Received: 22 November 2021 | Revised: 16 October 2022 | Accepted: 25 October 2022

DOI: 10.4218/etrij.2021-0446

ORIGINAL ARTICLE

ETRI Journal WILEY

PartitionTuner: An operator scheduler for deep-learning compilers supporting multiple heterogeneous processing units

Misun Yu | **Yongin Kwon** | **Jemin Lee** | **Jeman Park** | **Junmo Park** | **Taeho Kim**

Artificial Intelligence Research Laboratory, Electronics and Telecommunications Research Institute, Daejeon, Republic of Korea

Correspondence
Misun Yu, Artificial Intelligence Research Laboratory, Electronics and Telecommunications Research Institute, Daejeon, Republic of Korea.
Email: msyu@etri.re.kr

Funding information
IITP/MSIT, Grant/Award Numbers: 2018-0-00769, 2022-0-00454

Abstract

Recently, embedded systems, such as mobile platforms, have multiple processing units that can operate in parallel, such as centralized processing units (CPUs) and neural processing units (NPUs). We can use deep-learning compilers to generate machine code optimized for these embedded systems from a deep neural network (DNN). However, the deep-learning compilers proposed so far generate codes that sequentially execute DNN operators on a single processing unit or parallel codes for graphic processing units (GPUs). In this study, we propose PartitionTuner, an operator scheduler for deep-learning compilers that supports multiple heterogeneous PUs including CPUs and NPUs. PartitionTuner can generate an operator-scheduling plan that uses all available PUs simultaneously to minimize overall DNN inference time. Operator scheduling is based on the analysis of DNN architecture and the performance profiles of individual and group operators measured on heterogeneous processing units. By the experiments for seven DNNs, PartitionTuner generates scheduling plans that perform 5.03% better than a static type-based operator-scheduling technique for SqueezeNet. In addition, PartitionTuner outperforms recent profiling-based operator-scheduling techniques for ResNet50, ResNet18, and SqueezeNet by 7.18%, 5.36%, and 2.73%, respectively.

KEYWORDS

deep neural network, deep-learning compiler, parallel processing, partitioning

1 | INTRODUCTION

Deep neural networks (DNNs) have become a key solution to a range of application such as image classification, object detection, and speech recognition. As deep learning (DL) expands into various fields, improving the performance of DNN inference on resource-constrained embedded systems has become a challenge. As one way to increase the performance of DNN inference, embedded systems with various processing units (PUs) such as centralized processing units (CPUs), graphic processing units (GPUs), and neural processing units (NPUs) [1–3] have

been proposed. To efficiently run a DNN on the various PUs, it is essential to convert the DNN operators into executable code optimized to these PUs.

Existing DL compilers [4–8] automatically transform the operators of a DNN into optimized machine code that can be run on a target system. For example, GLOW [8] and Tensorflow XLA [6] can generate optimized code for CPUs and GPUs, respectively. TVM [4], a representative open-source DL compiler, additionally supports NPU as a backend, which can perform computationally intensive operations with low power and low latency.

These DL compilers map operators in a DNN to CPUs, GPUs, or NPUs in a target system according to the choice of developers or the type of operators. However, they cannot intelligently map DNN operators to heterogeneous PUs to minimize the total inference time. In addition, the resource utilization of the codes generated from the DL compilers is not high. The current DL compilers such as TVM, GLOW [8], and XLA adopt a sequential approach for scheduling the execution of DNN operators. Specifically, it is impossible to generate parallel code that simultaneously performs independent operators in a single DNN on different PUs.

Recently, operator-scheduling techniques [9] have been proposed to support automatic mapping to heterogeneous PUs in a system to reduce the total inference time, but they target server systems with GPUs. If the target system contains NPUs, there are more considerations for operator-scheduling than if the hardware system contains only GPUs. Because the NPU is specialized for certain operations, such as general matrix multiplication (GEMM), the NPU backend of the DL compiler may only support those operations. Also, even if the operation is supported by the NPU backend, it may be better not to use the NPU because the overall execution time can take more than the CPU due to the data-transfer overhead. Therefore, a compiler-integrated NPU-supported operator-scheduling technique is needed to increase the performance of DNN inference on NPU-embedding hardware systems.

In this study, we introduce PartitionTuner (PT), an operator-scheduling framework, which is integrated to NEST-C [10], an open-source DL compiler that supports NPU and CPU backends. PT determines which operators in the DNN are mapped to which PUs in order to minimize the total inference time of the final code generated by the DL compiler. Based on this mapping result, it creates an operator-scheduling plan and final inference code that can simultaneously execute several groups of operators. To the best of our knowledge, PT is the first profile-based operator-scheduling framework integrated with an open-source DL compiler that supports multiple heterogeneous PUs including NPUs. Our contributions are summarized as follows:

- We propose PT, which is a profile-based DNN-operator-scheduling framework supporting operator-level parallelism. PT is platform agnostic and can serve as a general technique for DL compilation frameworks such as TVM or XLA.
- We implemented PT and two other operators scheduling techniques to demonstrate the feasibility and efficiency of PT in an open-source DL compilation framework named NEST-C that supports CPU and NPU backends.
- We applied PT to a CPU-NPU hardware architecture and performed performance experiments with seven DNNs. PartitionTuer is 5.03% better than a static type-based operator-scheduling technique for SqueezeNet and 7.18%, 5.36%, and 2.73% better than a profiling-based operator-scheduling technique for ResNet50, ResNet18, and SqueezeNet, respectively.

2 | MOTIVATION

In this section, we first discuss the weakness of previous operator-scheduling techniques to apply to CPU-NPU architecture, and then briefly explain our approach using a simple DNN example.

