directly call libraries such as cuBLAS/cuDNN; For other memory access intensive operators, XLA will conduct fully automatic OP fusion and underlying code generation (CodeGen) [23–27]. In addition to the compilation body, XLA also includes a set of static execution engines. This statically is reflected in the static fixed shape compilation (that is, a complete compilation is carried out for each set of input shapes at run time and the compilation results are retained), the static operator scheduling sequence, and the static display memory / memory optimization. It is expected that better performance / storage optimization results can be obtained compared with tensorflow for the dynamic interpretation and execution of the calculation graph [13–15].

LLVM began a research project at the University of Illinois. Its goal is to provide a modern SSA based compilation strategy that can support static and dynamic compilation of any programming language [1–3]. LLVM has developed into a comprehensive project composed of many sub-projects, many of which are widely used in the production of various commercial and open source projects, as well as academic research. The LLVM core library provides a modern optimizer independent of source code and target, and provides code generation support for many popular CPUs (and some less common CPUs). These libraries are built around an intermediate code representation called LLVM IR. Clang is the compiler front end provided by LLVM, which can provide fast compilation and accurate error and warning information. LLC is responsible for IR parsing, instruction selection, optimization, register allocation, assembly code generation, machine code generation and other functions. LLVM intermediate code is an IR (immediate representation) in the form of SSA. The instruction set used is LLVM virtual instruction set. The instruction set is a three address instruction set similar to RISC (reduced instruction set computer). It contains simple control instructions and memory access instructions with type pointers. It has syntax independent of high-level language and target processor, and is easy to conduct code analysis and optimization. LLVM intermediate code instruction set has rich types of instructions, including scalar instructions such as bit and bit by bit instructions and vector instructions such as extraction element, insertion element and shift [28–31].

3 Implement

3.1 Compiler Structure

As shown in Fig. 1 Compiler framework, the compiler is mainly divided into two parts. The front end supports the input of various models, such as tensorflow and pytorch. MLIR provides a basic framework for converting these models into corresponding dialects. XLA can convert tensorflow graphs into HLO dialects, which is a dialect supported by MLIR. Then, using the basic framework of MLIR, the compiler can convert HLO dialect to dialect designed for DSP architecture [18–20], and finally convert it to LLVM IR. The compiler will perform some high-level operator optimization in the MLIR phase, including data format conversion, dead code elimination, Op fusion and other operations. We call the infrastructure provided by MLIR to complete these optimization. In addition, we can also do some architecture related optimization at the front end, which is also the focus of this article, the lower optimization of tensor multiplication.

LLC is a reusable compiler back-end, which does not distinguish the front-end language framework, so we can use LLC to complete the process from LLVM to machine code. LLC takes LLVM IR as input, and finally generates machine code through instruction selection, register allocation and other processes.

The SIMD instruction of DSP provides powerful vector operation capability [16].First, we define the corresponding instruction at the backend and provide the form of intrinsics function to call the intrinsics function in LLVM IR. Assume that the vector width supported by DSP is 2048bit (Fig. 2).

In order to realize architecture related optimization in the MLIR phase, it is necessary to call DSP supported instructions in the MLIR. Therefore, we need to expose some

