creative commons
COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

M.S. THESIS

Automatic Generation of Efficient Execution Plan for Convolutional Neural Networks

합성곱 신경망의 효율적인 실행을 위한 실행 계획 자동 생성

AUGUST 2020

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

김민수

M.S. THESIS

Automatic Generation of Efficient Execution Plan for Convolutional Neural Networks

합성곱 신경망의 효율적인 실행을 위한 실행 계획 자동 생성

AUGUST 2020

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

김민수

Automatic Generation of Efficient Execution Plan for Convolutional Neural Networks

합성곱 신경망의 효율적인 실행을 위한 실행 계획 자동 생성

지도교수 Bernhard Egger

이 논문을 공학석사 학위논문으로 제출함

2020 년 06 월

서울대학교 대학원

컴퓨터 공학부

김 민 수

김민수의 공학석사 학위논문을 인준함

2020 년 06 월

위 원 장	박 근 수	(인)
부위원장	Bernhard Egger	(인)
위 원	Srinivasa Rao Satti	(인)

Abstract

Over the past years, a large number of architectures and accelerators for Deep Neural Networks (DNNs) have been proposed. While exhibiting common features, the number and arrangement of processing elements, the sizes and types of on-chip memory, and the possibilities of parallel execution vary significantly especially in the embedded system domain. The number of off-chip memory accesses and the performance of a DNN on a given accelerator depends not only on the supported computational patterns and the available on-chip memory but also on the sizes and shapes of each layer. Finding a computational pattern that minimizes off-chip memory accesses while maximizing performance is thus a tedious and error-prone task. This thesis presents *e-PlaNNer*, a compiler framework that generates an optimized execution plan for a given embedded accelerator and Convolutional Neural Network (CNN). For each layer, *e-PlaNNer* determines the performance-optimal configuration by considering the data movement, tiling, and work distribution. The generated execution plan is transformed to code, allowing for a fast development cycle with different CNNs and hardware accelerators. Evaluated with five neural networks under varying memory configurations and compared to previous works on the Nvidia Jetson TX2, *e-PlaNNer* achieves 6× speedup and 21.14 % reduction of off-chip memory access volume on average. In addition, *e-PlaNNer* shows meaningful performance compared to well-known deep learning frameworks in terms of end-to-end execution.

Keywords: Convolutional Neural Network, Compiler, Execution Plan

Student Number: 2018-22990

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Thanks to excellent performance in image classification and object detection tasks, convolutional neural networks (CNNs) have received an unprecedented amount of attention. In the quest for higher accuracy, the initially simple CNNs comprising only a limited number of sequential layers continue to evolve into more and more complex and deeper networks [2, 3, 4, 5, 6, 7]. At the same time, a large number of dedicated CNN accelerators have been proposed and implemented on ASICs, FPGAs, and CGRAs [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]. Common architecture features such as processing elements (PE) and on-chip memory are shared by all accelerators, however, the number and arrangement of the PEs, the size and types of the on-chip memory, and the possibilities for parallelizing a workload vary significantly. Moreover, if the available on-chip memory is typically not large enough to hold all the required data at once (e.g. embedded system or partitioned server system), the calculation of a layer has to be split into smaller sub-tasks through a task known as *tiling*. The type of the network layer, the parallelization strategy, and the size of these tiles

determine what computations can be performed and which data chunks need to be brought off/on-chip at what times. These architectural features combined with the different options for orchestration of the computation lead to a large number of possibilities with which a layer of a network can be processed. However, finding the best configuration and generating code for a given CNN and specific accelerator in this large optimization space are tedious and error-prone tasks.

The computations of the layers in CNN (convolutional and fully-connected) can be represented as nested loops. A common feature of all layers is that the loops carry no true dependences, allowing for a large number of valid executions plans in terms of tiling, loop interchange, and loop unrolling. A convolutional layer in VGGNet-16 [6], for example, allows for around 2×10^9 different execution plans that vary considerably with respect to the computational complexity and the amount of data transfer between on- and off-chip memory. To achieve good performance, the utilization of the available computational resource needs to be maximized while minimizing the amount of off-chip memory accesses that consume two to three orders of magnitude more energy than on-chip memory accesses [20].

This thesis presents *e-PlaNNer*, a framework that facilitates the task of finding an optimized execution plan for each layer of a given CNN algorithm and accelerator. *e-PlaNNer* takes as an input the structure of a CNN and the description of a hardware accelerator. Building on the idea of the *Roofline model* [21], *e-PlaNNer* enumerates and identifies the execution plan that maximizes the performance per off-chip memory access volume. Since evaluating all possible execution plans is infeasible, the vast search space is first pruned using heuristics. The expected performance of an execution plan is determined by a model that considers both computation and memory transfers with respect to

the dataflow and tiling of the execution plan. The selected execution plan is then transformed into an intermediate representation (IR) that can easily be translated into code executable on the target accelerator.

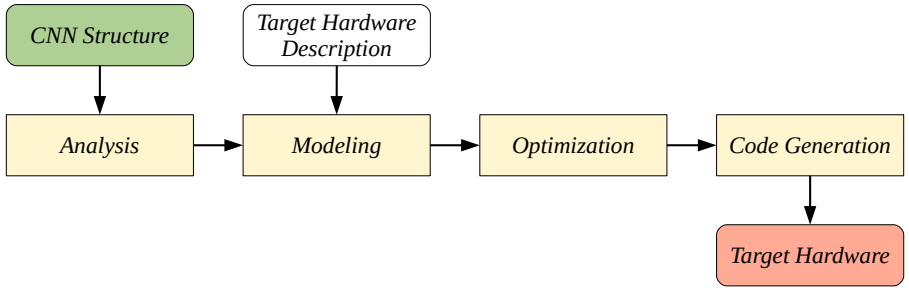

Figure 1.1: Overview of e-PlaNNer framework processing.

The contributions of this thesis are as follows. First, it presents some heuristics to prune the large optimization space down to a manageable size using the *Roofline model* and two analytic models that calculate the performance and amount of transferred memory based on an architecture template. Second, it presents the design of *e-PlaNNer* that applies the heuristics to find and generate code for the best execution plan for each layer of a CNN. Third, it shows evaluations with five well-known CNNs and different memory limitations on the Nvidia Jetson TX2 platform. Compared to the best static dataflow and tiling, *e-PlaNNer* achieves 6× speedup and 21.14% reduction of of off-chip memory access volume on average.

The remainder of this thesis is organized as follows. Chapter 2 introduces related works of this thesis and discusses the difference of this thesis from them. Chapter 3 briefly discusses the anatomy of convolutional layers and the Roofline model. Chapter 4 shows a representation of CNN as a graph and optimization process of *e-PlaNNer* at the graph level. Chapter 5 analyses the possible tilings and dataflow patterns of a layer. Chapter 6 introduces the models and the

methodology to generate an efficient execution plan. Code generation to intermediate and target representation are discussed in Chapter 7, and Chapter 8 shows evaluation results on *e-PlaNNer* including performance and memory access on the target hardware platforms. Chapters 9-10 discuss and conclude this thesis.

Chapter 2

Related Work

There are several other approaches to map CNN operations to the specific devices. Commonly, they follow some steps. First, extract some features of convolutional operation and characterize each execution style according to the features. Second, construct a model to estimate or evaluate the execution so that it is a barometer of decision which execution is better. Some works focus on the accuracy of the model compared to the actual execution on the hardware [22, 23]. Finally, explore a search space, which comprises the possible executions, to find the best execution.

Zhang et al. [16] represents a convolutional operation as a nested loop form and explains a mechanism of mapping the operation to hardware accelerator using loop optimization techniques; *loop unrolling*, *loop pipelining*, and *loop tiling*. This work determines the optimal hardware design for target CNN model by design space exploration based on *roofline model*. At the same time, this work determines an optimized execution plan represented by unified tiling factors through the whole layers. This work has similar approach with this thesis, but

its target hardware has limited execution structure, so it is too hard to apply layer-wise execution planning. So, this work applies cross-layer optimization to determine the identical tiling factors regardless of layer features.

Ma et al. [13] investigates the optimal design variables for the CNN accelerator design. It also proposed an loop-based analysis for convolutional operation to determine the optimal execution plan expressed by tiling factors and loop order. This work divides the hardware system by two parts; off-chip and on-chip, and determines the optimization factors for each part to minimize computing latency, memory access size, and hardware resource. To find the optimization factors, this work proposes a few steps of decision flow for each part. However, it takes too much time to follow the decision flow for every possible execution plan; almost 5 to 9 hours on two desktops, so this work randomly samples 0.005% of possible plans to construct the search space. On the other hand, this thesis proposes some heuristics that guarantees the final execution plan can reach to the global optimum as well as a light weight decision model to determine the optimal execution plan.

Chen et al. [24], TVM, presents an end-to-end deep learning framework that generates optimized execution code for wide range of hardware target. Since each hardware including not only CPU and GPU but custom accelerator supports only its own low-level library, TVM just applies an extensible solution by providing higher level abstraction and lowers it to the supported library. TVM proposes a machine-learning (ML) based cost model which predicts the performance of an execution plan from a large search space. However, like many other ML based solutions, TVM's ML model may need so many training data and trials with time consuming until the model shows reasonable accuracy. Even though TVM collects the training data and improve itself at runtime, it shows less speedup at the small number of trials.

Li et al. [25], Smart-Shuttle, presents an adaptive partitioning and scheduling approach to minimize the size of DRAM access, which has the greatest impact on power efficiency. In order to determine the DRAM access size, this work introduces a DRAM access pattern which is determined by the tiling factors and data reusability. Smart-Shuttle applies an empirical rule in order to find the optimal solution instead of exploring the solution in a large search space. The rule divides all layer cases into only two conditions, depending on the ratio between output feature map and filter size. And it prioritizes each tiling factor to give larger number under the limited global memory size. Even if the rule removes the search space exploration, it cannot be applied to diverse hardware platforms and CNN models, because the execution patterns on the various hardware and models cannot be determined that simple rule.

Parashar et al. [23], Timeloop, introduces a systematic approach to evaluate deep learning accelerator. This work is based on two keywords; *model* and *mapper*. The model represents methods to evaluate such mapping on such deep learning accelerator by providing performance, area and energy. The mapper constructs a search space of various mappings and explores the best mapping in the search space. It introduces some heuristic algorithms like exhaustive linear search or random sampling depending on the size of the search space.