TVM is a state-of-the-art DL compiler that supports a NPU backend that can generate optimized code for NPU. When generating code for a CPU-NPU architecture from an input DNN, TVM maps NPU backends to Convolution operators and CPU backends to other types of operators because the NPUs (VTAs) supported by TVM can only efficiently execute GEMM operations. In particular, TVM maps an operator to a backend according to the type of the operator. In addition, TVM generates code that executes DNN operators sequentially on PUs. That is, TVM's operator-scheduling technique enforces type-based ping and sequential execution policy (TypeSeq). Figure 1A shows the final inference time of a DNN when using TypeSeq on a CPU-NPU architecture. In the figure, $v_1, ..., v_n$ represent the operators of an input DNN. Table 1 lists the execution time of individual and grouped operators (Convolution + ReLU) on the CPU and NPU. According to TypeSeq, all Convolution operators are mapped to the NPU and the total inference time becomes 235 ms.

TypeSeq is simple and effective because Convolution operators can be executed on NPUs faster than on CPUs. However, according to the size of input data of operators, using NPU may require more time than using CPU due to data-transfer overheads. Therefore, recent DL

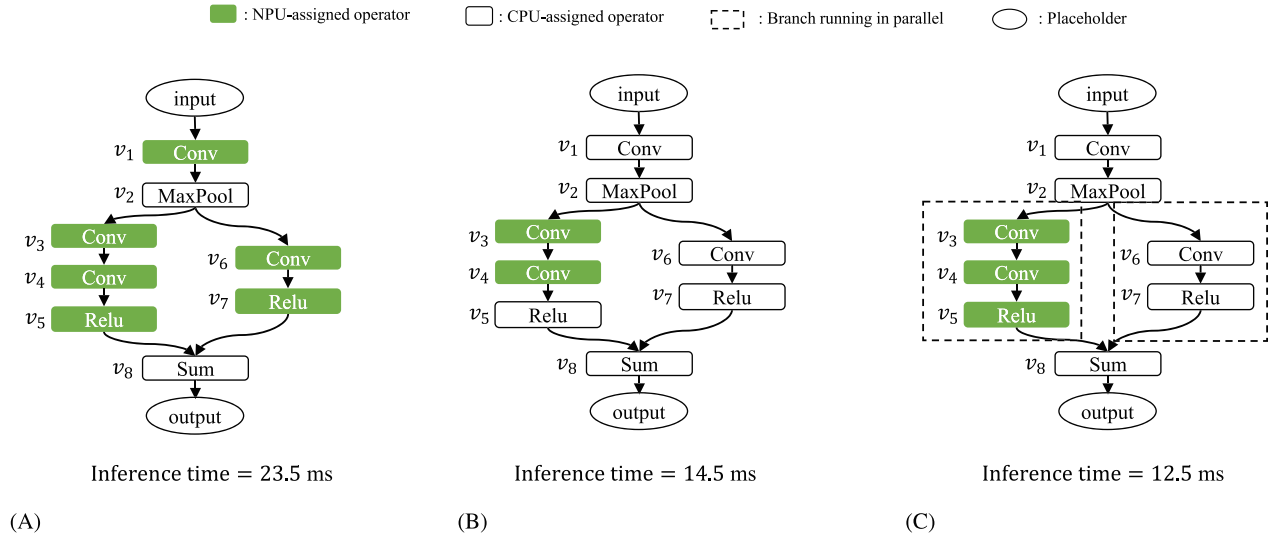

FIGURE 1 DNN inference time for different backend-mapping policies: (A) TypeSeq: static type-based ping + sequential execution, (B) OpSeq: ping based on the execution time of individual operator + sequential execution, and (C) PartitionTuner: ping based on the execution time of individual/grouped operators + parallel execution

TABLE 1 Example of operator execution time on CPU and NPU

	Execution time (ms)			Execution time (ms)	
Operator	**CPU**	**NPU**	**Operator**	**CPU**	**NPU**
v_1	1	10	v_6	1.0	1.5
v_2	2	-	v_7	1.0	-
v_3	10	2	v_8	5.0	-
v_4	10	2	$v_4 + v_5$	10.5	2.5
v_5	1	-	$v_6 + v_7$	1.5	2.0

Note: Execution time: computing time + data-transfer time; -: not supported.

compilers [9,11,12] adopt profiling-based operator-scheduling policy in order to minimize the DNN inference time. In these policies, the final backends of DNN operators are determined by the profiled execution time of each DNN operator using the final code from the DL compilers on PUs. Figure 1B shows the final inference time of the DNN based on the execution time of each operator instead of the TVM's type-based ping (OpSeq). By simply replacing the type-based backend-assignment policy to the profiling-based one, we can reduce the inference time up to 9 ms.

However, recent operator-scheduling techniques only consider CPUs and GPUs that are generally designed to run all types of operations. Different from CPUs and GPUs, NPUs are designed to execute certain types of operations. In addition, NPU backends of DL compilers can perform various kinds of operator fusions to reduce computing time and remove communication overheads. For example, TVM combines Convolution, BatchNormalization, and ReLU operators into a single one not to use memory for saving intermediate results.

Therefore, we propose an operator-scheduling technique named PT that map operators to the backends by considering the execution time of individual operators and grouped (adjacently located) operators. In addition, PT schedules parallel branches to run in parallel in case of parallel branches with no data dependencies (independent) and other backends mapped.

Figure 1C shows the backend-mapping result of PT and the total inference time of the input DNN, which consider the execution time of an individual operator and grouped operators. The NPU is mapped to a ReLU if it can execute Convolution and ReLU groups faster than the CPU, although the CPU can execute ReLU operators faster than the NPU. That is, while OpSeq maps the CPU backend to v_5, PT maps the NPU backend to reduce the total inference time of the DNN. Figure 1B,C shows that

by executing v_4 and v_5 on NPU, we can reduce the inference time of the DNN by 0.5 ms.

