Fine-Grained Compiler Optimization with Split-Schedule-Merge for Specialized Domains

Seungbin Song

The Graduate School

Yonsei University

Department of Electrical and Electronic Engineering

Fine-Grained Compiler Optimization with Split-Schedule-Merge for Specialized Domains

A Dissertation Submitted
to the Department of Electrical and Electronic Engineering
and the Graduate School of Yonsei University
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical and Electronic Engineering

Seungbin Song

June 2024

This certifies that the Dissertation of Seungbin Song is approved.

Thesis Supervisor	Prof. Hanjun Kim
Thesis Committee Member	Prof. Won Woo Ro
Thesis Committee Member	Prof. William Jinho Song
Thesis Committee Member	Prof. Youngsok Kim
Thesis Committee Member	Prof. Seonyeong Heo

The Graduate School
Yonsei University
June 2024

ACKNOWLEDGMENTS

”Surely your goodness and love will follow me all the days of my life,
and I will dwell in the house of the LORD forever.” - *Psalms 23:6 (NIV)*

항상 선한 길로 인도하시고 인자하심으로 이끌어주신 하나님께 감사드립니다.

학부생의 시작부터 박사 과정의 끝까지 지도해주신 김한준 교수님께 진심으로 감사드립니다. 10년이 넘는 시간 동안 수많은 조언과 가르침으로 저를 지도하시어 제가 한 명의 연구자로 성장할 수 있도록 해주셨습니다. 교수님께서 주신 가르침을 항상 마음 속에 담고, 앞으로 스스로의 가치를 증명하고 발전하는 연구자가 될 수 있도록 더욱 정진하겠습니다.

학위 논문 심사를 맡아주신 노원우 교수님, 송진호 교수님, 김영석 교수님, 허선영 교수님께 감사드립니다. 교수님들께서 주신 값진 피드백과 의견을 반영하여 연구를 더욱 발전시킬 수 있었고, 학술대회 논문을 준비함에 있어 좋은 결과를 얻을 수 있었습니다. 본 연구가 학위 논문에서 그치는 것이 아니라 향후에 연구를 더 발전시키도록 노력하겠습니다.

저의 박사 과정 동안의 연구는 컴파일러 최적화 연구실의 수많은 선후배들의 도움이 없었다면 불가능했을 것입니다. 포항에서부터 저를 이끌어주신 선배님이신 이경민 박사님, 김봉준 박사님, 김창수 박사님, 허선영 교수님께 진심으로 감사드립니다. 그리고 연구실에서 동고동락하며 함께한 이용우 박사, 정신녕, 김동관, 이재호, 최희림, 천선영, 윤성우, 이주민, 권현호, 염호윤, 이찬, 정건모, 정해은에게 감사드립니다.

저를 낳아주시고 길러주신 사랑하는 부모님께 감사드리고, 박사 학위를 받은 형에게 축하를 전합니다. 신앙적으로 안식처가 되어준 연세대학교회 대학청년부 목사님과 친구들에게도 감사드립니다. 마지막으로 박사 과정동안 사랑으로 함께한 정민정에게 감사와 사랑을 전합니다. 이들의 삶 속에 하나님의 사랑과 축복이 가득하길 진심으로 기도합니다.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ABSTRACT

Fine-Grained Compiler Optimization with
Split-Schedule-Merge for Specialized Domains

Domain-specific languages support programmabilities for programmers to implement and extend functions that fulfill the users' demands. Defining operations and interfaces of functions with some granularity allows programmers to compose domain-specific programs with the functions easily. Although the encapsulated functions entirely express the programs' functionalities, existing compilers do not fully optimize the programs because of the coarse granularity.

In software-defined Networking (SDN), existing compilers miss opportunities to parallelize fine-grained functions. They treat each packet processing table, which includes both match and action functions, as a single task unit. Therefore, they parallelize the programs without breaking down the match and action functions and analyzing dependencies between them.

In the domain of deep learning inference, existing compilers do not fully optimize fine-grained convolutions of tensor-decomposed deep learning models. They apply tensor decomposition on convolution weights and generate decomposed convolution sequences. However, because they only replace convolutions with the corresponding decomposed convolution sequences, they do not reorder or fuse the decomposed convolutions in a whole model perspective and lose opportunities to minimize memory usage.

This research proposes novel fine-grained compilers using a split-schedule-merge scheme for network programming and deep learning inference. It presents a new compiler named PSDN for network programming, which splits packet processing tables into match and action functions, schedules them into a pipeline, and merges the functions to reduce synchronization overheads. Additionally, for deep learning inference, it introduces a new compiler called TeMCO that splits decomposed convolution sequences into separated convolution layers, schedules the execution order of restore layers, and merges the decomposed convolution layers with non-decomposed layers. Through the split-scheme-merge scheme, the compiler can find more fine-grained parallelism opportunities in SDN programs and reduce peak memory usage in tensor-decomposed deep learning models.

The compilers of this work enhance the performance of domain-specific programs with the split-schedule-merge schemes. Compared to previous approaches, the PSDN compiler achieves a 12.1% reduction in packet processing time and a 3.5% decrease in resource utilization of seven network programs. The TeMCO compiler reduces peak memory usage of internal tensors by 75.7% with 1.08× to 1.70× inference time overheads of 10 decomposed models of five deep learning architectures. The compilers of this work can achieve performance gains on their domain-specific programs by utilizing the split-schedule-merge schemes tailored to their specific domains.

Keywords: Compilers, Networks, Deep Learning

1. Introduction

The demand for high-performance computing in server environments using domain-specific accelerators is ever-increasing. As applications become more complex and data volumes grow exponentially, traditional CPU-based server architectures struggle to keep pace with the required processing speeds. To address these challenges, server-side acceleration technologies such as smart network interface cards (SmartNICs) and graphic processing units (GPUs) have emerged as powerful solutions. SmartNICs offload and accelerate network processing tasks, freeing CPU resources and enhancing overall system efficiency, particularly in network function virtualization (NFV). As another example, GPUs, with their parallel processing capabilities, are well-suited for compute-intensive tasks, providing significant performance boosts for workloads such as machine learning, data analytics, and scientific computing.

Coupled with these hardware advancements, the rise of domain-specific languages (DSLs) offers tailored support for specific application domains, facilitating more efficient and expressive programming models. DSLs enable developers to write concise code that directly utilizes specialized accelerators like SmartNICs and GPUs. This synergy between server-side acceleration technologies and DSLs allows for optimized performance and simplified development workflows, driving innovations across various fields, from high-frequency trading to deep-learning acceleration. By harnessing the

power of SmartNICs and GPUs with DSL support, modern server architectures can meet the demands of applications, pushing the boundaries of what is achievable in computing performance and efficiency.

One of the key points that DSLs can provide efficient and expressive programmability for GPUs and SmartNICs is *abstraction*. That is, DSLs support programmers to implement their specialized functionalities by granulating them into functions. In the domain of Software-Defined Networking (SDN), recent network programming languages enable programmers to create a program that is composed of multiple functional units called *packet processing tables*. In the field of deep learning inference, tensor decomposition provides methods that decompose a convolution layer into the corresponding *decomposed convolution sequence*, regarding the sequence as the representative functional unit of the convolution layer.

Recent advancements in network programming languages enable programmers to construct network services on programmable network switches equipped with multiple subdivided functional units. The OpenFlow specification [1] initially outlined a programmable switch architecture that utilizes these multiple units. Subsequently, the P4 programming language [2] introduced a programming model based on these divisions. Each unit is represented by a *packet processing table* that includes *match* and *action* functions. The *match* function evaluates packet header values against the rules in the control plane, while the *action* function modifies internal metadata or packet header values based on the results of this comparison. For instance, programmers can use these tables to implement features like access control lists (ACLs) [3], Ethernet switching (layer-2) [4], IP routing (layer-3) [5], and equal-cost multi-path (ECMP) routing [6]. Net-

work service providers then integrate these tables into a control flow and deploy the configured network service on various platforms such as CPUs [7, 8, 9], FPGAs [10], or specialized packet processors [11, 12, 13].

In deep learning inference, tensor decomposition schemes decompose a convolution weight into factorized weights and generate a *decomposed convolution sequence* to substitute for the original convolution layer. Tensor decomposition [14, 15, 16, 17, 18, 19] is one of model compression schemes such as pruning [20, 21, 22, 23, 24, 25], quantization [26, 27, 28, 29, 30, 31], and knowledge distilation [32, 33, 34, 35, 36]. Tensor decomposition lowers computational complexity by applying mathematical methods to break down a convolution into a series of smaller convolutions. This technique involves factorizing a large-weight tensor from an original convolution into multiple smaller-weight tensors. As a result, the *decomposed convolution sequence* of factorized weight tensors generates an output tensor that closely approximates the output of the original convolution using the initial weight tensor. Therefore, existing research using tensor decomposition substitutes the convolution layers in the original model with decomposed convolution sequences, and they become functional units that represent the original convolutions.

Even though encapsulated functions express functionalities of domain-specific programs, existing compilers do not fully optimize the programs due to their coarse granularity. The existing network compilers [37, 38, 39] regard each packet processing table as a task unit and parallelize programs while match and action functions are aggregated. On the other hand, the existing tensor decomposition methods [14, 15, 16, 17, 18, 19] encapsulate decomposed convolutions into a sequence and do not reorder or fuse

the convolutions individually, so they lose opportunities to minimize memory usage by using internal tensors within the encapsulated decomposed convolutions.

A P4 program consists of match and action functions that read and modify various packet header values, allowing some parts of these functions to be executed concurrently. However, existing compilers [37, 38, 39] treat each packet processing table as a single task unit and schedule the execution order of the tables while keeping match and action functions intact. These compilers manage data dependencies between tables in a coarse-grained manner and assign all the tables to the physical pipeline of packet processors. As a result, they miss opportunities for fine-grained parallelism between match and action functions. To fully exploit these opportunities for parallelism, a network compiler should split packet processing tables into individual match and action functions and strategically schedule these match and action functions, rather than the whole tables, into the pipeline.

In decomposed convolution sequences in tensor-decomposed deep learning models, tensors within the decomposed convolution sequences have reduced sizes and memory usage. However, the existing tensor decomposition methods [14, 15, 16, 17, 18, 19] do not decapsulate the decomposed convolution sequences and lose opportunities to minimize memory usage of whole model inference by using the reduced tensors. These methods replace convolution layers with the corresponding decomposed convolution sequences. Inside the sequences, the channel sizes of tensors are reduced with decomposed convolutions, but they soon recover to their original sizes. This is because following non-decomposed layers, such as pooling or activations, require the tensors with the original sizes as their inputs, and skip connections store the tensors with the original

sizes for the latter layers. Therefore, these methods do not reduce the memory usage of whole model inference. To minimize the peak memory usage of tensor-decomposed models, a deep-learning compiler should utilize the reduced tensors not only in the decomposed convolution sequences but also through the model by replacing the original tensors with the reduced tensors.

To overcome the coarse granularity of the existing domain-specific compilers, this work proposes fine-grained compilers using split-schedule-merge schemes that optimize network programs and decomposed deep learning models. A splitting scheme decomposes the coarse-grained abstracted functional units into fine-grained functions, analyzing data and control dependencies among them. A scheduling scheme calculates and estimates computation and memory overheads of the decomposed functions and schedules the execution order of the functions while preserving the dependencies. Finally, a merging scheme fuses the functions to minimize synchronization overheads and function calls. These methods are augmented in the network compiler called PSDN and the tensor-decomposed deep learning compiler called TeMCO.

This work introduces PSDN, a novel compiler that reorganizes the network function programs with fine-grained functional units written in the P4 language to reduce packet processing latency. The PSDN compiler follows split, schedule, and merge schemes to parallelize network programs with subdivided functional units. First, the compiler splits packet processing tables into decomposed match and action functions and analyzes the control and data dependencies between them. The compiler generates a program dependence graph (PDG) for the split functions. Next, the compiler estimates the processing latency for each function based on their execution behavior and strategi-

cally schedules these functions within a packet processing pipeline, considering both PDG and estimated latencies. To minimize the pipeline's length, the compiler assigns independent functions wherever possible to the same pipeline stage. In the final phase, to reduce synchronization overheads among the functions, the compiler merges concurrently running functions within the same stage, as well as serially executed functions in the pipeline. The compiler generates a program in PX language [40], which is suitable for synthesis into FPGA-based network switches [10].

This work also proposes TeMCO, compiler optimizations for minimizing tensors' memory usage across tensor decompositions of deep learning inference. The TeMCO compiler replaces the usage of original internal tensors with reduced tensors by utilizing a split-schedule-merge scheme. First, the compiler splits a decomposed convolution sequence into the first, core, and last convolution layers and considers them individual optimizable units. Second, the compiler schedules the execution order of restore layers required in skip connections and duplicates the layers at the end of these connections to substitute original internal tensors with reduced tensors. Finally, the compiler merges non-decomposed activation layers with decomposed convolution layers. These fused layers avoid allocating input and output internal tensors and instead operate solely with the reduced tensors. As a result, these compiler transformations enable the continuous and unimpeded use of reduced tensors throughout the inference process, eliminating the need for tensor restoration.

The compilers of this work utilize split-schedule-merge schemes to improve the performance of domain-specific programs. The compilers split abstracted functional units of the domain-specific programs into fine-grained functions, exposing optimization op-

portunities for parallelization and memory usage reduction. Then, the compilers schedule the execution orders of fine-grained functions, maintaining the original dependencies and semantics of the programs. Finally, the compilers merge the functional units with subsequent ones to reduce computation costs and memory overheads. The split-schedule-merge scheme is shared in the PSDN compiler and the TeMCO compiler, improving the packet processing latency of network programs and the peak memory usage of tensor-decomposed deep learning models, respectively.

This work evaluates the PSDN compiler prototype with seven P4 programs [41] and synthesizes these programs onto the NetFPGA-SUME board [10]. The evaluation experiment analyzes the end-to-end packet processing latency through HDL simulation and measures the resource consumption of arithmetic logic units (ALUs), registers, and memories by synthesizing the compiled program into the SmartNIC. In comparison to prior work [39], the PSDN compiler achieves a 12.1% reduction in packet processing latency and a 3.5% decrease in resource utilization.

This work evaluates the TeMCO compiler prototype using 10 models from five deep learning architectures, including image classification with deep learning models [42, 43, 44, 45], as well as image segmentation with UNet [46]. The evaluation experiment benchmarks these against baseline models decomposed using the Tucker decomposition [47], with TeMCO's optimizations applied. The evaluation results demonstrate that TeMCO reduces the peak memory usage of internal tensors by 75.7% while adding an inference time overhead of between 1.08× and 1.70× across various batch sizes. Importantly, the optimizations implemented by TeMCO do not compromise the accuracy of the decomposed models.

Contributions of this work are:

- fine-grained compiler optimizations that decapsulate functional units and reorganize domain-specific programs with split-schedule-merge schemes,
- a new fine-grained network compiler named PSDN that converts a P4 networking program into a PX program for Xilinx SDNet,
- a splitting scheme for a network program that decouples match and action functions of packet processing tables,
- a scheduling scheme for a network program that schedules the decoupled functions into a pipeline with dependency analysis and clock cycle estimation,
- a merging scheme for a network program that combines the functions in the concurrent and subsequent stages to decrease both latency and resource usage,
- TeMCO's compiler optimizations that replace the uses of original tensors with reduced tensors and reduce peak memory usage in decomposed model's inferences,
- a splitting scheme for a decomposed deep learning (DL) program that considers decomposed convolution layers as individual layers,
- a scheduling scheme for a decomposed DL program that schedules the execution orders of restore layers in skip connections,
- and a merging scheme for a decomposed DL program that fuses the decomposed convolution layers with non-decomposed activation layers.

2. Background & Motivation

This section provides background on network programming and tensor decomposition in deep learning inference. In the domain of network programming, it describes the concept of software-defined networking (SDN) and introduces the structure of data plane language. In the field of deep learning inference, it describes tensor decomposition and the memory usage of tensor-decomposed models. Finally, it investigates the limitations of existing network compilers and tensor decomposition compilers.

2.1 Network Programming

2.1.1 Software-Defined Networking

Software-defined networking (SDN) decouples the control plane from the data plane in network switches, enabling them to be both controllable and programmable. OpenFlow [1] provides APIs to facilitate communication between the control and data planes, while ONOS [48] offers a control platform.

Figure 2.1 illustrates the packet processing structure of SDN. In the control plane, network service providers define *packet processing rules* for how network switches process packets based on their headers or metadata. In Figure 2.1, the Controller/OS provides packet processing rules: `forward` and `broadcast`. First, the rule `forward` checks

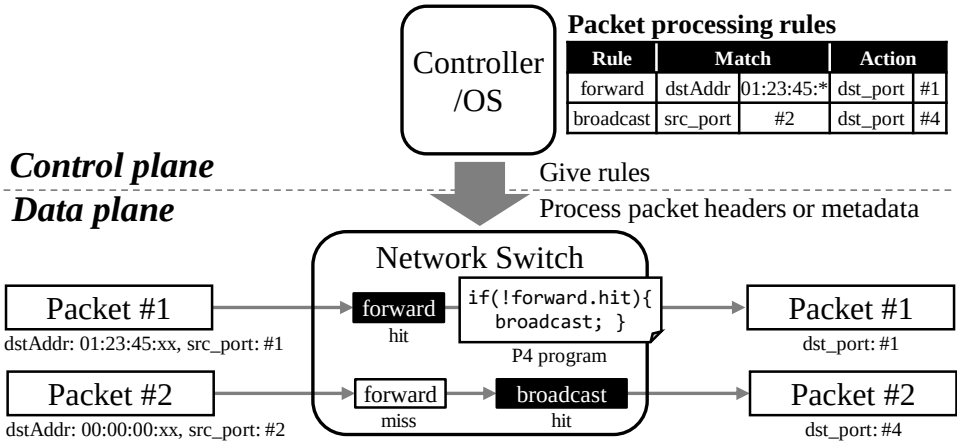


Figure 2.1: A brief structure of software-defined networking

whether the packer header value `dstAddr` starts with `01:23:45` and modifies the metadata `dst_port` to #1. Second, the rule `broadcast` checks whether the packet header value `src_port` is #2 and modifies the metadata `dst_port` to #4.

In the data plane, a network switch processes packets according to these predefined rules. In Figure 2.1, the network switch is programmed with a program written in the P4 language [2]. The rule `forward` and `broadcast` are applied to the corresponding *packet processing tables*. The example P4 program describes that if the table `forward` misses, apply the table `broadcast`. In the first case, the programmed network switch processes `Packet #1`, whose `dstAddr` starts with `01:23:45`. Because the match of the table `forward` hits, the switch modifies the metadata `dst_port` to #1 and passes the table `broadcast`. In the second case, the switch processes `Packet #2`, whose `dstAddr` starts with `00:00:00` and `src_port` is #2. Because the match of the table `forward` misses, the switch applies the table `broadcast` and modifies the metadata `dst_port` to #4 as `src_port` is #2 and the table `broadcast` matches.

The development of data plane languages and reconfigurable network switch architectures has made the data plane programmable. Network service providers can now program switches to support newly defined protocols or execute multiple network functions across different protocols using data plane languages rather than relying solely on built-in functions. One popular data plane language is P4 [2]. For instance, in-band Network Telemetry (INT) [49], load balancing [50] and in-network computation [51] have all been implemented using the P4 language. In Figure 2.1, the network switch can be programmed with the P4 program to conduct designated network packet processing.

2.1.2 Structure of P4 Language

P4 [2] is a domain-specific language designed for packet processing. While this work focuses on the P4 language, other languages, such as Huawei's Protocol-Oblivious Forwarding [52, 53], share a similar structure. A data plane program in these languages comprises three main components: a parser, a table pipeline, and a deparser. The parser accepts packets and generates packet headers and metadata according to network protocols. Using the parsed headers and metadata, the table pipeline modifies them with tables applying rules defined in the control plane. Finally, the deparser packages all the information and emits the modified packets.

Figure 2.2 illustrates a segment of the simplified P4 grammar related to the table pipeline. To simplify the example, this paper omits the definitions of headers, metadata, the parser, and the deparser. The table pipeline includes action functions, table declarations, an apply function, and extern functions. The extern functions are implemented outside of the program, such as Verilog modules. Because the P4 program does

```
table_pipeline := control table_pipeline_name(...) {
                    action_decl_list
                    table_decl_list
                    extern_decl_list
                    apply { stmt_list; } }
action_decl_list := action_decl_list action_decl
                 := action_decl
    action_decl := action name(args) { stmt_list; }
table_decl_list := table_decl_list table_decl
                 := table_decl
     table_decl := table table_name {
                    key = { key_list; }
                    actions = { action_name_list; }
                    default_action = action_name; }
action_name_list := action_name_list; action_name
                 := action_name
   extern_decl := extern extern_name(args);
     stmt_list := stmt_list; stmt
               := stmt
          stmt := id = expr
               := if(expr) {stmt_list;}
               := if(expr) {stmt_list;} else {stmt_list;}
               := table_name.apply()
               := action_name(expr_list)
               := extern_name(expr_list)
```

```
          args := arg, args
               := arg
           arg := type id
      key_list := key_list; key
               := key
           key := id : match_type
    match_type := exact
               := lpm
               := ternary
            id := packet header
                  or metadata field
     expr_list := expr_list, expr
               := expr
          expr := bool
               := int
               := id
               := (expr)
               := lop expr
               := expr op expr
               := table_name\
                    .apply().hit
```

Figure 2.2: The table pipeline's simplified grammar in the P4 language

not describe the implementation details of the extern functions, these functions are considered as a black box. On the other hand, the action functions are composed of various statements such as assignments, if conditions, and function calls. These functions modify packet header values or metadata or call other functions or tables.

The table declaration consists of a list of *keys* and *actions* defined within the table pipeline. These keys include the IDs of match variables, such as packet headers or metadata, and the types of matches. The types of matches include `exact`, `lpm`, and `ternary`. The `exact` match checks whether the values are the same, the `lpm` finds the longest prefix matches, and the `ternary` match finds the most similar match, including *don't care* term. The tables can be called using a `apply` method, and `apply().hit` returns the result of the table matches. The table compares the keys with rules from the control plane and conducts the designated action of the matched rule.

Figure 2.3 illustrates an example P4 pseudo program in Figure 2.1. The table pipeline in the example includes action definitions, tables, and an apply function. An *action* definition is a function that modifies packet headers (`hdr`) or metadata (`meta`). Some actions, such as `set_output_port` and `set_broadcast`, require arguments. These argument values are in packet processing rules provided by the control plane. Additionally, actions can modify packet headers or metadata either directly or by using external functions. A *table* definition includes *keys* consisting of packet headers or metadata for matches and *actions* that invoke actions when the keys are matched. The *apply* function serves as the main function, outlining the control flow. It can include conditional branches like `if` statements, but the P4 language does not support loops or iterations. Therefore, the P4 program results in an acyclic control flow for the table pipeline.

```
parser Parser(packet_in packet, out headers hdr,
    inout metadata meta) {...}

control TablePipeline(inout headers hdr, inout metadata meta) {
  action set_output_port(port_t port) {
    meta.dst_port = port;
  }
  table forward {
    key = { hdr.ethernet.dstAddr: lpm; }
    actions = {
      set_output_port;
      NoAction;
    }
    default_action = NoAction;
  }
  action set_broadcast(port_t port) {
    meta.dst_port = port;
  }
  table broadcast {
    key = { meta.src_port: exact; }
    actions = {
        set_broadcast;
        NoAction;
    }
    default_action = NoAction;
  }
  apply {
    // forward based on destination Ethernet address
    if (!forward.apply().hit) {
      // miss, then broadcast
      broadcast.apply();
    }
  }
}

control Deparser(packet_out packet, in headers hdr) {...}

Switch(Parser(), TablePipeline(), Deparser()) main;
```

Figure 2.3: An P4 program example

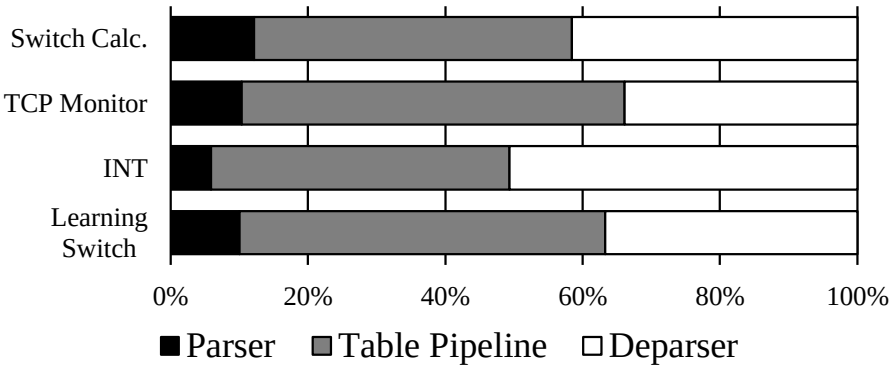


Figure 2.4: The latency percentage of a parser, a table pipeline, and a deparser

In the P4 program shown in Figure 2.3, the table `forward` uses the Ethernet destination address (`hdr.ethernet.dstAddr`) with the `lpm` match option as a key (Line 9) and includes two actions: `set_output_port` and `NoAction` (Lines 10 to 13). When the control plane provides a rule with the Ethernet destination address (`01:23:45:xx`), the network switch compares this address with the destination address of incoming packets. If the addresses match, the switch sets the destination port number (`meta.dst_port`) to the value #1 specified by the control plane using the `set_output_port` action. If the addresses do not match, the table `forward` passes the packet by default action `NoAction`. In this case, the table `broadcast` is applied, as defined in the `apply` function (Lines 27 to 33). The table `broadcast` compares the source port number `meta.src_port` with the rule and modifies `meta.dst_port` to #4 if the value exactly matches.