	Layer-wise	**Modeling**		**Exploration**	
	Optimization	**Time**	**Method**	**SSE[1]time**	**SSE method**
e-PlaNNer (Ours)	Yes	Short	Loop-based	Short	Search Space Pruning
Zhang et al. [16]	No	Short	Loop-based	Long	Full Search
Ma et al. [13]	Yes	Short	Loop-based	Long	Random Sampling
TVM [24]	Yes	Long	ML-based	Short	Simulated Annealing
Smart-Shuttle [25]	Yes	Short	Loop-based	Short	Empirical Rules
Timeloop [23]	Yes	Short	Loop-based	Long	Random Sampling

Table 2.1: Comparison with related works in terms of modeling and exploration method.

[1]Search Space Exploration

Chapter 3

Background

3.1 Convolutional Neural Networks

Deep Neural Networks (DNNs) is the most popular algorithm in machine learning. In particular, CNN is one of the famous DNN algorithms in the vision processing. The CNN consists of two parts; *feature extraction* and *classification*. A convolutional layer is the most influential layer in the feature extraction. It is computed by sliding a 4-dimensional kernel over a 3-dimensional input, yielding a 3-dimensional output as illustrated in Figure 3.1.

The most common arithmetic operation of the convolutional layer is the multiply-accumulate (MAC) operation that accumulates the product of an input element with a kernel element into a partial output. Typically, a kernel of size ($OC \times IC \times KH \times KW$) operates on an input ($IC \times IH \times IW$) to obtain an output of size ($OC \times OH \times OW$), where I, O, and K stand for *input*, *output*, and *kernel*, and C, H, and W for *channel*, *height*, and *width*, respectively. Figure 3.2 shows the code for a convolution expressed as a six-fold nested loop.

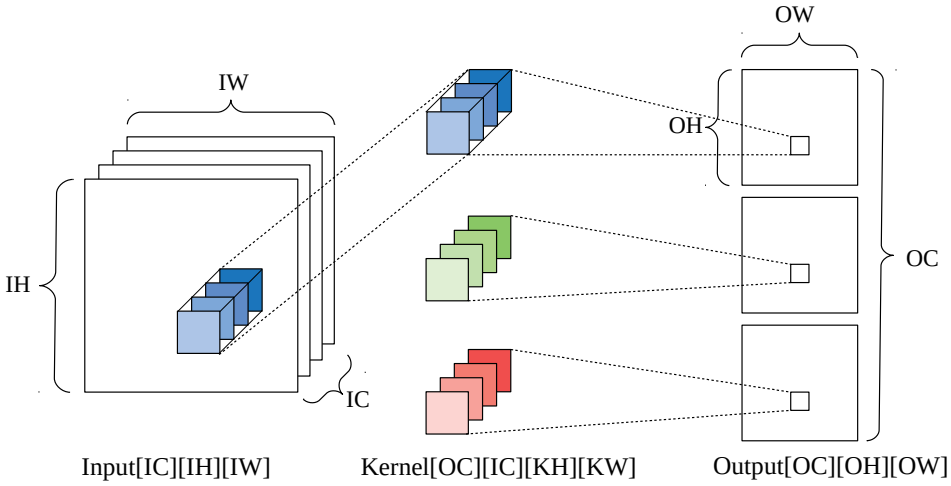

Figure 3.1: A convolutional layer

```
for (oc=0; oc<OC; oc++)
  for (ic=0; ic<IC; ic++)
    for (oh=0; oh<OH; oh++)
      for (ow=0; ow<OW; ow++)
        for (kh=0; kh<KH; kh++)
          for (kw=0; kw<KW; kw++)
            Output[oc][oh][ow] +=
              Input[ic][oh*stride+kh][ow*stride+kw]
              * Weight[oc][ic][kh][kw];
```

Figure 3.2: General loop structure of a convolutional layer

A fully connected layer is the main component of the classification. It is similar to the convolutional layer except that the input and output are 1-dimensional data, and kernel is 2-dimensional data without width and height dimensions.

3.2 DNN Accelerator

Thanks to noticeable growth of hardware computing power, the performance of CNN algorithms are more reasonable than before. Some kinds of hardware accelerators are used in order to accelerate the execution of CNN algorithms.

Especially, a distributed computing architecture is suitable for CNN which consists of multiple repetitive and simple calculations like MAC. Figure 3.3 shows a common architecture of DNN accelerator. It consists of the array of processing elements (PEs) and hierarchical memories.

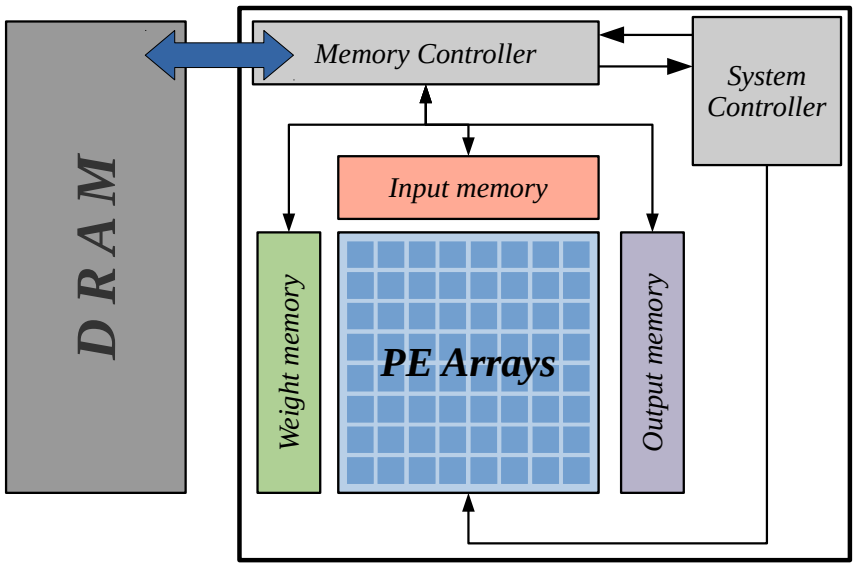

Figure 3.3: A common architecture of DNN accelerator.

Chen et al. [9], *Eyeriss*, for instance, has 168 PEs with 12 rows and 14 columns and three-layered memory hierarchy; DRAM, 108 KB of global buffer, and register files. Chen et al. [8], *DianNao*, has 496 floating point PEs, including multipliers and adders, with SIMD architecture and stores input, kernel and output data separately on the 44 KB of local on-chip memory. Jouppi et al. [12], *Google TPU*, has 64K of matrix multiply units with systolic architecture and 24 MiB of local unified buffer for input and output data.

Most of DNN accelerators focus on high *throughput* and *power efficiency*. High throughput can be achieved with the high utilization of the PEs. And, since memory access requires relatively higher energy consumption than com-

putation, power efficiency can be achieved by reducing memory access size. So, the execution of the DNN algorithm on the accelerator should consider how to reduce the memory transfer size while PEs are highly utilized.

3.3 Roofline Model

The roofline model [21] expresses the achievable throughput of a computation in terms of arithmetic intensity. Based on how many computations are performed per data unit; *arithmetic intensity*, either the memory bandwidth or the computational throughput limits the maximum attainable performance as shown in Figure 3.4.

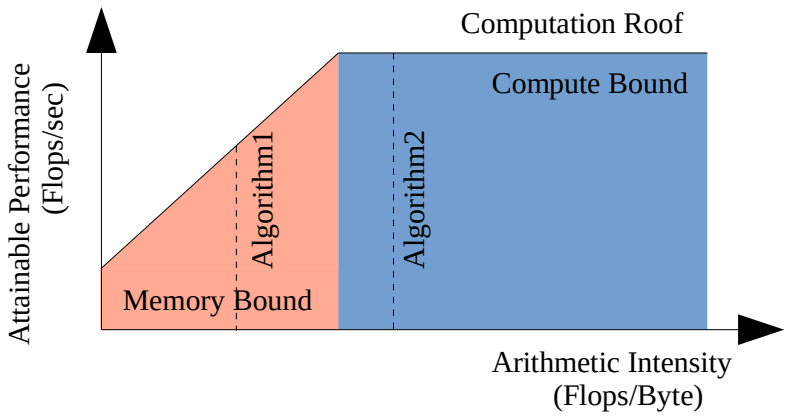

Figure 3.4: The roofline model.

Algorithm 1 is in the memory bound with low arithmetic intensity; only a relatively small number of operations are applied to the transferred data, hence the memory becomes the bottleneck. *Algorithm 2*, on the other hand, has a high arithmetic intensity in the compute bound. The memory bandwidth is sufficient to keep all available PEs occupied at all time.

$$Performance = \min \begin{cases} Computation\ Roof \\ Arithmetic\ Intensity \times Bandwidth \end{cases} \tag{3.1}$$

Equation 3.1 depicts the attainable performance of roofline model. When the arithmetic intensity is in the compute bound, the attainable performance is same as computation roof. On the other hand, in the memory bound, the performance is same as the arithmetic intensity times memory bandwidth. A conclusion of the roofline model is that minimizing the data transfer size (and thus increasing arithmetic intensity) leads to higher performance as long as the computation roof has not been reached.

Chapter 4

Graph Level Processing

The layers in the convolutional neural network form a *directed acyclic graph (DAG)*. So, DAG can give a good start for the end-to-end CNN framework to execute CNN algorithm. This chapter introduces the way how *e-PlaNNer* constructs the CNN graph. In addition, this chapter explains the way to reduce an exploration overhead; *schedule caching* by grouping and caching some identical layers in the graph level.

4.1 Graph Construction

DAG consists of vertices and directed edges. In *e-PlaNNer*, each vertex represents a layer in the CNN, and each directed edge represents a tensor, N-dimensional data, passed between the layers. Figure 4.1 represents a simple example of DAG in *e-PlaNNer*. It has three layers CNN; two convolutional layers and one max pooling layer with passing 4-dimensional tensors. The edge passes an information of tensor dimensions to the next vertex as an input data. The layer parameters in the vertex calculate next tensor dimensions and they are delivered to the next edge.

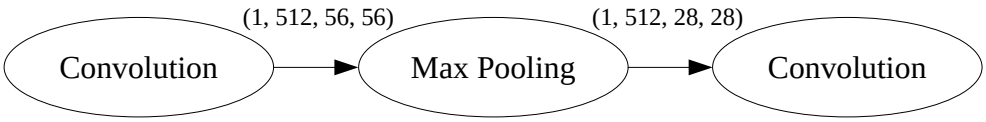

Figure 4.1: Directed acyclic graph in *e-PlaNNer*.

The vertex includes the layer parameters representing layer processing. For example, *convolutional* layer has the information of kernel tensor dimensions, stride and padding. It also includes a function to calculate output tensor dimensions which are passed to the next vertex as an input. The vertex also includes scheduling factors which are explained in chapter 5.