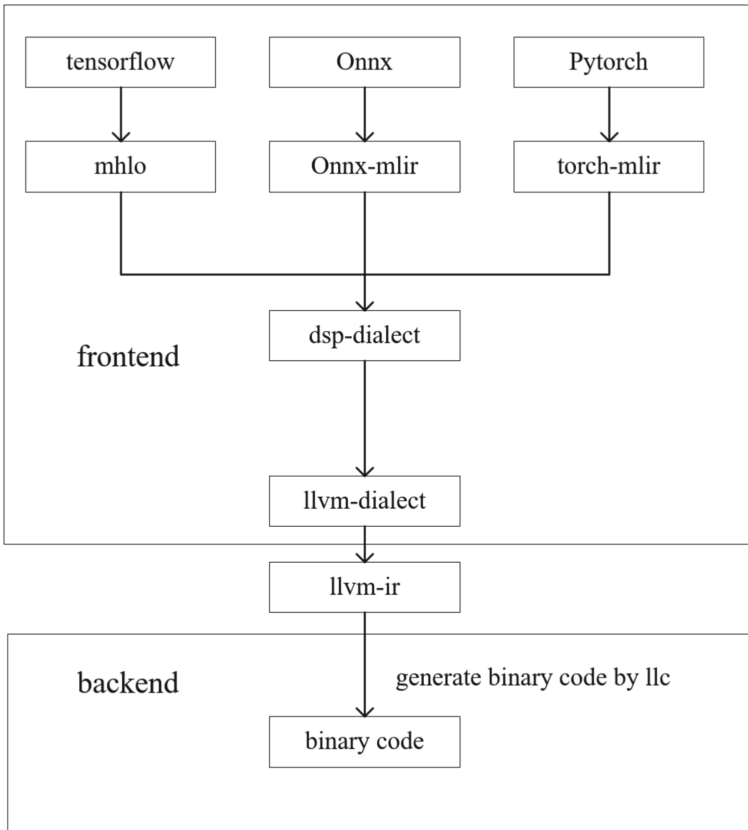


Fig. 1. Compiler framework.

instructions in the DSP to the front end of the MLIR in the form of intrinsic instructions. And rewrite the conversion pattern in the process of dialect conversion to complete the conversion from matrix multiplication to vector multiplication. For other operations, we convert them to affine and other dialects, which are the infrastructure provided by MLIR, and for which MLIR defines the relevant patterns converted to LLVM. Through the delivery optimized infrastructure provided by MLIR, all the code is finally lowered to the LLVM IR phase (Fig. 3).

```
def int_dsp_vfmul_10  :GCCBuiltin<"__builtin_dsp_vfmul_10">,
        Intrinsic<[llvm_v256i8_ty], [llvm_v256i8_ty, llvm_v256i8_ty],
                  [IntrNoMem]>;

def int_dsp_vfmul_20  :GCCBuiltin<"__builtin_dsp_vfmul_20">,
        Intrinsic<[llvm_v128i16_ty], [llvm_v128i16_ty, llvm_v128i16_ty],
                  [IntrNoMem]>;

def int_dsp_vfmul_40  :GCCBuiltin<"__builtin_dsp_vfmul_40">,
        Intrinsic<[llvm_v64i32_ty], [llvm_v64i32_ty, llvm_v64i32_ty],
                  [IntrNoMem]>;

def int_dsp_vfmac_10  :GCCBuiltin<"__builtin_dsp_vfmac_10">,
        Intrinsic<[llvm_v256i8_ty], [llvm_v256i8_ty, llvm_v256i8_ty, llvm_v256i8_ty],
                  [IntrNoMem]>;

def int_dsp_vfmac_20  :GCCBuiltin<"__builtin_dsp_vfmac_20">,
        Intrinsic<[llvm_v128i16_ty], [llvm_v128i16_ty, llvm_v128i16_ty, llvm_v128i16_ty],
                  [IntrNoMem]>;

def int_dsp_vfmac_40  :GCCBuiltin<"__builtin_dsp_vfmac_40">,
        Intrinsic<[llvm_v64i32_ty], [llvm_v64i32_ty, llvm_v64i32_ty,llvm_v64i32_ty],
                  [IntrNoMem]>;
```

Fig. 2. Intrinsic function.

```
def LLVM_FmacOp : LLVM_BinaryIntrinsicOp<"Fmac">;
def LLVM_VaddOp : LLVM_BinaryIntrinsicOp<"Vadd">;
def LLVM_VsumOp : LLVM_BinaryIntrinsicOp<"Vsum">;
def LLVM_VsubOp : LLVM_BinaryIntrinsicOp<"Vsub">;
```

Fig. 3. Mlir intrinsic op.

3.2 Tensor Multiply

Tensor multiplication in AI model is mainly divided into two types: the first is full connection and dot. The two inputs of this tensor multiplication are feature and weight. Feature is the feature extracted from the input data, which can only be determined at runtime. Weight is the value that is continuously updated by iteration through the back-propagation algorithm during the model training process. Therefore, the weight data is represented as static data in the model reasoning process, that is, the data that can be determined in the model compilation process, which is similar to the literal constant in the C language program. Therefore, we can optimize the layout of weight data during compilation, and convert it to a data format more suitable for SIMD instructions, so as to speed up the operation.