Although a data plane program includes a parser, a table pipeline, and a deparser, this work concentrates on optimizing the table pipeline, as it consumes the majority of the execution time in the data plane. Figure 2.4 shows the latency percentage of four P4 programs in HDL simulations. The latency of table pipelines takes 49.4% of overall

latency in geomean. The parser extracts packets with predefined protocols, and the deparser packages the packets with the processed data. Compared to the parser and the deparser, the table pipeline incurs overheads due to its reading and writing of packet header fields and metadata. Moreover, previous research [54] shows that the latency of the table pipeline increases sharply as the number of tables grows. Therefore, this work aims to optimize the table pipeline to reduce the packet processing latency in programmable switches.

2.1.3 Limitations of Existing Network Compilers

Although P4 [2] supports programmability and flexibility of network switches, current P4 compilers [11, 37, 38, 39] do not fully optimize network programs due to the lack of fine-grained dependency analysis. The existing compilers [11, 38] detect data dependencies in the case that a table modifies a packet header or metadata field and the following table uses this field in a match (match dependency) or alters it in an action (action dependency). However, these approaches only focus on table-level data dependencies, resulting in coarse-grained pipeline scheduling that misses potential parallelism opportunities among match and action operations.

Table-level dependency analysis treats a packet processing table as an atomic unit, overlooking the finer details of matches and actions. Figure 2.5 illustrates the definition of *use* and *def* within a table. Although match keys and actions have their own uses and defs, existing compilers analyze these only at the table level to identify data dependencies. To fully utilize parallelism opportunities at a finer granularity, it is necessary to decouple matches and actions from the tables in data dependency analysis.

$$
\begin{aligned}
\text{U(key)} &:= \{\ \text{id}\ \} \\
\text{D(key)} &:= \emptyset \\
\text{U(key\_list)} &:= \text{U(key\_list)} \cup \text{U(key)} \\
&:= \text{U(key)} \\
\text{D(key\_list)} &:= \text{D(key\_list)} \cup \text{D(key)} \\
&:= \text{D(key)} \\
\text{U(stmt)} &:= \{\ \text{id} \mid \text{id} \in \text{expr}\ \} \\
&:= \{\ \text{id} \mid \text{id} \in \text{expr}\ \} \cup \text{U(stmt\_list)} \\
\text{D(stmt)} &:= \{\ \text{id}\ \} \\
&:= \text{D(stmt\_list)} \\
\text{U(stmt\_list)} &:= \text{U(stmt\_list)} \cup (\text{U(stmt)} \setminus \text{D(stmt\_list)}) \\
\text{D(stmt\_list)} &:= \text{D(stmt\_list)} \cup \text{D(stmt)} \\
\text{U(action\_decl)} &:= \text{U(stmt\_list)} \\
\text{D(action\_decl)} &:= \text{D(stmt\_list)} \\
\text{U(action\_name)} &:= \text{U(action\_decl} \mid \text{action\_decl.action\_name} = \text{action\_name)} \\
\text{D(action\_name)} &:= \text{D(action\_decl} \mid \text{action\_decl.action\_name} = \text{action\_name)} \\
\text{U(action\_name\_list)} &:= \text{U(action\_name\_list)} \cup \text{U(action\_name)} \\
\text{D(action\_name\_list)} &:= \text{D(action\_name\_list)} \cup \text{D(action\_name)} \\
\text{U(table\_decl)} &:= \text{U(key\_list)} \cup \text{U(action\_name\_list)} \\
\text{D(table\_decl)} &:= \text{D(action\_name\_list)}
\end{aligned}
$$

Figure 2.5: A P4 table's *use* (U) and *def* (D)

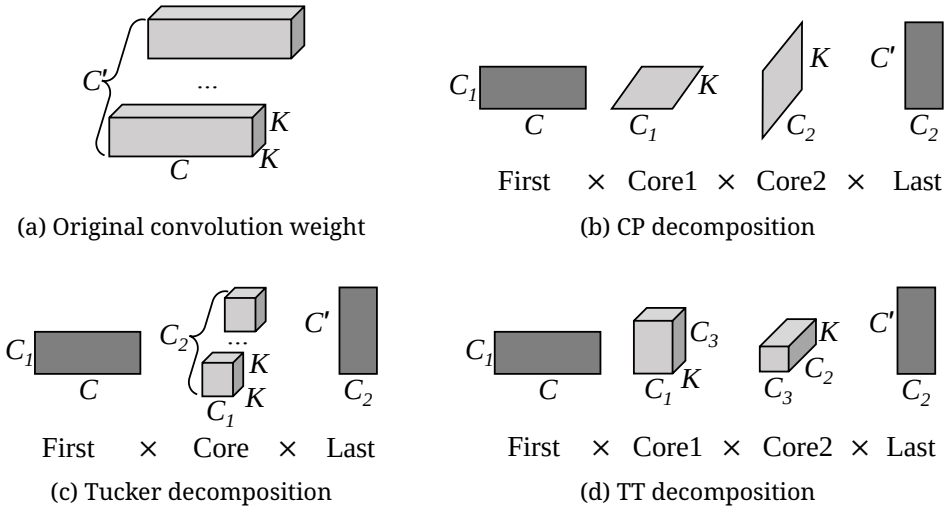
While decomposing tables into match functions and action functions can uncover additional parallelism opportunities, it may also lead to increased computation and area overheads in the synthesized hardware. Existing compilers [37, 39] allocate tables to hardware pipeline stages, synthesizing control flows at the table level. Combining match and action functions within a single module can reduce synchronization overheads because the compiler does not have to place synchronization barriers for every function but for tables.

However, a fine-grained compiler scheme that separates match and action functions into different pipeline stages may result in an increased number of stages and more complex control flows. This complexity can lead to unnecessary synchronization, potentially increasing the overall execution time despite shorter individual pipeline stages. Therefore, it is essential for a P4 compiler to simplify the control flows by pipeline scheduling and to reduce the number of pipeline stages by merging functions.

2.2 Tensor Decomposition in Deep Learning

2.2.1 Tensor Decomposition Methods

Tensor decomposition methods decompose weight tensors into core weights and low-ranked factor matrices. Tensor decomposition types include Canonical Polyadic Decomposition (CP) [55], Tucker Decomposition [15, 47], and Tensor Train Decomposition (TT) [17, 56]. Figure 2.6 depicts how different tensor decompositions break down a convolution weight tensor into smaller tensors. There are two 2D factor matrices: *first* and *last*. The width and the height of the first factor matrix are the channel size of the input (C) and reduced channel (C_1), and the width and the height of the last factor matrix are reduced channel (C_2) and the channel size of the output (C'). The *core weights* have lower dimensions or reduced channels than the original weight tensor. The shape of the core weights is different depending on the tensor decomposition methods, but the methods in Figure 2.6 all have the first and the last 2D factor matrices. The following figures in this paper will use a core weight of Tucker decomposition (Figure 2.6c) as a base example for simplicity, but the core weights can be replaced depending on the de-



(a) Original convolution weight

(b) CP decomposition

(c) Tucker decomposition

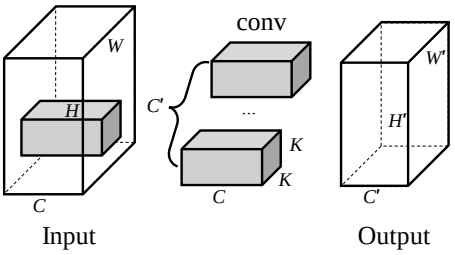
(d) TT decomposition

Figure 2.6: Tensor decomposition methods

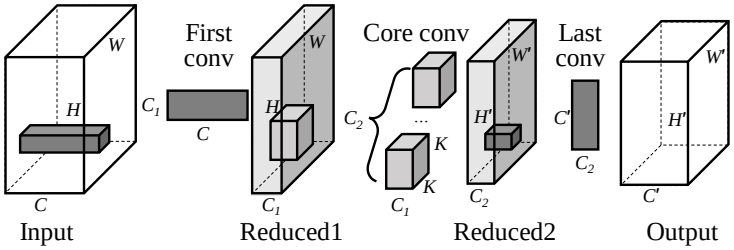
composition types. Multiplication of the core weights and the 2D factor matrices is an approximation of the original weight tensor.

In tensor decomposition, selecting appropriate *ranks* influences the quality of the decomposition and its effectiveness in compressing tensors. In Figure 2.6, the channel sizes C_1 to C_3 represent ranks of the decomposition. The ranks of the core weights and the factor matrices determine the level of approximation achieved and the amount of compression attained. Generally, higher ranks result in better approximation accuracy but require more computational and spatial resources. Conversely, lower ranks may lead to more significant compression but at the expense of increased approximation error. Retraining of the decomposed model compensates for approximation accuracy.

With the factor matrices and the core tensor, tensor decomposition can construct a *decomposed convolution sequence* that approximates the computation of a convolution



(a) Original convolution layer



(b) Decomposed convolution sequence

Figure 2.7: Tensor decomposition on a convolution layer

layer. Figure 2.7 describes an original convolution layer and the corresponding decomposed convolution sequence. In the two factor matrices in Figure 2.6, the First matrix and the Last matrix become weights of 1×1 convolution layers named First conv and Last conv in Figure 2.7b, respectively. The core convolution layer(s) (Core conv in Figure 2.7b) has the decomposed core weight(s) (Core in Figure 2.6). Note that while the core convolutions vary across different types of tensor decomposition, the first and final convolution layers remain consistent among these methods.

The first convolution layer reduces an input channel size (C) to a reduced channel size (C_1). The core convolution layer performs a reduced-sized convolution, accepting the reduced input channel size (C_1) and generating a reduced output channel size (C_2). Finally, the last convolution layer restores the reduced channel size (C_2) to an output

channel size (C'). In the rest of the paper, this work will refer to the first convolution layer as *fconv*, the last convolution layer as *lconv*, and the internal tensors within a decomposed convolution sequence (Reduced1 and Reduced2 in Figure 2.7b), of which the channel sizes are reduced, as *reduced tensors*.

2.2.2 Number of Operations of Decomposed Convolution Sequences

This work measures the reduction of the number of operations achieved through tensor decomposition. By decomposing tensors, tensor decomposition effectively reduces the computational complexity of convolution operations, primarily by reducing the channel size of the core convolution layer. The original convolution layer in Figure 2.7a requires the number of multiplication operations described in Equation (2.1).

$$CC'K^2H'W' \tag{2.1}$$

In Equation (2.1), K^2 represents the size of the convolution kernel, C denotes the number of input channels, and C', H', and W' represent the number of output channels, height, and width of the output tensor, respectively. The height and width of the output tensor (H' and W') are calculated based on the input dimensions (H and W) and the size of the convolution kernel (K), where $H' = H - K + 1$ and $W' = W - K + 1$.

The decomposed convolution sequence in Figure 2.7b requires the number of multiplication operations described in Equation (2.2).

$$CC_1HW + C_1C_2K^2H'W' + C_2C'H'W' \tag{2.2}$$

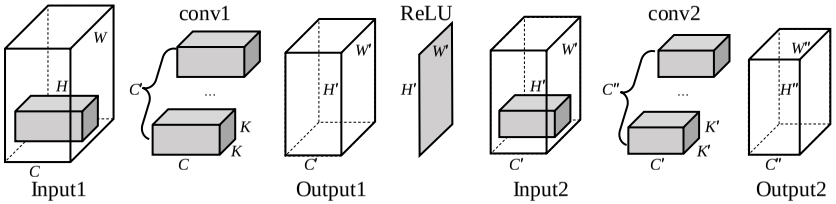
In Equation (2.2), C represents the number of input channels, C_1, C_2, and C' denote the number of channels after each decomposition step, and H and W represent the height and width of the input tensor, respectively. The dimensions H' and W' of the output tensor are calculated similarly to the original convolution layer, where $H' = H - K + 1$ and $W' = W - K + 1$. Comparing Equation (2.1) with Equation (2.2), the decomposed convolution sequence involves fewer multiplication operations due to the reduction in the number of input and output channels after each decomposition step. Assuming that the Tucker decomposition reduces channel sizes with decomposition ratio r, the reduced channel sizes C_1 and C_2 follow these equations: $C_1 = rC$ and $C_2 = rC'$. Then, Equation (2.2) can be rewritten as follows in Equation (2.3):

$$rC^2HW + r^2CC'K^2H'W' + rC'^2H'W' \tag{2.3}$$

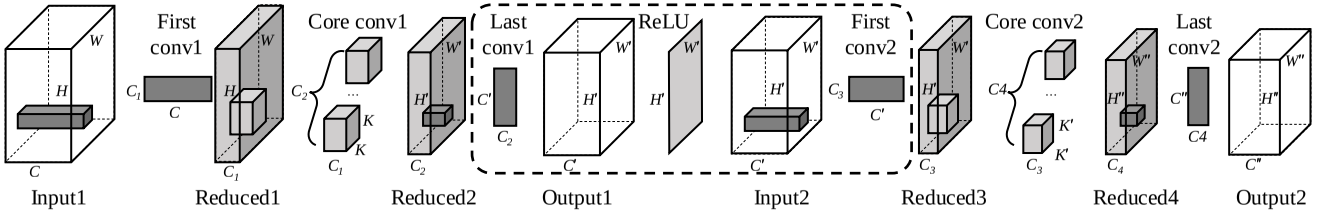
This example shows that tensor decomposition can reduce the number of operations. Previous work [15] proposes a tensor decomposition scheme that finds decomposition ratios (i.e., ranks) of convolution layers to reduce FLOPS and inference time of a model. However, the scheme fails to reduce memory usage of the model inference. The following section describes the memory usage analysis of tensor-decomposed models and why the previous scheme fails to reduce the memory usage.

2.2.3 Memory Usage of Decomposed Convolution Sequences

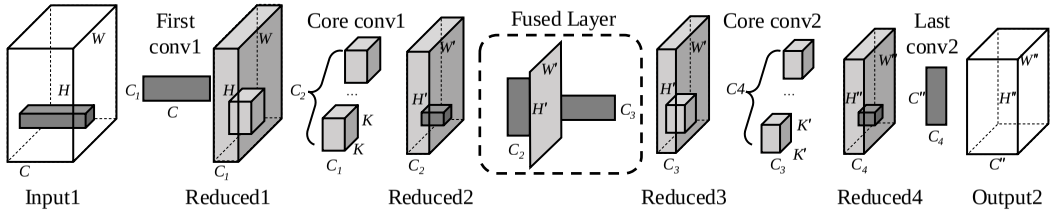
This work analyzes the peak memory usage of models decomposed by tensor decomposition. To inspect peak memory usage, we analyze a sequence of two consecutive



(a) Original convolution layers with an activation layer



(b) Two decomposed convolution sequences with an activation layer



(c) Optimized convolution sequences by TeMCO

Figure 2.8: Tensor decomposition on a convolution sequence

convolution layers and one activation layer between them rather than a single convolution layer. In Figure 2.8, we observe two convolution layers and one activation layer between them, as well as two sequences of decomposed convolutions with an activation layer. The peak memory usage of a model is determined by the memory requirements of two key types of tensors: *weight tensors* (parameters, filters, and kernels) and *internal tensors* (input and output tensors of layers, also known as feature maps). When measuring the peak memory usage influenced by internal tensors, it is essential to include activation layers like ReLU followed by convolution layers for an accurate assessment of peak memory usage.

In the context of model inference, deep learning frameworks, such as PyTorch [57] and TensorFlow [58], adopt a strategy of loading complete weight tensors before executing the inference. This approach involves initializing and storing all the weight tensors associated with the model's convolution layers. In Figure 2.8a, the size of the weight tensors for the convolution layers is outlined by Equation (2.4):

$$CC'K^2 + C'C''K'^2 \tag{2.4}$$

In Figure 2.8b, the size of the weight tensors in the decomposed convolution sequences is described by Equation (2.5):

$$CC_1 + C_1C_2K^2 + C_2C' + C'C_3 + C_3C_4K^2 + C_4C'' \tag{2.5}$$

If the tensor decomposition reduces channel sizes with decomposition ratio r, the reduced channel sizes C_1, C_2, C_3 and C_4 of reduced tensors follow these equations:

$C_1 = rC, C_2 = C_3 = rC'$ and $C_4 = rC''$. Then, Equation (2.5) can be rewritten as follows in Equation (2.6):

$$rC^2 + r^2CC'K^2 + 2rC'^2 + r^2C'C''K^2 + rC''^2 \tag{2.6}$$

While tensor decomposition effectively reduces the memory usage of weight tensors, it does not directly impact the memory usage of internal tensors allocated by deep learning frameworks. These frameworks manage memory allocation and deallocation for internal tensors dynamically. This dynamic memory management strategy involves allocating memory only for the internal tensors required by the currently active layer during model inference. Additionally, memory allocated to tensors that are no longer needed for subsequent inference tasks is promptly released.

In Figure 2.8, this work analyzes the peak memory usage of internal tensors by computing the maximum size of input and output tensors for each layer. This approach provides insights into the memory requirements imposed by the internal tensors within the model architecture. For the case of the two convolution layers illustrated in Figure 2.8a, the peak memory usage related to internal tensors is quantified using Equation (2.7):

$$MAX\,(CHW + C'H'W',\ 2C'H'W',\ C'H'W' + C''H''W'') \tag{2.7}$$

Assuming that $H \approx H'$, $W \approx W'$ and $C \approx C' \approx C''$, Equation (2.7) is reduced as follows in Equation (2.8):

$$2CHW \tag{2.8}$$

Similar to calculating the peak memory usage of the internal tensors in the convolution sequences associated with an activation layer, this work calculates the peak memory usage of the internal tensors in Figure 2.8b as follows in Equation (2.9):

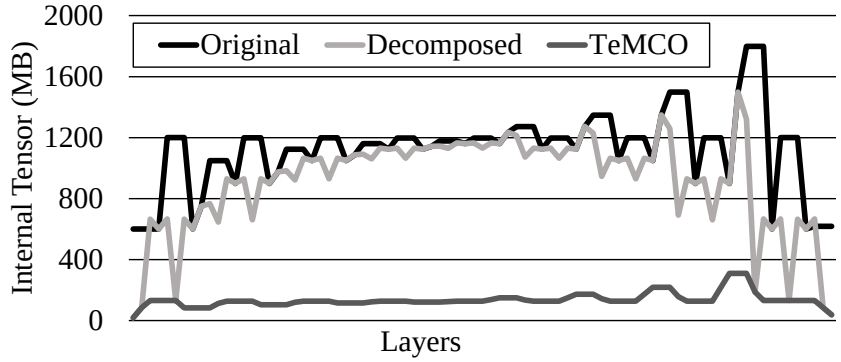
$$\begin{aligned} MAX(&CHW + C_1HW,\ C_1HW + C_2H'W', \\ &C_2H'W' + C'H'W',\ 2C'H'W',\ C'H'W' + C_3H'W', \\ &C_3H'W' + C_4H''W'',\ C_4H''W'' + C''H''W'') \end{aligned} \tag{2.9}$$

The channel sizes of the reduced tensors (denoted as C_1 to C_4) are smaller compared to those of the internal tensors (denoted as C to C''). This is because tensor decomposition reduces the channel sizes of convolution kernels. Then, this work simplifies Equation (2.9) as follows in Equation (2.10):

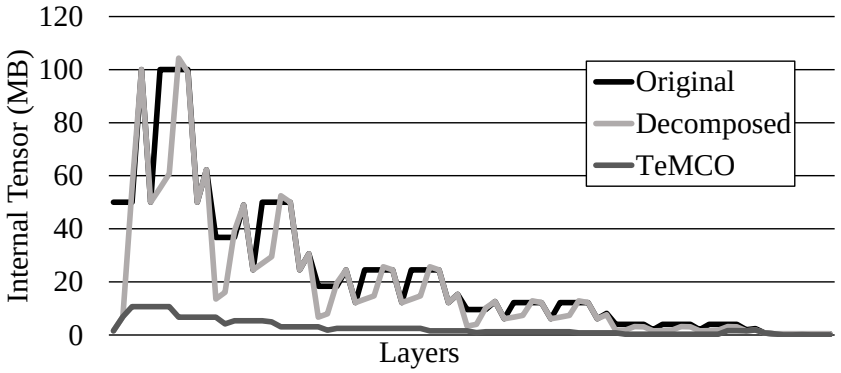
$$2C'H'W' \tag{2.10}$$

Here, the resulting size calculated from Equation (2.10) closely resembles the approximate size calculated using Equation (2.8). The memory usage of the internal tensors within the activation layer has a significant influence on the overall peak memory usage of the decomposed convolution sequences. Therefore, it is necessary to hide the allocation of internal tensors within the activation layer to reduce peak memory usage.

This analysis highlights that while tensor decomposition reduces the memory usage of weight tensors, it does not directly reduce the peak memory usage associated with internal tensors. The dominance of internal tensor memory usage within the activation



(a) UNet



(b) VGG-16

Figure 2.9: Memory usage of internal tensors

layers underscores the challenge of further reducing peak memory usage solely through decomposition methods. Consequently, achieving comprehensive memory reduction in tensor-decomposed models requires complementary strategies beyond tensor decomposition. The following section examines the actual memory usage of decomposed deep learning models and discusses the limitations of previous work.

2.2.4 Limitations of Existing Tensor Decompositions

The existing tensor decomposition schemes [14, 15, 16, 17, 18] miss the opportunity to further reduce peak memory usage of internal tensors despite tensor decomposition using less memory for decomposed convolution sequences. To illustrate the impact of skip connections and activation layers, this work compares the memory usage of internal tensors in the original and decomposed models using Tucker decomposition [47]. Figure 2.9 show the memory usage of internal tensors during 4-batch inference with UNet [46] and VGG-16 [43] on RTX 4090.

In Figure 2.9a, skip connections account for 76.2% of the peak memory usage by internal tensors in the tensor-decomposed model of UNet. The UNet architecture features an hourglass shape with skip connections horizontally linking the downsampling and upsampling blocks. During decomposed model inference, the downsampling blocks' decomposed convolution sequences restore reduced tensors to their original sizes, while the original tensors remain in the skip connections until the upsampling blocks use them. As a result, memory usage by skip connections in the decomposed model is comparable to that of the original model.

Conversely, in Figure 2.9b, the peak memory usage by internal tensors occurs during the computation of non-decomposed activation layers in VGG-16. VGG has a linear sequence of convolution, activation, and pooling layers. In the decomposed model of VGG-16, decomposed convolution sequences reduce the internal tensor sizes during core convolutions. However, these sequences soon restore the reduced tensors to their original sizes to be processed in non-decomposed activation layers. Thus, the peak mem-

ory usage caused by non-decomposed layers in the decomposed model is similar to that in the original model.

To address this issue, a new compiler optimization is needed to decapsulate the decomposed convolution sequences and to transform the decomposed deep-learning model to use only reduced tensors, as shown in Figure 2.8c. Figure 2.8c displays the optimized convolution sequence comprising reduced tensors, achieved by fusing *lconv1*, ReLU and *fconv2*, and removing Output1 and Input2 from Figure 2.8b. Since Figure 2.8b did not include the skip connections, the compiler optimization schemes require more sophisticated steps to handle complex data flows with decapsulated convolutions.

2.3 Motivation

To overcome the limitations of existing domain-specific compilers for network programming and tensor decomposition, this work proposes compilers that operate in finer granularity. These compilers decapsulate coarse-grained functional units into finer-grained functions, analyze their data and control dependencies, optimize programs by scheduling the functions, and fuse the functions to minimize synchronization overheads. This *split-schedule-merge* concept is shared in both the proposed network programming compiler and tensor decomposition compiler, decomposing coarse-grained functional units into finer-grained functions, scheduling the functions preserving dependencies, and merging the functions to reduce computation and memory overheads.

This work proposes PSDN, a compiler utilizing the split-schedule-merge scheme for network function programs written in P4. To overcome the limitations of existing compilers that optimize programs at the table level, PSDN splits packet processing tables into

matches and *actions* and schedules and merges the split functions of the programs. The PSDN compiler consists of table decomposition and dependency analysis as a splitting scheme, cycle estimation and pipeline scheduling as a scheduling scheme, and function fusion and code generation as a merging scheme.