4.2 Schedule Caching

The tensor parameters like dimensions are main factors for layer scheduling in *e-PlaNNer*. If some layers have identical tensor parameters, they are scheduled as same scheduling factors. Since finding the best scheduling factors in the large search space is time consuming, the exploration time can be reduced by grouping identical layers as a same group and caching the best schedule of them. Especially, most of CNNs have repeated layers which have identical layer parameters. Table 4.1 shows the reduced number of scheduled layers before and after applying the schedule caching. On average, 37.14 % of layers are repeated, so that they can be grouped as a same scheduling during graph level processing.

VGGNet-16 [6]		**ResNet50** [2]		**SqueezeNet** [4]		**YOLOv2** [26]	
Uncached	Cached	Uncached	Cached	Uncached	Cached	Uncached	Cached
16	12	50	21	26	18	23	15

Table 4.1: The number of scheduled layers before & after caching.

Chapter 5

Convolutional Layer Analysis

In all except the simplest cases, the entire data required by a convolution is not fit in the on-chip memory of an embedded device. While loop tiling [27] reduces the memory requirements to a supported level, the repeated reloading of data leads to an increased energy consumption and can cause the convolution to become memory-bound. Finding a tiling that meets the hardware requirements while minimizing data reloading and maximizing arithmetic intensity is thus an important optimization goal. The general loop structure of a convolution (Figure 3.2) carries no true loop dependence. The loops are thus interchangeable, leading to several degrees of freedom in the order of processing a convolution. This order impacts the loop tiling and presents opportunities to improve data reusability, if exploited properly, can greatly reduce the amount of reloaded data. The remainder of this section discusses opportunities for optimizations of convolutional layers.

5.1 Loop Structure

```
1  // off-chip loop nest
2  for (oc=0; oc<OC; oc+=OCt)
3    for (ic=0; ic<IC; ic+=ICt) {
4      ... // skipped remaining dimensions for brevity
5
6      /** move data from off-chip to on-chip memory **/
7
8      // on-chip loop nest
9      for (toc=oc; toc<min(oc+OCt, OC); toc+=OCp)
10       for (tic=ic; tic<min(ic+ICt, IC); tic+=ICp) {
11         ...
12
13         /** move data from on-chip memory to register files **/
14
15         // parallelization loop nest
16         for (poc=toc; poc<min(toc+OCp, min(oc+OCt, OC)); poc++)
17           for (pic=tic; pic<min(tic+ICp, min(ic+ICt, IC)); pic++) {
18           ...
19
20           Output[poc][poh][pow] +=
                 Input[pic][poh*stride+pkh][pow*stride+pkw] *
                 Kernel[poc][pic][pkh][pkw];
21         }
22     }
23 }
```

Figure 5.1: Tiled loop structure for a convolutional layer on a generic CNN accelerator.

Anticipating loop tiling and based on the general architecture of accelerators, a convolution is represented as three nested loop groups; *off-chip*, *on-chip*, and *parallelization* (Figure 5.1) each containing the basic loop form in Figure 3.2. This loop structure reflects the execution order for the data movements between off-chip (DRAM) and on-chip memory (*off-chip* loops), the on-chip memory management (*on-chip* loops), and the computation (*parallelization* loops).

5.2 Loop Tiling

The off-chip loop nest from Figure 5.1 allows a large number of valid tilings that exhibit different performance characteristics. Figure 5.2 plots the performance of the possible tilings of the 9th convolutional layer (input: 512x28x28, tensor: 512x512x3x3, output: 512x28x28) from VGGNet-16 [6] for a fixed on-chip memory size and the maximum attainable performance under the roofline model.

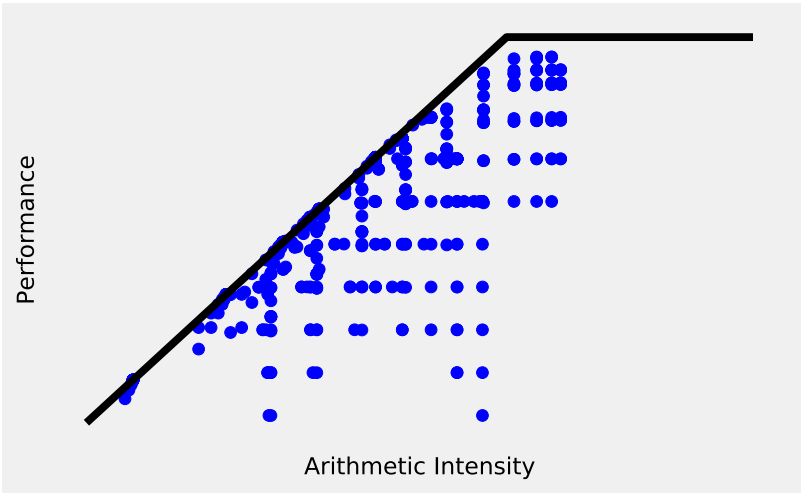

Figure 5.2: Roofline analysis of tilings for `VGGNet-16`'s 9^{th} convolutional layer.

The larger the tiles the lower the total number of iterations in a loop nest. Increasing the tile size in the off-chip loop nest, for example, leads to fewer data transfers between off- and on-chip memory. Under consideration of the architectural constraints, it may be possible to completely eliminate a loop dimension by setting its tiling size equal to its loop bounds.

5.3 Dataflow

The dataflow, i.e., the order of the computation, is a key optimization to improve performance and data reusability. While the dataflow of the *parallelization* loop nest is restricted by the capabilities of the accelerator, it can be freely chosen for the outermost *off-chip* loop nest. An optimization in one direction, however, can negatively affect the other dimensions because the data access patterns are intertwined.

The chosen dataflow dictates the order with which data tiles are brought on/off-chip. We use the terms introduced in [28] to refer to the three basic dataflow patterns *input stationary*, *weight stationary*, and *output stationary*. Figure 5.3 illustrates the effect of the dataflow patterns on the order of computation and data reuse.

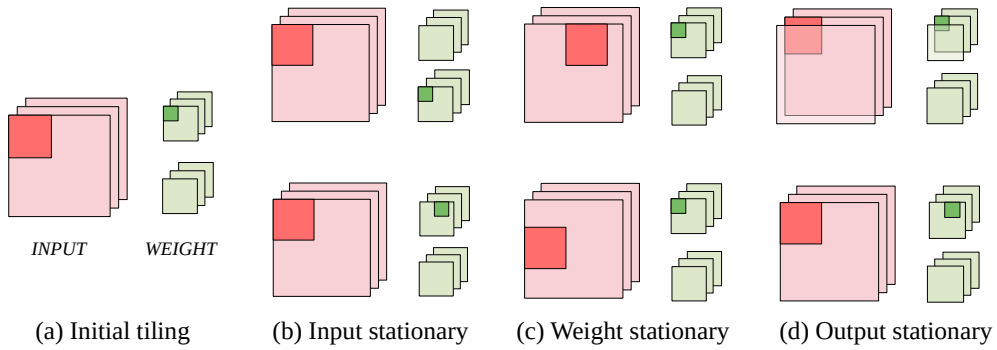

Figure 5.3: Data reuse patterns for the off-chip and the on-chip loop nest.

Input stationary exploits the reusability of the input by keeping an input tile on on-chip until all computations requiring that tile have completed. while weight and output stationary dataflow exploit data reusability of weight and output data in a similar fashion. Figure 5.3 shows two kinds of dataflow patterns

for each stationary. *Input stationary* is achieved by moving the OC (above) or KW, KH (below) loops to the innermost position. The data load pattern is *weight stationary* with the OW (above) or OH (below) loop at the innermost position. For *output stationary*, the IC (above) or KW, KH (below) loops are moved to the innermost position.

The dataflow pattern is also affected by the loop tiling. If a tile holds all data along the KW, KH, and OC dimensions, for example, the dataflow pattern becomes input stationary. In this case, there is no chance to move those loop dimensions to innermost because they are removed as their iteration count becomes one. *e-PlaNNer* maximizes data reuse combining loop tiling and loop interchange under consideration of the architectural constraints in terms of the supported execution patterns in the parallelization loop nest and the available on-chip memory.

Chapter 6

Execution Planning

This chapter discusses methodologies to find an efficient execution plan for CNN accelerators based on the convolutional layer analysis in chapter 5. There are two main issues to focus on. The one is *exploring the search space*. As shown in chapter 5, there are many combinations of loop tiling and interchange, and they form a huge search space. This chapter discusses more efficient way to explore that huge search space. The other is *modeling* to compare each combination in order to decide which combination is better. This thesis focuses on performance and power efficiency to evaluate an execution plan, but the definition of a better execution plan can vary. This chapter proposes some modeling methodologies using a *roofline model* to estimate memory and computation performance.

6.1 Architecture Configurations

The target architecture properties required by the compiler to generate an execution plan are provided in abstracted form and include information about the computational array (# and organization of PEs, supported dataflows), the on-chip memories, and the memory bandwidth. Figure 6.1 shows an example.

```
{
  "bandwidth": 1.6,      // off-chip bandwidth (GB/s)
  "frequency": 1.2,      // PE clock frequency (GHz)
  "mem_size": [368,288,144], // on-chip memories (KB)
                         // (input, weight, output)
  "pe_len": [32, 32],    // PE array (width, height)
  "pe_mapping":
    [ ["IC"],            // PE column mapping
      ["OC"] ]           // PE row mapping
}
```

Figure 6.1: Example of Hardware Configuration File

`pe_len` describes the organization of PEs abstracted as a two-dimensional array as shown in Figure 6.2. `pe_mapping` contains information about supported computational patterns. `bandwidth` and `frequency` are used to compute the memory bandwidth and the computational performance; `mem_size` describes the sizes of the different on-chip memories. Many CNN accelerators, including GPUs, can be abstracted by this form [9, 8, 28, 29, 30].

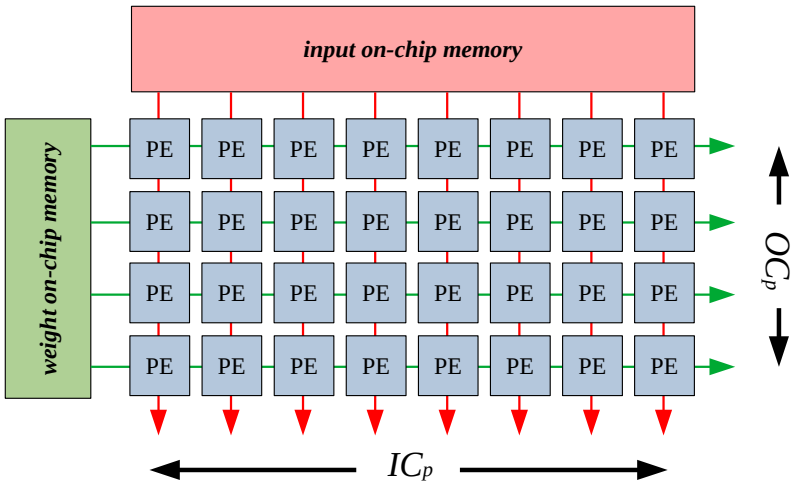

Figure 6.2: Abstract representation of a CNN accelerator.

