Automatic Deep Learning Parallelization for Vector Multicore Chips with the OSCAR Parallelizing and the TVM Open-Source Deep Learning Compiler

Fumiaki Onishi[1], Ryosei Otaka[1], Kazuki Fujita[1], Tomoki Suetsugu[1], Tohma Kawasumi[1], Toshiaki Kitamura[1,2], Hironori Kasahara[1,3], and Keiji Kimura[1,4]

[1] Waseda University,27 Waseda-machi, Shinjuku-ku, Tokyo, 1620042 {nueshitone, suan, kazuki_fujita, sue2gu, tohma}@kasahara.cs.waseda.ac.jp
[2] toshi.kitamura@aoni.waseda.jp
[3] kasahara@waseda.jp
[4] keiji@waseda.jp

Abstract. Deep Neural Networks (DNNs) are indispensable for AI robots and autonomous driving vehicles. Low power and high performance of DNN processors are critical to realize long battery life and long-term reliable controller operations, keeping their actions flexible. In addition, high program productivity is also essential for their cost-effective product development. We have been developing the OSCAR parallelizing and power-reducing compiler and its co-designed homogeneous and heterogeneous multicore processor chips. In this paper, the TVM, an open-source deep learning compiler, is utilized with the OSCAR compiler to automatically parallelize various DNN inference models. However, the current TVM does not generate a C program compatible with vectorization. In this paper, we propose a code generation method for TVM to generate vectorization-friendly code by transforming the memory layout of tensors to keep a long vector length at an innermost vectorized loop. The parallelized coarse grain task parallelization program is translated into NEC machine code with vector instructions by the NEC compiler. The execution performance of the proposed method with pre-trained DNN inference models is evaluated on NEC SX-Aurora TSUBASA vector multicore. The evaluation result shows that the proposed method achieves 31.3× speedup on seven cores with a ResNet model and 37.6× speedup with a VGG model, compared with the compilation flow that does not include the proposed method.

Keywords: Vector multicore · Parallelization compiler · Vectorization · Deep neural network · TVM.

1 Introduction

Emerging AI robots and autonomous driving vehicles rely on sophisticated deep neural network (DNN) technologies, such as image and voice recognition [1].

They utilize these DNN technologies to recognize the surrounding environment and decide the most suitable action in the future.

These DNN processes require high-clock frequency, many computational resources, and high memory bandwidth, particularly for convolutional neural network (CNN) processing in image recognition. However, these requirements lead to high-energy dissipation and high-heat dissipation. This is a problem for AI robots and vehicles since they need large battery and cooling modules to handle the energy and heat dissipation. These modules can reduce the flexibility of their actions. Thus, low-power processors are expected to enable the necessary DNN calculations for AI robots and autonomous vehicles on a limited energy budget.

To overcome this problem, parallel processing along with controlling the processor's clock frequency and voltage is one of the promising approaches. If a target program has a real-time deadline, parallel processing with more processing elements (PEs) can satisfy it with a lower clock frequency and voltage, resulting in lower power dissipation since the power dissipation is proportional to the cube of the clock frequency [2]. Appropriately scheduling the number of PEs and clock frequency according to the required amount of computation can realize high-performance and low-power AI processors.

Based on this approach, we have developed the OSCAR automatically parallelizing compiler cooperative OSCAR vector multicore [3–5]. Each core in the OSCAR vector multicore consists of a CPU (RISC-V), a vector accelerator, a data transfer unit (DTU), local data memory (LDM), and distributed shared memory (DSM). The OSCAR compiler controls the OSCAR vector multicore by performing vectorization, parallelization, memory usage optimization, and power optimization [6]. The vector accelerator module is suitable for processing DNN applications, taking into consideration memory access overhead.

While the OSCAR compiler can automatically employ parallelization and power control for C and Fortran programs, DNN models are usually developed on DNN frameworks such as TensorFlow [7] and PyTorch [8]. To deal with many existing DNN inference models created by these frameworks on the OSCAR vector multicore, we incorporate the deep learning compiler stack TVM [9] into the compilation flow. TVM takes a DNN model represented in the standard DNN exchange format ONNX [10] and generates a C program. Then, the OSCAR compiler performs the optimization above and develops the parallelized C programs for CPU and vector cores. Finally, the backend compilers for the target CPU and the accelerator core generate the executable binary. Fig. 1 depicts this compilation flow.