The PSDN compiler decouples match and action functions and analyzes the control and data dependencies among them. It then estimates the processing latency of each function based on their execution behaviors and allocates the functions efficiently in a pipeline while respecting these dependencies and latency estimations. To minimize pipeline length, the compiler places independent functions into the same pipeline stage. Finally, the PSDN compiler generates a program written in the PX language [40], which can be synthesized into FPGA-based network switches [10]. To simplify the resulting hardware and reduce synchronization overheads, the compiler also fuses concurrent functions within the same pipeline stage and consecutive functions in the pipeline. With these optimizations, the PSDN compiler reduces the latency of packet processing by 12.1% and utilization of LUTs, registers, and memory by 3.5% in geomean.

This work also proposes the TeMCO compiler that utilizes the split-schedule-merge scheme for the deep learning models with tensor decomposition applied. TeMCO splits *fconv* and *lconv* from decomposed convolution sequences, schedules the execution orders of restore layers in skip connections and copies them, and merges *lconv* and *fconv* with non-decomposed activation layers in the decomposed models. The TeMCO compiler consists of inlining as a splitting scheme, skip connection optimization as a scheduling scheme, and activation layer fusion and concatenation layer transformation as a merging scheme.

The TeMCO compiler inlines the decomposed convolution layers and analyzes the dependencies of the layers in a model. It then finds skip connections, identifies reduced tensors of skip connections, copies required layers that restore the original tensors of the skip connections, inserts the copied layers into the end of skip connections, and replaces the original tensors with the reduced tensors. To avoid allocation of the original tensors in non-decomposed activation layers, the TeMCO compiler fuses the activation layer with *lconv* and *fconv*, which are alongside the activation layer. Finally, TeMCO transforms concatenation layers that are placed at the end of skip connections to apply activation layer fusion fully. With these optimizations, TeMCO reduces the memory usage of internal tensors by 75.7% of tensor-decomposed models, with 1.08× to 1.70× overheads of inference time in 4 to 32 batch sizes. Furthermore, the compiler optimizations of TeMCO do not reduce the accuracy of the decomposed models.

3. Split, Schedule, Merge for Network Programs

This chapter presents an overview of a split-schedule-merge scheme of the PSDN compiler for network programs. The compiler 1) splits match and action functions in packet processing tables, analyzing data and control dependencies and generates a program dependency graph, 2) schedules execution orders of functions into the pipeline regarding the dependencies and estimated clock cycles, and 3) merges concurrent and subsequent action functions and finally generates a PX program which is synthesizable to FPGA-based SmartNICs. The PSDN compiler reduces latency and resource utilization of network function programs with these optimizations.

3.1 Overview

Figure 3.1 provides an overview of the PSDN compiler. The frontend of the PSDN compiler is the p4c open-source compiler [59], which generates P4 intermediate representation (IR). Based on the P4 IR, the PSDN compiler decomposes P4 tables into match functions and action functions (Section 3.2.1). It then analyzes data and control dependencies among the functions and combines them into a program dependence graph (PDG), with prefetching of read-only functions (Section 3.2.2). After generating the PDG, the compiler performs cycle estimation and schedules the order of the function executions into a pipeline (Section 3.3). To minimize pipeline length, it allocates independent func-

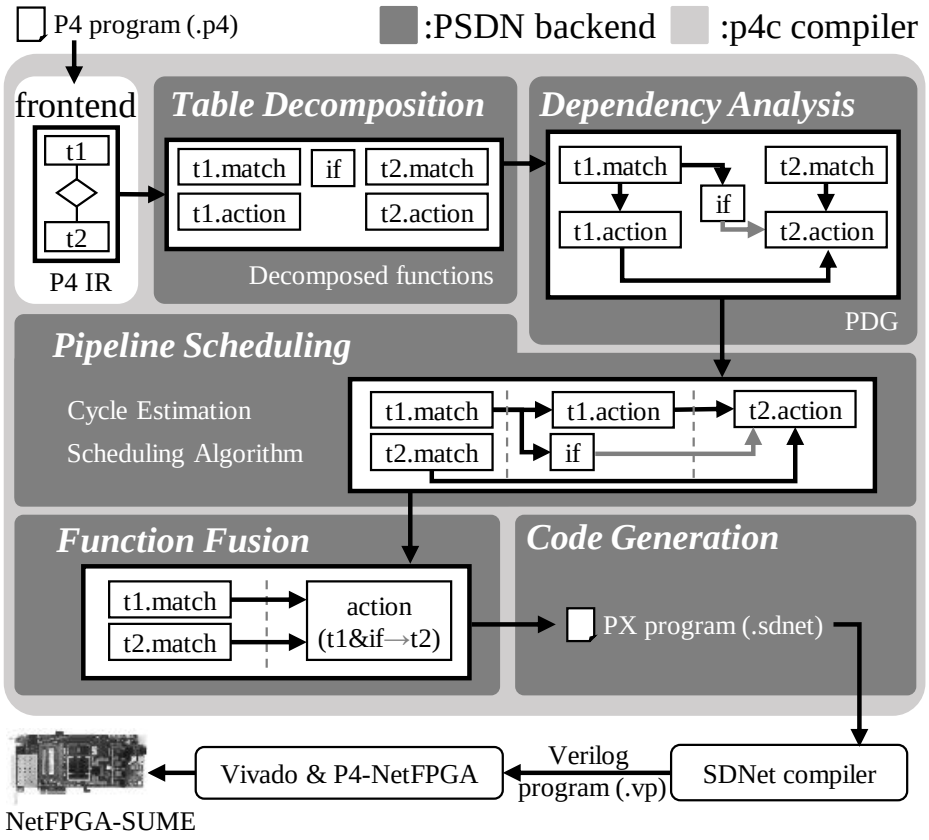


Figure 3.1: The PSDN compiler

tions to the same pipeline stage. Finally, the compiler performs function fusion, merging adjacent action functions (Section 3.4).

The role of this compiler is to translate a P4 IR to a PX program and optimize it. Other parts of the compilation process utilize the p4c compiler as the frontend and the SDNet compiler [60] as the backend. The PSDN compiler is implemented on top of the p4c compiler, with backend passes developed to translate P4 IR to an optimized PX program. Once the PSDN compiler generates the optimized PX program, the SDNet compiler trans-

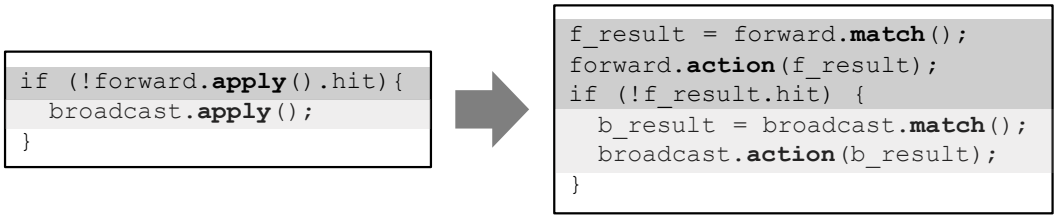


Figure 3.2: Table decomposition example of Lines 27 to 33 in Figure 2.3

lates it into a protected (encrypted) Verilog program, concealing the implementations of translated modules. Since the SDNet compiler hides the details of the translated modules, RTL (register transfer-level) optimizations are impossible. Currently, this work follows the workflow of P4-NetFPGA-SUME [41], which includes the SDNet compiler. Still, future work aims to develop an end-to-end compiler for high-level synthesis and to overcome pipeline limitations. The Verilog programs are synthesized using Vivado 2018.2 and tested on the NetFPGA-SUME board [10]. The evaluation infrastructure is detailed in Section 5.1.

3.2 Splitting Scheme

3.2.1 Table Decomposition

The PSDN compiler employs a table decomposition scheme that splits match functions and action functions for P4 tables. Figure 3.2 illustrates the decomposition of the table pipeline (Lines 27 to 33) from Figure 2.3. Initially, the PSDN compiler decomposes `forward.apply()` into `forward.match()` and `forward.action()`. The transformed code calls `forward.match()` and stores the return value in `f_result`. The `forward.action()` function then performs actions using `f_result` as an argument.

```
1 typedef enum { exact, lpm, ternary } LookupType;
2 typedef struct { bool hit; int action_id;
3   int* args; } Result;
4
5 Result ForwardTable::match(LookupType type, int* keys) {
6   Result result = PERFORM_MATCH(type, keys);
7   return result;
8 }
9
10 void ForwardTable::action(Result result) {
11   if(result.hit) {
12     switch(result.action_id){
13       //case 1: set_output_port(), default: NoAction()
14       case 1: meta.dst_port = result.args[0]; break;
15       default: break;
16     }
17   }
18 }
```

Figure 3.3: Match and action functions of Table `forward` in Figure 2.3

The `if` statement requires the hit-or-miss result of the table `forward` as the condition is `!f_result.hit`. The table decomposition also decomposes the table `broadcast` into `broadcast.match()` and `broadcast.action()`.

The table decomposition decouples match and action functions into separated instruction blocks. Figure 3.3 lists the decoupled match and action functions of the table `forward` from Figure 2.3. For simplicity, this work uses C++ semantics to describe the match and action functions. The match function handles the key match operation of the P4 table, receiving key variables as input, comparing with packet processing rules in the control plane, and returning a hit-or-miss result, a selected action number, and arguments. The action function determines whether to execute the actions based on the match result and performs the specified action with the given arguments. Note that the table decomposition inlines each action function into the switch cases of

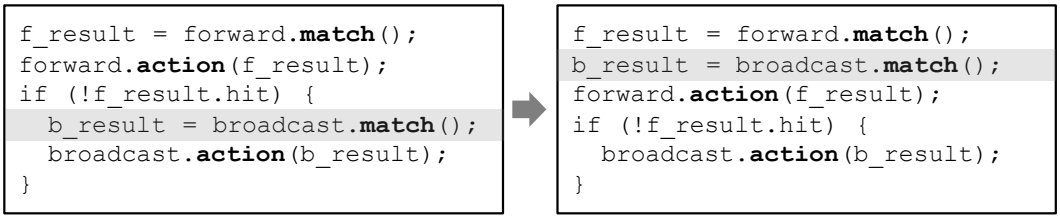


Figure 3.4: Code motion on match functions

`ForwardTable::action` (Line 14 in Figure 3.3). By applying the table decomposition, these match and action functions become separated instruction blocks.

P4 semantics allow direct instructions in the table pipeline, meaning the P4 apply block can contain conditional branches (e.g., Line 29 in Figure 2.3) or assignment statements. To simplify dependency analysis, the PSDN compiler translates these instructions into separate instruction blocks. Finally, the table decomposition generates instruction blocks with matches, actions, and condition instructions.

3.2.2 Dependency Analysis

The PSDN compiler's dependency analysis follows traditional data and control dependency analysis methods [61, 62]. It identifies data dependencies using the *use* and *def* information illustrated in Figure 2.5. The compiler constructs a program dependence graph (PDG) by combining data and control dependencies. Before constructing the PDG, the compiler redirects control dependencies from the table to the action function to prefetch read-only match functions.

The PSDN compiler conducts *code motion* on read-only functions and prefetches them. Match functions and certain extern functions are *stateless*, meaning that they

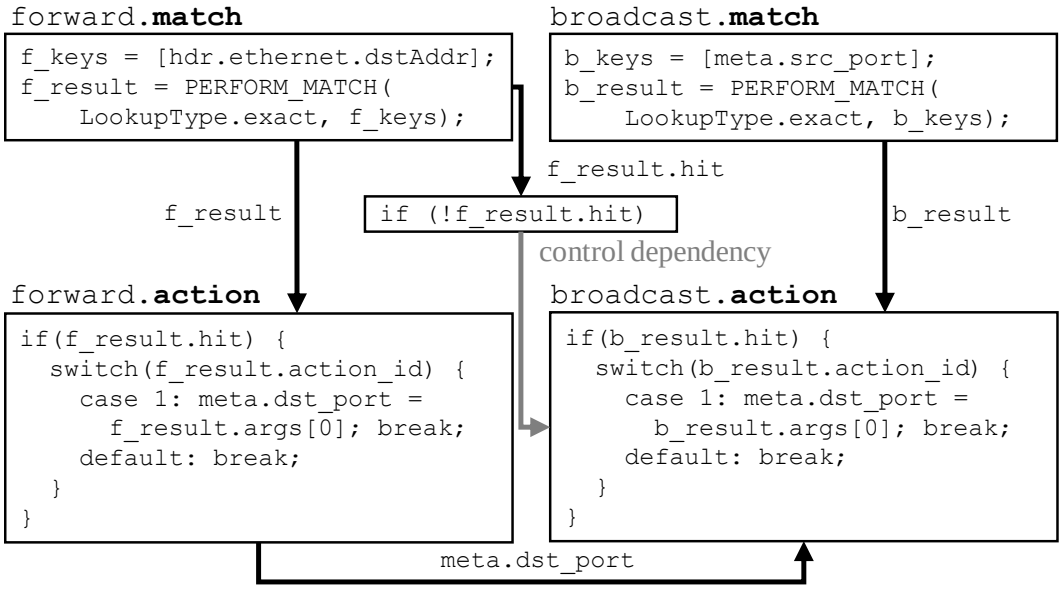


Figure 3.5: Program dependence graph

neither modify the program's state nor depend on the control flow. Figure 3.4 demonstrates how the PSDN compiler handles stateless functions. The match function only compares keys with control plane rules, reading variables without changing the program's state. Similarly, some extern functions like register-read and hash operate in the same way. The PSDN compiler moves condition-invariant function calls outside the conditional branches for stateless functions, thereby combining the control dependency with the action function. The prefetching enhances parallelization opportunities, as seen with the match functions of `forward` and `broadcast` in Figure 3.4 and reduces the execution time after the `if` statement.

Figure 3.5 presents the PDG for the example code in Figure 2.3. The match and action functions can have data dependencies. The `forward` and `broadcast` action functions

have a data dependency because they modify the same metadata field. The `if` statement requires `f_result.hit` to determine the execution of the `broadcast` match function. Since the `broadcast` match function is stateless and not data-dependent on the if statement, the PSDN compiler draws the control dependency from the if statement to the broadcast action function.

In terms of data dependence, the compiler can resolve *false* dependencies through variable renaming with a static single assignment (SSA) form. However, this work opts not to resolve false dependencies. Variable renaming requires additional instructions, which can introduce overhead on resource utilization and latency. For instance, the `forward` and `broadcast` action functions have write-after-write dependencies. A compiler could allocate these functions in parallel by renaming `meta.dst_port` another name, but this would require additional resources on registers to save the local results. Furthermore, the metadata field `meta.dst_port` is a global variable, meaning that the local modification in both `forward` and `broadcast` should be synchronized. Therefore, this work decides not to resolve these dependencies to avoid additional resource usage and synchronization overheads.

3.3 Scheduling Scheme

The PSDN compiler schedules the execution order of functions in the PDG and maps them into a pipeline. To reduce the pipeline length, the pipeline scheduler allocates independent functions within the same pipeline stage. It begins by estimating the latencies in clock cycles of each function and then uses a greedy-based algorithm to allocate the functions efficiently.

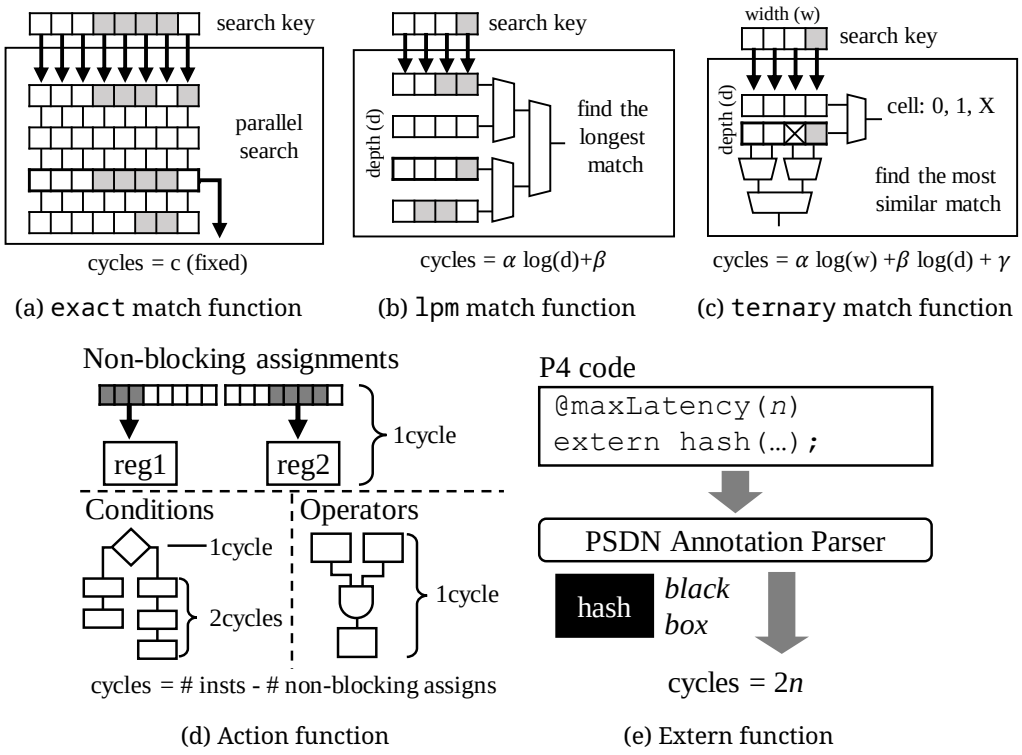


Figure 3.6: Function cycle estimation

3.3.1 Clock Cycle Estimation

Figure 3.6 shows clock cycle estimation of the PSDN compiler. The PSDN compiler estimates the required clock cycles for each function block. This paper describes the estimation methods of match, action, and extern functions.

The cycles for match functions depend on their types: `exact` match, `ternary` match, and `lpm` match. Although the cycle estimation formulas for match functions depend on their specific implementations, this work abstracts the execution models of these functions with the number of entries (depth, d) and length of a key (width, w). Table 3.1

Table 3.1: Cycle estimation of match functions

Lookup type	**Estimated cycles**
exact	6
lpm	20 $(= 2\log(256) + 4)$
ternary	$2\lfloor\log(\lceil w/40\rceil)\rfloor + 6$

shows the cycle estimation of the match functions used by the PSDN compiler, setting $d = 256$ and referenced from Xilinx documentation [63, 64, 65].

An `exact` match function (Figure 3.6a) uses a content-addressable memory (CAM) that conducts a parallel search to find an exact match of entries with a key. Therefore, the `exact` match function takes a constant latency.

An `lpm` match function (Figure 3.6b) finds the longest prefix match (LPM) to search keys. Calculating the length of the prefix match of each entry is constant, but finding the longest length takes more time when the number of entries (depth) is large. This process requires multiplexers to compare each entry with a binary search. Therefore, the estimated latency of the `lpm` match function is a linear function of $\log(d)$.

A `ternary` match function (Figure 3.6c) uses a ternary-CAM (TCAM) module that accepts don't-care terms (X) as elements of entries. The TCAM module finds the most similar match of entries with a key. Thus, TCAM requires multiplexers for width and depth dimensions to calculate the similar match for each entry and find the most similar match. The estimated cycles are a linear function of $\log(w)$ and $\log(d)$.

For action functions (Figure 3.6d), non-blocking assignments that are independent of each other take one cycle. Each blocking assignment and conditional statement also takes one cycle. For conditional branches, the estimator sums up the cycles of the longest

branch. Therefore, the estimated latency of the action functions equals the number of instructions in the longest branch minus the number of non-blocking assignments.

For extern functions (Figure 3.6e), the PSDN compiler treats them as a *black box*. Since the compiler does not know how a programmer implements the extern function, the programmer should provide information about the maximum latency of the extern function through an annotation (`@maxLatency`). Then, the PSDN compiler parses this annotation to determine the estimated cycles of the extern function.

The backend FPGA hardware uses different clock rates for action functions and match/extern functions. The clock period for match and extern functions is twice as long as for instructions in action functions. This difference in clocks is also considered when estimating the cycles. Therefore, the estimated clock cycles of the extern function (Figure 3.6e) are twice as long as the annotated cycles, and the estimated clock cycles of the match functions (Table 3.1) are multiple of two.

3.3.2 Pipeline Scheduling Algorithm

With a PDG and estimated cycles of functions, the pipeline scheduler allocates the functions to a pipeline. Algorithm 3.1 illustrates the pipeline scheduling of the PSDN compiler. The algorithm first identifies the functions that do not have any dependencies (I) from the PDG and identifies the next-independent functions (N) whose dependencies are resolved once the functions in I are allocated. Next, the algorithm identifies the functions (R) that are required to resolve the dependencies of the functions in N.

The algorithm allocates all functions in R within the pipeline stage and determines where to allocate the functions in $I \setminus R$. Where to allocate the function $v \in I \setminus R$ in a

Algorithm 3.1: Pipeline scheduling algorithm

Input : A program dependence graph $G = (v, e)$
Output : A scheduled pipeline P that is a list of pipeline stages
// PRED(v): predecessors of v
// SUCC(v): successors of v

```
1  P ← ∅
2  while G ≠ ∅ do
3      I ← {v | v ∈ G s.t. PRED(v) = ∅ }
4      N ← ∪_{v∈I} SUCC(v)
5      R ← ∪_{v∈N} PRED(v)
6      V ← Sort v ∈ I \ R in assending order of Latency(v)
7      for v ∈ V do
8          if Latency(v) ≤ MaxLatency(R) then
9              R ← R ∪ {v}
10         else if MaxLatency(R) > MaxLatency(N) then
11             R ← R ∪ {v}
12         end
13     end
       // ⊕: concatenation operator
14     P ← P ⊕ R
15     G ← G \ R
16 end
```

pipeline stage depends on `Latency(`v`)` over `MaxLatency(`R`)` and `MaxLatency(`N`)`. The scheduling algorithm selects either the current stage (R) or the next stage (N) whose `MaxLatency` is larger than `Latency(`v`)`. If `Latency(`v`)` is larger than `MaxLatency(`R`)` and `MaxLatency(`N`)`, the scheduler opts for the stage with the longer `MaxLatency`. These steps are repeated until all functions in the PDG (G) are allocated.

Figure 3.7 illustrates an example of pipeline allocation for the PDG shown in Figure 3.5. The pipeline scheduling algorithm allocates functions that are independent of each other to the same pipeline stage. In Figure 3.7, `forward` and `broadcast` match

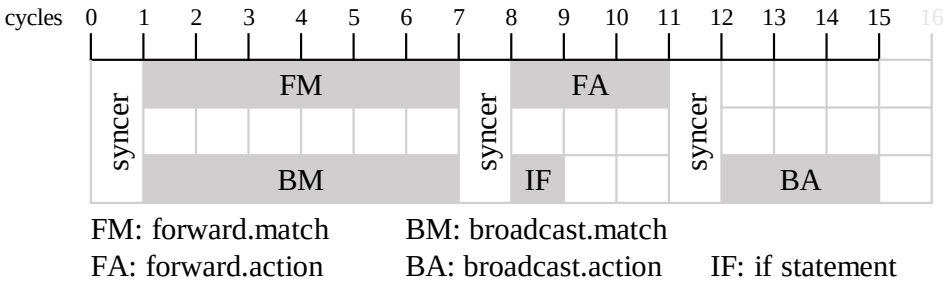


Figure 3.7: A scheduled pipeline

functions are located in the same stage. Functions allocated to the same pipeline stage are executed in parallel.

To synchronize data of each pipeline stage, the PSDN compiler inserts a synchronization barrier called a *syncer* between stages. A syncer receives results from the preceding stage, updates metadata and global packet header values, and transmits them to the next stage. Because the syncer introduces additional clock cycles, inserting too many syncers would increase latencies and diminish parallelization benefits. Consequently, the PSDN compiler employs a function fusion scheme to minimize the number of syncers, as discussed in the following section.

3.4 Merging Scheme

3.4.1 Code Generation

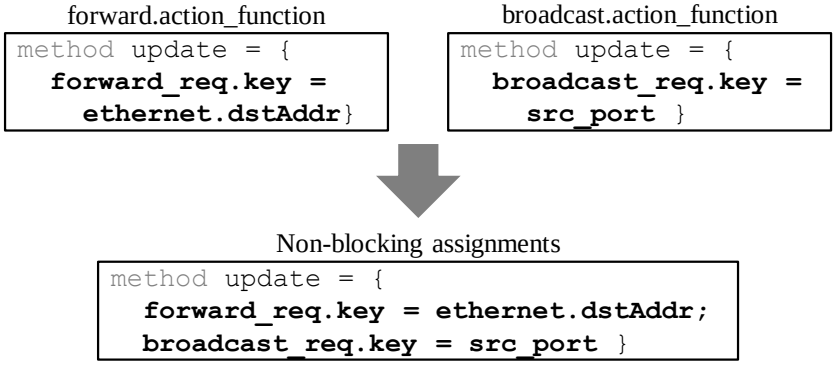
The code generation of the PSDN compiler follows a similar approach to the P4-SDNet compiler [39]. The PSDN compiler translates instructions in action functions and basic blocks into PX instructions within tuple engines. A tuple engine in the PX language [40]

consists of multiple *sections* that contain assignment statements (*update*) and a jump operation to the next section (*move_to_section*). The PSDN compiler groups non-blocking assignment statements into the same section and translates conditional statements into the *move_to_section* operation.