Furthermore, when independent branches are executed in parallel, the total execution time of the DNN is theoretically reduced by 1.5 ms. In this example, the total inference time becomes 12.5 ms. Even when the v_3, v_4, v_5 and v_6, v_7 are executed sequentially, total inference time is 14 ms, which is faster than OpSeq. Therefore, the PT generates an operator-scheduling plan that takes less total inference time than other operator-scheduling policies.

In the next section, we describe the detailed structure and operator-scheduling algorithm of PT.

3 | DESIGN AND IMPLEMENTATION OF PT

Figure 2 shows the structure of NEST-C, our open-source DL compiler, integrated with PT. NEST-C, like recent DL compilers [4, 6, 8], uses its frontend to perform hardware-independent optimizations based on high-level intermediate representation in the forms of a computation graph and then uses its backend to perform hardware-dependent optimizations and code generation.

In NEST-C, machine code for multiple heterogeneous PUs is generated by backends using partitions that are mapped to the backends by PT. Each partition is a subgraph of an input graph, and the execution order of partitions is determined by PT. Figure 2 shows that PT takes a computation graph in form of a directed acyclic graph and generates backend-mapped partitions, as well as partition-scheduling plan and code. For each partition, different hardware-dependent optimizations are used as per the mapped backends.

PT's main functions are to profile the performance of operators, map a backend to each operator, partition the graph according to the mapped backends, and find partitions that can run in parallel. These features are provided by the five components of PT: profiler, branch extractor, backend finer, graph partitioner, and partition scheduler.

The results of NEST-C includes PU-dependent and scheduling codes. These codes are built together and deployed to the target system. The target system herein is an embedded system equipped with one CPU and NPU.

Figure 3 shows the execution steps of PT using these five components and the brief description of each component.

1. *In the first step*, the profiler maps backends to the individual and group operators of the input graph, generates machine code from the indivisual and grouped operators using the NEST-C backend, and measures the execution time of the generated code. The measured execution time is recorded in the performance profile.
2. *In the second step*, the branch extractor extracts the information of sequential and parallel branches from the input graph.
3. *In the third step*, the backend finder maps backends and operators using the execution time of individual/grouped operators in the performance profile to minimize execution time.
4. *In the fourth step*, a partition is a group of consecutive operators that are mapped the same backend. Each

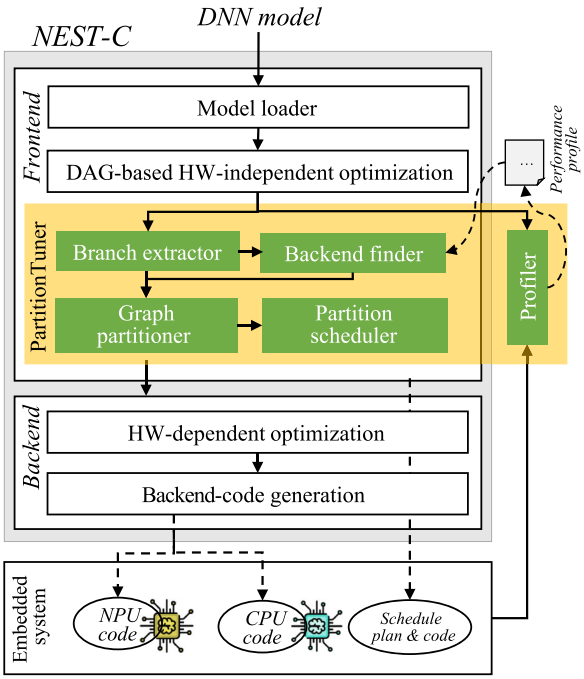

FIGURE 2 Overall structure of PartitionTuner

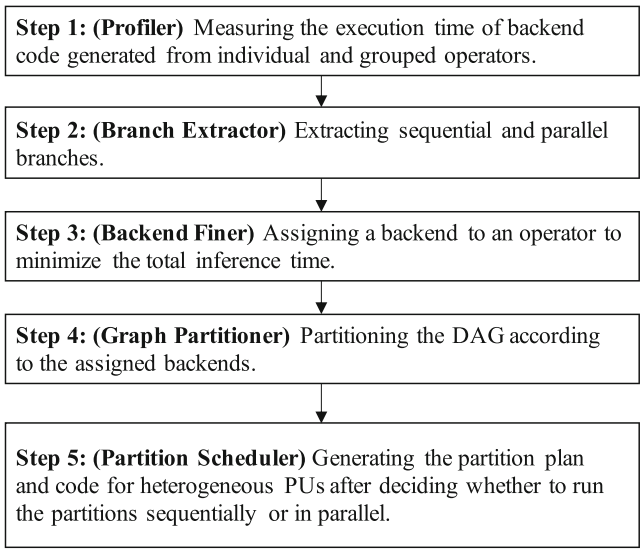

FIGURE 3 Execution steps of PartitionTuner

partition is sent to the HW-dependent optimization and backend-code generation modules to generate the machine code that can be run on a specific PU.

5. *In the last step*, the partition scheduler generates a schedule plan that contains information about the order in which the partitions created in the previous step will be executed. At this stage, the partition scheduler creates a schedule plan so that if parallel branches are mapped to different backends, each branch will run in parallel on a different PU. In this case, each branch is a partition. The partition scheduler also generates C/C++ source code from the schedule plan.

The following sections describe each component of PT in detail.