Though introducing TVM can handle many existing DNN models, it does not generate innermost loops whose length is sufficiently long for efficient vectorization. Therefore, this paper proposes a code generation method for TVM that transforms the tensor layout in a learning model to generate code whose innermost loops have sufficient loop length suitable for vectorization. We implement the proposed method in TVM and evaluate it on the NEC vector multicore personal supercomputer SX-Aurora TSUBASA instead of the currently developing OSCAR vector multicore. The OSCAR compiler parallelizes the TVM-generated

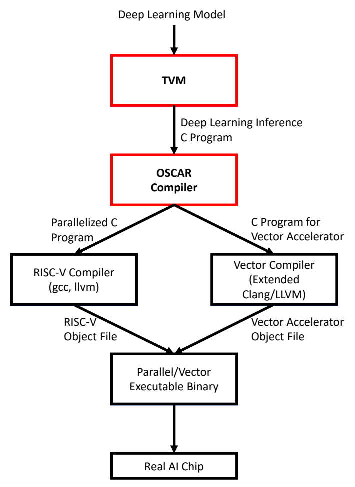

Deep Learning Model
TVM
Deep Learning Inference C Program
OSCAR Compiler
Parallelized C Program
C Program for Vector Accelerator
RISC-V Compiler (gcc, llvm)
Vector Compiler (Extended Clang/LLVM)
RISC-V Object File
Vector Accelerator Object File
Parallel/Vector Executable Binary
Real AI Chip

Fig. 1. Compilation flow of deep learning models using TVM and OSCAR

code, and the NCC vectorizes the OSCAR-parallelized code in this evaluation. The main contribution of this paper can be summarized as follows:

- We propose a vector accelerator-friendly TVM code generation method.
- We construct a vector multicore compilation flow consisting of the extended TVM, the OSCAR parallelizing compiler, and NCC.
- We evaluate the proposed method on a real vector multicore, NEC SX-Aurora TSUBASA.

The rest of this paper is organized as follows: Section 2 describes OSCAR automatically parallelizing compiler. Section 3 explains an overview of TVM, followed by the description of the proposed code generation method in Section 4. Section 5 reports the evaluation result. Finally, Section 6 concludes this paper.

2 OSCAR Compiler

The OSCAR automatically parallelizing compiler is an automatic parallelizing compiler for C and Fortran programs. It works as a source-to-source compiler. It generates a parallelized C or Fortran program by inserting OSCAR API directives, an OpenMP-compatible parallelizing API [11]. The compiler performs multi-grain parallel processing [12]. Multi-grain parallel processing includes three kinds of parallel processing as follows:

- Coarse-grained task parallel processing for parallelism among basic blocks, loops, and function calls.
- Loop-iteration level parallel processing.
- Near-fine-grain parallel processing for parallelism among statements within a basic block.

The OSCAR compiler consists of three modules: Frontend, Middle pass, and Backend. First, the Frontend translates a source C or Fortran program into the compiler's intermediate representation (IR) through lexical and syntax analysis. Next, the Middle pass decomposes the obtained IR into basic blocks, repetition blocks, and subroutine blocks, each of which is a parallelization unit of coarse-grain task parallel processing, and they are called macro tasks (MTs). The Middle pass analyzes control flow and data dependence among MTs and represents them as a macro flow graph (MFG). Then, it employs the earliest executable condition analysis to exploit coarse grain parallelism from MFG and represents its result as a macro task graph (MTG). In addition, the Middle path performs cache, memory, and low-power optimization based on the MTG and generates an optimized IR. Finally, the Backend generates a parallelized C or Fortran program by inserting OSCAR API directives.

In this paper, we integrate TVM before the OSCAR compiler to deal with existing pre-trained DNN models, as in Fig. 1. TVM translates a model into a sequential C program; then, the OSCAR compiler performs the parallelization and corresponding optimization techniques.

3 TVM

TVM (Tensor Virtual Machine) is an open-source deep learning compiler and inference runtime. It optimizes the pre-trained deep-learning models developed by multiple deep-learning frameworks such as PyTorch, TensorFlow, and Keras. Code generation supports multiple target hardware, such as CPUs, GPUs, and DSPs. TVM optimizes the input model at the graph and operator levels, generating efficient code tailored to the target hardware backend. Graph-level optimization includes operator fusion, which combines multiple operations to reduce memory access to intermediate results, and constant folding, which pre-computes values that can be determined statically to save execution costs. Operator-level optimization includes schedule transformations determining execution details such as loop structure and parallel patterns. Fig. 2 depicts the overall structure of TVM.

The figure shows that the TVM frontend translates the model into a graph-level intermediate representation named Computational Graph. The outline of the Computational Graph is depicted in Fig. 3.

The Computational Graph is an abstract syntax tree (AST) and can be expressed as Python code or text, as depicted in the lower left of Fig. 3. TVM performs graph-level optimization on this intermediate representation, such as the fusion of convolutional operations and activation function calculations. It then performs the operator-level optimization followed by the code generation. Operator-level optimization is performed using an intermediate representation called Tensor Expression, represented in the form depicted in Fig. 4. This intermediate representation is described based on the output shape of tensors and operation rules. Operator-level optimization includes optimization such as the fusion of multiple loops in a multi-loop and sorting loop nests.