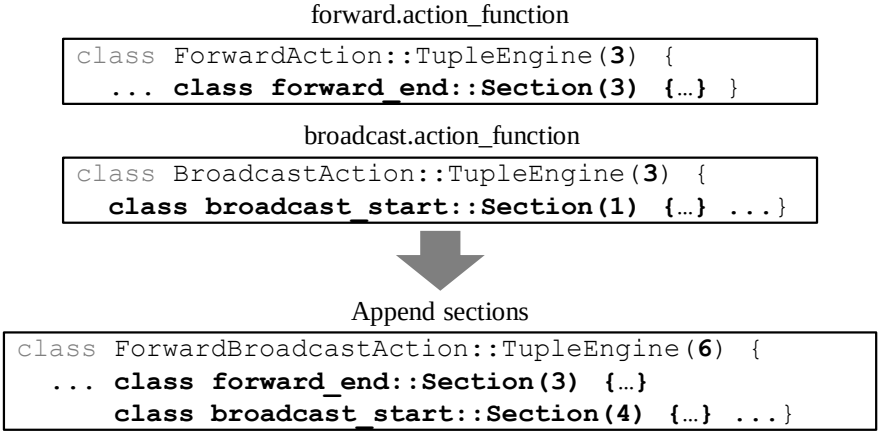
3.4.2 Backend Optimization and Function Fusion

During the code generation process, the PSDN compiler performs function fusion on action functions to reduce latency. First, it *merges* concurrent functions within the same pipeline stage into one tuple engine. For example, if the instructions in the action functions have no dependencies on each other, the compiler places these instructions into one section and merges the functions (Figure 3.8a). Second, the PSDN compiler *concatenates* an action function with the neighboring action function in the adjacent pipeline stages into one tuple engine. For instance, it concatenates the following sections into preceding sections, increases the number of the following sections by the number of the preceding sections, and sets the `move_to_section` operation of `forward_end` to point to the following `broadcast_start` section (Figure 3.8b).

Figure 3.9 illustrates the fused pipeline. The PSDN compiler fuses `if` statement, `forward` action function, and `broadcast` action function into a single action function. The instructions for `if` statement and `foward` action are located in the same section, and the instructions for `broadcast` action are located in the following sections. In this way, the PSDN compiler successfully deletes the syncer between the second and the third pipeline stages in Figure 3.7. Compared to Figure 3.7, the fused pipeline has a shorter processing time due to the reduction of the syncer.



(a) Merge action functions in the same pipeline stage



(b) Concatenate action functions in the adjacent pipeline stage

Figure 3.8: Function fusion methods

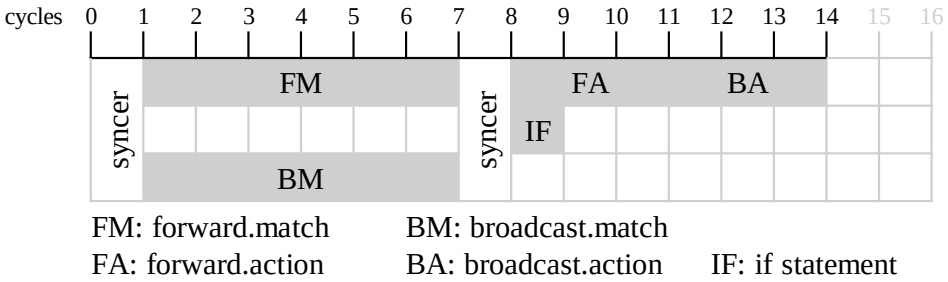


Figure 3.9: A fused pipeline

4. Split, Schedule, Merge for Deep Learning Models

This chapter describes an overview of a split-schedule-merge scheme of the TeMCO compiler for tensor-decomposed deep learning models. 1) The splitting scheme includes tensor decomposition [47, 55, 56] and inlining by TeMCO. Next, the compiler 2) schedules the execution orders of restore layers and optimizes skip connections. Finally, the compiler 3) merges non-decomposed activation layers with adjacent *lconv* and *fconv*. By performing these optimizations, the TeMCO compiler replaces the uses of internal tensors with reduced tensors produced by decomposed convolution sequences, reducing the peak memory usage of internal tensors.

4.1 Overview

This work introduces TeMCO, which is designed to optimize tensor-decomposed deep learning models to reduce peak memory usage incurred by internal tensors. The compilation process of the TeMCO compiler, illustrated in Figure 4.1, involves several sequential steps to achieve the replacement of internal tensors with reduced tensors (Section 4.2). Firstly, the compiler receives a tensor-decomposed deep learning model as an input and inlines decomposed convolution layers, analyzing the dependencies of layers. Secondly, it optimizes skip connections to replace the used internal tensors with reduced tensors, rescheduling precedent restore layers and analyzing memory and computation

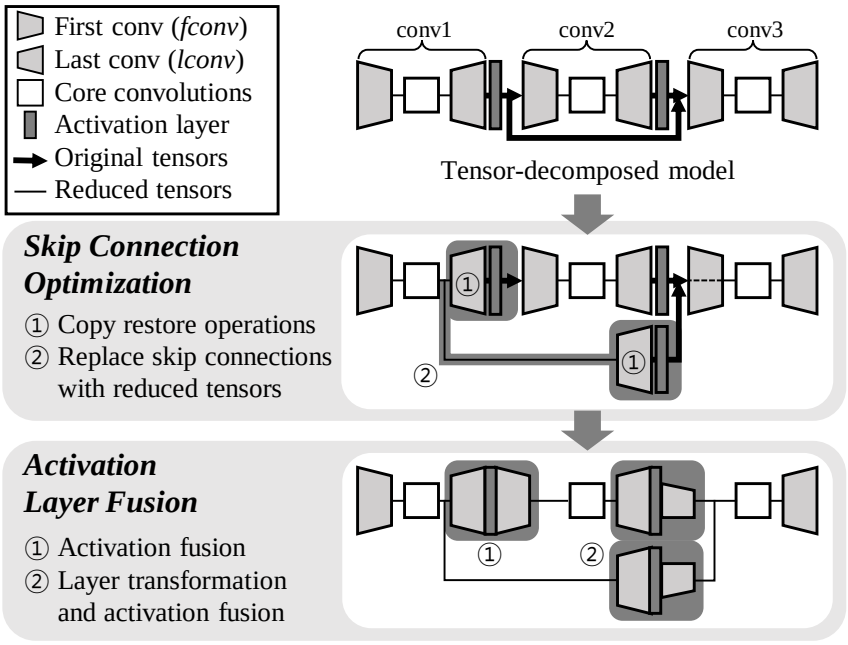
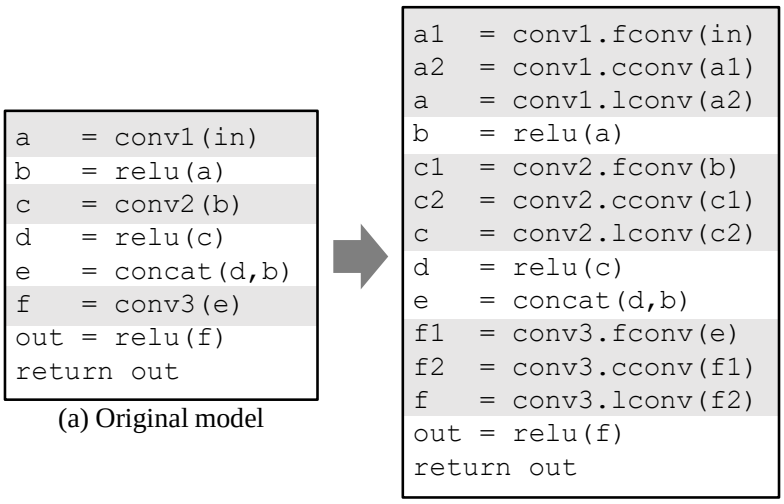


Figure 4.1: The TeMCO compiler

overheads of copying the layers (Section 4.3). Thirdly, it merges non-decomposed activation layers with neighboring *lconv* and *fconv* and generates specialized kernels not to allocate internal tensors but to compute with reduced tensors only (Section 4.4.1). Lastly, it transforms concatenation layers and *lconv* layers to reduce the number of layer calls and to be suitable for activation layer fusion (Section 4.4.2).

The actual implementation of TeMCO uses specific environments, but the concept of TeMCO's optimizations can be applied in broader environments. First, the TeMCO prototype implementation accepts Tucker-decomposed [47] models, but the optimizations can be applied to other tensor decomposition methods, described in Section 2.2.1, which have 1×1 factor convolution layers in front of and at the end of core convolutions (*fconv*



(a) Original model

(b) Decomposed model

Figure 4.2: Tensor decomposition example

and *lconv*). Second, the fused layer implementation of TeMCO uses CUDA [66], but the tiling mechanism can be augmented for CPU-based environments. Finally, the implementation of TeMCO uses PyTorch [57] FX graph compilers, but the transformation can be implemented by MLIR [67] infrastructures or other deep learning compilers.

4.2 Splitting Scheme

4.2.1 Tensor Decomposition and Inlining

The TeMCO compiler accepts tensor-decomposed deep learning models as inputs, employing tensor decomposition methods [47, 55, 56] as a splitting scheme. Tensor decomposition schemes decompose a convolution layer into several decomposed convolution layers, as described in Section 2.2.1. However, the previous tensor decomposition compilers [15, 16] encapsulate these layers into a decomposed convolution sequence. These

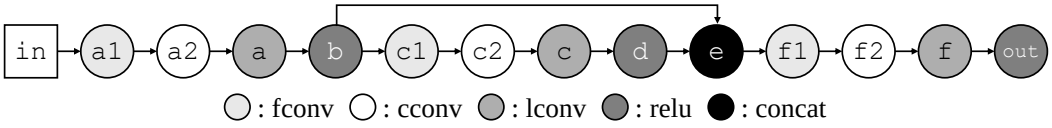


Figure 4.3: Program dependence graph

compilers define a class that represents the decomposed convolution sequence and includes all convolution operations inside the class.

Compared to the previous compilers, the TeMCO compiler inlines the decomposed convolution layers into a model as a form of the first convolution (*fconv*), core convolution (*cconv*), and the last convolution (*lconv*). Figure 4.2 describes an inlined tensor decomposition example. The decomposed convolution layers (`fconv`, `cconv`, `lconv`) of `conv1`, `conv2`, and `conv3` are inlined into the model, and they use reduced tensors (`a1`, `a2`, `c1`, `c2`, `f1`, `f2`) to propagate the internal results. The actual operation of *cconv* depends on the type of tensor decomposition methods (Section 2.2.1), but TeMCO's optimizations only use *fconv* and *lconv*, so the TeMCO compiler can conduct the optimizations in regardless of the type of *cconv*.

4.2.2 Dependency Analysis

The TeMCO compiler analyzes data dependencies of the inlined decomposed model and generates a program dependence graph (PDG). The TeMCO compiler employs an existing dependency analysis [68] to build a PDG. Figure 4.3 illustrates a PDG of Figure 4.2b. The PDG describes dependencies among layers, and the compiler can acquire lists of predecessors and successors of a layer from the PDG. For example, predecessors of a layer `e` (`PRED(e)`) are `b` and `b`, and successors of a layer `b` (`SUCC(b)`) are `c1` and `e`.

Algorithm 4.1: Skip connection optimization

Input : An ordered tensor node list L in SSA form,
Program dependence graph G of L,
Liveness analysis result $live \leftarrow$ `Liveness`(L, G)

Output : Skip-connection-optimized tensor node list O

```
// SUCC(v, G): successor list of v in G
// DISTANCE(a, b): distance of two node a, b
```

```
1  O ← L
2  for n in L do
       // Identify skip connections with distance
3      d ← DISTANCE(live[n].begin, live[n].end)
4      if d > DISTANCE_THRESHOLD then
           // Find reduced tensors and restore operations
5          l ← FindReduced(n, G)
           // Calculate overheads
6          if Overhead(n, l) then
7              for s in SUCC(n, G) do
                   // Insert operations l.list before s
8                  O ← InsertBefore(O, s, COPY(l.list))
9              end
10 end
```

4.3 Scheduling Scheme

To address the memory usage problem of skip connections in a tensor-decomposed model, the TeMCO compiler performs skip connection optimization. TeMCO schedules and reorders the execution of restore layers in skip connections. The optimization process for skip connections involves the following steps (Algorithm 4.1):

1. **Identify skip connections**: The compiler first identifies all skip connections present within the model architecture. To identify skip connections, the compiler performs liveness analysis on the model.

2. **Find precedent reduced tensors and required restore layers**: For each identified skip connection, the compiler traverses the dependence graph of the model and recursively finds the reduced tensors comprised by the skip connection's internal tensor and required restore layers.

3. **Evaluate FLOPS and memory trade-offs**: The compiler assesses the trade-offs regarding floating-point operations (FLOPS) and memory associated with copying the necessary restore layers that precede the skip connection. This evaluation determines whether copying these layers would result in more computational cost or memory overheads.

4. **Replace skip connections**: Based on the evaluation, the compiler copies and pastes the layers right before the use of the skip connections and replaces the original skip connections with the reduced tensors.

4.3.1 Identifying Skip Connections

To find skip connections, the TeMCO compiler first performs a *liveness analysis* of tensors on the whole model. Then, the TeMCO compiler finds skip connections whose tensors are long-lived through the inference, calculating the distance from the beginning to the end of the liveness.

Algorithm 4.2 shows the liveness analysis of the TeMCO compiler. The TeMCO compiler follows traditional liveness analysis algorithms [69, 70], but simplifies the algorithms to find the first *def* and the last *use* as *begin* and *end*, respectively. This can be done because a deep learning model is a form of a directly acyclic graph (DAG) that does

Algorithm 4.2: Liveness analysis

Input : An ordered tensor node list L in SSA form,
Program dependence graph G of L

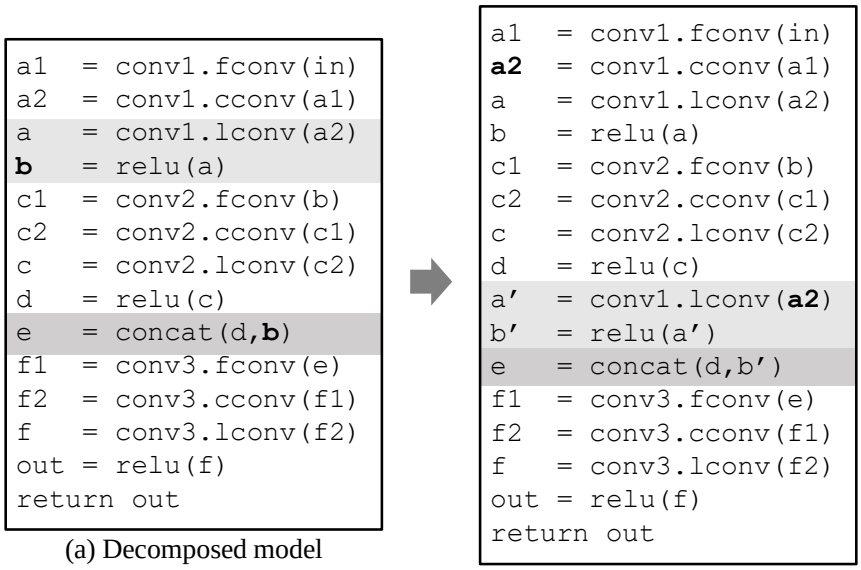
Output : Liveness analysis result list *live* for each node n in L

```
// PRED(v, G): predecessor list of v in G
1 Function Liveness(L,G):
2     live ← {}
3     for n in L do
4         live[n].begin ← n
5         for p in PRED(n, G) do
6             live[p].end ← n
7         end
8     end
9     return live
```

not have loops inside, and the model is a form of a single static assignment (SSA) that all the variable *def* is occurred once.

Skip connection optimization uses the result of liveness analysis to identify skip connections. TeMCO calculates the live distance of all nodes and finds long-lived tensors by comparing `DISTANCE_THRESHOLD` (Lines 3 to 4 in Algorithm 4.1). If a tensor remains live across multiple layers from its creation to its final usage, the compiler identifies it as a skip connection and continues the rest of the optimization steps.

Figure 4.4 shows a code example of skip connection optimization. In this example, the compiler identifies tensor `b` as a skip connection. The tensor `b` is live across multiple layers, and the distance of *begin* and *end* is larger than `DISTNACE_THRESHOLD` $= 3$. The identified skip connections become targets of skip connection optimization, and the compiler performs the following evaluations and optimizations on these skip connections to reduce memory usage precisely.



(a) Decomposed model

(b) Skip connection optmized model

Figure 4.4: Skip connection optimization example

4.3.2 Finding Precedent Reduced Tensors and Restore Layers

The TeMCO compiler finds precedent reduced tensors of a skip connection and identifies required layers to restore the skip connection by the reduced tensors. Algorithm 4.3 describes how the compiler finds precedent reduced tensors and required restore layers. The function `FindReduced` (Lines 17 to 31) recursively traverses the program dependence graph (PDG, G) of a model and searches the precedent reduced tensors. To do that, the function `FindReduced` uses the function `IsLConv` (Lines 1 to 7) that returns whether the node is *lconv* or not. The predecessor of *lconv* is the reduced tensor whose channel size is restored by the *lconv*. The function `IsLConv` identifies the layer as *lconv* if the operator type is a convolution layer with 1×1 kernels and the size of the output channel is larger than the size of the input channel. In Figure 4.4, the function `FindReduced`

Algorithm 4.3: Finding reduced tensors and restore layers

Input : A tensor node v, a program dependence graph G

Output : Results *res* of reduced tensor node list, size, peak memory usage

```
// PRED(v, G): predecessor list of v in G
// SIZE(v): tensor size of v by shape inference
```

Function IsLConv(v):
  $op \leftarrow$ OP(v) // operator of v
  if $op.type = conv$ **then**
    if $op.kernel\_size = op.stride = (1,1)$ **then**
      if $op.out\_channels > op.in\_channels$ **then**
        return True
  return False

Function Compare(a, b):
  return $a.size + b.peak < b.size + a.peak$

Function Peak(l, v):
  $peak \leftarrow 0,\ resided \leftarrow 0$
  for e *in* l **do**
    $peak \leftarrow$ MAX($resided + e.peak, peak$)
    $resided \leftarrow resided + e.size$
  end
  return $\leftarrow$ MAX($resided +$ SIZE(v), $peak$)

Function FindReduced(v, G):
  if IsLConv(v) **then**
    $res.list \leftarrow [v],\ res.size \leftarrow$ SIZE(v)
    $res.peak \leftarrow$ SIZE(v) + SIZE(PRED(v, G)[0])
    return *res*
  else
    $predList \leftarrow []$
    for n *in* PRED(v, G) **do**
      $predList \leftarrow predList \cup$ FindReduced(n,G)
    end
    $orderedList \leftarrow$ ORDER(Compare, $predList$)
    $res.list \leftarrow$ CONCAT($e.list$ for e in $orderedList$) $\cup [v]$
    $res.size \leftarrow$ SIZE(v), $res.peak \leftarrow$ Peak($orderedList, v$)
    return *res*
  end

is applied on the skip connection tensor b, and the function `IsLConv` identifies the layer `a = conv1.lconv(a2)` as a restore layer.

Because the number of the required restore layers could be larger than one, the TeMCO compiler schedules the execution order of the restore layers to minimize the peak memory usage on restoring. The function `Peak` (Lines 10 to 16) calculates the peak memory usage of the tensor node v with the ordered predecessor list l by accumulating the peak memory usage (*peak*) of the predecessors in l and the tensor size (*size*) of v. With calculated *peak* and *size*, the function `FindReduced` orders the predecessor list with the function `Compare` (Lines 8 to 9). Finding the optimal execution order that has a minimum peak memory usage is NP-hard, so this work employs a heuristic comparison to reduce the peak memory usage of the predecessors. Other previous work [69, 71, 72, 73] introduces scheduling algorithms to optimize peak memory usage, and this work will improve the scheduling algorithm with their ideas.

The function `FindReduced` finally returns the ordered list of predecessors that are scheduled to reduce peak memory usage, the size of current tensor node v, and the peak memory usage calculated by the function `Peak`. As `FindReduced` returns the *lconv* layers as leaf nodes, the predecessor list takes the reduced tensors as the arguments only. In Figure 4.4 the result list of `FindReduced(b)` is `[a=conv1.lconv(a2), b=relu(a)]`. The precedent reduced tensor of the skip connection `b` is the tensor `a`, and the restore layers that are required to restore the tensor `a` to `b` is `conv1.lconv` and `relu`. The following evaluation assesses the overheads of copying these restore layers, estimating whether the skip connection optimization is valuable or not.

Algorithm 4.4: Computation and memory overhead check

Input : Target skip connection node n, Reduced tensor node list l,
Program dependenge graph G,
Thresholds COMPUTE_THRESHOLD

Output : Boolean value that overheads are less than thresholds

```
// PRED(v, G): predecessor list of v in G
// SIZE(v): tensor size of v by shape inference
// FLOPS(v): FLOPS of v by calculating weight sizes
```

```
1 Function Overhead(n,l):
2     c ← 0, m ← SIZE(n)
3     for e in l.list do
4         c ← c + FLOPS(e)
5     end
6     for p in PRED(n,G) do
7         m ← m + SIZE(p)
8     end
9     return c ≤ COMPUTE_THRESHOLD and l.peak ≤ m
```

4.3.3 Evaluating FLOPS and Memory Trade-Offs

Before substituting skip connections with reduced tensors, the TeMCO compiler evaluates the FLOPS and memory trade-offs associated with introducing additional restore layers. Tensor decomposition reduces the FLOPS of the original model, as explained in Section 2.2.2, but replacing skip connections with reduced tensors necessitates additional computation from restore layers found by Algorithm 4.3. As the restore layers are copied and inserted right before the original skip connection is used, the compiler should estimate whether replacing with the reduced tensors and copying and inserting these restore layers is profitable or not.

Algorithm 4.4 describes how the TeMCO compiler evaluates the memory and computation overheads. For the computation overhead, the function Overhead (Algorithm 4.4)

accumulates the FLOPS of restore layers. The function `FLOPS` estimates the FLOPS of the convolution layers by calculating weight sizes as described in Section 2.2.2.

For example, the TeMCO compiler calculates the FLOPS of copying restore layers in Figure 4.4 and checks whether the `COMPUTE_THRESHOLD` satisfies. Here, this example sets the `COMPUTE_THRESHOLD` as the FLOPS of the original model without decomposition. The FLOPS comparison is outlined by Equation (4.1), where $F(\texttt{layer})$ represents the FLOPS of a `layer`:

$$\begin{aligned} & F(\texttt{conv1.fconv}) + F(\texttt{conv1.cconv}) \\ & + 2F(\texttt{conv1.lconv}) + 2F(\texttt{relu}) \\ & \leq F(\texttt{conv1}) + F(\texttt{relu}) \end{aligned} \tag{4.1}$$

In Equation (4.1), the FLOPS of the required restore layers (`conv1.lconv` and `relu`) are added. $F(\texttt{relu})$ can be removed on both sides, and Equation (4.1) can be expanded by Equation (2.1) and Equation (2.2), described in Equation (4.2):

$$\begin{aligned} & C_{in}C_{a_1}H_{in}W_{in} + C_{a_1}C_{a_2}K^2H_aW_a \\ & + 2C_{a_2}C_aH_aW_a + F(\texttt{relu}) \\ & \leq C_{in}C_aK^2H_aW_a \end{aligned} \tag{4.2}$$

Equation (4.2) can be further expanded by using Equation (2.3) with decomposition ratio r, described in Equation (4.3):

$$
\begin{aligned}
& rC_{in}^2 H_{in} W_{in} + r^2 C_{in} C_a K^2 H_a W_a \\
& + 2rC_a^2 H_a W_a + F(\texttt{relu}) \\
& \leq C_{in} C_a K^2 H_a W_a
\end{aligned}
\tag{4.3}
$$

During the compilation process, the TeMCO compiler computes the FLOPS based on tensor sizes and verifies if Equation (4.3) is met. By comparing the computation overheads with the original model, the TeMCO compiler aims not to introduce more latency than the original model without decomposition. Note that the skip connection optimization copies the required restore layers, so the additional computation overheads occur compared to the decomposed model.

For memory overheads, the TeMCO compiler compares the peak memory usage ($l.peak$) of restore layers with the accumulated restored sizes of predecessors. This means that the copied restore layers should use less memory than the restored tensors. In Figure 4.4, the compiler derives the comparison formula, described in Equation (4.4):

$$
SIZE(a_2) + SIZE(a) \ \leq \ SIZE(a) + SIZE(b) \tag{4.4}
$$

The duplicated size of a can be removed, and Equation (4.4) can be expanded by using Equation (2.9), described in Equation (4.5):

$$
C_{a_2} H_a W_a \ \leq \ C_a H_a W_a \tag{4.5}
$$

With the decomposition ratio $r < 1$, C_{a_2} is always smaller than C_a as $C_{a_2} = rC_a$ satisfies. Then, Equation (4.5) becomes always true. Therefore, the TeMCO compiler decides to copy the restore layers if Equation (4.3) is true. After evaluating the computation and memory trade-offs, the compiler copies the restore layers and replaces the skip connection with the reduced tensor.