6.2 Modeling Off-Chip Memory Accesses

Off-chip memory access volume significantly affects performance and energy consumption [20]. Dynamic off-chip memory accesses typically constitute one of the largest shares of the total energy consumption [9], and the arithmetic intensity of a convolution is improved by reducing off-chip memory access volume. An energy-efficient and high-performance execution plan must thus aim at minimizing off-chip memory access volume.

The total size of data loaded and stored to off-chip memory is the summation of the data size multiplied by the reload factors for input, weight, and output data:

$$memory\ access\ volume = \sum_{d=\{input,weight,output\}} volume_d \times reload_d \tag{6.1}$$

The reload factor $reload_d$ depends on the dataflow and the tiling of a convolution. For input stationary dataflow, the input reload factor is one[1]. Otherwise, the reload factor depends on the iteration count of loops not related to the input dimensions, i.e., the kernel width/height, and the number of output channels. The input data reload factor is given by Equation 6.2 for non-input-stationary dataflow:

$$reload_{input} = \left\lceil \frac{KW}{KW_t} \right\rceil \times \left\lceil \frac{KH}{KH_t} \right\rceil \times \left\lceil \frac{OC}{OC_t} \right\rceil \tag{6.2}$$

[1] Ignoring the overlap necessary to compute a convolution.

Similarly, for weight stationary dataflow, the weight data reload factor is one. Otherwise, the reload factor is determined by the iteration count of the output height and width loops as follows:

$$reload_{weight} = \left\lceil \frac{OW}{OW_t} \right\rceil \times \left\lceil \frac{OH}{OH_t} \right\rceil \tag{6.3}$$

Output data is incrementally computed by accumulating the product of input and weight elements. Unlike (the read-only) input and weight data, partially computed output data (*psum*) needs to be written back to off-chip memory before being replaced on-chip. For non-output-stationary dataflows, the psum reload factor is given by Equation 6.4:

$$reload_{psum} = 2 \times (\left\lceil \frac{KW}{KW_t} \right\rceil \times \left\lceil \frac{KH}{KH_t} \right\rceil \times \left\lceil \frac{IC}{IC_t} \right\rceil - 1) \tag{6.4}$$

The term -1 at the end of Equation 6.4 considers that initially each psum is zero and does not need to be loaded from off-chip memory. Since the psum data is transferred twice between off-/on-chip memory for both psum load and write-back, the psum reload factor includes a factor 2 at the front of Equation 6.4. Finally, the output reload factor is determined by the psum reload as follows:

$$reload_{output} = reload_{psum} + 1 \tag{6.5}$$

Since the final output data does not need to reload from off-chip to on-chip memory, the term $+1$ at the end of Equation 6.5 stands for the last write-back of final output data to off-chip memory. If the dataflow is output stationary, i.e., $(KW = KW_t) \wedge (KH = KH_t) \wedge (IC = IC_t)$, the output data reload factor evaluates to 1 regardless of Equation 6.5.

6.3 Modeling Performance

Performance is one of the important metrics when comparing different combinations of loop tiling and dataflow. *e-PlaNNer* estimates the performance for each possible combination at compile time using a *roofline model*-based performance estimation model.

The performance of a tiling is estimated by multiplying the number of active PEs with their clock frequency:

$$perf_{compute} = frequency \times active\ PEs \tag{6.6}$$

The PE utilization is estimated by analyzing the on-chip and parallelization loop from Figure 5.1.

$$\begin{aligned} active\ PEs &= \#\ of\ PEs \times PE\ utilization \\ PE\ utilization &= \frac{\#\ of\ MAC\ operations}{\#\ of\ iterations \times \#\ of\ PEs} \end{aligned} \tag{6.7}$$

The number of MAC operations per tile is the product of all tiling factors in on-chip loop nest, and the total number of iteration is equal to the number of parallelization tiles created by a tiling.

$$\#\ of\ MAC\ operations = \prod factor_t \tag{6.8}$$

$$\#\ of\ iterations = \prod \left\lceil \frac{factor_t}{factor_p} \right\rceil \tag{6.9}$$

Because of the rounding up function in Equation 6.9, the remainder of the division can affect the number of iterations, even though it is quite small. The large number of iterations reduces PE utilization in Equation 6.7. So, to maximize PE utilization, the tiling factors should be a multiple of the hardware parallelization given by `pe_len`.

The performance of a memory-bound tiling is the product of arithmetic intensity by the off-chip memory bandwidth

$$perf_{memory} = bandwidth \times arith.\ intensity \tag{6.10}$$

where the arithmetic intensity is estimated as

$$arith.\ intensity = \frac{\#\ of\ total\ operations}{off\text{-}chip\ memory\ access\ size} \tag{6.11}$$

The estimated performance of a tiling is the smaller of the two performance estimates

$$performance = min(perf_{compute}, perf_{memory}) \tag{6.12}$$

Since the model is only used to select the best tiling, the correlation between a performance estimation from the modeling and an actual measured performance on the hardware is important rather than the absolute accuracy. Figure 6.3 shows a positive correlation between the two around the regression line (the red linear line) with 0.64 of *PCC (Pearson Correlation Coefficient)* [31], which statistically measures the correlation between paired data. It means that it appears *strong positive correlation* according to the guide from Evans [32].

6.4 Search Space Exploration

Enumerating and evaluating all possible dataflows and tilings for a given network is not a feasible strategy as there are too many combinations to consider. For example, a single convolutional layer of VGGNet-16 has about 2×10^9 combinations. This section discusses the three heuristics used to prune the large search space to a manageable number of combinations that are evaluated using the models discussed in the section 6.2 and 6.3. *e-PlaNNer* prunes the search space by setting a lower bound of tiling and dataflow in a following few steps:

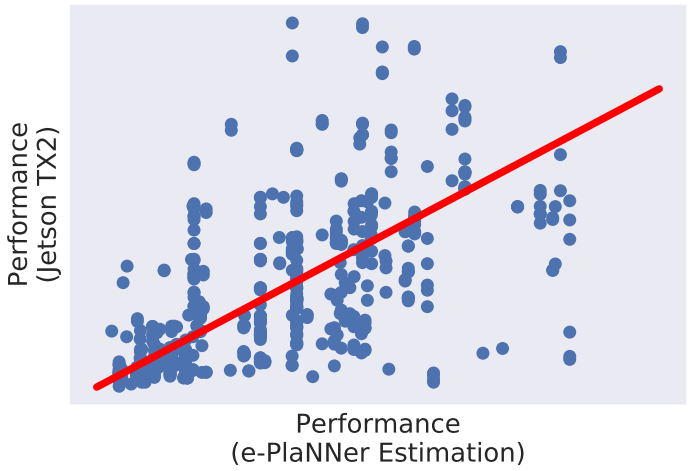

Figure 6.3: Correlation between the performance from *e-PlaNNer* and Nvidia Jetson TX2 [1].

1. Tiling factor initialization. This heuristic prunes the search space by applying some preconditions to the possible tiling factors. First, *e-PlaNNer* excepts the cases to tile a weight tensor in the KH and KW dimension because doing so generates small tiles with a high reload count. Second, *e-PlaNNer* also maximizes the tiling along the innermost dimension of the input, output, and weights (typically the W dimensions) to exploit the contiguous data load in the row-major data layout. Finally, too small tilings can lead to severe PEs underutilization. The tiling factors associated with the loops mapped directly onto the PE array are initialized with the length of the PE array in order to maximize PE utilization. The result of tiling factor initialization is illustrated in Figure 6.4 from (a) to (b).

Pseudo-code for this step is shown in Algorithm 1. Initially, all the tiling factors along all dimensions are initialized to 1. Next, the weights are fully tiled in KH and KW dimension. Line 3 initializes the tiling factor along the OW

```
for (oc=0; oc<OC; oc+=1)
  for (ic=0; ic<IC; ic+=1)
    for (oh=0; oh<OH; oh+=1)
      ...
      // loop body
```

(a) Before Pruning

```
for (oc=0; oc<512; oc+=32)
  for (ic=0; ic<256; ic+=32)
    for (oh=0; oh<56; oh+=1)
      ...
      // loop body
```

(b) Tiling Factor Initializing

```
for (oc=0; oc<512; oc+=32)
  for (ic=0; ic<256; ic+=32)
    for (oh=0; oh<56; oh+=56)
      ...
      // loop body
```

(c) Loop Elimination

```
for (ic=0; ic<256; ic+=32)
  for (oc=0; oc<512; oc+=32)
      ...
      // loop body
```

(d) Loop Interchange

Figure 6.4: Search space pruning through loop transformations.

dimension to exploit contiguously stored data. Lines 4–8 initialize the tiling factors in accordance with the execution model of the accelerator (refer to Figure 6.1).

Algorithm 1: Tiling Factor Initializing

Result: Initialized Tiling Lower Bound
Input: Layer paramters *param*,
Hardware configuration *config*

1: $T[kw, kh, ow, oh, ic, oc] \longleftarrow [1, 1, 1, 1, 1, 1]$
2: $T[kw, kh] \longleftarrow param[kw, kh]$
3: $T[ow] \longleftarrow param[ow]$
4: $col\_mapping\_dim \longleftarrow config.pe\_mapping[col]$
5: $row\_mapping\_dim \longleftarrow config.pe\_mapping[row]$
6: **for** $map \in [col\_mapping\_dim,\ row\_mapping\_dim]$ **do**
7: $\quad T[map] \longleftarrow config.pe\_len[map]$
8: **end for**
9: **return** T

2. Loop elimination. The intention of this second step is to fully tile iterations and thereby eliminate reloads from off-chip memory. Figure 6.4 (b) to (c) shows the effect of loop elimination optimization. There are four possible loops that can potentially be eliminated (the KW, KH, and W loops have been fully tiled in step 1). A possible ways to choose the loop dimension to be eliminated

Algorithm 2: Loop Elimination

Result: Tiling Lower Bound
Input: Layer parameters *param*,
Hardware configuration *config*,
Initialized tiling factors $initT$

```
1:  bestT ⟵ initT
2:  minReload ⟵ MAX_INT
3:  for set ⊂ [OC, IC, OH] do
4:      T ⟵ bestT
5:      for f ∈ set do
6:          T[f] ⟵ param[f]
7:      end for
8:      if T.size < config.memSize then
9:          if minReload > ReloadCount(T) then
10:             minReload ⟵ ReloadCount(T)
11:             bestT ⟵ T
12:         end if
13:     end if
14: end for
15: return bestT
```

is to select the loop with the highest iteration count. This, however, does not always lead to the best outcome because eliminating two smaller dimensions can be better than eliminating one large dimension. For example, when a large OC is eliminated by fully tiling it, only input reloading can be minimized. On the other hand, if IC and OH are eliminated, both weight and partial sum reloads can be minimized. There are eight possible set combinations along the

loop dimensions (*IC*, *OC*, *OH*). *e-PlaNNer* tries to apply loop elimination for all eight combinations and selects the one that minimizes the amount of reloaded data. Since loop elimination is restricted by the size of the on-chip memory, a required memory size should be checked for every combinations.