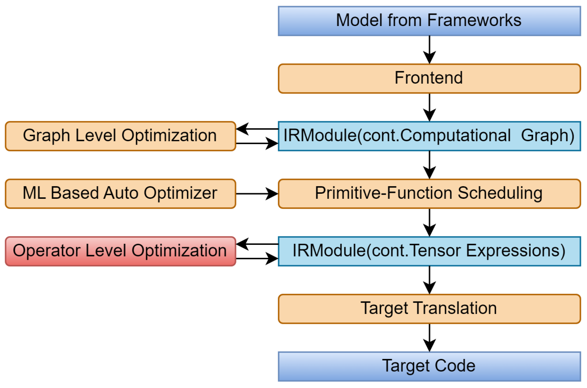
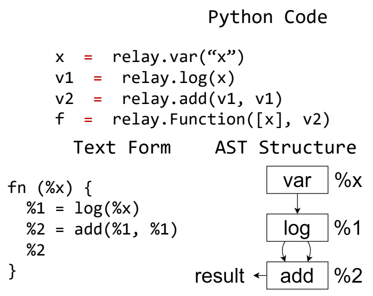
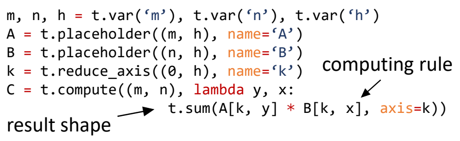

Model from Frameworks
Frontend
Graph Level Optimization
IRModule(cont.Computational Graph)
ML Based Auto Optimizer
Primitive-Function Scheduling
Operator Level Optimization
IRModule(cont.Tensor Expressions)
Target Translation
Target Code

Fig. 2. Structure of TVM [13]



```
x  =  relay.var("x")
v1  =  relay.log(x)
v2  =  relay.add(v1, v1)
f  =  relay.Function([x], v2)
```

```
fn (%x) {
  %1 = log(%x)
  %2 = add(%1, %1)
  %2
}
```



```
m, n, h = t.var('m'), t.var('n'), t.var('h')
A = t.placeholder((m, h), name='A')
B = t.placeholder((n, h), name='B')
k = t.reduce_axis((0, h), name='k')
C = t.compute((m, n), lambda y, x:
                t.sum(A[k, y] * B[k, x], axis=k))
```

Fig. 3. Computational Graph [14] **Fig. 4.** Tensor Expression [9]

We extend the TVM's compilation flow to achieve faster DNN inference execution on vector multicores.

4 Proposed TVM Extension for Vector Multicores

This section describes the proposed TVM code generation extension for DNN inference acceleration on vector multicore chips.

4.1 Tensor Layout Transformation

When vectorizing a DNN program that includes loops, tensors in the program must have a data layout that allows for efficient vectorization by ensuring continuous data element placement along a vectorizing dimension (axis) as long as possible. This feature can improve the vector pipeline's computation throughput since the data can be continuously provided to its functional units with fewer stall cycles.

Here, we focus on convolution calculation, a fundamental calculation in convolutional neural networks (CNN) used in image recognition. Fig.5 depicts how to vectorize a convolutional calculation. A convolution calculation takes N batches of an input tensor consisting of C' channels of $H \times W$ feature maps and a $C' \times C \times k \times k$ (C is the number of output channels) kernel tensor, then generates an output C channels of $H \times W$ feature maps. An output tensor along the channel can be calculated by multiplying an input feature map point and kernel tensor along the output channel elements, which can be vectorized. We can keep a long vector length if we continuously locate the output channel dimension and process it at the innermost loop.

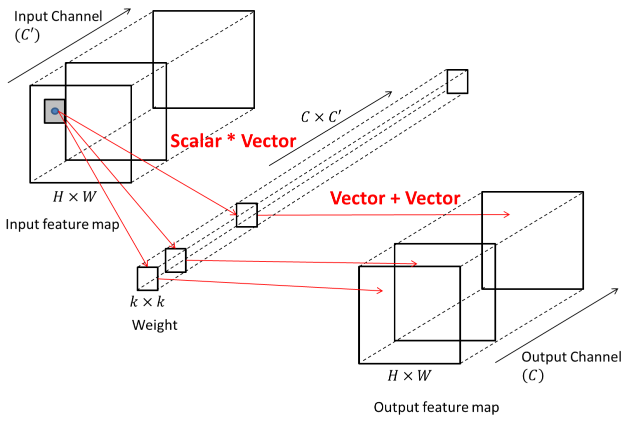


Fig. 5. Vectorization in convolutional calculation

However, the current TVM does not ensure it. Some DNN frameworks, such as PyTorch, use the NCHW layout. Its data structure is as Fig. 6, where the order of dimensions is the batch size (N), number of channels (C), heights (H), and width (W).