4.3.4 Replacing Skip Connections

With the results of the evaluation, the TeMCO compiler duplicates and inserts the restore layers right before the use of the skip connection and replaces the original skip connection with the reduced tensor. In Figure 4.4, the compiler finds the sequence of the restore layer `[a=conv1.lconv(a2), b=relu(a)]` and evaluates the overheads of the sequence with thresholds. If the overheads are less than the thresholds, the compiler copies the sequence and inserts it right before the use of the skip connection `e=concat(d,b)`. To preserve the SSA form of a model, the compiler renames the copied tensors from `a`, `b` to `a'`, `b'`, respectively. By duplicating the restore layers, the skip connection optimized model in Figure 4.4 has the reduced tensor `a2` as the replaced skip connection instead of the original tensor `b`. Then, the optimized model reduces the peak memory usage as long as the skip connection lives. If the size of the tensor `b` is $C_aH_aW_a$ and the decomposition ratio $r < 1$, the size of the reduced tensor `a2` is $rC_aH_aW_a$. Then, the optimized model uses less memory of $(1-r)C_aH_aW_a$ than the decomposed model as long as the skip connection lives.

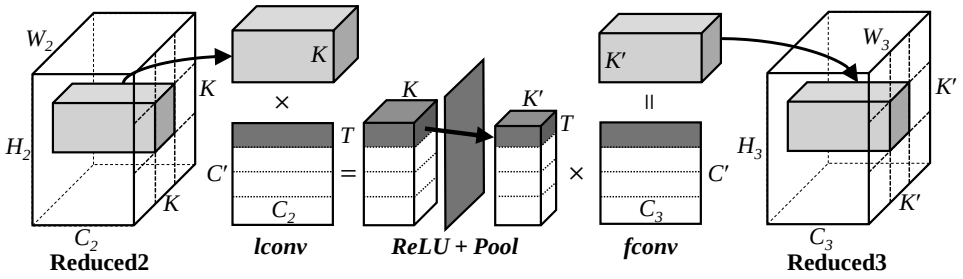


Figure 4.5: Fused layer in Figure 2.8c

4.4 Merging Scheme

4.4.1 Activation Layer Fusion

The TeMCO compiler employs activation layer fusion as a merging scheme. Despite the compiler's efforts to optimize skip connections, the model in Figure 4.8 still contains internal tensors used in non-decomposed activation layers, which contribute to the overall peak memory usage as described in Section 2.2.3. Therefore, the TeMCO compiler fuses non-decomposed activation layers with the preceding *lconv* and the succeeding *fconv* not to allocate internal tensors used within these layers. Then, the fused layer conducts convolution, activation, and convolution operations using a reduced tensor and skip allocating the original tensor.

This work implements fused layers with tiling of convolution, activation, and convolution to fuse the operation sequence of these operations. Figure 4.5 illustrates the operation of a fused layer. The purpose of fusing these layers is not to allocate the original tensors (Output1 and Output2 in Figure 2.8b) that are used in the activation layer. Therefore, the fused layer uses tiled memory space in terms of kernel dimension (K

and K') and tiled channel dimension (T over C'). Finally, the fused layer performs operations with the reduced tensors (Reduced2 and Reduced3) and gets reduced peak memory usage.

The types of activation and pooling layers can vary in the implementation of various deep learning models. For example, VGG [43] has sequences of Conv - ReLU - Conv and Conv - ReLU - MaxPool - Conv. However, most of the fusing target sequences are in the form of Conv - ReLU - Conv, and the other sequences are in the form of Conv - ReLU - Pool - Conv. Once this work implements the generalized fused layers, activation layer fusion can be applied to other models by replacing the actual operations of activation layers, such as SiLU [74] or GeLU [75]. The activation layer fusion scheme can be further developed when automatic operation fusion is implemented.

This work handles various activation layer fusions, including various activation sequence combinations between *lconv* and the activation layer. Firstly, Figure 4.6 depicts a fused layer implementation of *lconv* - ReLU - *fconv* in Figure 4.5. This research introduces a fused kernel approach aimed at executing parallelized operations across the $C2$, H, and W dimensions. The fused kernel begins by performing nested and tiled matrix multiplications using 1×1 convolutions on the input tensor `IN` and weight tensor `W1`. This convolutional operation is designed to iterate matrix multiplications over each channel $C1$ within the tensor. The output `v1` from these convolutions then undergoes an activation process using the ReLU function, and the resulting values are stored within a designated tile named `tileMID`. This approach enables efficient parallelized computation, utilizing tiled computation within each stage of the fused kernel to memory footprints during deep learning inference.

```
Tensor fused_crc(Tensor IN,W1,B1,W2,B2){
    /* T: tile size
       bx,by,bz: block idx of x,y,z dimension
       tx,ty,tz: tile idx of x,y,z dimension
    */
    c2 = bx * T + tx; //index of C2 with x
    h = by * T + ty; //index of H' with y
    w = bz * T + tz; //index of W' with z

    v3 = B2[c2]; //load bias
    for(i=0; i<C/T; i++){
        v1 = B1[c]; //load bias
        for(j=0; j<C1/T; j++){

            //load IN and W1 into tile
            tileIN[tx][ty][tz] = IN[j*T+tx][h][w];
            tileW1[tx][ty] = W1[i*T+tx][j*T+ty];
            __syncthreads();

            //perform lconv
            for(k=0; k<T; k++)
                v1+=tileIN[k][ty][tz]*tileW1[tx][k];
            __syncthreads();
        }
        //apply activation and load W2
        tileMID[tx][ty][tz] = relu(v1);
        tileW2[tx][ty] = W2[c2][i*T+ty];
        __syncthreads();

        //perform fconv
        for(l=0; l<T; l++)
            v3+=tileMID[l][ty][tz]*tileW2[tx][l];
        __syncthreads();
    }
    //update result to OUT
    OUT[c2][h][w] = v3;
    return OUT;
}
```

Figure 4.6: The fused kernel code of *lconv* - ReLU - *fconv*

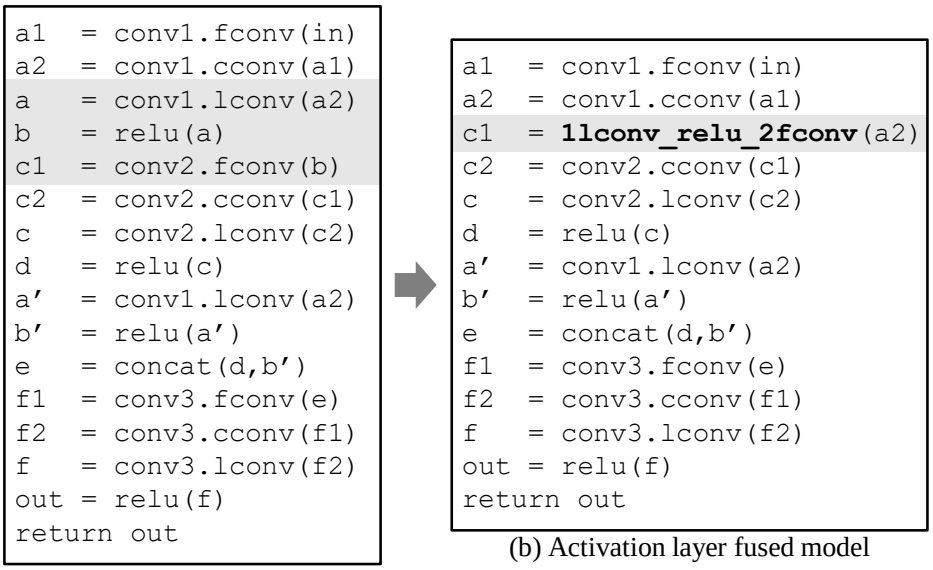
```
Tensor fused_crpc(Tensor IN,W1,B1,W2,B2,p){
    // p: kernel size of pooling layer
    c2 = bx * T + tx; //index of C2 with x
    h = by * T + ty; //index of H' with y
    w = bz * T + tz; //index of W' with z

    v3 = B2[c2]; //load bias
    for(i=0; i<C/T; i++){
        v1 = B1[c]; //load bias
        for(j=0; j<C1/T; j++){

            //load IN and W1 into tile
            tileIN[tx][ty][tz] = IN[j*T+tx][h][w];
            tileW1[tx][ty] = W1[i*T+tx][j*T+ty];
            __syncthreads();

            //perform lconv
            for(k=0; k<T; k++)
                v1+=tileIN[k][ty][tz]*tileW1[tx][k];
            __syncthreads();
        }
        //apply activation, pooling, and load W2
        tileMID[tx][ty][tz] = relu(v1);
        __syncthreads();
        tileMID2[tx][ty][tz] = pool(tileMID[tx][p*ty][p*tz],...);
        tileW2[tx][ty] = W2[c2][i*T+ty];
        __syncthreads();

        //perform fconv
        for(l=0; l<T; l++)
            v3+=tileMID2[l][ty][tz]*tileW2[tx][l];
        __syncthreads();
    }
    //update result to OUT
    OUT[c2][h][w] = v3;
    return OUT;
}
```

Figure 4.7: The fused kernel code of *lconv* - ReLU - Pool - *fconv*

This work also implements *lconv* - ReLU - Pool - *fconv* to cover another variant of *lconv*, activation, *fconv* sequences. This sequence appears when the inference of a convolution block is ended and moves to the next convolution block. Figure 4.7 describes a fused layer implementation of *lconv* - ReLU - Pool - *fconv*. If the fusion sequence includes a pooling layer, the fused kernel proceeds to apply pooling operations across the spatial dimensions H and W within the `tileMID`. Following the activation and pooling steps, the fused kernel further performs a 1×1 convolution across the entire channel C, updating the result `v3` to produce the final output tensor `OUT`.

The implementation of the fused kernel involves utilizing tiled buffers like `tileMID` in shared memory rather than allocating the entire size of the decompressed tensor `MID` in Figure 4.5. By adopting this approach, the activation layer fusion can strategically bypass the need to fully allocate decompressed internal tensors, resulting in a notable reduction in peak memory usage. This optimization is facilitated by the TeMCO compiler, which replaces sequences such as *lconv* - ReLU - *fconv* or *lconv* - ReLU - Pool - *fconv* with their corresponding fused layers.

In more detail, when the activation layer fusion is applied, the fused kernel efficiently manages memory resources by leveraging tiled buffers to store and process intermediate results. For instance, after conducting nested and tiled matrix multiplications using 1×1 convolutions, the resulting activations are stored within the designated tile `tileMID` in shared memory. This approach minimizes unnecessary memory allocation and deallocation operations, does not use the internal tensors that have significant memory overheads and optimizes memory usage throughout the inference. The TeMCO compiler seamlessly integrates these optimizations by replacing conven-



(a) Skip connection optimized model

(b) Activation layer fused model

Figure 4.8: Activation layer fusion code example

tional sequence-based layers like *lconv* and *fconv* in tensor-decomposed models with more memory-efficient fused layers, enhancing the overall memory efficiency of deep learning inference tasks.

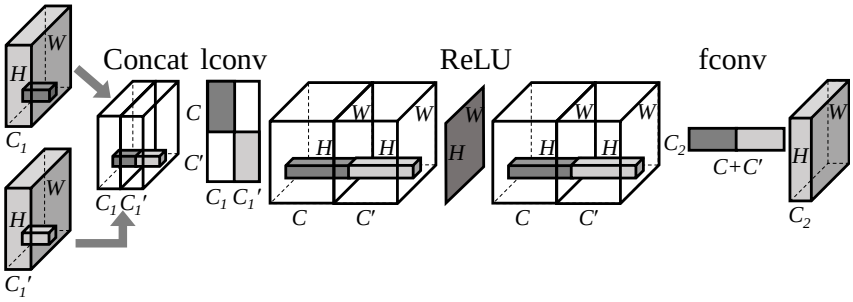
Figure 4.8 shows the code example of activation layer fusion. In this example, the sequence of `conv1.lconv-relu-conv2.fconv` is the target of activation layer fusion. Activation layer fusion replaces the sequence with the fused layer `1lconv_relu_2fconv`. Here, the allocation of the tensors `a` and `b` from the activation layer `relu` is the significant bottleneck of peak memory usage. However, activation layer fusion fuses the non-decomposed activation layer with decomposed convolution layers, hiding the allocation of these tensors and using the reduced tensors `a2` and `c1` for the operations.

4.4.2 Concatenation Layer Transformation

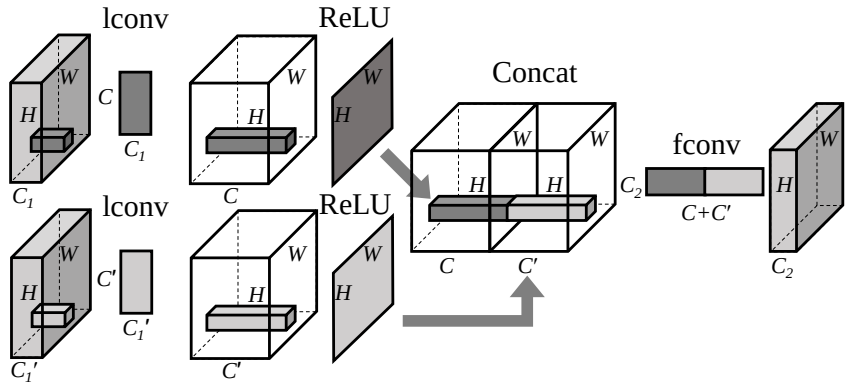
The TeMCO compiler transforms a concatenation layer with decomposed convolution layers that are neighboring it to apply activation layer fusion. Figure 4.9 shows the three forms of concatenation layer transformation. This section describes how the original form in Figure 4.9b is generated and how the concatenation layer transformation transforms this form to Figure 4.9c and Figure 4.9a.

Deep learning models [45, 46] with skip connections use concatenation layers to merge the results of skip connections with the mainstream and apply convolutions on the concatenated tensors. In terms of decomposed models, a concatenation layer precedes a decomposed convolution sequence, so the sequence is a form of Concat - *fconv*. As the TeMCO compiler applies skip connection optimization described in Section 4.3 and copies restore layers into the skip connection, the concatenation layer has the restore layers (*lconv* - ReLU) as its predecessors. Therefore, the original concatenation sequence is the form of Figure 4.9b after the skip connection optimization.

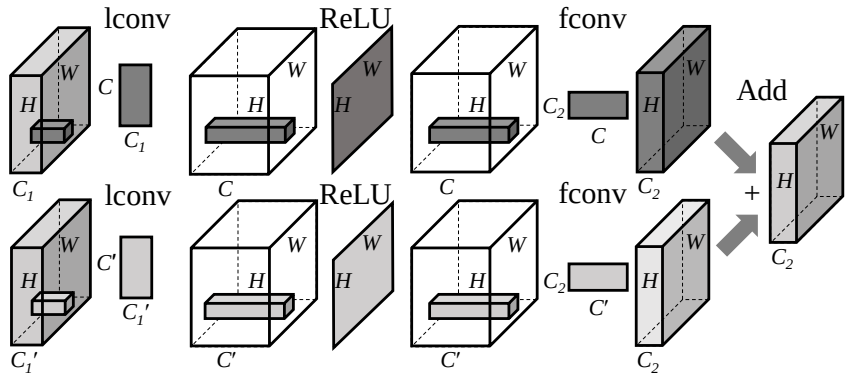
The TeMCO compiler can transform the original concatenation sequence by dividing *fconv* and replacing the concatenation layer with an add layer, as described in Figure 4.9c. The *fconv* layer is an 1×1 convolution layer that performs matrix-vector multiplication on each channel. On the other hand, the concatenation layer concatenates the result tensors of the ReLU layers in the dimension of the channel. Here, the concatenated channel is multiplied by the weight matrix of *fconv*. The matrix-vector multiplication is a type of multiply-accumulate operation, so the addition of two matrix-vector multiplications that are divided into two channels shows the same result as the



(a) Merging *lconv*



(b) Original concatenation sequence



(c) Dividing *fconv*

Figure 4.9: Concatenation layer transformation

original multiplication. In Figure 4.9c, the weight matrix of *fconv* is divided into two channels: C and C'. The results of two *fconv* layers are added with the add layer. Here, the add layer conducts the role of accumulation in the multiply-accumulate operations. Therefore, the results of Figure 4.9b and Figure 4.9c are mathematically the same.

On the other hand, the TeMCO compiler can transform the original concatenation sequence by merging *lconv* layers and placing the concatenation layer in front of *lconv*, as described in Figure 4.9a. The *lconv* layer is also an 1×1 convolution layer. Before merging *lconv*, the compiler places the concatenation layer for reduced tensors, and the layer concatenates them into the channel dimension. To ensure the results are the same, the compiler merges the weight matrices of *lconv* layers in a diagonal way, filling zeros with empty spaces. Then, the concatenated results are the same. This transformation is applied when the types of the activation layers are the same. In Figure 4.9b, two sequences use the same activation layer ReLU, so the transformation safely transforms them into merging *lconv*, as described in Figure 4.9a.

Figure 4.10 and Figure 4.11 show the code examples of concatenation layer transformation. The TeMCO compiler can transform the concatenation layer in both ways. After the layer transformation is applied, the model is well-formed to apply activation layer fusion. In other words, concatenation layer transformation replaces the concatenation layer with other forms and generates the sequence of *lconv* - ReLU - *fconv*. In this way, concatenation layer transformation expands the applicability of activation layer fusion, reducing peak memory usage on concatenation layers.

In the application on DenseNet [45], the TeMCO compiler merges *lconv* (Figure 4.9a) to reduce the number of layer calls and divides *fconv* (Figure 4.9c) to manage different

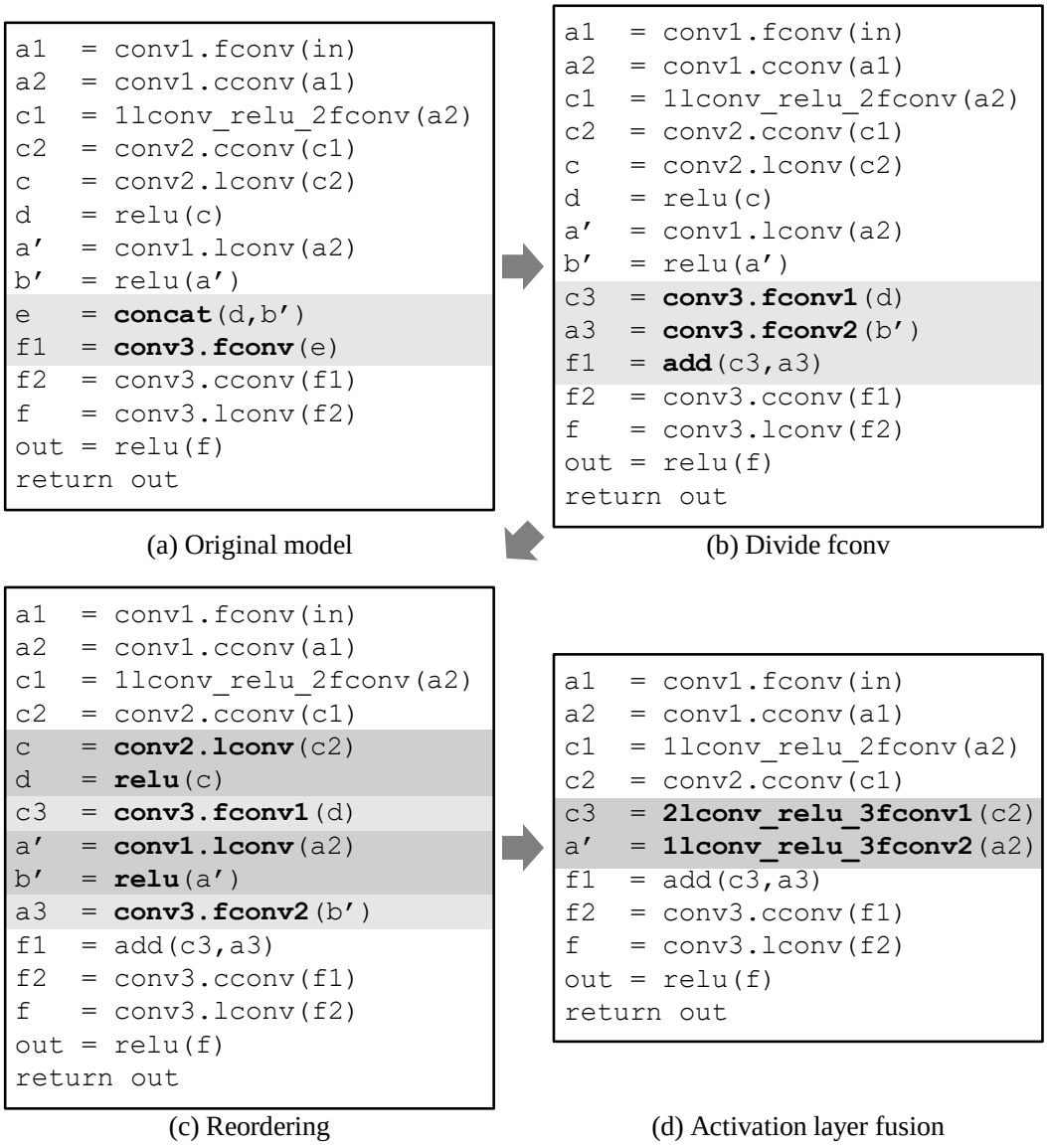


Figure 4.10: Dividing *fconv* code example

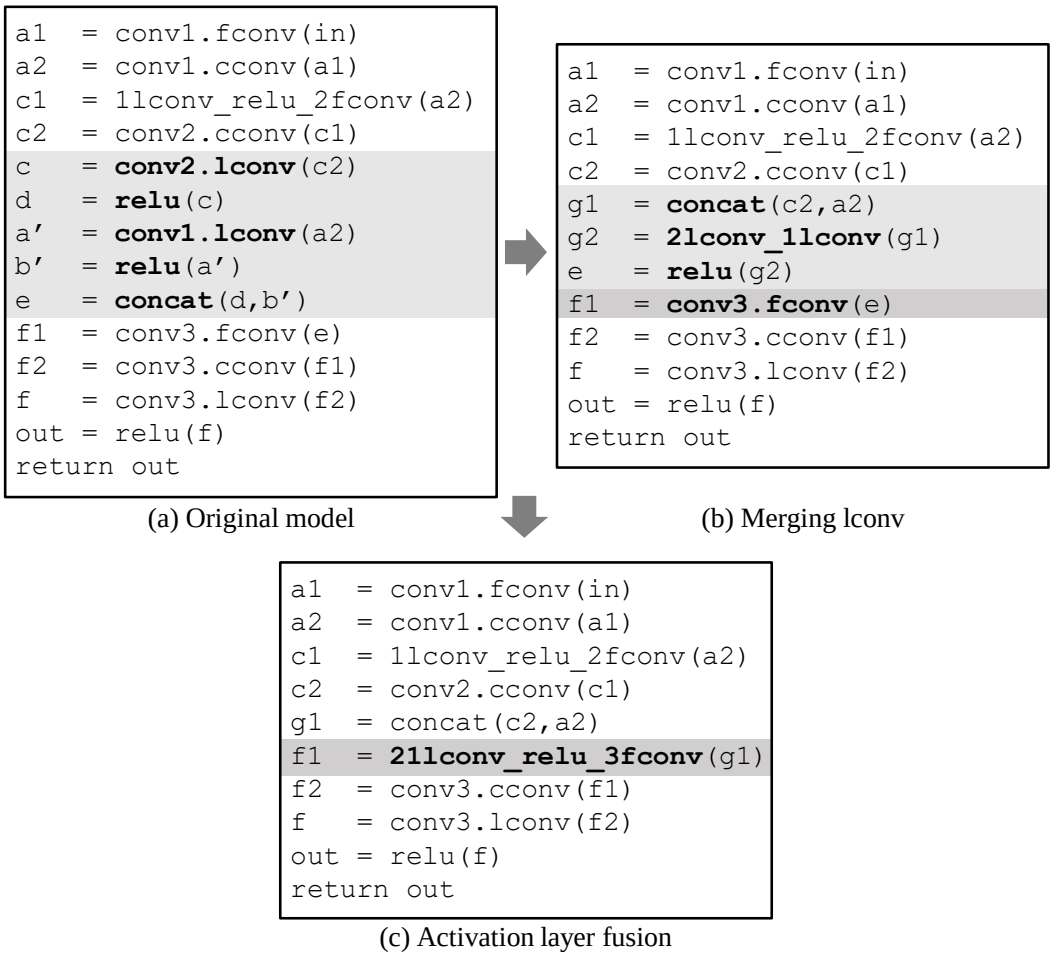


Figure 4.11: Merging *lconv* code example

activation layers. DenseNet has an activation sequence of ReLU - Pool in the mainstream, and other skip connections only have ReLU. Therefore, the compiler divides *fconv* to handle the mainstream and merges *lconv* of skip connections to fuse numerous *lconv* - ReLU - *fconv* sequences. Because DenseNet has all-to-all skip connections inside, reducing the number of layer calls is significant to reduce computation overheads. Therefore, the TeMCO compiler applies the merge of *lconv* to reduce the number of layer calls.