The arrangement of data has a great impact on matrix multiplication. At present, in order to adapt to the tensor core or matrix operation unit, most feature matrices are arranged in rows, while weight is transposed, which is arranged in columns. This format is very suitable for chips with matrix computing units, and has the advantage of sequential access to memory. However, for deploying voice model on DSP, storing weight matrix

by line can generate better code. Therefore, in order to make full use of the operation characteristics of DSP, we can transpose the weight matrix arranged in columns. We can insert a transfer node into DSP dialect to represent the transpose of weight matrix. However, the DSP chip does not provide matrix transpose instructions, or load and store instructions accessed according to stripe. One method is to insert transpose instructions during compilation, but the load and store instructions are expensive, and additional memory is required to implement matrix transpose, which puts great pressure on the memory of DSP chip. However, the weight data is obtained in the training process and remains unchanged in the reasoning process, so we can transpose it in the compilation stage.

The second is that the two tensors that perform operations are dynamic, and the data is determined at runtime. This kind of tensor is obtained through pre-node operation. Because the pre-running and post-running environment cannot be determined for this kind of data, its data format cannot be changed in the compiler. Fortunately, current compiler frameworks are arranged in rows for dynamic tensors. Therefore, we propose an algorithm to realize matrix multiplication by exchanging matrix operation sequence. This algorithm does not need additional memory overhead, and is better than transpose algorithm in time complexity.

In the traditional algorithm, one value of an objective matrix is obtained each time. For example, for the evaluation of multiplication, $C = AB$, we can get the value of c_{11} by $c_{11} = a_{11}b_{11} + a_{12}b_{21} + \cdots + a_{1m}b_{m1}$. the basic idea of our algorithm is to obtain only one item of all elements in all rows of the objective matrix each time, and continue to multiply and accumulate. Finally, after m cycles, the elements in one row of the objective matrix are obtained.

The description of the algorithm is as follows:
Input: matrix A(m*n) and matrix B(n*k), both of which are closely arranged in rows.
Output: product of matrix A and matrix B.

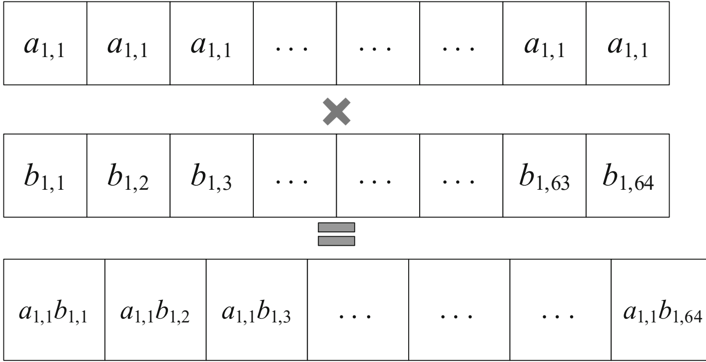
Because the pipeline of DSP chip has been carefully designed and optimized, it is considered that after the pipeline is started, the average execution cycle of each instruction is 1 cycle, that is, the next instruction can directly use the calculation result of the previous instruction without waiting.

For i = 0 to n:
For j = 0 to m:

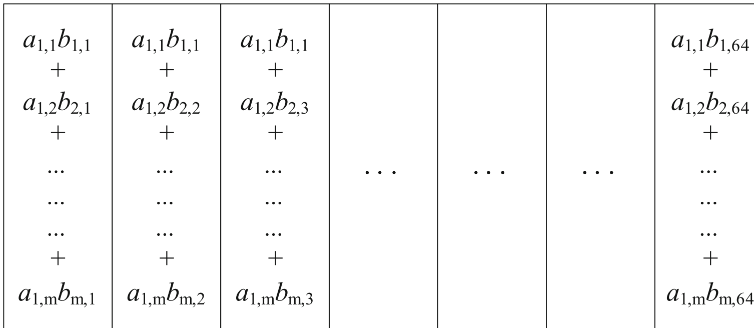
1. vloadr a_{ij} : there are 64 a_{ij} duplicate in a vector register VR1
2. load $b_1, b_2, \cdots, b_{jg}$ to vector register
3. vfmac VR0, VR1, VR2: multiply VR1 and VR2, add their results to VR0, and store the results in VR0

store VR0: 64 target values can be calculated per cycle

So the final time complexity is: $3mnk/64$.
Inner circulation diagram:



outer circulation diagram:



The image below is part of the LLVM IR generated by the compiler front end, corresponding to the innermost loop in the algorithm:

```
1  for.body:
2    %j.019 = phi i32 [ 0, %for.cond1.preheader ], [ %inc, %for.body4 ]
3    %0 = call <64 x i32> @LLVM.dsp.vloadr.32(i32 x, i32 x) #1, !dbg !49
4    %1 = call <64 x i32> @LLVM.dsp.vload.32(i32 x, i32 x)
5    %2 = call <64 x i32> @LLVM.dsp.vfmac.40(<64 x i32> %3, <64 x i32> %4, <64 x i32> %5) #3, !dbg !43
```