Instead of the NCHW layout, we set the tensors as the NCHWc layout to vectorize the innermost loop efficiently. Fig. 7 depicts the data structure of the NCHWc layout. This is a data layout for 5-dimensional tensors, where the order of dimensions is NCHWc and "c" stands for "channel-block". This layout divides the C (channel) dimension of the NCHW layout into channel-blocks.

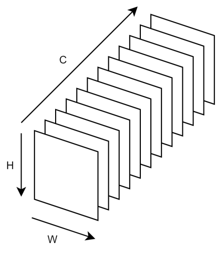


Fig. 6. Overview of NCHW Layout

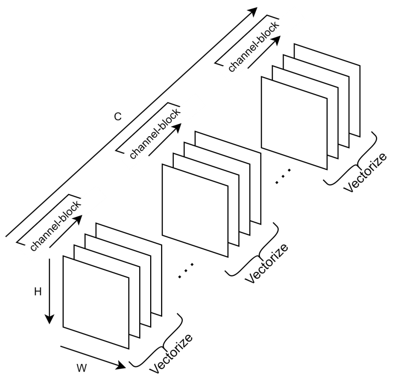


Fig. 7. Overview of NCHWc Layout

As discussed above, convolution operations in CNNs can be vectorized in the dimension of the output channel ("C" in Fig. 6) direction. In addition, laying out tensors, each of which has its innermost dimensions at channel-block, enables contiguous memory access [15]. However, the channel-block size of the program generated by the current TVM is not sufficiently long. Therefore, we expand the channel-block size to increase the vector calculation. We also transformed the layout of kernels accordingly.

5 Experimental Evaluation

5.1 Evaluation Environment

This section evaluates the proposed code generation method for vector multicore chips like the NEC SX-Aurora TSUBASA personal vector multicore supercomputer. The configuration of the NEC SX-Aurora TSUBASA used for evaluation is shown in Table 1.

This table shows that SX-Aurora TSUBASA used in the evaluation has eight vector processing elements (PEs) with an operating frequency of 1.4GHz and a

Table 1. Specifications of NEC SX-Aurora TSUBASA Vector Engine

Vector PE	8cores
Operating Frequency(VE)	1.4GHz
Memory Capacity	24GiB
L1d Cache	32KiB / PE
L1i Cache	32KiB / PE
L2 Cache	256KiB / PE
L3 Cache (LLC)	16MiB
Vector Registers	64 (per PE)
Vector Length	256 entries

maximum vector length of 256 entries. It also has a 16MiB last-level cache (LLC). We used NEC's compiler NCC version 3.5.1 as a backend compiler to generate executable binaries and both the OSCAR compiler and NCC for parallelization. After parallelization by the OSCAR compiler, vector binary codes were generated using NCC.

5.2 Evaluation Method

Table 2 shows the evaluation target. For the evaluation, we used a pre-trained ONNX format model for ImageNet1000 image classification including ResNet50 [16] and VGG19 [17]. VGG is a CNN network model that won the ILSVRC 2015 challenge with a top-5 error rate of 3.57%. The structure of VGG is straightforward. It has a feature extraction part consisting of multiple convolutional layers and a part that performs class classification from the extracted features. ResNet can have up to 152 layers, much deeper than previous architectures such as AlexNet and VGG. It uses "skip connections" or "shortcut connections" that allow the network to bypass some layers. This helps alleviate the vanishing gradients problem, which can occur when training DNNs.

Table 2. Evaluation target

TVM	Ver.0.8.0 [18]
model	ResNet50(ImageNet1000 image classification) [19]
	VGG19(ImageNet1000 image classification) [20]
input image	$224 \times 224 \times 3$(cat.png) [21]

TVM translated these models into sequential C source code using the proposed method. We also evaluated the performance of the original TVM for comparison. After TVM's code generation, the OSCAR compiler optimized and parallelized it. For comparison, we also parallelized the same program with NCC. Since the number of iterations of the loops for parallel processing in the evaluation programs is a multiple of 7, we generated parallelized code for up to 7PEs, which obtains the best load balancing up to 8PEs available in SX-Aurora TSUBASA. In this evaluation, NCC vectorized the programs. In addition to measuring the execution time per image, we also measured the average vector length and vector operation rate. We confirmed the equivalency of the parallelized programs' execution result to that of the sequential program.

5.3 Evaluation Result of The Original TVM

First, we evaluated the execution time of the code obtained from the original TVM, which was parallelized by the OSCAR compiler. Table 3 shows the evaluation result with the speedup compared to the sequential execution. The execution time of ResNet50 was about 2.1 seconds at 1PE, and the speedup on 7PEs was 2.3× compared to 1PE, and the execution time of VGG19 was about 20.2 seconds at 1PE.