In the application on ResNet [44], the TeMCO compiler transforms the sequence of an add layer (Figure 4.9c) to the merged *lconv* sequence (Figure 4.9a). ResNet merges residual paths into the mainstream with add operation. To reduce the number of layer calls, the compiler transforms the sequence of Figure 4.9a to Figure 4.9c. In this way, ResNet has one fused layer instead of two in the add sequences.

5. Evaluation

5.1 PSDN Compiler

5.1.1 Evaluation Setup

This work evaluates the PSDN compiler's ability to reduce latencies and resource utilization of packet processing through HDL simulation and synthesis using P4 benchmarks from the P4-NetFPGA GitHub repository [76]. The benchmark suite contains seven P4 programs: Learning Switch, In-Network Telemetry (INT), TCP Monitor, Switch Calculator (Switch Calc.), Basic Fair Random Early Detection (Basic FRED), Heavy Hitter, and Flow Rate. Table 5.1 provides descriptions and specifications for each program. Some of these programs feature independent and parallelizable functions, while others have sequential structures for monitoring and calculating packet information. The degree of parallelism or sequential nature in these programs will influence the reduction of latencies achieved by the PSDN compiler.

To measure the end-to-end packet processing latency, this work uses HDL simulation with Vivado 2018.2. Additionally, the compiled programs are synthesized to assess resource utilization. The testing platform is a NetFPGA-SUME board [10] equipped with a Xilinx Vertex-7 FPGA module and four 10 Gbps network ports. This work further evaluates the throughput of the switches synthesized with the programs.

Table 5.1: P4 benchmarks from P4-NetFPGA GitHub [76]

Program	**Description**	**# tables**	**# registers**	**# hashs**	**# timestamps**
Learning Switch	learns forwarding rules from packets	3	0	0	0
INT	collects network states in real-time	1	1	0	1
TCP Monitor	monitors TCP connections	1	2	1	0
Switch Calc.	conducts basic arithmetic operations	1	1	0	0
Basic FRED	drops packets based on queuing lengths	3	4	0	1
Heavy Hitter	finds over-flowed packets	2	3	0	1
Flow Rate	calculates flow rate of packets	2	7	0	1

To demonstrate the effectiveness of the PSDN compiler in reducing packet processing latency, resource utilization, and throughput, this work compares the proposed methods with previous work [39]:

- **Previous work**: applies table-level pipeline scheduling and optimizations.
- **Pipeline scheduling**: applies table decomposition and function-level pipeline scheduling without applying function fusion.
- **Pipeline scheduling** + **Function fusion**: applies table decomposition, function-level pipeline scheduling, and function fusion.

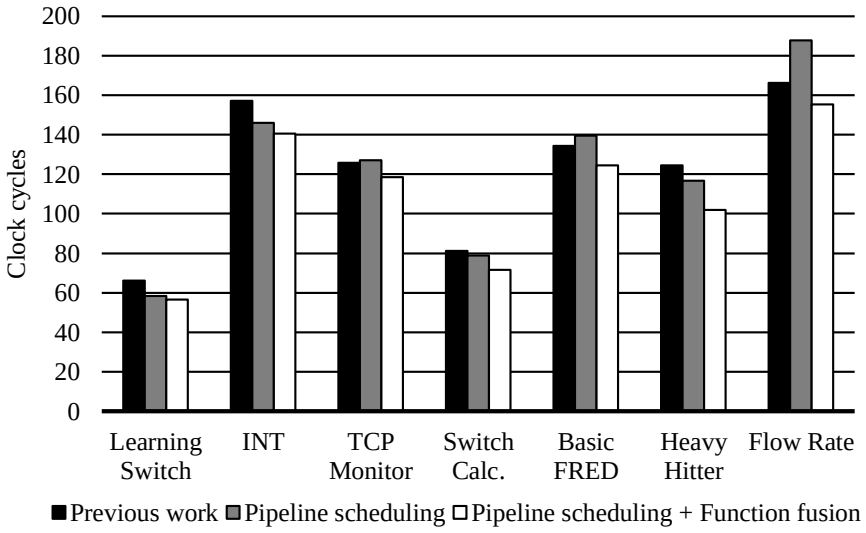


Figure 5.1: Packet processing latency

5.1.2 Latency

This work evaluates the end-to-end packet processing latency of the compiled programs using HDL simulation. Figure 5.1 illustrates the clock cycles required by the programs compiled by each option. Compared to previous work [39], the PSDN compiler achieves a 12.1% reduction in latency, as calculated by the geometric mean.

However, function-level pipeline scheduling alone does not decrease latencies for the TCP Monitor, Basic FRED, and Flow Rate programs compared to the previous work. These three P4 programs contain fewer parallelizable functions than the others. For these programs, decomposing every table into separate functions increases the number of functions and elongates the pipeline.

By employing function fusion, which merges adjacent functions, the compiler reduces the number of action functions and minimizes synchronization overheads. This

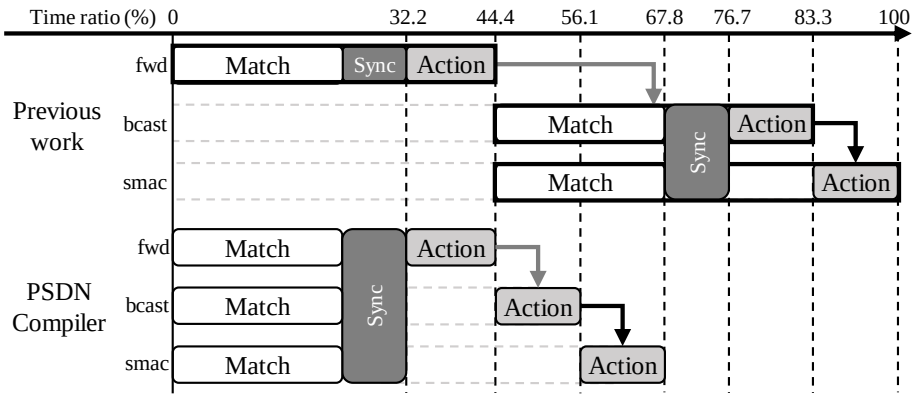


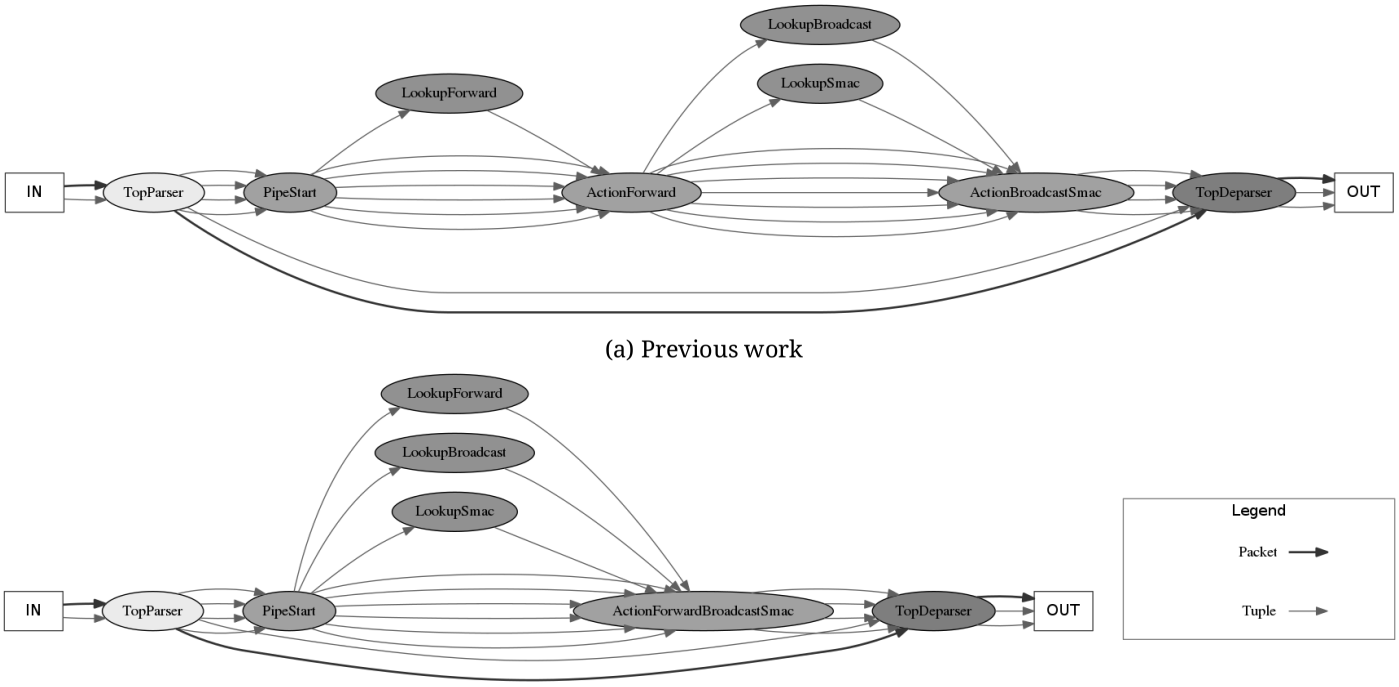
Figure 5.2: Latency of functions in Learning Switch

approach reveals performance improvements by enhancing function-level parallelization, particularly in programs where the initial table decomposition increased complexity without providing significant parallel execution benefits.

To thoroughly analyze how the PSDN compiler effectively reduces latency, this study measures the clock cycles of functions in the table pipeline of the Learning Switch benchmark and compares the results with previous work (Figure 5.2). The Learning Switch program includes three tables: `forward` (`fwd`), `broadcast` (`bcast`), and `smac`. There is a control dependency between the `fwd` and `bcast` tables (grey arrows) and a data dependency between the actions of the `bcast` and `smac` tables (black arrows).

Previous work [39] treats tables as execution units and sequentially allocates the `fwd` and `bcast` tables. Since it only considers match or action dependencies [11], it parallelizes only the match functions of the `bcast` and `smac` tables. Therefore, the previous work loses the opportunity of parallelizing the `fwd` table.

In contrast, the PSDN compiler decomposes the tables into separate match functions and action functions, redirecting control dependencies to action functions. This redirec-



(a) Previous work

(b) PSDN compiler

Figure 5.3: PDG of Learning Switch translated into SDNet IR

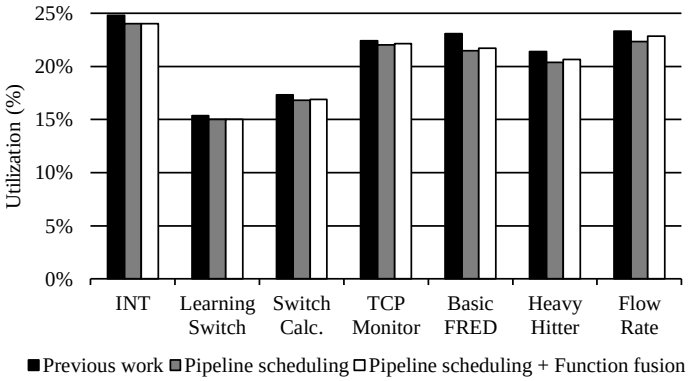
tion allows prefetching of the read-only match functions. Consequently, the PSDN compiler can parallelize all match functions across the three tables, significantly improving performance over the previous methods.

Figure 5.3 shows the actual compilation results of the previous work and the PSDN compiler, including the PDG of SDNet IR. In Figure 5.3a, the match functions (Lookups in SDNet IR) are not parallelized as described in Figure 5.2. However, in Figure 5.3b, all the match functions are parallelized. Table decomposition and prefetching the match functions allow the parallelization opportunity to the PSDN compiler, improving packet processing latency.

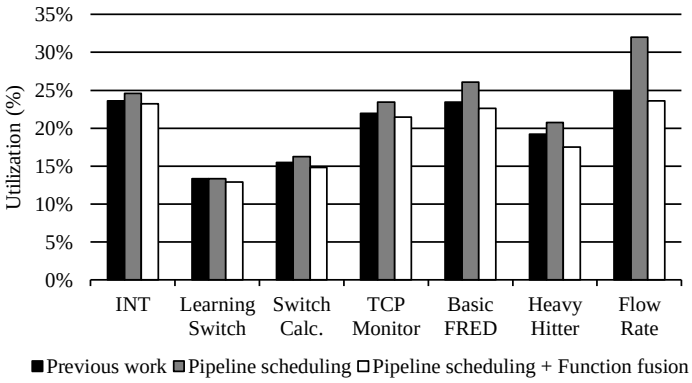
5.1.3 Resource Utilization

This study measures the resource utilization of the synthesized programs in terms of lookup tables (LUTs; representing the number of combinational logics, not packet processing tables), registers, and memory. The synthesized programs are installed on the NetFPGA-SUME board, which is equipped with a Xilinx xc7vx690t FPGA module. Figure 5.4 shows the resource utilization percentages for each unit. Compared to previous work, the PSDN compiler reduces resource usage by 3.5% on average, as measured by the geometric mean.

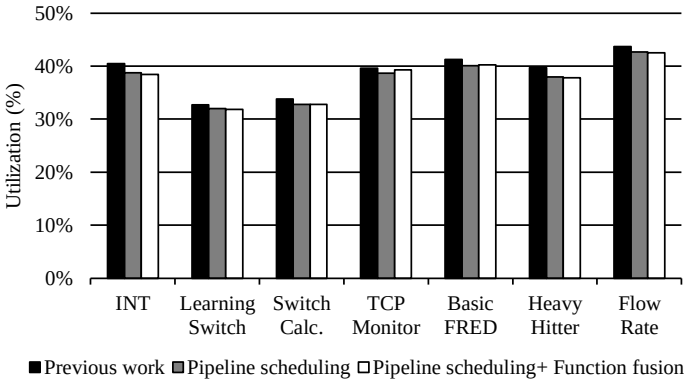
In Figure 5.4a, table decomposition and function-level pipeline scheduling reduce the usage of combinational logic (LUTs) compared to the previous work. Decomposing tables reduces the number of conditional branches by executing separate action functions in parallel, thereby decreasing combinational logic usage. Although the function fusion scheme merges functions and introduces additional conditional branches, result-



(a) LUTs

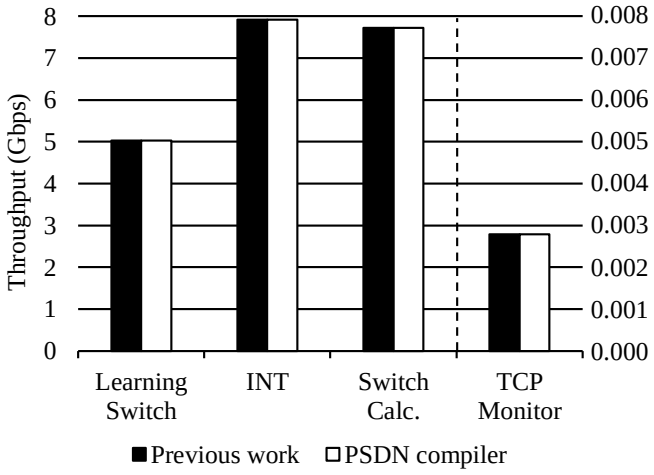


(b) Registers

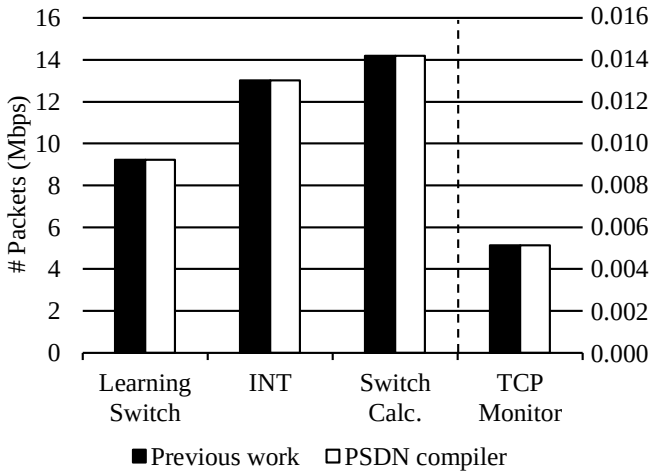


(c) Memory

Figure 5.4: Resource utilization



(a) Throughput



(b) The number of packets

Figure 5.5: Throughput and the number of processed packets

ing in slightly higher LUT usage than without fusion, the PSDN compiler still achieves a 3.07% reduction in LUT usage on average compared to the previous methods.

Regarding register usage (Figure 5.4b), table decomposition initially increases the number of registers used compared to previous work. Each function must transfer data to subsequent functions in the pipeline, which consumes additional registers for data storage. Despite the increase in register usage due to the creation of more functions, applying the function fusion scheme to the decomposed pipeline allows the PSDN compiler to reduce register usage by 4.29% on average compared to the previous work.

In Figure 5.4c, pipeline scheduling reduces memory (BRAM) usage compared to the previous work. Memory utilization is related to the internal synchronization buffers required for match functions and extern functions. By placing independent match and extern functions into the same pipeline stage, pipeline scheduling reduces the need for synchronization buffers. As a result, the PSDN compiler decreases memory resource usage by 3.13% on average compared to the previous methods.

5.1.4 Throughput

Figure 5.5 shows throughput and the number of processed packets of 4 programs compiled by previous work and the PSDN compiler. The results show that the compiled programs by the PSDN compiler have equal performance of throughput to the programs by the previous work. While Figure 5.5 does not evaluate all benchmarks, the throughput is the same in both the previous work and PSDN.

The network provider should install duplicate programs on the switch to increase the throughput of a network switch program. To do this, the size of the network switch

program should be reduced. The PSDN compiler reduces the resource usage of P4 programs, so there are opportunities to increase the throughput of the programs by duplication. This work further evaluates the impacts of reducing resource usage on throughput as future work.

5.2 TeMCO Compiler

5.2.1 Evaluation Setup

This study implements a prototype compiler for TeMCO using PyTorch 2.2 [57] and fused kernels (Section 3.2) with CUDA and evaluates performance on an RTX 4090 GPU. The benchmark set includes image classification models such as AlexNet [42], VGG [43], ResNet [44], and DenseNet [45], as well as the image segmentation model UNet [46]. The ILSVRC 2012 [77] dataset is used for image classification, while the Carvana dataset is used for image segmentation.

This work applies Tucker decomposition [47] to the 10 models with a decomposition ratio of 0.1, using these decomposed models as the baseline, referred to as **Decomposed** in the subsequent graphs. Since AlexNet and VGG do not have skip connections, only activation fusion is applied to these models, which are labeled as **Fusion**. For models with skip connections, such as ResNet, DenseNet, and UNet, both skip connection optimization and activation fusion are applied. The effects on memory usage and latency of skip connection optimization with and without activation fusion are evaluated, denoted as **Skip-Opt** and **Skip-Opt+Fusion**, respectively.

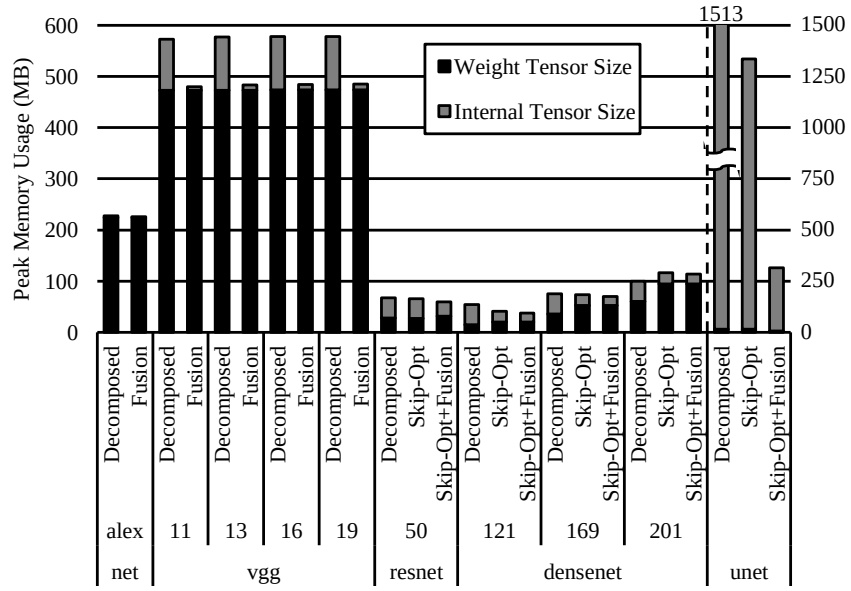
5.2.2 Peak Memory Usage

This study measures the peak memory usage during end-to-end inference for 10 models using 4-batch and 32-batch inference. Figure 5.6 illustrates the peak memory usage by weights and internal tensors. Compared to the original models, this work achieves a 75.7% reduction in memory usage for internal tensors on average.

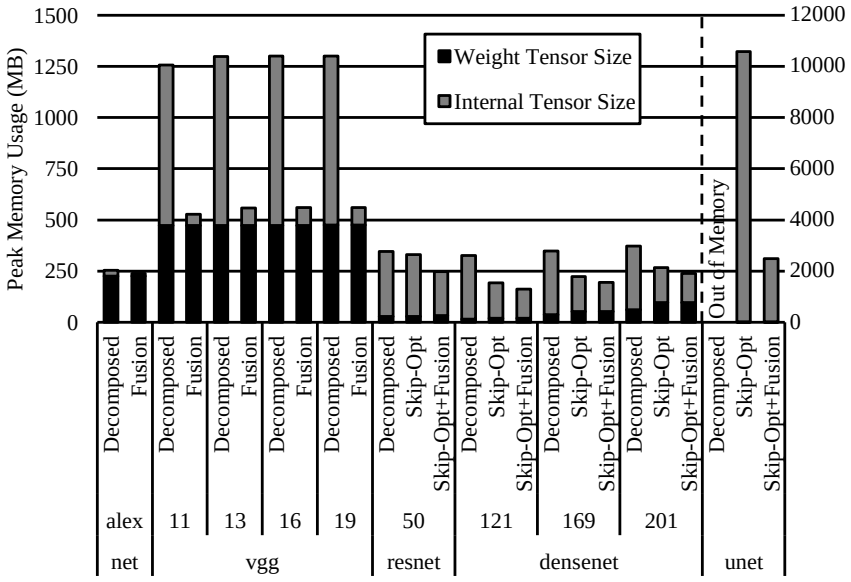
For AlexNet and VGG, the TeMCO optimized models significantly benefit from activation layer fusion, reducing internal tensor memory usage by 49.4% and 90.7%, respectively. Because activation layer fusion skips allocation of internal tensors and uses reduced tensors, the peak memory usage is proportionally reduced with the designated decompression ratio.

In the case of ResNet, the TeMCO compiler reduces the internal tensor memory usage by 30.7%. ResNet's deep skip connections result in a high amount of computation due to replay-dependent restore layers. Skip connection optimization selectively enhances skip connections based on their included operations to mitigate the computation overheads of copying the restore layers.

DenseNet shows a 54.0% reduction in internal tensor size, owing to its numerous skip connections. However, DenseNet's restore operations mainly involve *lconv* and ReLU, which are simpler. While copying the restore layers of the skip connections requires more memory spaces on weight tensors, skip connection optimization, and activation layer fusion reduce the memory usage of internal tensors. Because DenseNet has numerous skip connections, skip connection optimization takes a large portion of reducing internal tensor size.



(a) Batch size 4



(b) Batch size 32

Figure 5.6: Peak memory usage

For UNet, with its hourglass structure and long-lived skip connections, TeMCO reduces internal tensor memory usage by 79.3%. Skip connection optimization in this context involves copying *lconv* layers, and concatenation layer transformation fuses the *lconv*. Although merging *lconv* requires additional memory for weight tensors, it reduces overall peak memory usage by decreasing the internal tensor size of the long-lived skip connections.

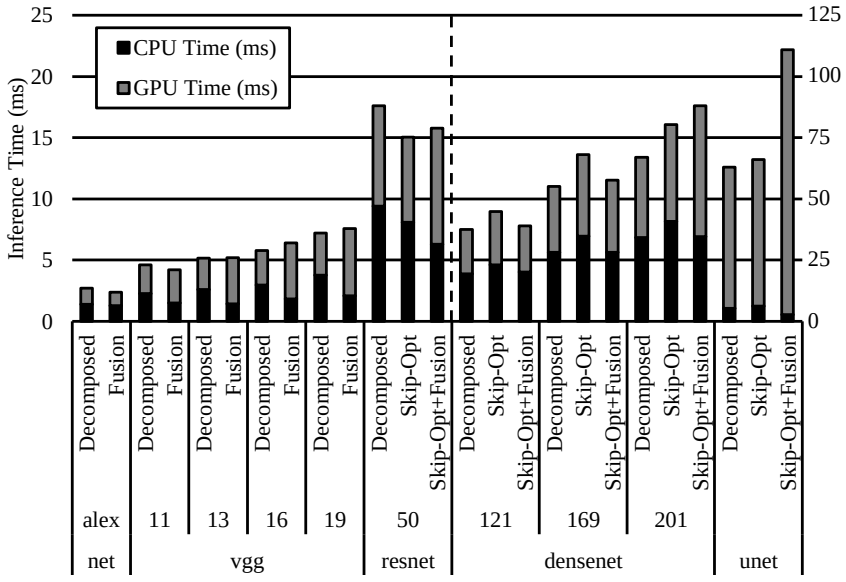
As a result, the TeMCO compiler significantly reduces internal tensor sizes and prevents out-of-memory issues in complex deep-learning architectures.