3.3 Kernel Fusion

The basic form of full connection formula in deep learning is that $Y = WX + b$, we can see that there is an offset b in the formula. The general operator method is to multiply first and then add once. In the stage of model reasoning, we can fuse the two operations through reasonable optimization. Therefore, based on the matrix multiplication proposed above, the offset calculation algorithm is integrated as follows.

The description of the algorithm is as follows:
Input: matrix A(m*n) and matrix B(n*k), both of which are closely arranged in rows.
Output: product of matrix A and matrix B.

Because the pipeline of DSP chip has been carefully designed and optimized, it is considered that after the pipeline is started, the average execution cycle of each instruction is 1 cycle, that is, the next instruction can directly use the calculation result of the previous instruction without waiting.

For i = 0 to n:
For j = 0 to m:

1. vloadr a_{ij} : there are 64 a_{ij} duplicate in a vector register VR1
2. load $b_{j1}, b_{j2}, \cdots, b_{jg}$ to vector register
3. vfmac VR0, VR1, VR2: multiply VR1 and VR2, add their results to VR0, and store the results in VR0

vadd VR0, VR0, VR3: add VR0 with VR3, and store result to VR0

store VR0: 64 target values can be calculated per cycle

By operator coincidence, we can optimize the space-time cost of repeated load of the target matrix, so as to improve the performance.

3.4 Loop Tiling

Tensor multiplication is a typical computation intensive operator, but it still needs to access a large amount of memory space. In order to speed up matrix operation, we should reduce the overhead caused by memory access as much as possible. When the input matrix used for tensor multiplication cannot be placed in the SRAM, and in order to adapt to the computational bit width of the vector instruction, the data needs to be divided into tiles so that each tile can be placed in the SRAM.

3.5 Quantization

The training of neural network is a process of continuously fine adjustment of weights, which usually requires floating-point precision representation and operation, and cannot be directly replaced by fixed-point numbers. However, in the model prediction stage, due to the strong robustness of the deep neural network model, it can well deal with a certain intensity of input noise and eliminate the interference of irrelevant information in the data. Therefore, if the low-precision operation is regarded as a noise source, the neural network model should be able to give relatively accurate results. Therefore, we reduce the precision of the tensor multiplication of fp32 to fp16 for operation, which greatly improves the operation efficiency while retaining the expected results [32].

4 Evaluation

Based on the manually written assembler library, we compare the performance of a single operator with that of the whole model by taking the cycles required for the end of the running of the machine code generated by the compiler as the performance index.

Take the 1024*1024 tensor multiplication operator as an example. At the single operator scale, because we have optimized the operator implementation at the operator level, the code generated by the AI compiler and the handwritten assembly library have similar performance.

However, in AI model reasoning with more operators, the AI compiler shows better performance because the AI compiler supports both operator level and instruction level optimization, and can schedule instructions across operators. Instructions between operators without dependency can be executed in parallel, further improving performance. As shown in Fig. 4. Performance comparison between mlir compiler and tradition compiler which uses assembler library, in the test of a part of the recommendation model, the AI compiler achieved 21.27% performance optimization compared with the manually optimized operator library. The Fig. 5 explains why code generation through IR is more advantageous in the model formed by the combination of multiple operators. Store operations of different operators and computations of subsequent operators, such as vfmac, can be executed in parallel.