Table 3. Execution time of the C code generated by the original TVM

# of PEs	ResNet50		VGG19	
	Execution Time [ms]	SpeedUp	Execution Time [ms]	SpeedUp
1	2077.9	—	20157.7	—
2	1489.3	1.4	10253.8	2.0
4	974.7	2.1	5390.1	3.7
7	898.0	2.3	4383.8	4.6

Table 4 shows the evaluation results when parallelizing by NCC instead of the OSCAR compiler. NCC's parallelization did not give us speedup even with multiple PEs.

Table 4. Execution time of the C code generated by the original TVM (NCC only)

# of PEs	ExecutionTime[ms]	
	ResNet50	VGG19
1	2181.3	5468.6
2	2181.3	5466.1
4	2181.2	5465.6
7	2181.2	5465.5

5.4 Evaluation Result of The Proposed Method

Next, we evaluated the proposed method. Fig. 8 shows the result of ResNet50.

This figure includes parallelizing with OSCAR (OSCAR + NCC) and NCC (NCC only). As in Fig. 8, the execution time of the original TVM by NCC only

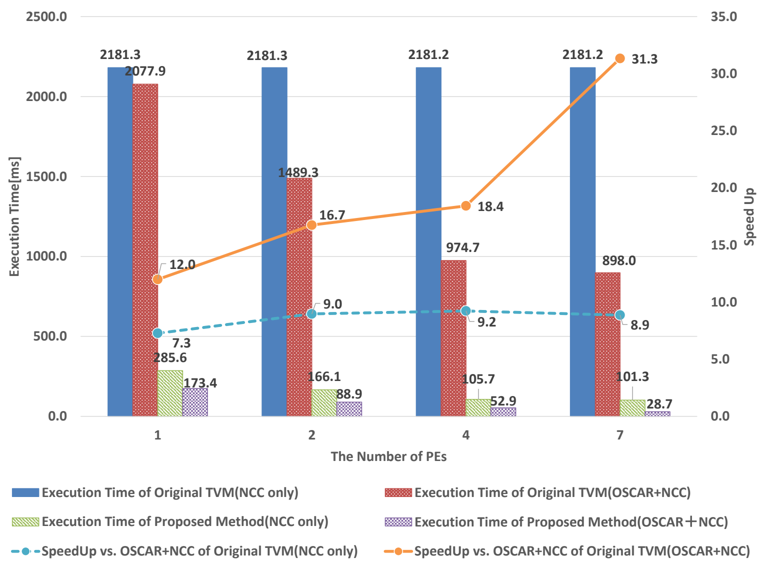


Fig. 8. Execution time after tensor layout transformation(ResNet50)

was 2181.3 ms for 1PE and 2181.2 ms for 8PEs. Also, the execution time of the original TVM by OSCAR+NCC was 2077.9 ms for 1PE and 898.0 ms for 7PEs. Next, the execution time of the proposed method was 285.6 ms with 1PE and 101.3 ms with 7PEs by using NCC only. It was reduced to 173.4 ms with 1PE and 28.7 ms with 7PEs when parallelizing with OSCAR. Therefore, the proposed method using OSCAR+NCC (28.7 ms) gave us 76.0× speedups for the same 7PEs compared with the original TVM using NCC only (2181.2 ms).

Fig. 9 shows the result of VGG19.

The execution time of the original TVM using only NCC was 5468.6 ms for 1PE and 5465.5 ms for 8PEs. The proposed method using OSCAR + NCC reduced the execution time for 1PE to 728.2 ms and 116.7 ms for 7PEs. Therefore, the proposed method gave us 46.8× more speedups than the original TVM with NCC on the same 7PEs.

Next, we examined the speedup on each core against the sequential execution to investigate how much parallelism had been achieved. The results are shown in Fig. 10.

When using the original TVM with OSCAR, the speedup on 7PEs was 2.3× when executing ResNet50 and 4.6× when executing VGG19, respectively. On the other hand, the proposed method achieved 6.1× for ResNet50 and 6.2× for VGG19. In the case of using OSCAR, the speedup of the proposed method was higher than that of the original TVM. Fig. 11 depicts the number of executed instructions (when processing on 7PEs).

The original TVM parallelized by NCC (Original TVM (NCC only) in Fig. 11) executed 82.8 billion instructions for ResNet50 and 199.5 billion instructions