Pseudo-code for this step is shown in Algorithm 2. After the initialization, the loop iterates through all eight set combinations and completely unrolls the dimensions in the set (lines 5–7). If the results fits into the on-chip memory (line 8) and the combination has a lower reload count that that so-far best case, it is remembered as the new best combination (lines 9–12).

3. Loop interchange. In this step, the loop with the highest iteration count is moved to the innermost position. The loop at the innermost position does not cause repeated data reloads, hence moving it to the innermost position helps to reduce the amount of reloaded data. Pseudo-code for this step is shown in Algorithm 3. The initial loop sequence is initialized randomly in line 1; the front is the inner loop. And the loop sequence is sorted in descending order by the number of iterations for each loop dimension.

Algorithm 3: Loop Interchange

Result: Most Efficient Loop Sequence
Input: Layer parameters *param*,
Tiling Lower Bound T

1: $loopSeq \longleftarrow [kw, kh, ow, oh, ic, oc]$
2: $loopSeq.SortBy(\lceil \frac{param[dim]}{T[dim]} \rceil,\ dim \in [kw, kh, ow, oh, ic, oc])$
3: **return** *loopSeq*

4. Explore Search Space. All remaining tiling candidates above the lower bound are compared with the metric:

$$optimization\ metric = \frac{estimated\ performance}{off\text{-}chip\ memory\ access\ size} \tag{6.13}$$

where *estimated performance* is obtained from the performance estimation model (Equation 6.12) and *off-chip memory access size* is the amount of accessed off-chip memory (Equation 6.1). Figure 6.4 (c) to (d) and Algorithm 4 illustrate

Algorithm 4: Search Algorithm

Result: Tiling of best execution plan
Input: Layer parameters *param*,
Hardware configuration *config*

1: $tilingList \longleftarrow tilingsInPrunedSearchSpace$
2: $bestTiling \longleftarrow tilingList[0]$
3: $largestMetric \longleftarrow 0$
4: **for** $tiling \in tilingList$ **do**
5: $ai \longleftarrow GetArithmeticIntensity(param,\ tiling)$
6: $cr \longleftarrow GetComputationRoof(config)$
7: $perf \longleftarrow min(cr,\ ai \times config.bandwidth)$
8: $metric \longleftarrow \frac{perf}{OffChipAccess(tiling)}$
9: **if** $metric > largestMetric$ **then**
10: $bestTiling \longleftarrow tiling$
11: $largestMetric \longleftarrow metric$
12: **end if**
13: **end for**
14: **return** $bestTiling$

function `OffChipAccess`(*tiling*)
15: $size \longleftarrow 0$
16: $size \longleftarrow size + GetInputAccessSize(tiling)$
17: $size \longleftarrow size + GetWeightAccessSize(tiling)$
18: $size \longleftarrow size + GetOutputAccessSize(tiling)$
19: **return** $size$

this step. Conceptually, all feasible tilings are enumerated (line 1) and the best tiling and metric are remembered (lines 2–3). Each feasible tiling (line 4) is evaluated according to the performance model from Section 6.3 (lines 5–7) and compared against the optimization metric (line 8). If the new best tiling has been found, it is stored in the *bestTiling* variable (lines 9–12) which is returned at the end of the search algorithm (line 14).

Figure 6.5 shows the pruned space after each step. As the pruning heuristics proceed, the considered tilings converge to the left top to a point with few off-chip accesses and high performance.

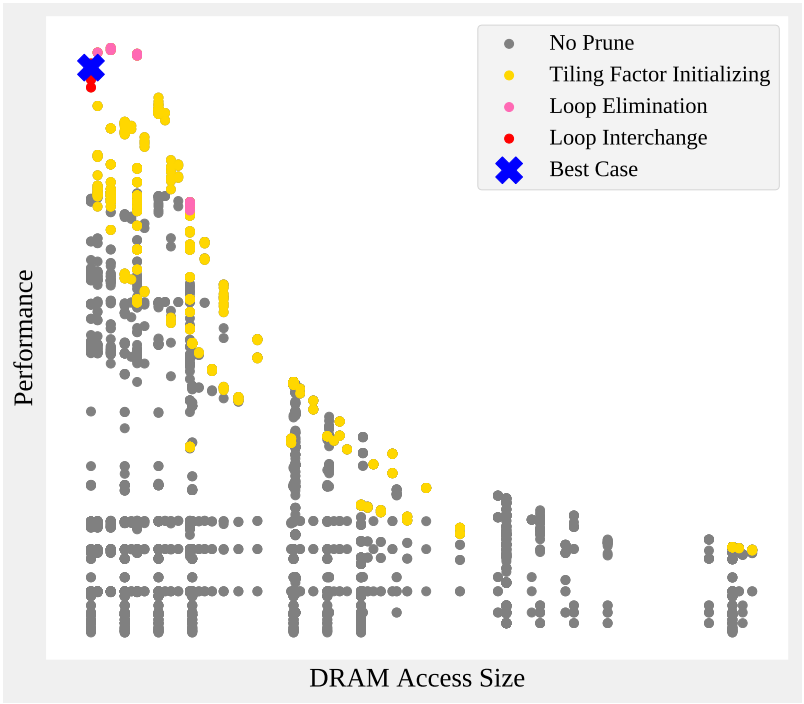

Figure 6.5: Search space after pruning.

Chapter 7

Code Generation

Most of accelerators in embedded system are typically part of a System-on-Chip with a general purpose processor acting as a system controller, a memory interface to off-chip memory, and the accelerator itself for the purpose of a neural network acceleration (Figure 3.3). The accelerator is programmed by the system controller through an custom runtime library supported by accelerator to trigger computations and data movement operations. An execution plan is a sequence of operations from the library to orchestrate the evaluation of a neural network for a given input.

e-PlaNNer includes a template-based code generator that automatically generates target code from an execution plan. In a first step, *e-PlaNNer* outputs the sequence of operations in a programming-language independent intermediate representation (IR) that defines the exact sequence of operations and specifies the tiles to be loaded/stored. In a second step, the generated IR is transformed into target code that can be compiled by a compiler for the target platform.

7.1 Intermediate Representation

An execution plan in IR form consists of the three sections `info`, `var`, and `text`; an example is shown in Figure 7.1. The `info` section contains information

```
[info]
// (batch, channel, height, width)
INPUT(1,128,56,56)
WEIGHT(256,128,3,3)
OUTPUT(1,256,56,56)

[var]
// Memory variables with size
IN_MEM_0(131072) // Input memory
WT_MEM_0(65536) // Weight memory
OT_MEM_0(131072) // Output memory
// Tiles (address, batch, channel, height, width)
INPUT_0[0](1,65,17,56)
INPUT_1[840](1,65,18,56)
INPUT_2[1736](1,65,18,56)
INPUT_3[2632](1,65,9,56)
INPUT_4[203840](63,17,56)
INPUT_5[204680](63,18,56)
...
WEIGHT_0[0](56,65,3,3)
WEIGHT_1[585](56,63,3,3)
...
OUTPUT_0[0](1,56,16,56)
OUTPUT_1[896](1,56,16,56)
...

[text]
LOAD  WT_MEM_0 WEIGHT_0
LOAD  IN_MEM_0 INPUT_0
CONV  OUTPUT_0 INPUT_0 WEIGHT_0  1  (1,1,1,0)
STORE OUTPUT_0 OT_MEM_0
...
```

Figure 7.1: An example of intermediate representation for convolutional layer.

about the layer parameters such as the number of batches and channels, and the height and width of the data. The section `var` defines the data tiles and assigns a name for each tile. The `var` section defines the on-chip memories

and the data tiles in off-chip memory. For each hardware on-chip memory, a corresponding variable is defined as configured in the hardware configuration file (Figure 6.1). This information allows the low-level code generator to verify that the size of all data allocated to the on-chip memory actually fits. The tile variables define the offset relative to the start of the data array, the number of batches, channels, and its height and width for each tile. At the moment, the offset is computed assuming a row-major data layout with the dimensions input, output and weight tensors laid out as given in Section 3.1. The `text` section, finally, contains a sequence of commands implementing the data movement and computation operations. The first argument to a `LOAD` and `STORE` command refers to a destination of the data transfer and the second argument defines the source of the transfer. A convolution operation, `CONV`, expects five arguments: output, input, and weight tiles and two parameters expressing the stride and the padding.

7.2 Target Code Generation

e-PlaNNer contains a template-based backend code generator that transforms an execution plan in IR format to target code. As a proof of concept, we have implemented a Nvidia CUDA backend. Figure 7.2 shows part of the CUDA code generated from the IR shown in Figure 7.1. The function `conv` (line 4) is a template function invoking a CUDA convolution. The variables defined in the `var` section of the IR are transformed into CUDA/C code at the beginning of the `main` function. The `LOAD` operations and the `STORE` operation are mapped to `cudaMemcpy` functions in lines 16, 17, and 19.

```
#include <cuda.h>
#include <cuda_runtime.h>

__global__ void conv(float* in, float* wt, float* ot,
                     int stride, int* paddings, ...)
{
    // CUDA kernel code for convolutional operation.
}

int main(void)
{
    init_variables();
    // GPU memory allocations.
    float *in_mem_0, *wt_mem_0, *ot_mem_0;
    cudaMalloc((void**)&in_mem_0, sizeof(float)*131072);
    ...
    cudaMemcpy(wt_mem_0,
               weight_0 + 0,
               sizeof(float)*56*65*3*3,
               cudaMemcpyHostToDevice); // LOAD WT_MEM_0 WEIGHT
    cudaMemcpy(in_mem_0,
               input_0 + 0,
               sizeof(float)*1*65*17*56,
               cudaMemcpyHostToDevice); // LOAD IN_MEM_0 INPUT
    conv<<<numBlocks, numThreads>>>(in_mem_0, wt_mem_0, ot_mem_0,
                                    1, {1, 1, 1, 0}, ...);
                  // CONV OUTPUT_0 INPUT_0 WEIGHT_0 1 (1,1,1,0)
    cudaMemcpy(output_0 + 0,
               ot_mem_0,
               sizeof(float)*1*56*16*56,
               cudaMemcpyDeviceToHost); // STORE OUTPUT_0 OT_MEM_0
    ...
}
```

Figure 7.2: Example of generated CUDA code

A runtime library with CUDA helps to execute the generated CUDA code on GPU. The runtime library includes basic operations of CNNs like *Convolutional*, *Max Pooling*, and *Fully Connected*. The runtime library can be replaced with well-known library like *LAPACK (Linear Algebra PACKage)* [33]. In addition, it can be also replaced with a custom library of an accelerator as well as a CPU/GPU library.