3.1 | Profiler

The profiler of PT measures and records the execution time of individual and grouped operators of an input graph. Specifically, using the NEST-C backends, the profiler can generate machine code for a partition that contains a single or grouped operator. Users can specify the operator types to be grouped. The execution time of generated codes is then measured. Each partition has its ID, which is created by successively attaching the attributes of operators in a partition. The attributes of an operator in PT represent the characteristics of parameters to execute the operator such as the type name, input size, and filter size. Therefore, there may exist partitions with the same ID, and in this case, only one partition's execution time is recorded.

3.2 | Branch extractor

In PT, the backend mapping and the graph partitioning are done on the branches of a graph. We make the definition of a branch and parallel branch as Definitions 1 and 2.

Definition 1 (Branch) For any directed acyclic graph DAG $G(V,E)$, a branch consists only of sequential nodes and forms a separate path with other branches in the DAG.

Definition 2 (Parallel branch) Two branches are parallel if they have the same input and output node.

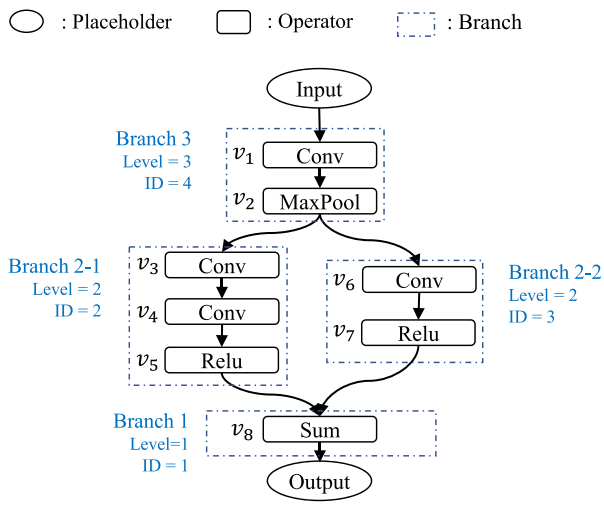

FIGURE 4 Example of branches

Figure 4 shows an illustrative example of a graph that has branches. In Figure 4, Branch 3 includes two operators v_1–v_2 and sends its output to Branches 2-1 and 2-2. Branches 2-1 and 2-2 share the input and output node; thus, they are parallel. Branch 1 includes a single node v_8. In PT, a branch has two attributes to represent the identity and parallelism: ID and *level*. Every branch has a different ID, and only parallel branches have the same *level*.

3.3 | Backend finder

The backend finder of PT maps a backend to an operator based on the performance profile generated from the profiler. The performance profile includes the execution time of machine codes that execute individual or grouped operators on PUs. In NEST-C, available backends are the CPU and NPU backends, which can generate machine codes for ARM/X86 CPU and EVTA NPU.[1] Therefore, the profiling and backend mapping of PT is performed for these two PUs. The backend mapping of PT is performed on a dynamic programming basis to minimize the total execution time. In addition, users can specify the types of consecutive operators that can be grouped for the profiler and backend finder. We specified Convolution and ReLU operators, as a group.

3.4 | Graph partitioner

After finishing the mapping of backends to the operators in an input graph, the graph partitioner of PT partitions

the graph. Partitioning is splitting a graph as subgraphs or partitions. Operators in a partition are mapped to the same backend. The left side of Figure 5 shows four partitions of the graph in Figure 4. The graph partitioner takes a branch list as an input from the branch extractor, traverses the operators of each branch sequentially, groups consecutive operators, and maps the same backend to the same partition. When an operator in which a new backend is mapped appears, a new partition is created.

3.5 | Partition scheduler

After partitioning a graph, the partition scheduler of the PT decides the execution order of the graph partitions (subgraphs). Partitions in the sequential branches are executed sequentially. Partitions in the parallel branches can be executed in parallel when different backends are mapped to them.

Figure 5 shows an example of partitions that are executed sequentially or in parallel. In this figure, $p2$ and $p3$ are parallel branches and mapped different backends. Therefore, they can run in parallel on target platforms containing two different PUs (CPU and NPU).

Based on the decided execution order, the partition scheduler generates a scheduling plan and program code. The scheduling plan and program code are logically identical. The difference is that the scheduling plan is independent of specific target hardware or programming language. An example of a scheduling plan and program code is shown on the right side of Figure 5. In the scheduling plan, we use the "parallelPartitions" keyword to indicate that $p2$ and $p3$ are parallel partitions that can be run in parallel on different PUs. In the program code, $p2$ and $p3$ are executed in parallel using threads.

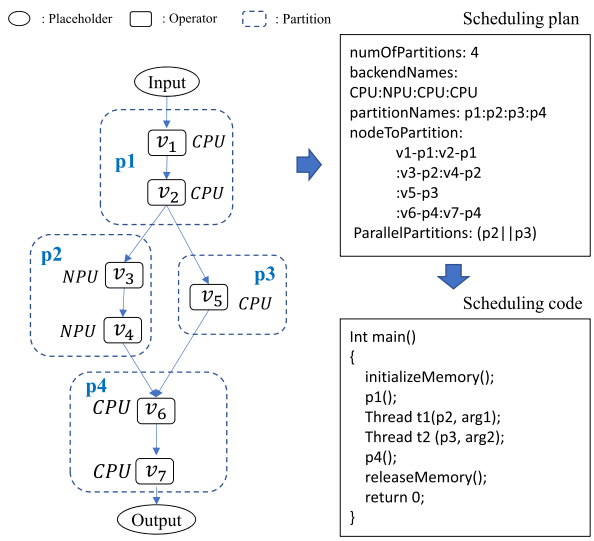

FIGURE 5 Example of a partitions and schedule plan

4 | EXPERIMENTS AND EVALUATION

4.1 | Implementation setup

We implemented PT in our NEST-C open-source DL compilation framework that is based on the GLOW [8]. We reused GLOW's CPU backend to generate the machine code for the quad-core ARM Cortex-A53 CPU of the Xilinx ZCU102 platform. The frequency of the CPU is set at 1200 MHz. To generate machine code for NPU, we implemented the EVTA backend in NEST-C. EVTA is a type of NPU and an extension of the VTA [13]. EVTA supports only the 8-bit integer precision and conducts a 16×16 GEMM operation at once. Therefore, we quantize the weight and input data of operators to compute the operators on EVTA. The frequency of EVTA is set at 333 MHz. We implemented the EVTA on the FPGA of the Xilinx ZCU102 platform.