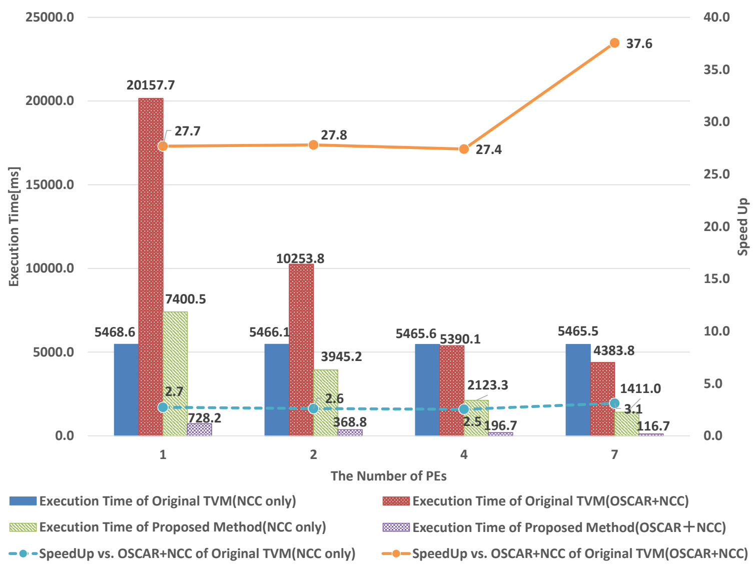


Fig. 9. Execution time after tensor layout transformation(VGG19)

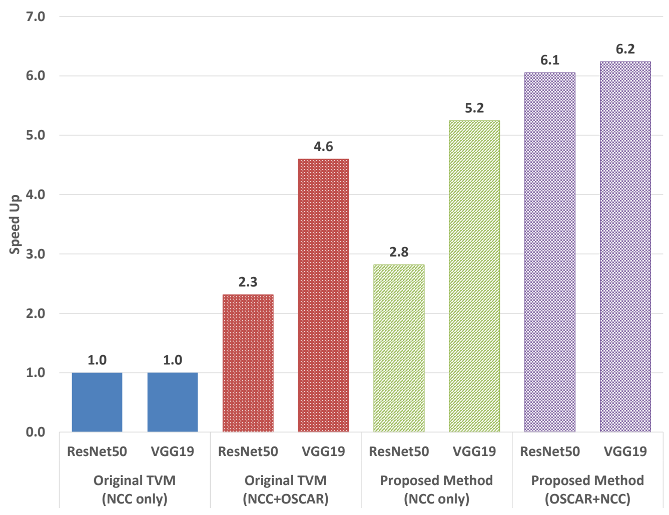


Fig. 10. Speedup ratio of 7PEs execution compared to 1PE

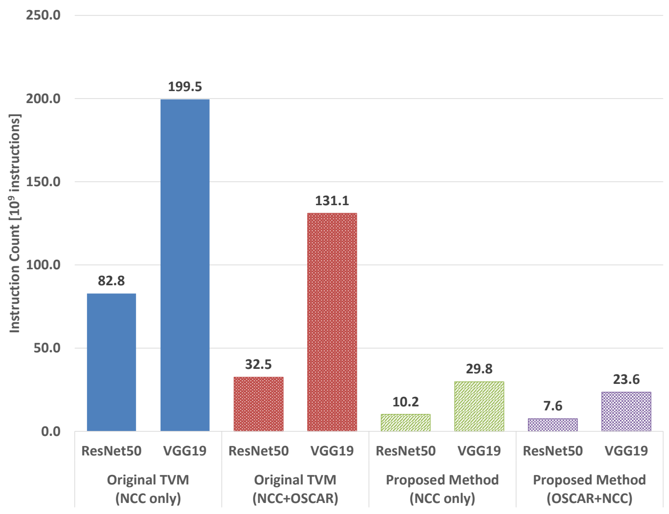


Fig. 11. Total number of executed instructions

for VGG19. In comparison, that parallelized by OSCAR (Original TVM (OSCAR + NCC) executed 32.5 billion instructions for ResNet50 and 131.1 billion instructions for VGG19, respectively. On the other hand, the proposed method parallelized by OSCAR (Proposed Method (OSCAR + NCC) in Fig. 11) executed only 7.6 billion instructions for ResNet50 and performed only 23.6 billion instructions for VGG19, indicating a significant reduction in the number of executed instructions. Reducing the number of executed instructions is one factor contributing to the reduction in execution time. In addition, when comparing "OSCAR + NCC" and "NCC only", the OSCAR compiler's parallelization can suppress the number of executed instructions even on 7PEs. One of its reasons is increasing the vector operation ratio and average vector length. Fig. 12 shows the percentage of vector instructions in the programs.

The percentage of vector instructions was between 73.3% and 86.5% when using the code generated by the original TVM. On the other hand, the proposed method increased them to more than 96.0%. Also, Fig. 13 depicts the average vector length and the execution time of vector operation.

The average vector length by the original TVM was about 15 for ResNet50 and about 26 for VGG19, respectively. The proposed method increased it to more than 147.9 for ResNet50 and 161.6 for VGG19, respectively, indicating that the vector length has increased by more than seven times. In addition, when parallelizing with OSCAR, the proposed method reduces the total execution time of vector load instructions. Fig. 14 shows the execution time of the vector load instructions.