Chapter 8

Evaluation

8.1 Experimental Setup

To demonstrate the effect of the optimized execution plan, *e-PlaNNer* is evaluated with five representative CNNs and four memory configurations. The evaluated CNNs are *VGGNet-16* [6], *ResNet50* [2], *AlexNet* [5], *SqueezeNet* [4], and *YOLOv2* [26] executed with 32-bit single-precision floating point values. *VGGNet-16*, *ResNet50*, and *AlexNet* show the combination of the convolutional layer and the fully connected layer while *SqueezeNet* and *YOLOv2* include only convolutional layers. Table 8.1 describes the CNNs for the experiment.

Model	**Size (MB)**	**# ops (Gflops)**	**# layers**	**Model description**
VGGNet-16	527.741	14.408	16	13 conv + 3 fc layers
ResNet50	86.723	4.536	50	49 conv + 1 fc layers
AlexNet	237.914	1.057	8	5 conv + 3 fc layers
SqueezeNet	4.698	1.029	26	26 conv
YOLOv2	194.265	27.898	23	23 conv

Table 8.1: Representative CNNs for the experiment.

The memory sizes are quite limited less than the size of input, weight, and output tensors so that it can show the impact of tiling and dataflow effectively. Table 8.2 represents experimental setups of the on-chip memory configurations for target embedded systems.

	Memory Size		
	Input	**Weight**	**Output**
Setup A	256 KB	128 KB	256 KB
Setup B	512 KB	256 KB	512 KB
Setup C	256 KB	256 KB	256 KB
Setup D	256 KB	512 KB	256 KB

Table 8.2: Evaluated memory configurations.

The code generation capabilities of *e-PlaNNer* are demonstrated by executing the generated CUDA code and measure its end-to-end execution time on the Nvidia Jetson TX2 platform [1]. The Jetson TX2 contains 256 CUDA cores running at 1.02 GHz and achieves a memory bandwidth of 60 GB/s for data movement operations.

e-PlaNNer is compared with some related works [16, 13, 25] which have their own planning strategy to find the best execution plan in terms of tiling and dataflow. Zhang et al. [16] and Ma et al. [13] try to minimize the psum reload because the partial sum should be transferred twice between the off- and on-chip memory. Zhang et al. [16] places IC loop at the innermost of the off-chip loop nest, so that the psum data reload goes 0 by setting the dataflow as *output stationary*. Ma et al. [13] fully tiles KW, KH and IC loop dimensions in priority in order to eliminate those dimensions. It makes the psum data reload 0 according to Equation 6.4. Li et al. [25], Smart-Shuttle, applies empirical rules to minimize off-chip memory access volume. First, It compares the layer parameters $OW \times OH$ and $IC \times KW \times KH$. If $OW \times OH$ is larger than

$IC \times KW \times KH$, the dataflow is set as *output stationary*. Otherwise, the dataflow is set as *weight stationary*. Second, this work gives every tiling factor a priority to be allocated maximum under the constraint of on-chip memory size. For example, in *output stationary* case, the priority sequence of the tiling factors is $OC_t > OH_t > IC_t$. This means that IC_t and OH_t are maximized after maximizing OC_t which has the highest priority. On the other hand, in *weight stationary* case, the priority sequence shows $OC_t, IC_t > OH_t$.

To compare them fairly on my runtime backend, they are given some beneficial conditions. All of them are given fully tiled W dimension to exploit the contiguous data load in the row-major data layout. Zhang et al. [16] and Ma et al. [13] are given the best tiling factors and dataflow for remaining factors generated from *e-PlaNNer*. Originally, without the additional conditions, Zhang et al. [16] should find a sub-optimal fixed tiling factors regardless of layer parameters using *Cross-layer Optimization* which is explained ambiguously in [16]. Ma et al. [13] randomly samples the possible execution group from a large search space to find the remaining factors from the group. Table 8.3 shows the planning strategies and given additional conditions for the comparison with the related works.

	Planning Strategy	**Additional Conditions**
Zhang et al. [16]	Output Stationary	Fully tiled W, Best factors for remainders
Ma et al. [13]	Minimize output reload	Fully tiled W, Best factors for remainders
Smart Shuttle [25]	Empirical rules	Fully tiled W

Table 8.3: Related work settings for the experiment.

8.2 Performance Results

Performance is evaluated as the speedup by comparing end-to-end execution time of generated code from *e-PlaNNer* with the related works which have fixed execution rules summarized in Table 8.3. Figure 8.1 shows results of the performance comparison. On average, *e-PlaNNer* achieves a 6.12× of speedup compared to the related works. *e-PlaNNer* shows more improvement of speedup in smaller memory configuration. In setup A (640 KB on-chip memory), *e-PlaNNer* achieves 8.30× of speedup in comparison to the 3.23× speedup in setup B (1,280 KB on-chip memory).

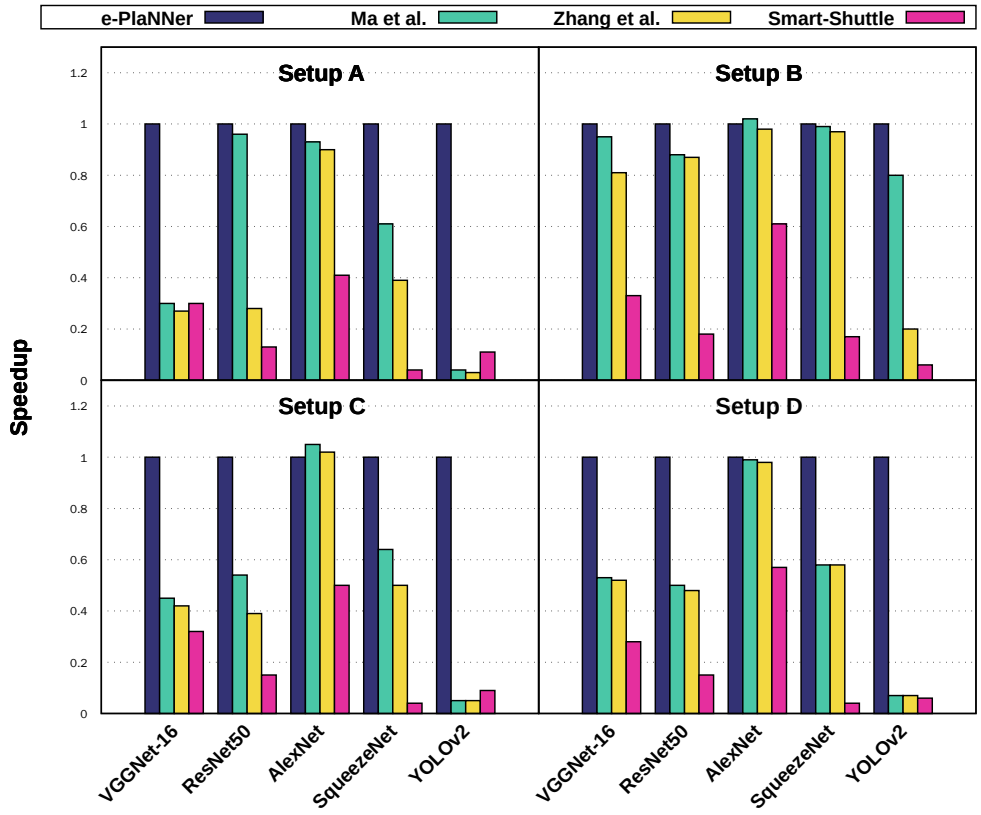

Figure 8.1: Speedup of *e-Planner* generated plans relative to the related works with fixed execution rules.

e-PlaNNer also achieves more speedup in the CNN with the large amount of data. In *YOLOv2*, which has the largest amount of input and output tensors, *e-PlaNNer* achieves a 15.66× of speedup. On the other hand, *AlexNet*, which has the small number of convolutional layers, is given less benefit of *e-PlaNNer* strategy, so that *e-PlaNNer* achieves only 1.33× of speedup in it. Furthermore, *AlexNet* has three of fully connected layers which require a large number of data transfers between off-/on-chip memory. In fact, in case of *AlexNet*, fully connected layers have 12 times more data size than convolutional layers. Moreover, Fully connected layer has less optimization points than convolutional layer because fully connected layer has only single width and height for input, weight, and output. Table 8.4 shows the average speedup results compared to comparative group for each memory configurations and CNNs.

	VGGNet-16	**ResNet50**	**AlexNet**	**SqueezeNet**	**YOLOv2**	**Avg.**
Setup A	3.435 x	4.023 x	1.536 x	9.450 x	23.068 x	8.302 x
Setup B	1.767 x	2.608 x	1.213 x	2.630 x	7.929 x	3.229 x
Setup C	2.569 x	3.770 x	1.304 x	9.106 x	17.083 x	6.767 x
Setup D	2.458 x	3.632 x	1.257 x	8.982 x	14.573 x	6.180 x
Avg.	2.557 x	3.508 x	1.328 x	7.542 x	15.663 x	6.120 x

Table 8.4: Average speedup of *e-PlaNNer* compared to the comparative group.

8.3 Comparison of Off-chip Memory Access

Off-chip memory access volume is measured by adding up data transfer sizes for each memory access event in the generated code. Figure 8.2 compares the off-chip memory access volume of *e-PlaNNer* with the related works. The optimal case assumes that there is no data reload, which means the total tensor size for each CNN. *e-PlaNNer* shows 21.14 % reduction of memory access volume on average. Especially, *e-PlaNNer* shows outstanding results on the environment with the small memory size and CNN with deep network and large number of operations. In setup A, *e-PlaNNer* shows 26.36 % reduction of memory access

volume in setup A while it shows only 16.57 % of reduction in setup B. And, in *YOLOv2*, it shows 52.07 % of reduction while it shows only 2.67 % in *AlexNet* which is the shallowest.