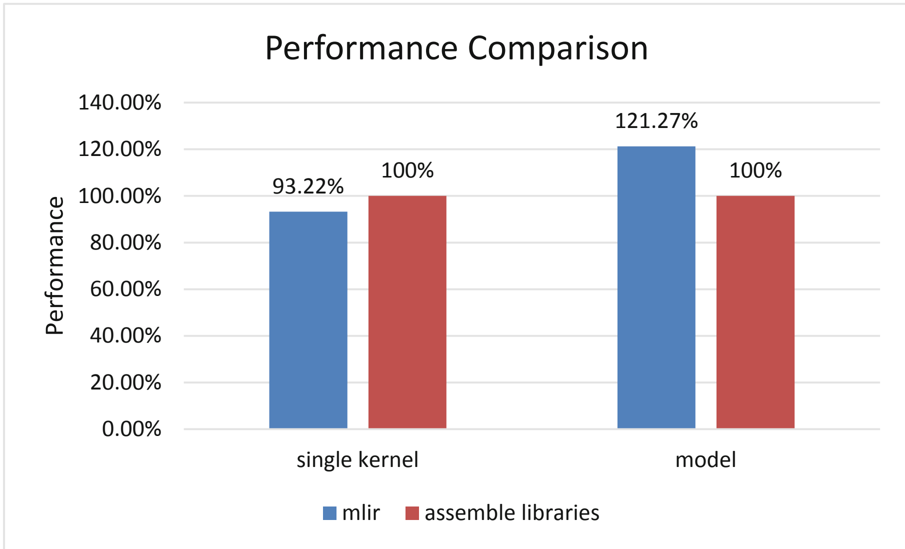


Fig. 4. Performance comparison between mlir compiler and tradition compiler which uses assembler library.

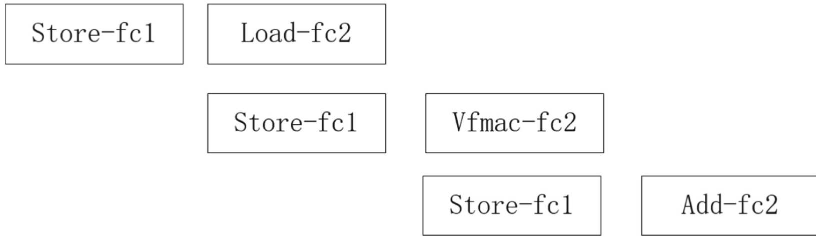


Fig. 5. Instructions in parallel

5 Conclusion

This paper presents an AI compiler, which can deploy the trained AI model to the embedded processor DSP. This compiler architecture can optimize the operator level and instruction level, and further reduce the model reasoning overhead. This AI compiler can well lower the important tensor multiplication operation in AI reasoning to the SIMD processor supported by DSP, so as to support AI reasoning on the embedded processor, and its performance can be comparable to that of the handwritten assembly operator library. This work can reduce the overhead of manually designing algorithm libraries for various AI chips.

Acknowledgement. The authors would like to thank the editors and the reviewers for providing comments and suggestions for this paper. This work was supported by National Key R&D Program of China under Grant 2020YFA0711400, National Natural Science Foundation of China under Grants 61831018 and U21A20452, the Jiangxi Double Thousand Plan under Grant jxsq2019201125, and the S&T plan projects of Jiangxi Province Education Department GJJ201003.

References

1. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis and transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, ser. CGO 2004. IEEE Computer Society, Washington, DC, p. 75 (2004). http://dl.acm.org/citation.cfm?id=977395.977673
2. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst., **13**(4), 451–490 (1991). https://doi.org/10.1145/115372.115320
3. Vasilache, N., et al.: The next 700 accelerated layers: from mathematical expressions of network computation graphs to accelerated GPU kernels, automatically. ACM Trans. Archit. Code Optim. **16**(4), pp. 38:1–38:26 (2019). https://doi.org/10.1145/3355606
4. Lattner, C., et al.: MLIR: scaling compiler infrastructure for domain specific computation. In: 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 2–14 (2021). https://doi.org/10.1109/CGO51591.2021.9370308
5. Schweitz, E.: An MLIR dialect for high-level optimization of fortran. In: LLVM Developer Meeting, October 2019