The EVTA backend of NEST-C can generate instructions for Convolution operators of the ONNX operator set [14] and can fuse Convolution and ReLU to execute efficiently by not using slow external memory (RAM).

4.2 | Experimental setup

4.2.1 | Machine environment

We evaluate the performance of PT and two other operator-scheduling techniques mentioned in Section 2 in terms of inference time. All operator-scheduling techniques are integrated into NEST-C, and the performance is the inference time of the final code generated from NEST-C. All operator-scheduling techniques share NEST-C's frontends and backends. We also provide the performance using only the CPU backend (Baseline) to demonstrate the performance gains that can be achieved by using NPUs. All operator-scheduling techniques subject to performance comparison are as follows:

- Single partition on CPU (Baseline)
  All operators are mapped to the CPU backend and executed sequentially.
- Static type-based backend mapping and sequential execution (TypeSeq)
  Convolution and subsequent ReLU operators are mapped to the EVTA backend. Other operators are mapped to the CPU backend. All operators are executed sequentially.
- Backend mapping based on the execution time of individual operators and sequential execution (OpSeq)

Operators are mapped to backends according to the execution time of an individual operator on the CPU and EVTA. All operators are executed sequentially.

- PT
  Operators are mapped to backends according to the execution time of individual and grouped operators on the CPU and EVTA and can be executed in parallel if different backends are mapped to them.

Note that the EVTA backend that we implemented can handle only 16×16 GEMM operations; therefore, TypeSeq, OpSeq, and PT map the EVTA backend only to the Convolution operators with the number of input channels and the number of filters that are multiples of 16. Furthermore, the EVTA backend of NEST-C can fuse a Convolution operator and a consecutive ReLU operator into a single Convolution operator similar to other DL compilers [4, 15]. Therefore, we can use EVTA for a ReLU operator if a ReLU operator is placed immediately after a Convolution operator in a partition.

4.2.2 | Benchmarks and datasets

We focus our evaluation on inference time of the final code generated from NEST-C. We evaluate the execution time of compiled models that are trained on the ImageNet [16] data set. Our evaluation is performed using a set of seven well-known pretrained CNN models: ZFNet [17], AlexNet [18], GoogleNet [19], ResNet18/50 [20], ResNeXt [21], and SqueezeNet [22]. We downloaded these models from the ONNX model Zoo [23]. Among these CNN models, all models except ZFNet and AlexNet have parallel branches. The number of parallel branches in the models is listed in the second column of Table 3. Table 2 shows the inference time of benchmark models. The first and second columns of Table 2 list the name and size of benchmark models, and other columns list the inference time of Baseline, TypeSeq, OpSeq, and PT.

We compiled the seven benchmark models using our NEST-C compilation framework, which generates machine codes based on the backends that are mapped to partitions. We measured the averaged inference time of the compiled codes over 1,000 runs with different input images. There were little variations in all cases.

4.3 | Performance evaluation

Table 2 shows the end-to-end inference time of partitioned models by the Baseline and three operator-scheduling techniques (TypeSet, OpSeq, and PT). All three operator-scheduling techniques provide better performance than Baseline by using EVTA to compute the Convolution and ReLU operators. The performance improvement rates of the three operator-scheduling techniques compared with the Baseline are shown in Figure 6. We calculated the performance improvement rate (PI_{rate}) of a specific scheduling technique ($T1$) compared with another scheduling technique ($T2$), as shown in the formula as follows.

$$PI_{\text{rate}} = (T2 - T1)/T2 \times 100. \quad (1)$$

Figure 6 shows that for all benchmark models. Except AlexNet, the performance improvement is over 30%. Particularly, the performance improvement rate of three operator-scheduling techniques reaches approximately 80% for ResNet50 and ResNet18. In case of AlexNet, only a single Convolution operator was mapped to the NPU by the three operator-scheduling techniques as shown in the fourth to sixth columns of Table 3. Therefore, the performance improvement of AlexNet was relatively small

TABLE 2 Benchmark information and inference time

Benchmarks			Time (μs)		
Name	Size (Mbytes)	Baseline (CPU)	TypeSeq CPU + NPU	OpSeq	PT
ZFNet	349	2435.17	1694.38	1699.78	Same as TypeSeq
AlexNet	244	860.10	810.73	810.92	Same as TypeSeq
GoogleNet	171	1465.80	482.30	507.09	Same as TypeSeq
ResNet50	103	3183.74	636.19	684.15	635.04
ResNeXt50	100	6638.85	3959.94	3952.10	3928.46
ResNet18	47	1269.18	249.03	262.57	248.49
SqueezeNet	5	298.64	195.46	190.83	185.62

compared with other models, whereas more Convolution operators were mapped to the NPU for other models, and the performance improvement rate of the models was higher than that of AlexNet.