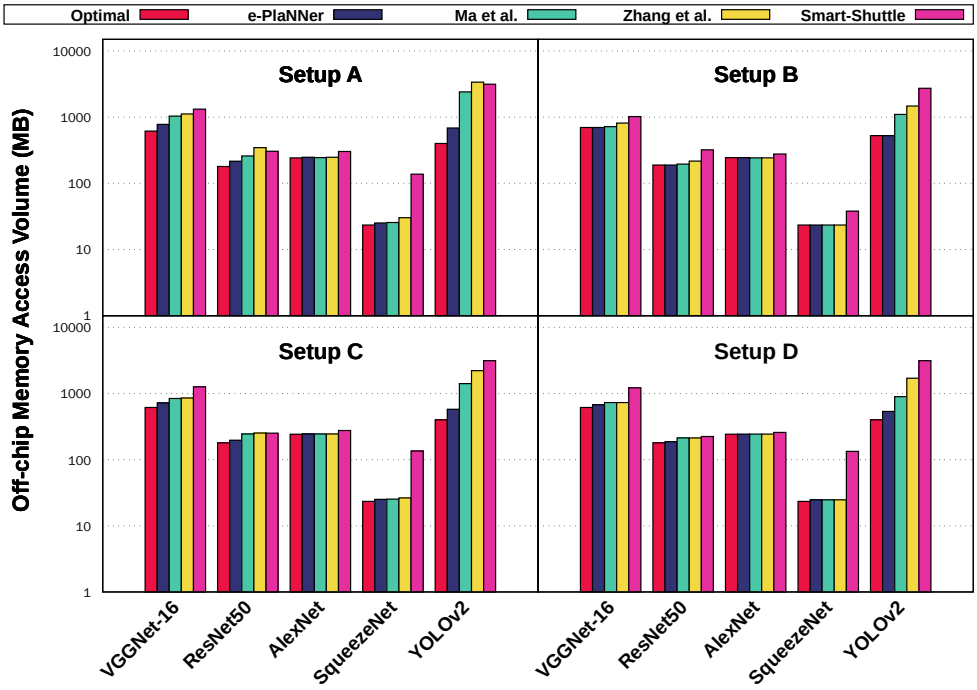

Figure 8.2: Comparison of off-chip memory access volume.

By analyzing the off-chip memory access volume, I can find the characteristics for each related work. Figure 8.3 stacks the memory access volumes of input, weight, and output data in *VGGNet-16* with setup A. The x-axis for each chart represents each layer in *VGGNet-16*. Because Ma et al. and Zhang et al. focus on minimizing reloading the output data, they care less about reloading the input data. Smart-Shuttle, which applies empirical rules, shows much larger output data reload at the back side of layers. That's because Smart-Shuttle chooses weight stationary at the back side of *VGGNet-16* layers while it chooses output stationary at the front side.

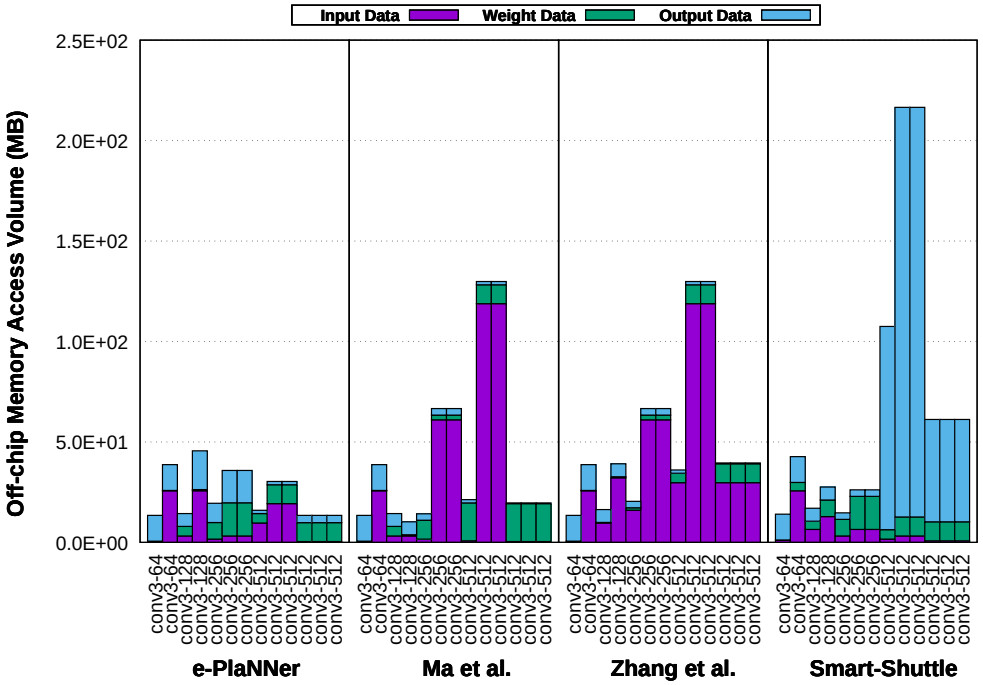

Figure 8.3: Off-chip memory access volume for each layer of VGGNet-16 with Setup A.

8.4 Framework Results

This section shows how *e-PlaNNer*'s techniques introduced in this thesis affect runtime performance. In addition, this section also shows performance of *e-PlaNNer* on the sufficient hardware resources compared to performance from well-known deep learning frameworks.

Table 8.5 shows the pruning rates for each case of memory configuration and CNN. As described in Section 6.4, by pruning the search space, *e-PlaNNer* effectively reduces the scheduling time to a reasonable level. In Table 8.5, the percentage numbers in parentheses describe the ratio of search space size between pruned and non-pruned. The numbers above the percentage numbers

represent the number of execution plan combinations before or after pruning. As shown in Table 8.5, all the cases prune the search space more than 99 %.

	VGGNet-16	**ResNet50**	**AlexNet**	**SqueezeNet**	**YOLOv2**
Non-pruned	3541.530 M	298.074 M	645.980 M	49.388 M	4104.669 M
Pruned					
Setup A	27,552 (0.00078 %)	41,040 (0.01377 %)	5,310 (0.00082 %)	4,596 (0.00931 %)	89,937 (0.00219 %)
Setup B	25,536 (0.00072 %)	21,984 (0.00738 %)	405 (0.00006 %)	1,530 (0.00310 %)	83,409 (0.00203 %)
Setup C	25,536 (0.00072 %)	23,232 (0.00779 %)	645 (0.00010 %)	4,596 (0.00931 %)	89,937 (0.00219 %)
Setup D	23,184 (0.00065 %)	12,864 (0.00432 %)	645 (0.00010 %)	4,530 (0.00917 %)	80,133 (0.00195 %)
Avg.	**0.00072 %**	**0.00831 %**	**0.00027 %**	**0.00772 %**	**0.00209 %**

Table 8.5: The size of non-pruned search space and pruning rates for each memory configuration and CNN.

Schedule caching, which is explained in Section 4.2, also affects *e-PlaNNer*'s scheduling time as well as pruning. Figure 8.4 shows the reduction of e-PlaNNer scheduling time by caching the repeated layers in CNN graph. Schedule caching reduces the scheduling time by 36.37 % compared to uncached scheduling time. Since *AlexNet* has no repeated layer, there is no difference of the scheduling time

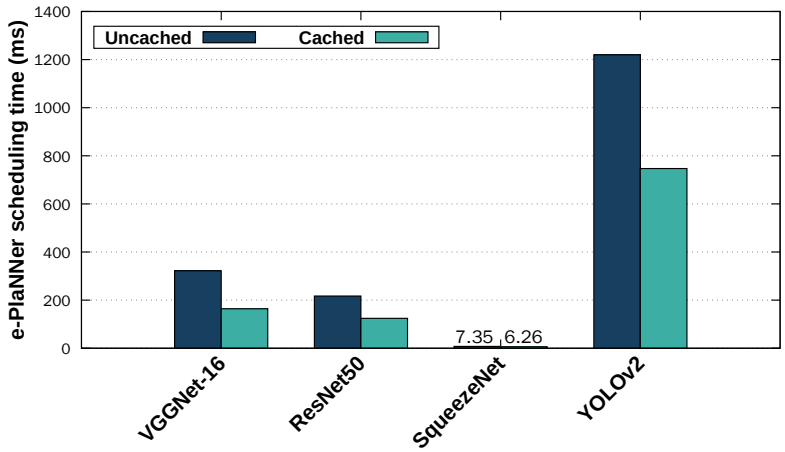

Figure 8.4: Reduction of *e-PlaNNer* scheduling time by caching. (Setup A)

between before and after caching in *AlexNet*; the scheduling time of *AlexNet* on setup A is 36.579 milliseconds. All scheduling times of CNNs are less than 1.4 seconds, so it can provide reasonable execution time to generate the best execution plan.

e-PlaNNer, as a deep learning framework, provides end-to-end execution of CNN on the various hardware settings not only on the limited resource but also on the sufficient resource. The sufficient resource means that the hardware has enough resource (i.e. on-chip memory) to run target CNN application without the tiling and scheduling explained in this thesis.

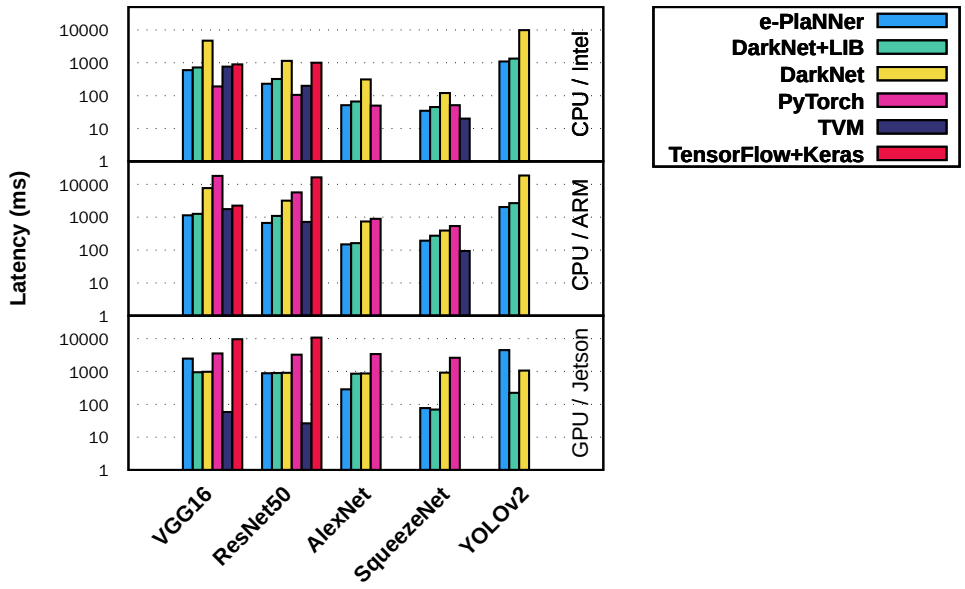

Figure 8.5: Comparison of end-to-end inference latency to well-known deep learning frameworks.