6. Li, M., et al.: The deep learning compiler: a comprehensive survey. IEEE Trans. Parallel Distrib. Syst. **32**(3), 708–727, 1 March 2021. https://doi.org/10.1109/TPDS.2020.3030548
7. Abadi, M., et al.: TensorFlow: a system for large-scale machine learning. In: Proceedings of 12th USENIX Symposium on Operating Systems Design Implementation, pp. 265–283 (2016)
8. Long, G., Yang, J., Zhu, K., Lin, W.: FusionStitching: deep fusion and code generation for tensorflow computations on GPUs (2018). arXiv:1811.05213
9. Xing, Y., Weng, J., Wang, Y., Sui, L., Shan, Y., Wang, Y.: An in-depth comparison of compilers for deep neural networks on hardware. In: Proceedings of IEEE International Conference on Embedded Software Systems, pp. 1–8 (2019)
10. 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)
11. Adachi, Y., Kumano, T., Ogino, K.: Intermediate representation for stiff virtual objects. In: Proceedings Virtual Reality Annual International Symposium 1995, pp. 203–210. IEEE (1995)
12. 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)
13. 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)
14. Wei, R., Schwartz, L., Adve, V.: DLVM: a modern compiler infrastructure for deep learning systems. arXiv preprint arXiv:1711.03016 (2017)
15. Griewank, A., Walther, A.: Evaluating derivatives: principles and techniques of algorithmic differentiation. SIAM (2008)
16. Ren, H., Zhang, Z., Jun, W.: SWIFT: A Computationally-intensive DSP architecture for communication applications. Mob. Netw. Appl. **21**(6), 974–982 (2016)
17. Mullapudi, R.T., Vasista, V., Bondhugula, U.: PolyMage: automatic optimization for image processing pipelines. In: International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 429–443 (2015)
18. Zerrell, T., Bruestle, J.: Stripe: tensor compilation via the nested polyhedral model. CoRR, vol. abs/1903.06498 (2019). http://arxiv.org/abs/1903.06498
19. Zhou, Y., Qin, J., Chen, H., Nunamaker, J.F.: Multilingual web retrieval: an experiment on a multilingual business intelligence portal. In: Proceedings of the 38th Annual Hawaii International Conference on System Sciences, Big Island, HI, USA, p. 43a (2005)
20. Korra, R., Sujatha, P., Chetana, S., Naresh Kumar, M.: Performance evaluation of Multilingual Information Retrieval (MLIR) system over Information Retrieval (IR) system. In: 2011 International Conference on Recent Trends in Information Technology (ICRTIT), Chennai, India, pp. 722–727 (2011)
21. Yang, H., Lee, C.: Multilingual Information Retrieval Using GHSOM. In: 2008 Eighth International Conference on Intelligent Systems Design and Applications, Kaohsuing, Taiwan, pp. 225–228 (2008)
22. Curzel, S., et al.: Automated generation of integrated digital and spiking neuromorphic machine learning accelerators. In: 2021 IEEE/ACM International Conference on Computer Aided Design (ICCAD), Munich, Germany, pp. 1–7 (2021)
23. Tian, R., Guo, L., Li, J., Ren, B., Kestor, G.: A high performance sparse tensor algebra compiler in MLIR. In: 2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC), St. Louis, MO, USA, pp. 27–38 (2021)
24. Wei, W., Zeng, Q., Ye, T., Lomone, D.: Adaptive differentiated integrated routing scheme for GMPLS-based optical Internet. J. Commun. Netw. **6**(3), 269–279 (Sept. 2004)

25. Li, H., Peng, Y.: Effective multi-level image representation for image categorization. In: 2010 20th International Conference on Pattern Recognition, Istanbul, Turkey, pp. 1048–1051 (2010)
26. Zhuhadar, L., Nasraoui, O., Wyatt, R., Romero, E.: Multi-language ontology-based search engine. In: 2010 Third International Conference on Advances in Computer-Human Interactions, Saint Maarten, Netherlands Antilles, pp. 13–18 (2010)
27. Siemieniuk, A., et al.: OCC: an automated end-to-end machine learning optimizing compiler for computing-in-memory. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **41**(6), 1674–1686 (June2022)
28. Komisarczyk, K., Chelini, L., Vadivel, K., Jordans, R., Corporaal, H.: PET-to-MLIR: a polyhedral front-end for MLIR. In: 2020 23rd Euromicro Conference on Digital System Design (DSD), Kranj, Slovenia, pp. 551–556 (2020)
29. Junod, P., Rinaldini, J., Wehrli, J., Michielin, J.: Obfuscator-LLVM - software protection for the masses. In: 2015 IEEE/ACM 1st International Workshop on Software Protection, Florence, Italy, pp. 3–9 (2015)
30. Wei, J., Thomas, A., Li, G., Pattabiraman, K.: Quantifying the accuracy of high-level fault injection techniques for hardware faults. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Atlanta, GA, USA, pp. 375–382 (2014)
31. Sharma, V.C., Haran, A., Rakamaric, Z., Gopalakrishnan, G.: Towards formal approaches to system resilience. In: 2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing, Vancouver, BC, Canada, pp. 41–50 (2013)
32. Alvarez, R., Prabhavalkar, R., Bakhtin, A.: On the efficient representation and execution of deep acoustic models. CoRR, abs/1607.04683 (2016)