When only the three operator-scheduling techniques were compared, the profiling-based techniques (OpSeq and PT) performed better than TypeSeq for ResNeXt50 and SqueezeNet. However, OpSeq showed lower performance than TypeSeq for all other benchmark models. This is because OpSeq maps the backend to operators based on the execution time of the individual operators running on the PU. This means that the Convolution and ReLU operators run fast on the NPU and CPU individually but can run faster when run sequentially on the NPU. However, OpSeq only considers the performance of individual operators, so it maps those operators to different backends.

OpSeq's backend-mapping method, which does not take into account the characteristics of NPU, divided four benchmark models (ZFNet, GoogleNet, ResNet50, and ResNet18) into more partitions, as shown in the Table 4, causing more execution time. Therefore, OpSeq allows NEST-C to generate less performing code than TypeSeq and PT. For AlexNet, the execution time of final code by OpSeq is slower than those by TypeSeq and PT although the number of partitions is the same. This is because one ReLU operator that appeared after a Convolution operator is mapped to the CPU instead of being mapped to the NPU as can be seen in the eighth column of Table 3.

PT showed higher than or equal to performance to other two techniques. For ZFNet, AlexNet, and GoogleNet, PT generated the same backend-mapping and execution-scheduling results to TypeSeq. As we can see, the second column of Table 2, the size of three benchmark models is relatively larger than other models. However, for other models such as ResNet18/50, ResNeXt50, ResNet18, and SqueezeNet, PT performed better than TypeSeq. From these results, it can be seen that PT is more effective than TypeSeq for small-sized DNN models in terms of inference time. Additionally, PT can generate the parallel execution schedule for parallel branches; thus, it could find two and eight branches that can be executed on the CPU and EVTA in parallel from ResNet18 and SqueezeNet as seen in the 10th column of Table 3.

As a result, PT generated faster code than TypeSeq by 5.03% for SqueezeNet and than OpSeq by 7.18%, 5.36%, and 2.73% for ResNet50, ResNet18, and SqueezeNet,

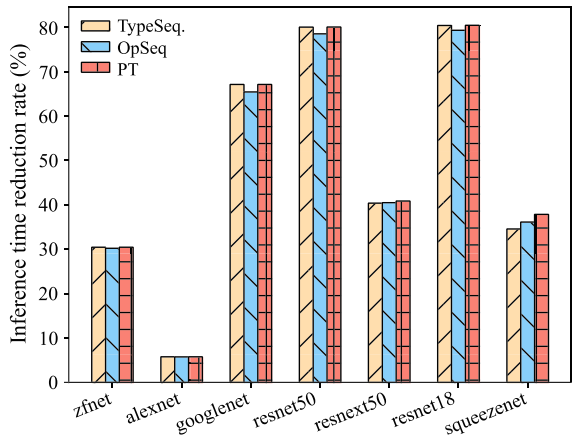

FIGURE 6 Reduction rate of inference time compared with the Baseline

TABLE 4 Number of partitions

Benchmarks	# Partitions			
	Baseline	TypeSeq	OpSeq	PT
ZFNet	1	5	10	5
AlexNet	1	3	3	3
GoogleNet	1	34	69	34
ResNet50	1	34	98	34
ResNexXt50	1	65	65	65
ResNet18	1	17	33	18
SqueezeNet	1	25	29	23

TABLE 3 Allocated backends for Convolution and ReLU operators

Benchmarks			# Convolution on VTA			# ReLU on VTA			# Parallelly executable partitions found by PT
Name	# Parallel branches	# Conv.	TypeSeq	OpSeq	PT	TypeSeq	OpSeq	PT	
ZFNet	0	5	4	4	4	6	0	6	0
AlexNet	0	5	1	1	1	1	0	1	0
GoogleNet	36	57	52	49	52	18	0	18	0
ResNet50	8	53	52	52	52	48	0	32	0
ResNeXt50	8	53	36	36	36	16	0	9	0
Resnet18	6	20	19	18	18	16	0	8	2
SqueezeNet	16	26	24	18	18	8	0	1	8

respectively. In particular, PT shows the highest performance for ResNet50, ResNeXt50, and ResNet18 despite the number of partitions are greater or equal to TypeSeq as seen in Table 4. This result represents that PT can map DNN operators to optimal backends that can minimize the total inference time.

Figure 7 shows the inference accuracy of Baseline, TypeSeq, OpSeq, and PT for four benchmark models. The accuracy of final NEST-C codes using all operator-scheduling techniques is lower than that using Baseline because it only uses the CPU backend performing floating-point operations but others use the EVTA backend performing integer operations for Convolution operators.

The accuracy of TypeSeq, OpSeq, and PT depends on the number of Convolution operators running on EVTA. That is, the greater the number of operators performed in EVTA, the lower the accuracy. Among TypeSeq, OpSeq, and PT, TypeSeq generated the code having the lowest accuracy for SqueezeNet, but there was little difference between the three operator-scheduling techniques for other benchmark models.

5 | RELATED WORK

5.1 | DL compilers

Because of the difficulty of deploying and efficiently executing DNNs on various hardware platforms, several DL compilers have been proposed such as Tensor Comprehensions [24], Tensorflow XLA [6], nGraph [5], GLOW [8], and TVM [4]. They take DNNs that are created from DL frameworks such as Tensorflow, Pytorch, and Caffe and generate machine codes that can run on the various hardware platforms embedding heterogeneous PUs.

XLA can generate codes for GPU and Google TPU [25]. However, the DL applications for an embedded system should be energy efficient and may have restrictions on the use of the GPU. The hybrid transformer of nGraph [5] maps complex operators (subgraphs) to Intel Nervana NNP to speed up the computation and the remaining operators to the CPU. Like nGraph, TVM can map the NPU backend to Convolution operators and the CPU backend to other types of operators. Glow [8] also can partition its computation graphs by operator types or graph size but can only generate CPU codes.