The proposed method reduces the time for vector load instructions from 1736.6 ms to 55.8 ms for ResNet50 and from 19164.2 ms to 347.5 ms for VGG19. Even without OSCAR, the execution time of vector load instructions was de-

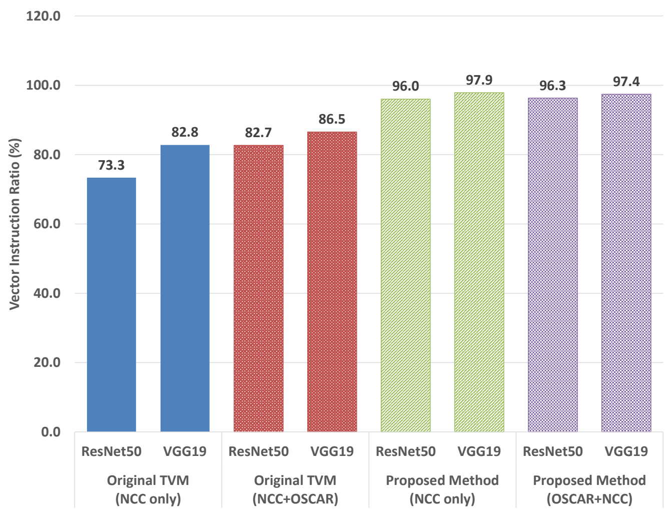


Fig. 12. Percentage of vector instructions

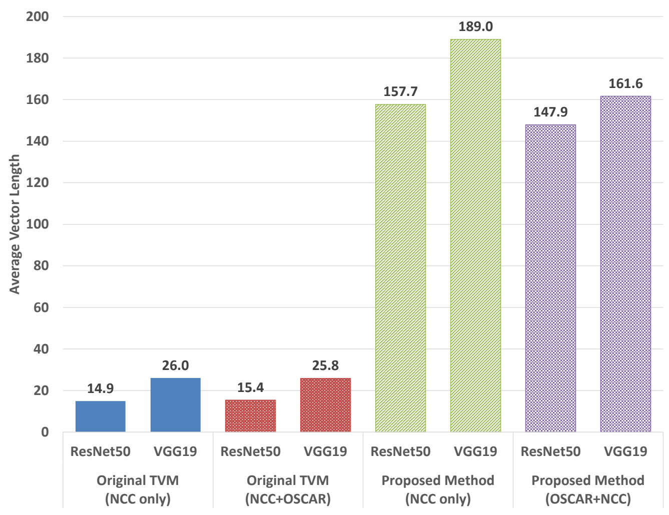


Fig. 13. Average vector length

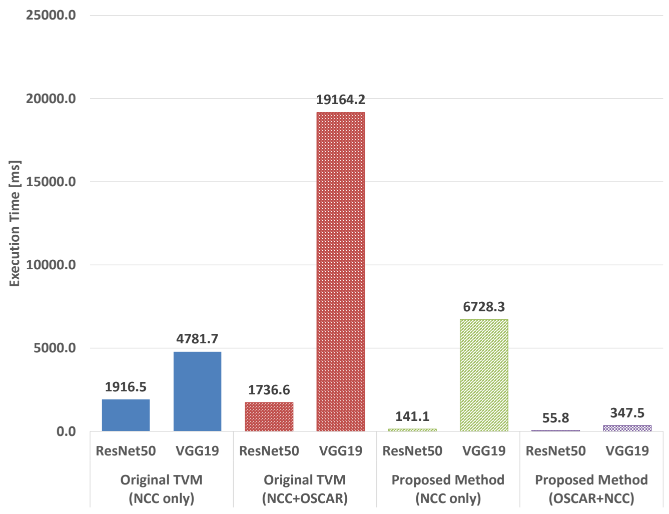


Fig. 14. Execution time of the vector load instructions

creased. From these results, it can be said that the efficiency of the vector pipeline has been improved by transforming the memory layout of the tensors, resulting in the execution time reduction.

6 Conclusion

This paper has proposed a vectorization-friendly code generation method for a deep learning compiler TVM. The proposed method transforms the memory layout of tensors to increase the length of the innermost loop for efficient vectorization. The output of TVM is parallelized by the OSCAR parallelizing compiler and vectorized by the vector accelerator compiler NCC.

The proposed method extended TVM and evaluated it on NEC SX-Aurora TSUBASA vector multicore supercomputer. Although the execution time of inference processing of ResNet50 without the proposed method was 2077.9 ms with 1PE and 898.0 ms with 7PEs, the proposed method reduced the execution time to 173.4 ms with 1PE and 28.7 ms with 7PEs. That is to say, it was 12.0× faster with 1PE and 31.3× faster with 7PEs for ResNet50. It also accelerated VGG19. The execution time was 27.7× faster from 20157.7 ms to 728.2 ms with 1PE and was 37.6× faster from 4383.8 ms to 116.7 ms with 7PEs. In addition, when using the proposed method, the speedup ratio between the sequential execution and 7PEs multi-core execution increased from 2.3 to 6.1 compared to the original TVM for ResNet50 and from 4.6 to 6.2 for VGG19, respectively. These results confirmed that deep learning models can be automatically parallelized by analyzing them with the OSCAR compiler after translating them into C source code using TVM. Furthermore, by transforming the tensor layout of the C source code generated by TVM so that the innermost loop length of convolution becomes larger, it was confirmed that the execution efficiency of vector processors improved, resulting in a faster execution time.