5.2.3 Inference Time

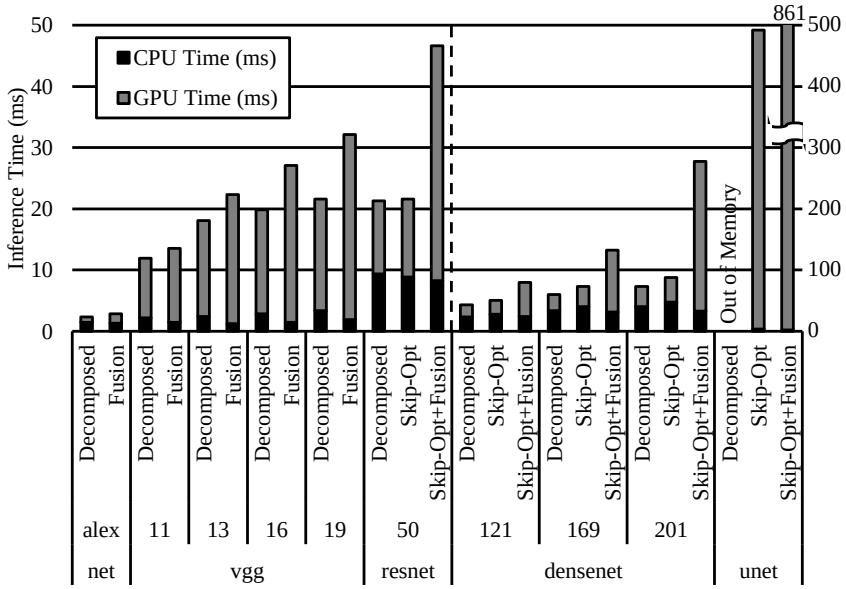
Figure 5.7 illustrates the end-to-end inference time of the 10 models in terms of CPU and GPU time. The CPU time is related to the layer function calls and synchronization overheads, and the GPU time is related to the actual computation of each layer. The inference time of the models optimized by TeMCO is 1.08× and 1.70× longer than the decomposition models in 4-batch and 32-batch inference, respectively.

For AlexNet, VGG, and ResNet, activation layer fusion reduces CPU overheads by decreasing the number of layer calls. However, since the fused layers perform tiled operations for large inputs, GPU overheads increase as the batch size grows. As the model depth increases, the number of fused layers also increases, leading to greater computational overheads. In summary, the GPU computational overheads of fused layers proportionally increase over batch size and model depth.

For DenseNet and UNet, skip connection optimization introduces CPU overheads due to the duplication of restore layers, which increases the number of layer calls. Al-



(a) Batch size 4



(b) Batch size 32

Figure 5.7: End-to-end inference time

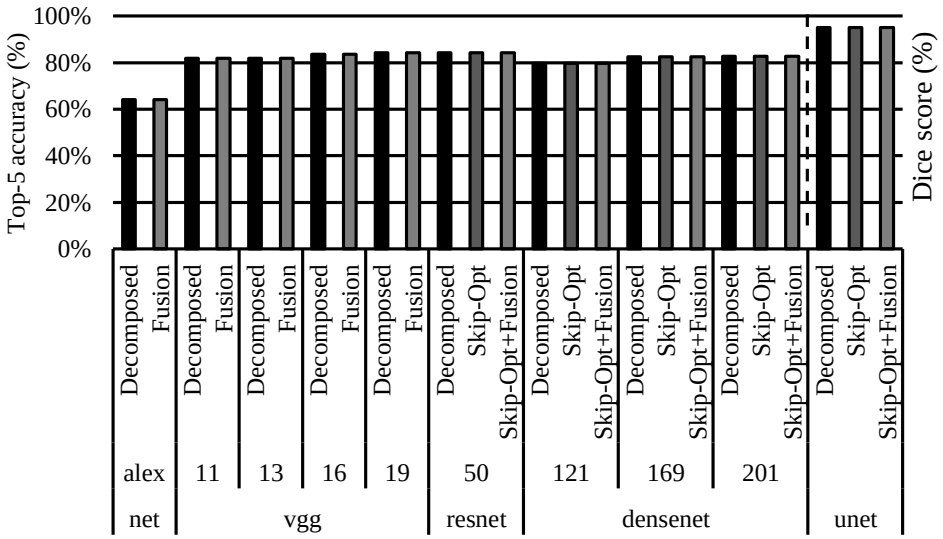


Figure 5.8: Accuracy

though activation layer fusion reduces CPU overheads by minimizing the number of layer calls, GPU overheads rise with larger batch sizes and deeper models because of the added complexity and number of fused layers.

5.2.4 Accuracy

Figure 5.8 presents the top-5 accuracy for AlexNet, VGG, ResNet, and DenseNet, along with the Dice score for UNet. The compiler optimizations implemented by TeMCO maintain the accuracy of the decomposed models because they preserve the original semantics of these models. This evaluation applies decompositions with a decomposition ratio of 0.1, and direct training is performed.

Previous work [15, 17] proposes ADMM training schemes specifically for decomposed models to achieve high accuracy with specified ranks. If a tensor decomposition

Table 5.2: Accuracy with ADMM training [15, 17]

Metrics	Base	ratio=0.5		ratio=0.1	
		Direct	ADMM	Direct	ADMM
Top-1 (%)	93.70	92.37	90.95	81.50	82.61
Top-5 (%)	99.51	99.14	98.99	98.35	98.03
Weight Tensor (MB)	528	492		474	
Internal Tensor (MB)	1568	784		172	
Total (MB)	2096	1276		646	

method offers a pre-trained decomposed model, the TeMCO compiler can effectively reduce the peak memory usage of internal tensors without compromising accuracy. This ability underscores TeMCO's advantage in preserving model integrity while optimizing peak memory usage.

This work implements ADMM training of the previous work and evaluates the accuracy and peak memory usage of the VGG-16 model across different compression ratios, as summarized in Table 5.2. Due to the nature of tensor decomposition requiring re-training to restore original accuracy, this work explores both direct training and ADMM-based training methods using CIFAR-10 datasets.

The findings indicate that higher decomposition ratios yield higher top-1 accuracy, showing a marginal drop in top-5 accuracy of within 1.5% across both ratios. The TeMCO compiler effectively reduces the memory footprint of internal tensors at specified decomposition ratios. The memory usage of weight tensors experiences only a slight decrease, primarily because the weight tensors of fully connected layers remain undecomposed and contribute to the overall model size of VGG. Overall, the TeMCO compiler achieves a substantial reduction in peak memory usage, amounting to 69.2% and 39.1%

with decomposition ratios of 0.1 and 0.5, respectively. This highlights the efficacy of TeMCO in optimizing memory utilization while maintaining competitive accuracy levels under different decomposition ratio settings.

6. Discussion

6.1 PSDN Compiler

The primary objective of this study is to prove the effectiveness of the PSDN compiler in reducing latency and resource usage in network packet processing. This work indicates that the PSDN compiler improves performance by reducing the packet processing latency and resource utilization in comparison to traditional methods. These results have profound implications for the future of high-performance network processing.

The analysis of this work demonstrates that the PSDN compiler achieves an average 12.1% reduction in latency due to its fine-grained pipeline scheduling and function fusion techniques. This improvement aligns with the theoretical expectations that finer-grained optimizations can expose more parallelism within the processing pipeline. Unexpectedly, this work observes that for models with fewer parallelizable functions, such as the TCP Monitor, the performance gains were less pronounced. This suggests that the benefits of the PSDN compiler are more significant in applications with inherently parallel structures.

Compared to previous work, which mainly focused on table-level optimizations, this work highlights the advantages of breaking down functions into smaller, more manageable units. This approach allows for more precise control over the scheduling and exe-

cution of tasks, as evidenced by the reduced latency and resource usage. The findings are consistent with the previous work [38], which also emphasized the importance of fine-grained parallelism in optimizing network processing tasks.

One of the key strengths of this work is the use of the PSDN compiler's novel approach to decomposing and scheduling functions. This method not only optimizes performance but also reduces synchronization overheads, a common bottleneck in network processing systems. Additionally, this work provides a comprehensive analysis of the compiler's impact across a variety of benchmark programs, demonstrating its broad applicability and effectiveness.

Despite the promising results, this work has several limitations. First, the PSDN compiler's performance was not tested on a wide range of hardware architectures, which may limit the generalizability of the findings. Additionally, the compiler's optimization techniques may not be as effective for applications with highly sequential processing requirements. Future research should explore the application of the PSDN compiler across different hardware setups and investigate methods to enhance its performance in less parallelizable contexts.

Future studies will focus on extending the PSDN compiler's capabilities to support more complex network functions and diverse hardware platforms. Extending the PSDN compiler's optimizations to the software pipeline and employing the multiple processes is the start of future work. This extension will make the PSDN compiler be applicable to multi-core CPU-based SmartNICs. Moreover, further investigation into automatically parallelizing the sequential applications or increasing throughputs could yield additional performance improvements.

In conclusion, the PSDN compiler represents a significant advancement in the field of network packet processing, offering substantial reductions in latency and resource usage through its fine-grained optimization techniques. This work underscores the potential of detailed function-level analysis and scheduling to enhance the efficiency of network processing systems. These findings pave the way for future innovations in compiler design and network function optimization.

6.2 TeMCO Compiler

This work aims to prove the TeMCO compiler's effectiveness in optimizing memory usage and processing efficiency for tensor-decomposed deep learning models. This work demonstrates that TeMCO significantly reduces peak memory usage by applying compiler optimization techniques such as skip connection optimization, concatenation layer transformation, and activation layer fusion. These results indicate that TeMCO offers a robust solution for handling complex, memory-intensive neural networks, thereby advancing the capabilities of deep learning deployment on hardware resources.

The evaluation shows that TeMCO reduces internal tensor memory usage by 75.7% on average across ten models during 4-batch inference (Figure 5.6). This reduction is particularly pronounced in models like VGG and UNet, which benefit from TeMCO's ability to fuse activation layers and optimize skip connections, leading to a 90.7% and 79.3% reduction in memory usage, respectively. These results align with expectations that reducing the uses of the internal tensors in non-decomposed activation layers and optimizing the skip connections can effectively minimize the memory footprint of tensor-decomposed deep learning models.

However, for models such as ResNet, which have deeper skip connections, TeMCO achieves a more modest 30.7% reduction in internal tensor memory usage. This is attributed to the extensive computation required to manage the replay-dependent operations inherent in deep-depth skip connections. Similarly, DenseNet, with its numerous but simpler skip connections, saw a 54.0% reduction, showcasing TeMCO's ability to handle varying model architectures effectively.

Compared to previous studies on network optimization and memory management, our work with TeMCO presents a novel approach that integrates activation fusion and skip connection optimization. Previous methods have often focused on either optimizing computational efficiency or managing memory usage but rarely combined these strategies holistically. TeMCO's approach of preemptively addressing memory overheads through layer fusion and efficient handling of skip connections places it in a unique position among contemporary optimization techniques.

The reductions in memory usage observed with TeMCO are consistent with findings from similar studies that advocate for tensor decomposition and layer fusion as effective means to optimize resource usage in deep learning models. For example, techniques employed in AlexNet and VGG, where activation layers are fused, corroborate the significant reductions in memory usage due to fewer active layers at any given time.

One of the key strengths of TeMCO is its ability to maintain the accuracy of tensor-decomposed models while substantially reducing their memory requirements. In Figure 5.8, TeMCO preserves the accuracies of the decomposed models. If the tensor decomposition gives a high-accuracy model with sophisticated training methods [15, 16], TeMCO preserves their accuracy and reduces the peak memory usage.

Additionally, TeMCO's approach to skip connection optimization and activation fusion allows it to be applied broadly across various model architectures, making it a versatile tool in the deep learning optimization toolkit. TeMCO also has concatenation layer transform schemes to apply activation layer fusion on broader structures. This versatility is evident in its consistent performance across models with different structures and levels of complexity.

Despite the significant improvements in memory usage and efficiency, there are limitations to the TeMCO compiler. The increase in GPU overheads for deeper models and larger batch sizes due to the fused kernels performing tiled operations highlights a trade-off between memory savings and computational cost. For models with extensive skip connections or deep architectures, the complexity added by managing replay operations can limit the extent of memory reduction achievable.

Furthermore, while TeMCO effectively reduces memory usage, the impact on real-time performance, such as inference latency, was not comprehensively evaluated in this study. Future work should investigate how these memory optimizations translate into actual runtime improvements in deployment scenarios.

Future research could explore extending TeMCO's capabilities to support dynamic batch sizes and adaptive layer fusion techniques that adjust based on real-time resource availability and workload characteristics. Additionally, integrating machine learning algorithms to predict optimal fusion strategies for different types of models could enhance TeMCO's efficiency and broaden its applicability.

Investigating TeMCO's performance in real-time applications and on a wider range of hardware platforms, including edge devices, would provide deeper insights into its

practical deployment potential. Moreover, exploring the integration of TeMCO with other compiler optimization techniques could yield further reductions in memory usage and computational overheads.

In conclusion, the TeMCO compiler represents a significant advancement in the optimization of deep learning models, offering substantial reductions in memory usage without sacrificing accuracy. By leveraging activation layer fusion and skip connection optimization, TeMCO effectively addresses the challenges posed by memory-intensive neural networks. These findings underscore the potential of TeMCO to enhance the deployment of memory-intensive deep learning models, paving the way for future innovations in compiler design and model optimization.

7. Related Work

7.1 Network Compilers

Compilers and languages for network packet processing [78, 79, 80] provide tools for writing and optimizing packet processing programs. Aspen [78] is a language that supports concurrency in network server applications. Code reuse in SDN programming [79, 80] simplifies network programming by offering reusable building blocks. This study uses the P4 language [2] as a frontend language, but the proposed parallelization method can be applied to any language with packet processing table pipeline semantics.

Previous research targeting RISC-based network processors has proposed compiler optimizations. These optimizations include register allocation [81, 82, 83], bit-level instruction partitioning [84], and resolving bank conflicts [85]. Although this work does not specifically target RISC network processors, it aims to improve the performance of CPU-based multi-core packet processors through parallelization.

Several studies have focused on optimizing packet parsers through compilers [86, 87, 88]. While packet parsers are part of the programmable data plane, their latency is generally shorter than that of the table pipeline, as discussed in Section 2.1.2. This is because parsers mainly read packet header values to identify protocol types, whereas the table pipeline reads and modifies packet headers and metadata. The table pipeline

is the primary bottleneck in the programmable data plane, and this work focuses on optimizing the pipeline to minimize overall latency.

To increase throughput in packet processing applications, researchers have proposed compilers that leverage application partitioning algorithms [89, 90, 91]. These implementations partition imperative packet processing programs into basic blocks and create pipelines to enhance throughput. Similarly, this work proposes table decomposition but specifically exploits the characteristics of match and action functions and employs the compiler's static analysis and optimizations.

Jose et al. [38] have adopted parallelization to optimize the latency of P4 [2] packet processing applications. Their compiler maps P4 logical tables into physical tables on packet processing architectures [11, 13]. This approach uses a table dependency graph to identify data dependencies and employs Integer Linear Programming (ILP) to map logical tables to physical tables. While their compiler schedules at the table level, the compiler in this work decouples match and action functions from tables and schedules them into the pipeline in a fine-grained manner.

P4FPGA [37] is a rapid prototyping compiler that translates P4 programs into Bluespec System Verilog [92, 93] programs. One of its approaches to reducing latency is colocating independent instructions within the same pipeline stage. Although this work uses a similar method in the function fusion scheme, P4FPGA does not address the reordering and mapping of packet processing tables into the pipeline while preserving table-level dependencies. In other words, P4FPGA focuses only on intra-table optimization, where the table is the optimization unit, whereas this work supports both intra- and inter-table optimization, where the matches and actions are the optimization units.

Previous research on programmable ASIC compilers [94, 95, 96] aims to overcome the limitations of chip-specific languages and architectures. Gao et al. [94] use domain-specific synthesis techniques to accelerate compilation and target multiple backends through a pipeline description language. To support hardware-independent programming, μP4 [95] increases the level of abstraction in target-specific packet processing pipelines and configurations. Lyra [96] provides a "one-big-pipeline" abstraction, allowing programmers to express their algorithms conveniently and generate chip-specific code for multi-vendor switches.

Recent publication [97] shows the automatic parallelization of network functions. This publication proposes a scheme that distributes and parallelizes a network program to multiple cores. While the previous work targets the parallelization of a program, this work focuses on the parallelization of functional units within the program. Furthermore, the previous work uses CPU-based network virtualization, but this work synthesizes the optimized network program into FPGA-based network switches.

7.2 Tensor Decomposition

Many tensor decomposition methods applied to deep learning models aim to achieve speedups and reduce computational costs while ensuring that any resulting accuracy drops remain within acceptable tolerances of the models. Previous research has explored techniques such as rank-1 expansion [98], SVD [99, 100], CP decomposition [101], and Tucker decomposition [15, 16]. Additionally, several studies [102, 103] have combined these decomposition methods to compress and accelerate convolution neural network (CNN) models.

Tensor-Train (TT) [56, 104, 105] and Tensor-Ring (TR) [106] decomposition are methods that decompose high-dimensional tensors into sequences of low-dimensional tensors. Compared to the previous decomposition, TT and TR decomposition achieve higher model compression by using low-dimensional tensors. For example, TT decomposition constructs 3D weight tensors for core convolution layers, while Tucker decomposition generates a 4D weight tensor for a core convolution layer. Additionally, previous studies [107, 108] have proposed deep learning accelerators tailored for these specific decomposition methods.

Ideally, TeMCO's optimization can be applied to the models with various tensor decomposition methods [47, 56, 101] that decompose a convolution with two factor matrix convolutions: *fconv* and *lconv*. As described in Section 2.2.1, TeMCO's optimization deals with *fconv* and *lconv*, and it does not care about the type of core convolutions. Therefore, if the decomposed convolution sequence has *fconv* and *lconv*, such as CP, Tucker, and TT decomposition, TeMCO's optimization can be applied to reduce the peak memory usage of internal tensors.

Previous tensor decomposition-based model compression schemes [15, 104] focus on reducing the FLOPS of a deep learning model but not reducing the memory usage. The previous work focuses on selecting appropriate ranks to satisfy higher accuracy and lower FLOPS. On the other hand, this work proposes compiler optimizations to reduce memory usage of the tensor-decomposed models. The TeMCO compiler can accept the decomposed models generated by the previous work to retarget from reducing FLOPS to reducing peak memory usage.

7.3 DNN Framework for Memory-Efficient Deep Learning

Previous studies [109, 110, 111, 112, 113] propose approaches aimed at reducing memory usage in deep learning devices through kernel division techniques. These methods involve partitioning layers into iterative sub-operations and accumulating their outcomes into unified results. By doing so, these techniques effectively address the constraints posed by limited scratchpad memory on accelerators.

The kernel division techniques resemble activation layer fusion in tiling. The fused layer tiles the operations of *lconv* - ReLU - *fconv* and uses small tiled memory space instead of internal tensor space. While this work implements the fused layer in GPU, it also aims to implement fused layers in CPU and accelerators in future work.

Earlier researches [69, 71, 72, 73] propose layer scheduling techniques aimed at minimizing memory usage. These researches show that execution orders of layers affect the memory usage of internal tensors. With their observations, this work implements scheduling of restore layers in Section 4.3. While finding the optimum scheduling of minimum memory usage is an open problem, this work can improve the scheduling algorithms by applying other heuristics.

On the point of TinyML, where memory constraints present a substantial bottleneck, quantization-based optimization [72, 114, 115] stands out as a model compression technique that reduces the bit-width of the model at the expense of accuracy. Quantization is effective in reducing both memory usage and computational overhead. However, despite these optimizations, the memory consumption attributable to internal tensors remains significant. Because tensor decomposition can be applied orthogonally on quan-

tized models, tensor decomposition plus quantization can reduce the memory usage by internal tensors.

In the domain of deep learning training, internal tensor compression has also been a subject of discussion. Previous work [116, 117] introduces compressed training schemes to compress internal tensors between inference and weight updates. However, the compressed tensors should be decompressed to proceed with further layer operations. This means the memory usage can be reduced when the tensor is compressed, but the memory usage will increase when the tensor gets decompressed. Compared to the previous work, the TeMCO compiler achieves the computations with reduced tensors by activation layer fusion, not to restore these reduced tensors to the corresponding internal tensors. Therefore, the TeMCO compiler's approach can significantly reduce the peak memory usage of internal tensors.

8. Conclusion

This work proposes a split-schedule-merge scheme for domain-specific programs and introduces two prototype compilers named PSDN and TeMCO that analyze the abstracted domain-specific programs in a fine granularity and optimize the programs with fine-grained functional units to achieve their purposes. The PSDN compiler splits match-action tables into match and action functions and performs fine-grained scheduling and merging of these functions to reduce packet processing latency. The TeMCO compiler splits decomposed convolution sequences, extracts *fconv* and *lconv* of decomposed deep learning models, schedules the execution order of the layers in skip connections, and merges the layers to reduce peak memory usage of internal tensors.

The PSDN compiler decouples match functions and action functions and performs dependency analysis on these functions. With the program dependence graph, the PSDN compiler schedules the execution of the functions, placing the functions that have no dependencies on each other in parallel. Then, the PSDN compiler fuses action functions to minimize synchronization overheads. Compared to previous work, the PSDN compiler reduces packet processing latency by 12.1% and resource usage by 3.5%.

The TeMCO compiler optimizes tensor-decomposed deep learning models with skip connection optimization and activation layer fusion. To achieve skip connection optimization, the TeMCO compiler analyzes the program dependence graph of a model and

finds a sub-graph of restore layers. The TeMCO compiler optimizes the execution order of the sub-graph to reduce memory usage and checks computation and memory overheads. If the overheads are not significant, the TeMCO compiler copies the restore layers and replaces the skip connection with the corresponding reduced tensor. In terms of activation layer fusion, the TeMCO compiler fuses sequences of *lconv* - activation - *fconv*, not to allocate internal tensors used in activation layers. To apply activation layer fusion in broader structures, the TeMCO compiler applies concatenation layer transformation and generates the sequences of *lconv* - activation - *fconv*. Compared to the decomposed models, the model optimized by TeMCO uses less internal tensor memory usage by 75.7% with inference time overhead of between 1.08× and 1.70× across 4 to 32 batch sizes.

The proposed PSDN and TeMCO compiler utilizes hidden opportunities of abstracted and coarse-grained domain-specific programs with fine-grained compiler optimizations that use split-schedule-merge schemes. This work successfully reduces computational and memory overheads in network programming and deep learning fields through compiler optimizations. This work will extend the split-schedule-merge scheme to other specialized domains like large language models or high-performance computing. This work will contribute to server-side high-performance computing using GPUs and SmartNICs, as well as domain-specific programs with divisible abstractions.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[2] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-independent Packet Processors," *SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[3] M. Gouda and X.-Y. Liu, "Firewall Design: Consistency, Completeness, and Compactness," in *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, 2004, pp. 320–327.

[4] R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," *Commun. ACM*, vol. 19, no. 7, pp. 395–404, Jul. 1976.

[5] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable High Speed IP Routing Lookups," in *Proceedings of the ACM SIGCOMM '97 Conference on Appli-*

cations, Technologies, Architectures, and Protocols for Computer Communication, ser. SIGCOMM '97, Cannes, France: Association for Computing Machinery, 1997, pp. 25–36, ISBN: 0-89791-905-X.

[6] C. E. Hopps, "Analysis of an Equal-Cost Multi-Path Algorithm," Internet Engineering Task Force, RFC 2992, Nov. 2000.

[7] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford, "PISCES: A Programmable, Protocol-Independent Software Switch," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16, Florianopolis, Brazil: ACM, 2016, pp. 525–538, ISBN: 978-1-4503-4193-6.

[8] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18, Heraklion, Greece: ACM, 2018, pp. 54–66, ISBN: 978-1-4503-6080-7.

[9] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The Design and Implementation of Open vSwitch," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Oakland, CA, May 2015, pp. 117–130, ISBN: 978-1-931971-218.

[10] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, “NetFPGA SUME: Toward 100 Gbps as Research Commodity,” *IEEE Micro*, vol. 34, no. 5, pp. 32–41, Sep. 2014.

[11] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, “Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN,” in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM ’13, Hong Kong, China: ACM, 2013, pp. 99–110, ISBN: 978-1-4503-2056-6.

[12] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall, “dRMT: Disaggregated Programmable Switching,” in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM ’17, Los Angeles, CA, USA: ACM, 2017, pp. 1–14, ISBN: 978-1-4503-4653-5.

[13] R. Ozdag, “Intel® Ethernet Switch FM6000 Series-Software Defined Networking,” *Intel Cooperation*, 2012.