Genesis [26] is a DL compiler that integrates graph partitioning functionalities into TVM. Genesis has a similar structure to NEST-C. However, PT of NEST-C automatically generates an optimal partition of a computation graph that simultaneously uses the heterogeneous PUs of a target platform.

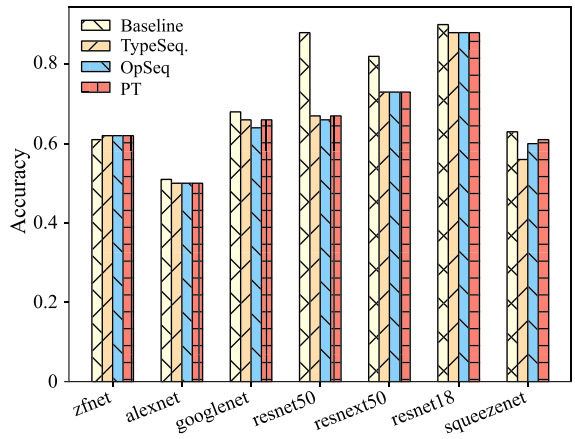

FIGURE 7 Inference accuracy

5.2 | DL graph partitioning and scheduling

Recently proposed techniques automatically partition a computation graph and find operator-level parallelism based on performance profiling to minimize inference time of a DNN. RAMMER [12] partitions a computation graph and maps them to virtualized parallel devices (vDevices) that are mapped to a physical accelerator during runtime. The objective of RAMMER is to maximize the utilization of a single hardware accelerator such as GPU and IPU. IOS [11] automatically schedules multiple operators on a GPU to reduce CNN inference time by improving the GPU utilization. IOS finds the parallelizable operators using a dynamic programming algorithm. DUET [9] automatically maps the branches of a DNN to CPUs and GPUs in a target system to improve concurrency and reduce the total inference time.

The objective of PT is to reduce the execution time of final code produced by the DL compilers supporting NPUs. To do that, it maps DNN operators to the backends of DL compilers and finds parallel branches that can be executed in parallel on heterogeneous PUs.

6 | CONCLUSION

With the advent of embedded devices including various DL accelerators, a hardware basis has been established to increase the performance of DNNs using the specialized functions of these accelerators and parallel processing technology using multiple accelerators. We proposed an operator-scheduling technique for efficiently using these multiple accelerators. The proposed technique integrated with a DL compiler can generate high-performance codes through profiling-based

performance analysis of PUs and parallelism analysis of DNNs. The experimental results show that the proposed technique generates faster code in terms of inference time than TypeSeq by 5.03% for SqueezeNet and than OpSeq by 7.18%, 5.36%, and 2.73% for ResNet50, ResNet18, and SqueezeNet, respectively. In the future, we will test the performance of PT in an embedded system with multiple DL accelerators. Also, we have a plan to integrate our PT into the other open-source DL compilers such as TVM and XLA to broadly validate its efficiency and usefulness.

ACKNOWLEDGMENT

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2018-0-00769: Neuromorphic Computing Software Platform for Artificial Intelligence Systems and No.2022-0-00454: Technology development of smart edge device SW development platform).

CONFLICT OF INTEREST

The authors declare that there are no conflicts of interest.

ORCID

Misun Yu https://orcid.org/0000-0001-7319-1053

REFERENCES

1. HISILICON, Kirin, 2022. https://www.hisilicon.com/en/products/Kirin
2. NVIDIA, Jetson, 2022. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/
3. Samsung, Exynos, 2022. https://semiconductor.samsung.com/processor/mobile-processor/
4. T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, and C. Guestrin, *TVM: An automated end-to-end optimizing compiler for deep learning*, (13th USENIX Symposium on Operating Systems Design and Implementation, Carlsbad, CA, USA), 2018, pp. 578–594.
5. S. Cyphers, A. K. Bansal, A. Bhiwandiwalla, J. Bobba, M. Brookhart, A. Chakraborty, W. Constable, C. Convey, L. Cook, O. Kanawi, R. Kimball et al., *Intel nGraph: An intermediate representation, compiler, and executor for deep learning*, arXive preprint, 2018. https://doi.org/10.48550/arXiv.1801.08058
6. C. Leary and T. Wang, *XLA: TensorFlow, compiled*, 2017. TensorFlow Dev Summit.
7. W.-F. Lin, D.-Y. Tsai, L. Tang, C.-T. Hsieh, C.-Y. Chou, P.-H. Chang, and L. Hsu, *ONNC: A compilation framework connecting ONNX to proprietary deep learning accelerators*, (IEEE International Conference on Artificial Intelligence Circuits and Systems, Hsinchu, Taiwan), 2019, pp. 214–218.
8. N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein, and J. Montgomery, *Glow: Graph lowering compiler techniques for neural networks*, arXive preprint, 2018. https://doi.org/10.48550/arXiv.1805.00907
9. M. Zhang, Z. Hu, and M. Li, *DUET: A compiler-runtime subgraph scheduling approach for tensor programs on a coupled CPU-GPU architecture*, (IEEE International Parallel and Distributed Processing Symposium, IEEE Portland, OR, 2021, pp. 151–161.
10. ETRI, NEST-C, 2021. https://github.com/etri/nest-compiler
11. Y. Ding, L. Zhu, Z. Jia, G. Pekhimenko, and S. Han, *IOS: Inter-operator scheduler for CNN acceleration*, Proc. Machine Learn. Syst. **3** (2021), 167–180.
12. L. Ma, Z. Xie, Z. Yang, J. Xue, Y. Miao, W. Cui, W. Hu, F. Yang, L. Zhang, and L. Zhou, *RAMMER: Enabling holistic deep learning compiler optimizations with rTasks*, (14th USENIX Symposium on Operating Systems Design and Implementation), 2020, pp. 881–897.
13. T. Moreau, T. Chen, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy, *VTA: an open hardware-software stack for deep learning*, arXive preprint, 2018. arXiv preprint arXiv: 1807.04188. https://doi.org/10.48550/arXiv.1807.04188
14. ONNX, ONNX operators, 2022. https://github.com/onnx/onnx/blob/main/docs/Operators.md
15. Y. Xing, S. Liang, L. Sui, X. Jia, J. Qiu, X. Liu, Y. Wang, Y. Shan, and Y. Wang, *DNNVM: End-to-end compiler leveraging heterogeneous optimizations on FPGA-based CNN accelerators*, IEEE Trans. Comput.-Aided Design Integrated Circ. Syst. **39** (2020), no. 10, 2668–2681.
16. J. Deng, W. Dong, R. Socher, L. J. Li, K. Li, and L. Fei-Fei, *ImageNet: A large-scale hierarchical image database*, (IEEE Conference on Computer Vision and Pattern Recognition IEEE, Miami, FL, USA), 2009, pp. 248–255.
17. M. D. Zeiler and R. Fergus, *Visualizing and understanding convolutional networks*, *European Conference on Computer Vision*, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, (eds.), Springer, Cham, 2014, pp. 818–833.
18. A. Krizhevsky, I. Sutskever, and G. E. Hinton, *ImageNet classification with deep convolutional neural networks*, Commun. ACM. **60** (2017), no. 6, 84–90.
19. C. Szegedy, W. Liu, Y. Jia, et al., *Going deeper with convolutions*, (Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Boston, MA, USA), 2015, pp. 1–9.
20. K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, (Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA), 2016, pp. 770–778.
21. S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, *Aggregated residual transformations for deep neural networks*, (Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Honolulu, HI, USA), 2017, pp. 1492–1500.
22. F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size*, arXive preprint, 2016. https://doi.org/10.48550/arXiv.1602.07360
23. ONNX, Onnx model zoo, 2022. https://github.com/onnx/models
24. N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, *Tensor comprehensions: Framework-agnostic high-performance*