Acknowledgments

A part of this research is supported by the Japan Science and Technology Agency Moonshot Research and Development Program (JPMJMS2031).

References

1. Hiroshi Ito, Kenjiro Yamamoto, Hiroki Mori, and Tetsuya Ogata. Efficient multi-task learning with an embodied predictive model for door opening and entry with whole-body control. *Science Robotics*, 7(65), 2022.
2. Tomohiro Hirano, Hideo Yamamoto, Shuhei Iizuka, Kohei Muto, Takashi Goto, Tamami Wake, Hiroki Mikami, Moriyuki Takamura, Keiji Kimura, and Hironori Kasahara. Evaluation of automatic power reduction with oscar compiler on intel haswell and arm cortex-a9 multicores. In *Proceedings of the 27th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*. LCPC, September 2014.
3. H. Kasahara, H. Honda, A. Mogi, A. Ogura, K. Fujiwara, and S. Narita. A multi-grain parallelizing compilation scheme for oscar (optimally scheduled advanced multiprocessor). In *Proc. 4th Intl. Workshop on LCPC*, pages 283–297, August 1991.
4. Hironori Kasahara, Motoki Obata, and Kazuhisa Ishizaka. Automatic coarse grain task parallel processing on smp using openmp. In *Languages and Compilers for Parallel Computing*, pages 189–207, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
5. M. Obata, J. Shirako, H. Kaminaga, K. Ishizaka, and H. Kasahara. Hierarchical parallelism control for multigrain parallel processing. In *Languages and Compilers for Parallel Computing*, pages 31–44, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
6. Hironori Kasahara, Keiji Kimura, Toshiaki Kitamura, Hiroki Mikami, Kazutaka Morita, Kazuki Fujita, Kazuki Yamamoto, and Tohma Kawasumi. Oscar parallelizing and power reducing compiler and api for heterogeneous multicores: (invited paper). In *2021 IEEE/ACM Programming Environments for Heterogeneous Computing (PEHC)*, pages 10–19, 2021.
7. Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.
8. Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.
9. Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. *arXiv preprint arXiv:1802.04799*, 2018.

10. J. Bai, F. Lu, K. Zhang, and et al. Onnx: Open neural network exchange, github (online). available from (`https://github.com/onnx/onnx`) (accessed 2023-08-30).
11. Keiji Kimura, Masayoshi Mase, Hiroki Mikami, Takamichi Miyamoto, Jun Shirako, and Hironori Kasahara. Oscar api for real-time low-power multicores and its performance on multicores and smp servers. In Guang R. Gao, Lori L. Pollock, John Cavazos, and Xiaoming Li, editors, *Languages and Compilers for Parallel Computing*, pages 188–202, Berlin, Heidelberg, 2010.
12. H. Kasahara, H. Honda, A. Mogi, A. Ogura, K. Fujiwara, and S. Narita. A multi-grain parallelizing compilation scheme for oscar (optimzally scheduled advanced multiprocessor). In *Proc. 4th Intl. Workshop on LCPC*, pages 283–297, August 1991.
13. TVM. Design and architecture (online). available from (`https://tvm.apache.org/docs/arch/index.html`) (accessed 2023-08-30).
14. TVM. Introduction to relay ir (online). available from (`https://tvm.apache.org/docs/arch/relay_intro.html`) (accessed 2023-08-30).
15. Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidynathan, Srinivas Sridharan, Dhiraj Kalamkar, Bharat Kaul, and Pradeep Dubey. Distributed deep learning using synchronous stochastic gradient descent. *arXiv preprint arXiv:1602.06709*, 2016.
16. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03, 2016.
17. Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.
18. APACHE. Tvm: Open deep learning compiler stack for cpu, gpu and specialized accelerators, github (online). available from (`https://github.com/apache/tvm/tree/v0.8`) (accessed 2023-08-30).
19. ONNX. Onnx models: Resnet model, github (online). available from (`https://github.com/onnx/models/tree/main/vision/classification/resnet/model/resnet50-v2-7.onnx`) (accessed 2023-06-25).
20. ONNX. Onnx models: Vgg model, github (online). available from (`https://github.com/onnx/models/blob/main/vision/classification/vgg/model/vgg19-7.onnx`) (accessed 2023-08-30).
21. DMLC. Mxnet.js: Javascript package for deep learning in browser, github (online). available from (`https://github.com/dmlc/mxnet.js/blob/master/data`) (accessed 2023-08-30).