[14] H. Kim, M. U. K. Khan, and C.-M. Kyung, “Efficient Neural Network Compression,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2019.

[15] L. Xiang, M. Yin, C. Zhang, A. Sukumaran-Rajam, P. Sadayappan, B. Yuan, and D. Tao, “TDC: Towards Extremely Efficient CNNs on GPUs via Hardware-Aware Tucker Decomposition,” in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’23, Montreal, QC, Canada: Association for Computing Machinery, 2023, pp. 260–273, ISBN: 979-8-4007-0015-6.

[16] M. Yin, H. Phan, X. Zang, S. Liao, and B. Yuan, “BATUDE: Budget-Aware Neural Network Compression Based on Tucker Decomposition,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 8, pp. 8874–8882, Jun. 2022.

[17] M. Yin, Y. Sui, S. Liao, and B. Yuan, “Towards Efficient Tensor Decomposition-Based DNN Model Compression With Optimization Framework,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2021, pp. 10 674–10 683.

[18] Y. Yang, D. Krompass, and V. Tresp, “Tensor-Train Recurrent Neural Networks for Video Classification,” in *Proceedings of the 34th International Conference on Machine Learning*, D. Precup and Y. W. Teh, Eds., ser. Proceedings of Machine Learning Research, vol. 70, PMLR, Aug. 2017, pp. 3891–3900.

[19] Y. Pan, J. Xu, M. Wang, J. Ye, F. Wang, K. Bai, and Z. Xu, "Compressing Recurrent Neural Networks with Tensor Ring for Action Recognition," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, pp. 4683–4690, Jul. 2019.

[20] Y. He, X. Zhang, and J. Sun, "Channel Pruning for Accelerating Very Deep Neural Networks," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct. 2017.

[21] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning Filters for Efficient ConvNets," in *International Conference on Learning Representations*, 2017.

[22] Z. Huang and N. Wang, "Data-Driven Sparse Structure Selection for Deep Neural Networks," in *Proceedings of the European Conference on Computer Vision (ECCV)*, Sep. 2018.

[23] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning Efficient Convolutional Networks Through Network Slimming," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct. 2017.

[24] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the Value of Network Pruning," in *International Conference on Learning Representations*, 2019.

[25] J.-H. Luo, J. Wu, and W. Lin, “ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression,” in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct. 2017.

[26] Y. Zhou, S.-M. Moosavi-Dezfooli, N.-M. Cheung, and P. Frossard, “Adaptive Quantization for Deep Neural Network,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, Apr. 2018.

[27] J. Yang, X. Shen, J. Xing, X. Tian, H. Li, B. Deng, J. Huang, and X.-s. Hua, “Quantization Networks,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2019.

[28] C. Zhu, S. Han, H. Mao, and W. J. Dally, “Trained Ternary Quantization,” in *International Conference on Learning Representations*, 2017.

[29] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2018.

[30] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, “HAQ: Hardware-Aware Automated Quantization With Mixed Precision,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2019.

[31] D. Lin, S. Talathi, and S. Annapureddy, “Fixed Point Quantization of Deep Convolutional Networks,” in *Proceedings of The 33rd International Conference on Machine Learning*, M. F. Balcan and K. Q. Weinberger, Eds., ser. Proceedings of Machine Learning Research, vol. 48, New York, New York, USA: PMLR, Jun. 2016, pp. 2849–2858.

[32] J. H. Cho and B. Hariharan, “On the Efficacy of Knowledge Distillation,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, Oct. 2019.

[33] F. Tung and G. Mori, “Similarity-Preserving Knowledge Distillation,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, Oct. 2019.

[34] B. Zhao, Q. Cui, R. Song, Y. Qiu, and J. Liang, “Decoupled Knowledge Distillation,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2022, pp. 11 953–11 962.

[35] S. I. Mirzadeh, M. Farajtabar, A. Li, N. Levine, A. Matsukawa, and H. Ghasemzadeh, “Improved Knowledge Distillation via Teacher Assistant,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 04, pp. 5191–5198, Apr. 2020.

[36] B. Minnehan and A. Savakis, “Cascaded Projection: End-To-End Network Compression and Acceleration,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2019.

[37] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon, “P4FPGA: A Rapid Prototyping Framework for P4,” in *Proceedings of the Symposium on SDN Research*, ser. SOSR ’17, Santa Clara, CA, USA: ACM, 2017, pp. 122–135, ISBN: 978-1-4503-4947-5.

[38] L. Jose, L. Yan, G. Varghese, and N. McKeown, “Compiling Packet Programs to Reconfigurable Switches,” in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Oakland, CA: USENIX Association, May 2015, pp. 103–115, ISBN: 978-1-931971-218.

[39] Xilinx, *P4-SDNet User Guide*, `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1252-p4-sdnet.pdf`, 2018.

[40] G. Brebner and W. Jiang, “High-Speed Packet Processing using Reconfigurable Computing,” *IEEE Micro*, vol. 34, no. 1, pp. 8–18, Jan. 2014.

[41] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, “The P4->NetFPGA Workflow for Line-Rate Packet Processing,” in *Proceedings of the 2019 ACM/SIGDA In-*

ternational Symposium on Field - Programmable Gate Arrays, ser. FPGA '19, Seaside, CA, USA: ACM, 2019, pp. 1–9, ISBN: 978-1-4503-6137-8.

[42] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25, Curran Associates, Inc., 2012.

[43] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015.

[44] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016.

[45] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul. 2017.

[46] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," in *Medical Image Computing and Computer-*

Assisted Intervention – MICCAI 2015, N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, Eds., Cham: Springer International Publishing, 2015, pp. 234–241, ISBN: 978-3-319-24574-4.

[47] L. R. Tucker, "Some mathematical notes on three-mode factor analysis," *Psychometrika*, vol. 31, no. 3, pp. 279–311, 1966.

[48] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "ONOS: Towards an Open, Distributed SDN OS," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14, Chicago, Illinois, USA: ACM, 2014, pp. 1–6, ISBN: 978-1-4503-2989-7.

[49] J. Hyun, N. Van Tu, and J. W.-K. Hong, "Towards Knowledge-Defined Networking using In-band Network Telemetry," in *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, Apr. 2018, pp. 1–7.

[50] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17, Los Angeles, CA, USA: ACM, 2017, pp. 15–28, ISBN: 978-1-4503-4653-5.

[51] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, “In-Network Computation is a Dumb Idea Whose Time Has Come,” in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XVI, Palo Alto, CA, USA: ACM, 2017, pp. 150–156, ISBN: 978-1-4503-5569-8.

[52] H. Song, “Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane,” in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN ’13, Hong Kong, China: ACM, 2013, pp. 127–132, ISBN: 978-1-4503-2178-5.

[53] S. Li, D. Hu, W. Fang, S. Ma, C. Chen, H. Huang, and Z. Zhu, “Protocol Oblivious Forwarding (POF): Software-Defined Networking with Enhanced Programmability,” *IEEE Network*, vol. 31, no. 2, pp. 58–66, Mar. 2017.

[54] H. T. Dang, H. Wang, T. Jepsen, G. Brebner, C. Kim, J. Rexford, R. Soulé, and H. Weatherspoon, “Whippersnapper: A P4 Language Benchmark Suite,” in *Proceedings of the Symposium on SDN Research*, ser. SOSR ’17, Santa Clara, CA, USA: ACM, 2017, pp. 95–101, ISBN: 978-1-4503-4947-5.

[55] F. L. Hitchcock, “The Expression of a Tensor or a Polyadic as a Sum of Products,” *Journal of Mathematics and Physics*, vol. 6, no. 1-4, pp. 164–189, 1927.

[56] I. V. Oseledets, “Tensor-Train Decomposition,” *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2295–2317, 2011.

[57] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “PyTorch: an imperative style, high-performance deep learning library,” in *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, Red Hook, NY, USA: Curran Associates Inc., 2019, pp. 8026–8037.

[58] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: A System for Large-Scale Machine Learning,” in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA: USENIX Association, Nov. 2016, pp. 265–283, ISBN: 978-1-931971-33-1.

[59] Barefoot, *P4_16 reference compiler*, `https://github.com/p4lang/p4c`, 2020.

[60] Xilinx, *SDNet Packet Processor User Guide*, `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1012-sdnet-packet-processor.pdf`, 2018.

[61] R. Cytron, J. Ferrante, and V. Sarkar, “Compact Representations for Control Dependence,” in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, ser. PLDI ’90, White Plains, New York, USA: ACM, 1990, pp. 337–351, ISBN: 0-89791-364-7.

[62] W. Pugh, “The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis,” in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing ’91, Albuquerque, New Mexico, USA: ACM, 1991, pp. 4–13, ISBN: 0-89791-459-7.

[63] Xilinx, *Exact Match Binary CAM Search IP for SDNet*, https://www.xilinx.com/support/documentation/ip_documentation/cam/pg189-cam.pdf, 2019.

[64] Xilinx, *Ternary Content Addressable Memory (TCAM) Search IP for SDNet*, https://www.xilinx.com/support/documentation/ip_documentation/tcam/pg190-tcam.pdf, 2017.

[65] Xilinx, *Longest Prefix Match (LPM) Search IP for SDNet*, https://www.xilinx.com/support/documentation/ip_documentation/lpm/pg191-lpm.pdf, 2017.

[66] NVIDIA, *CUDA*, https://developer.nvidia.com/cuda-toolkit, 2024.

[67] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation," in *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '21, Virtual Event, Republic of Korea: IEEE Press, 2021, pp. 2–14, ISBN: 9781728186139.

[68] J. Reed, Z. DeVito, H. He, A. Ussery, and J. Ansel, "Torch.fx: Practical program capture and transformation for deep learning in python," in *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu, Eds., vol. 4, 2022, pp. 638–651.

[69] J. Lee, S. Jeong, S. Song, K. Kim, H. Choi, Y. Kim, and H. Kim, "Occamy: Memory-efficient GPU Compiler for DNN Inference," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, IEEE, 2023, pp. 1–6.

[70] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006, p. 608, ISBN: 0321486811.

[71] L. Zhu, L. Hu, J. Lin, W.-M. Chen, W.-C. Wang, C. Gan, and S. Han, "PockEngine: Sparse and Efficient Fine-tuning in a Pocket," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 1381–1394.

[72] E. Liberis and N. D. Lane, *Pex: Memory-efficient Microcontroller Deep Learning through Partial Execution*, 2023. arXiv: 2211.17246 [cs.LG].

[73] Y. Pisarchyk and J. Lee, *Efficient Memory Management for Deep Neural Net Inference*, 2020. arXiv: 2001.03288 [cs.LG].

[74] S. Elfwing, E. Uchibe, and K. Doya, "Sigmoid-weighted linear units for neural network function approximation in reinforcement learning," *Neural Networks*, vol. 107, pp. 3–11, 2018, Special issue on deep reinforcement learning.

[75] D. Hendrycks and K. Gimpel, *Gaussian Error Linear Units (GELUs)*, 2023. arXiv: 1606.08415 [cs.LG].

[76] N. G. Organization, *P4-NetFPGA-public*, https://github.com/NetFPGA/P4-NetFPGA-public, 2018.

[77] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, pp. 211–252, 2015.

[78] G. Upadhyaya, V. S. Pai, and S. P. Midkiff, "Expressing and Exploiting Concurrency in Networked Applications with Aspen," in *Proceedings of the 12th*

ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ser. PPoPP '07, San Jose, California, USA: ACM, 2007, pp. 13–23, ISBN: 978-1-59593-602-8.

[79] H. Eran, L. Zeno, Z. István, and M. Silberstein, "Design Patterns for Code Reuse in HLS Packet Processing Pipelines," in *2019 IEEE 27th Annual International Symposium on Field - Programmable Custom Computing Machines (FCCM)*, Apr. 2019, pp. 208–217.

[80] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak, "Maple: Simplifying SDN Programming Using Algorithmic Policies," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13, Hong Kong, China: ACM, 2013, pp. 87–98, ISBN: 978-1-4503-2056-6.

[81] J. Wagner and R. Leupers, "C Compiler Design for an Industrial Network Processor," in *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, ser. LCTES '01, Snow Bird, Utah, USA: ACM, 2001, pp. 155–164, ISBN: 1-58113-425-8.

[82] J. Kim, S. Jung, Y. Paek, and G.-R. Uh, "Experience with a Retargetable Compiler for a Commercial Network Processor," in *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES '02, Grenoble, France: ACM, 2002, pp. 178–187, ISBN: 1-58113-575-0.

[83] L. George and M. Blume, “Taming the IXP Network Processor,” in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI ’03, San Diego, California, USA: ACM, 2003, pp. 26–37, ISBN: 1-58113-662-5.

[84] S. Carr and P. Sweany, “Automatic Data Partitioning for the Agere Payload Plus Network Processor,” in *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES ’04, Washington DC, USA: ACM, 2004, pp. 238–247, ISBN: 1-58113-890-3.

[85] X. Zhuang and S. Pande, “Resolving register bank conflicts for a network processor,” in *2003 12th International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2003, pp. 269–278.

[86] J. Santiago da Silva, F.-R. Boyer, and J. P. Langlois, “P4-Compatible High-Level Synthesis of Low Latency 100 Gb/s Streaming Packet Parsers in FPGAs,” in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field - Programmable Gate Arrays*, ser. FPGA ’18, Monterey, CALIFORNIA, USA: ACM, 2018, pp. 147–152, ISBN: 978-1-4503-5614-5.

[87] P. Benácek, V. Pu, and H. Kubátová, “P4-to-VHDL: Automatic Generation of 100 Gbps Packet Parsers,” in *2016 IEEE 24th Annual International Symposium on*

Field-Programmable Custom Computing Machines (FCCM), May 2016, pp. 148–155.

[88] A. Yazdinejad, A. Bohlooli, and K. Jamshidi, "P4 to SDNet: Automatic Generation of an Efficient Protocol-Independent Packet Parser on Reconfigurable Hardware," in *2018 8th International Conference on Computer and Knowledge Engineering (ICCKE)*, Oct. 2018, pp. 159–164.

[89] J. Dai, B. Huang, L. Li, and L. Harrison, "Automatically Partitioning Packet Processing Applications for Pipelined Architectures," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05, Chicago, IL, USA: ACM, 2005, pp. 237–248, ISBN: 1-59593-056-6.

[90] M. K. Chen, X. F. Li, R. Lian, J. H. Lin, L. Liu, T. Liu, and R. Ju, "Shangri-La: Achieving High Performance from Compiled Network Applications While Enabling Ease of Programming," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05, Chicago, IL, USA: ACM, 2005, pp. 224–236, ISBN: 1-59593-056-6.

[91] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking, "Packet Transactions: High-Level Programming for Line-Rate Switches," in *Proceedings of the 2016 ACM SIGCOMM*

Conference, ser. SIGCOMM '16, Florianopolis, Brazil: ACM, 2016, pp. 15–28, ISBN: 978-1-4503-4193-6.

[92] R. Nikhil, "Bluespec System Verilog: efficient, correct RTL from high level specifications," in *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.*, Jun. 2004, pp. 69–70.

[93] R. S. Nikhil, "Bluespec: A General-Purpose Approach to High-Level Synthesis Based on Parallel Atomic Transactions," in *High-Level Synthesis: From Algorithm to Digital Circuit*, P. Coussy and A. Morawiec, Eds. Dordrecht: Springer Netherlands, 2008, pp. 129–146, ISBN: 978-1-4020-8588-8.

[94] X. Gao, T. Kim, M. D. Wong, D. Raghunathan, A. K. Varma, P. G. Kannan, A. Sivaraman, S. Narayana, and A. Gupta, "Switch Code Generation Using Program Synthesis," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 44–61, ISBN: 978-1-4503-7955-7.

[95] H. Soni, M. Rifai, P. Kumar, R. Doenges, and N. Foster, "Composing Dataplane Programs with μP4," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Archi-*

tectures, and Protocols for Computer Communication, ser. SIGCOMM '20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 329–343, ISBN: 978-1-4503-7955-7.

[96] J. Gao, E. Zhai, H. H. Liu, R. Miao, Y. Zhou, B. Tian, C. Sun, D. Cai, M. Zhang, and M. Yu, "Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 435–450, ISBN: 978-1-4503-7955-7.

[97] F. Pereira, F. M. Ramos, and L. Pedrosa, "Automatic Parallelization of Software Network Functions," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, Santa Clara, CA: USENIX Association, Apr. 2024, pp. 1531–1550, ISBN: 978-1-939133-39-7.

[98] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Speeding up Convolutional Neural Networks with Low Rank Expansions," in *Proceedings of the British Machine Vision Conference 2014*, ser. BMVC '24, British Machine Vision Association, 2014, pp. 1–13.

[99] C. Tai, T. Xiao, Y. Zhang, X. Wang, and W. E, "Convolutional Neural Networks With Low-rank Regularization," in *International Conference on Learning Representations*, 2016.

[100] L. Liebenwein, A. Maalouf, D. Feldman, and D. Rus, "Compressing Neural Networks: Towards Determining the Optimal Layer-wise Decomposition," vol. 34, 2021, pp. 5328–5344.

[101] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky, "Speeding-up Convolutional Neural Networks Using Fine-tuned CP-Decomposition," in *International Conference on Learning Representations*, 2015.

[102] C.-C. Liang and C.-R. Lee, "Automatic Selection of Tensor Decomposition for Compressing Convolutional Neural Networks A Case Study on VGG-type Networks," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2021, pp. 770–778.

[103] G. Wu, S. Wang, and L. Liu, "Fast Video Summary Generation Based On Low Rank Tensor Decomposition," *IEEE Access*, vol. 9, pp. 127 917–127 926, 2021.

[104] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov, "Tensorizing Neural Networks," in *Advances in Neural Information Processing Systems*, C. Cortes, N.

Lawrence, D. Lee, M. Sugiyama, and R. Garnett, Eds., vol. 28, Curran Associates, Inc., 2015.

[105] F.-H. Meng, Y. Wu, Z. Zhang, and W. D. Lu, "TT-CIM: Tensor Train Decomposition for Neural Network in RRAM-Based Compute-in-Memory Systems," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 71, no. 3, pp. 1172–1183, 2024.

[106] L. Yuan, C. Li, D. Mandic, J. Cao, and Q. Zhao, "Tensor Ring Decomposition with Rank Minimization on Latent Space: An Efficient Approach for Tensor Completion," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, pp. 9151–9158, Jul. 2019.

[107] J.-F. Zhang, C.-H. Lu, and Z. Zhang, "TetriX: Flexible Architecture and Optimal Mapping for Tensorized Neural Network Processing," *IEEE Transactions on Computers*, pp. 1–13, 2024.

[108] Y. Gong, M. Yin, L. Huang, J. Xiao, Y. Sui, C. Deng, and B. Yuan, "ETTE: Efficient Tensor-Train-based Computing Engine for Deep Neural Networks," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23, Orlando, FL, USA: Association for Computing Machinery, 2023, ISBN: 979-8-4007-0095-8.

[109] A. Artemev, Y. An, T. Roeder, and M. van der Wilk, “Memory Safe Computations with XLA Compiler,” in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35, Curran Associates, Inc., 2022, pp. 18 970–18 982.

[110] T. W. Chris Leary, “XLA: TensorFlow, Compiled!” *TensorFlow Dev Summit*, 2017.

[111] J. Lin, W.-M. Chen, H. Cai, C. Gan, and S. Han, “MCUNetV2: Memory-Efficient Patch-based Inference for Tiny Deep Learning,” in *Advances in Neural Information Processing Systems*, 2021.

[112] H.-S. Zheng, Y.-Y. Liu, C.-F. Hsu, and T. T. Yeh, “StreamNet: Memory-Efficient Streaming Tiny Deep Learning Inference on the Microcontroller,” vol. 36, 2024.

[113] A. A. Khan, N. A. Rink, F. Hameed, and J. Castrillon, “Optimizing Tensor Contractions for Embedded Devices with Racetrack Memory Scratch-Pads,” in *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2019, pp. 5–18.

[114] S. Jaiswal, R. K. K. Goli, A. Kumar, V. Seshadri, and R. Sharma, “MinUn: Accurate ML Inference on Microcontrollers,” in *Proceedings of the 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2023, pp. 26–39.

[115] A. Kumar, V. Seshadri, and R. Sharma, “Shiftry: RNN Inference in 2KB of RAM,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.

[116] S. Jin, C. Zhang, X. Jiang, Y. Feng, H. Guan, G. Li, S. L. Song, and D. Tao, “Comet: A novel memory-efficient deep learning training framework by using error-bounded lossy compression,” *Proc. VLDB Endow.*, vol. 15, no. 4, pp. 886–899, Dec. 2021.

[117] J. Chen, L. Zheng, Z. Yao, D. Wang, I. Stoica, M. Mahoney, and J. Gonzalez, “ActNN: Reducing Training Memory Footprint via 2-Bit Activation Compressed Training,” in *Proceedings of the 38th International Conference on Machine Learning*, M. Meila and T. Zhang, Eds., ser. Proceedings of Machine Learning Research, vol. 139, PMLR, Jul. 2021, pp. 1803–1813.

Abstract in Korean

특수 도메인을 위한 분할-스케줄링-병합을 이용한 세분화된 컴파일러 최적화 기법

도메인 특화 언어는 사용자 요구를 충족시키기 위해 프로그래머들이 사용자의 요구를 충족시키는 함수의 기능을 구현하고 확장할 수 있는 프로그래밍 가능성을 지원한다. 프로그래머들은 함수의 연산과 인터페이스를 일정한 세분화 수준으로 정의하여 도메인 특화 프로그램을 쉽게 구성할 수 있다. 비록 캡슐화된 함수들이 프로그램의 모든 기능을 표현할 수 있지만, 기존의 컴파일러들은 이러한 프로그램을 완전히 최적화하지 못하고 있다. 그 이유는 기능들이 추상화되고 캡슐화되어 더 자세한 수준으로 분할되어 있지 않기 때문이다.

소프트웨어 정의 네트워킹에서는, 기존의 컴파일러들이 세밀하게 나누어진 함수들을 병렬화할 기회를 놓치고 있다. 이들은 매치 함수와 액션 함수를 포함하는 각 패킷 처리 테이블을 단일 작업 단위로 취급한다. 매치 함수와 액션 함수를 분리하면 더 많은 컴파일러 최적화 기회를 찾을 수 있으나, 기존의 컴파일러들은 패킷 처리 테이블 단위로 컴파일을 하고 있기 때문에 최적화 기회를 놓치게 된다.

딥러닝 추론 분야에서는, 기존의 컴파일러들이 텐서 분해를 통한 딥러닝 모델의 세밀한 합성곱을 완전히 최적화하지 못한다. 텐서 분해 기법은 합성곱 가중치에 대해 텐서 분해를 적용하고 분해된 합성곱 시퀀스를 생성한다. 하지만, 기존의 컴파일러들은 합성곱을 해당 분해된 합성곱 시퀀스로만 대체하며, 분해된 합성곱에 대한 각각의 레이어들을 재정렬하거나 융합하지 않기 때문에 메모리 사용량을 줄일 기회를 놓치게 된다.

본 연구는 네트워크 프로그래밍과 딥러닝 추론에 특화된 새로운 세분화 컴파일러를 제안한다. 네트워크 프로그래밍을 위한 새로운 컴파일러인 PSDN을 소개하며, PSDN 컴파일러는

패킷 처리 테이블을 매치 함수와 액션 함수로 분할하고, 함수들을 파이프라인에 스케줄링하며, 동기화 비용을 줄이기 위해 함수를 병합한다. 또한, 딥러닝 추론을 위해 TeMCO라는 새로운 컴파일러를 소개하며, TeMCO 컴파일러는 분해된 합성곱 시퀀스를 개별 합성곱 레이어로 분할하고, 합성곱 레이어를 스케줄링하여 스킵 연결을 최적화하며, 레이어 호출 비용을 줄이고 메모리 사용량을 줄이기 위해 합성곱 레이어와 활성 함수 레이어를 병합한다. 본 연구의 컴파일러는 분할-스케줄링-병합 기법을 이용하여 네트워크 프로그램의 병렬화 기회를 찾고 텐서 분해를 통한 딥러닝 모델의 최대 메모리 사용량을 줄인다.

본 연구의 컴파일러들은 분할-스케줄링-병합 기법을 이용하여 도메인 특화 프로그램의 성능을 향상시킨다. 이전 연구와 비교했을 때, PSDN 컴파일러는 7개의 네트워크 프로그램의 패킷 처리 시간을 12.1% 줄이고 자원 사용을 3.5% 감소시킨다. TeMCO 컴파일러는 10개의 딥러닝 모델에 대하여 배치 크기에 따라 추론 시간은 1.08배에서 1.70배 증가하나 내부 텐서의 최대 메모리 사용량을 75.7% 줄인다. 본 연구의 컴파일러들은 특정 도메인에 맞춘 분할-스케줄링-병합 기법을 활용하여 도메인 특화 프로그램에서 성능 향상을 이룰 수 있다.

핵심되는 말: 컴파일러, 네트워크, 딥러닝