machine learning abstractions, aeXive preprint, 2018. https://doi.org/10.48550/arXiv.1802.04730

25. N. P. Jouppi, C. Young, N. Patil, et al., *In-datacenter performance analysis of a tensor processing unit*, (Proceedings of the 44th Annual International Symposium on Computer Architecture, Association for Computing Machinery, Toronto, Canada), 2017, pp. 1–12.
26. Z. Chen, C. H. Yu, T. Morris, J. Tuyls, Y. H. Lai, J. Roesch, E. Delaye, V. Sharma, and Y. Wang, *Bring your own codegen to deep learning compiler*, arXive preprint, 2021. https://doi.org/10.48550/arXiv.2105.03215

AUTHOR BIOGRAPHIES

Misun Yu received the M.S. degree from the Department of Computer Science and Engineering at Pohang University of Science and Technology, Republic of Korea. She is a principal researcher at the Electronics and Communications Research Institute (ETRI), Daejeon, Republic of Korea. Her main research interests include concurrent program analysis, software testing, deep learning, and embedded systems.

Yongin Kwon received the B.Sc. degree in Electrical and Electronic Engineering from the Korea Advanced Institute of Science and Technology, South Korea, in 2008, and M.S. and Ph.D. degrees in Electrical and Computer Engineering from Seoul National University, South Korea, in 2010 and 2015, respectively. From 2015 to 2019, he worked at Samsung Electronics as a staff software engineer. He has been with Electronics and Telecommunications Research Institute (ETRI) since 2019, where he is currently a senior researcher. His research interests include neural processing units, compiler, deep learning, and embedded systems.

Jemin Lee received the B.S. and Ph.D. degrees in Computer Science and Engineering from Chungnam National University in 2011 and 2017, respectively. He is a senior researcher at the Electronics and Communications Research Institute (ETRI). He was a postdoctoral researcher at the Korea Advanced Institute of Science and Technology (KAIST) in 2017–2018. His research interests include energy-aware mobile computing and deep-learning compiler.

Jeman Park received his B.S., M.S., and Ph.D. degrees in Electronics and Computer Engineering in Hanyang University, Republic of Korea, in 2004, 2006, and 2014, respectively. Since 2019, he has been with Electronics and Telecommunications Research Institute, Daejeon, Republic of Korea, where he is now a senior researcher. His main research interests are computer network, edge computing, and AI compiler.

Junmo Park received the B.S. degree in Computer Science from Kwangwoon University in 2012 and the M.S. degree with the Graduate School of Convergence Science and Technology at Seoul National University, South Korea, in 2020. He joined Samsung Electronics, Hwaseong, South Korea, in 2012, where he is involved in compiler optimization and development. He has been working for mobile GPU compiler since 2020 as a staff software engineer. His research interests include deep learning, compiler, embedded systems, HW/SW codesign, and optimizations.

Taeho Kim received the B.S. degree from Sungkyunkwan University in 1995 and the M.S. and Ph.D. degrees from the Department of Computer Science, KAIST, in 1997 and 2005, respectively. He is currently assistant vice president of AI SoC Research Division, ETRI. His research interests are safety-critical and intelligent cyber-physical systems, system software, and software engineering.

How to cite this article: M. Yu, Y. Kwon, J. Lee, J. Park, J. Park, and T. Kim, *PartitionTuner: An operator scheduler for deep-learning compilers supporting multiple heterogeneous processing units*, ETRI Journal **45** (2023), 318–328. https://doi.org/10.4218/etrij.2021-0446