For this experiment, the runtime library of *e-PlaNNer* needs to be expanded to support three of hardware platforms; *Intel CPU*, *ARM CPU*, and *Nvidia Jetson GPU*. Intel CPU is Intel(R) Core(TM) i5-7500 64-bit quad-core CPU with 3.40 GHz frequency. ARM CPU has a big-little structure with dual-core

Nvidia Denver 2 64-bit CPU and quad-core ARM(R) Cortex-A57 MPCore. Nvidia Jetson GPU is Jetson TX2 introduced in Section 8.1. *e-PlaNNer* is compared with four of well-known deep learning frameworks; *DarkNet* [34], *PyTorch* [35], *TVM* [24], and *TensorFlow* [36] with *Keras* [37]. *DarkNet+LIB* combines DarkNet with an external library to enhance the performance of linear algebra and deep learning operations; *MKL (Math Kernel Library)* [38] for Intel CPU, *OpenBLAS* [39] for ARM CPU, and *cuDNN* [40] for Nvidia Jetson GPU.

Figure 8.5 shows the comparison of latency between *e-PlaNNer* and other frameworks. This experiment only measures the end-to-end inference latency for CNN models where the framework provides a pre-trained model. *e-PlaNNer* performs better than DarkNet with or without the external library on CPU; 492.63% better than DarkNet and 30.04% better than DarkNet with external library. On the other hand, DarkNet with *cuDNN* library performs better than *e-PlaNNer* on GPU. In case of CPU, *e-PlaNNer* runtime library also includes *MKL* and *OpenBLAS*, but it doesn't include *cuDNN* on GPU runtime library. Since *cuDNN* library is much more optimized for convolutional operation than *e-PlaNNer* by using Winograd algorithm [41], DarkNet with *cuDNN* shows better performance the *e-PlaNNer* on GPU except *AlexNet* which has less convolutional layers than the others.

PyTorch shows lower performance than *e-PlaNNer* on embedded system (ARM and Jetson) in contrast to the desktop (Intel). TensorFlow with Keras shows much slower inference latency than *e-PlaNNer* almost 7× on average. TVM shows similar performance to *e-PlaNNer*; *e-PlaNNer* is 3.35% faster than TVM on ARM, but TVM is 9.02% faster on Intel. Especially, TVM shows much better performance on Jetson than *e-PlaNNer*; almost 2× faster [1].

[1]The performance of TVM on Nvidia Jetson TX2 comes from incubator-TVM benchmark results. (`https://github.com/apache/incubator-tvm/wiki/Benchmark`)

Chapter 9

Discussion

This thesis lays a cornerstone of the hardware-software hierarchical integration for deep learning ecosystem. As deep learning algorithm becomes more diverse, the infrastructure of deep learning has to support a flexible execution platform. Unfortunately, the traditional hardware design process is too slow to follow the growth of deep learning algorithm. So, the recent deep learning hardware design approaches try to apply more flexible and reconfigurable way to design hardware. *DNN Weaver* [42], from high-level deep network models to FPGA acceleration, provides a template-based hardware design automation for various deep learning algorithms by parameterizing the features of them. *SiLago* framework [43] enhances the abstraction level of hardware design from RTL to micro-architecture, so that it simplifies the hardware design and reduces the design time. The deep learning compiler, this thesis presents, is the backbone of the integrated system between the algorithm and hardware stacks. If the deep learning compiler can be integrated with those hardware design automation tools, it can provide a hierarchical integration from compiler to hardware design automation for target deep learning algorithm.

Chapter 10

Conclusion

This thesis presents *e-PlaNNer*, a framework to generate efficient execution plans for a given convolutional network and CNN accelerator. *e-PlaNNer* predicts the performance of different dataflows and tilings using two estimation models to compute the off-chip memory access volume and the performance of execution plan and, by this so, *e-PlaNNer* selects the best execution plan for each convolutional layer. *e-PlaNNer* also provides a fast exploration of the best execution plan by pruning the large search space. *e-PlaNNer* outputs the best execution plan as target code which can be compiled on the target platform; in its current form, a CUDA backend has been implemented and tested. The evaluation with five common networks under various restricted memory setups on the Nvidia Jetson TX2 platform shows that *e-PlaNNer* outperforms the executions from previous works by a large margin both in terms of execution time and accessed off-chip data volume. In addition, *e-PlaNNer* never lags behind well-known deep learning frameworks in terms of end-to-end execution.

Bibliography

[1] NVIDIA. (2017) Nvidia jetson tx2. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2/

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[3] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[4] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size," *CoRR*, vol. abs/1602.07360, 2016. [Online]. Available: http://arxiv.org/abs/1602.07360

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. USA: Curran Associates Inc., 2012,

pp. 1097–1105. [Online]. Available: http://dl.acm.org/citation.cfm?id=2999134.2999257

[6] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014.

[7] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 2818–2826.

[8] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 269–284. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541967

[9] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan 2017.

[10] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *CVPR 2011 WORKSHOPS*, June 2011, pp. 109–116.

[11] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 243–254. [Online]. Available: https://doi.org/10.1109/ISCA.2016.30

[12] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080246

[13] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 45–54. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021736

[14] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *The 49th Annual IEEE/ACM International*

Symposium on Microarchitecture, ser. MICRO-49. Piscataway, NJ, USA: IEEE Press, 2016, pp. 17:1–17:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=3195638.3195659

[15] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput cnn inference on fpgas," in *Proceedings of the 54th Annual Design Automation Conference 2017*, ser. DAC '17. New York, NY, USA: ACM, 2017, pp. 29:1–29:6. [Online]. Available: http://doi.acm.org/10.1145/3061639.3062207

[16] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: ACM, 2015, pp. 161–170. [Online]. Available: http://doi.acm.org/10.1145/2684746.2689060

[17] M. Tanomoto, S. Takamaeda-Yamazaki, J. Yao, and Y. Nakashima, "A cgra-based approach for accelerating convolutional neural networks," in *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, Sep 2015, pp. 73–80.

[18] I. Bae, B. Harris, H. Min, and B. Egger, "Auto-tuning cnns for coarse-grained reconfigurable array-based accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 10, pp. 1–1, Oct 2018.

[19] B. Harris, I. Bae, and B. Egger, "Architectures and algorithms for on-

device user customization of cnns," *Integration*, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167926018302700

[20] M. Fagan, J. Schlachter, K. Yoshii, S. Leyffer, K. Palem, M. Snir, S. M. Wild, and C. Enz, "Overcoming the power wall by exploiting inexactness and emerging cots architectural features: Trading precision for improving application quality," in *2016 29th IEEE International System-on-Chip Conference (SOCC)*, 2016, pp. 241–246.

[21] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009. [Online]. Available: http://doi.acm.org/10.1145/1498765.1498785

[22] H. Kwon, M. Pellauer, and T. Krishna, "MAESTRO: an open-source infrastructure for modeling dataflows within deep learning accelerators," *CoRR*, vol. abs/1805.02566, 2018. [Online]. Available: http://arxiv.org/abs/1805.02566

[23] A. Parashar, P. Raina, Y. S. Shao, Y. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2019, pp. 304–315.

[24] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. Berkeley, CA, USA:

USENIX Association, 2018, pp. 579–594. [Online]. Available: http://dl.acm.org/citation.cfm?id=3291168.3291211

[25] J. Li, G. Yan, W. Lu, S. Jiang, S. Gong, J. Wu, and X. Li, "Smartshuttle: Optimizing off-chip memory accesses for deep learning accelerators," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 343–348.

[26] J. Redmon and A. Farhadi, "YOLO9000: better, faster, stronger," *CoRR*, vol. abs/1612.08242, 2016. [Online]. Available: http://arxiv.org/abs/1612.08242

[27] M. J. Wolfe, "Iteration space tiling for memory hierarchies," *Parallel Processing for Scientific Computing*, vol. 357, p. 361, 1987.

[28] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 367–379. [Online]. Available: https://doi.org/10.1109/ISCA.2016.40

[29] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 609–622. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.58

[30] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the

sensor," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 92–104. [Online]. Available: http://doi.acm.org/10.1145/2749469.2750389

[31] J. a. Benesty, *Pearson Correlation Coefficient*, 1st ed., ser. Noise Reduction in Speech Processing. Berlin, Heidelberg :: Springer Berlin Heidelberg :, 2009, vol. 2.

[32] J. Evans, *Straightforward Statistics for the Behavioral Science*, ser. Psychology Series. Brooks/Cole, 1995. [Online]. Available: https://books.google.co.kr/books?id=eokUtxKe8j4C

[33] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. D. J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. Philadelphia, Pennsylvania, USA: SIAM, 1999.

[34] J. Redmon, "Darknet: Open source neural networks in c," http://pjreddie.com/darknet/, 2013–2016.

[35] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, 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 *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[36] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[37] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[38] *Intel Math Kernel Library. Reference Manual.* Intel Corporation, 2009, santa Clara, USA. ISBN 630813-054US.

[39] Z. Xianyi, W. Qian, and Z. Yunquan, "Model-driven level 3 blas performance optimization on loongson 3a processor," in *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, 2012, pp. 684–691.

[40] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," 2014.

[41] A. Lavin, "Fast algorithms for convolutional neural networks," *CoRR*, vol. abs/1509.09308, 2015. [Online]. Available: http://arxiv.org/abs/1509.09308

[42] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[43] A. Hemani, N. Farahini, S. M. A. H. Jafri, H. Sohofi, S. Li, and K. Paul, *The SiLago Solution: Architecture and Design Methods for a Heterogeneous Dark Silicon Aware Coarse Grain Reconfigurable Fabric*. Cham: Springer International Publishing, 2017, pp. 47–94. [Online]. Available: https://doi.org/10.1007/978-3-319-31596-6_3

요약

지난 몇 년간 심층신경망을 위한 수많은 아키텍처와 가속기가 제안되었다. 이를 통해, 일반적인 심층신경망 수행 방식들이 함께 제안되었으나, 구체적인 연산 배치 방식과 온칩 메모리의 크기 및 종류, 그리고 병렬 실행 방식은 특히 내장형 시스템에서 다양하게 나타날 수 있다. 뿐만 아니라, 오프칩 메모리 접근 크기 및 신경망의 성능은 연산 형태 및 온칩 메모리의 크기 뿐 아니라 신경망 각 계층의 크기 및 형태에 따라서 달라질 수 있다. 따라서, 최대 성능을 내면서 오프칩 메모리 접근을 최소화하는 연산 형태를 일일이 찾는 것은 상당히 번거로운 작업이며, 많은 오류를 발생 시킬 수 있다. 본 논문에서 소개할 *e-PlaNNer*는 주어진 내장형 하드웨어 가속기와 합성곱 신경망에 대하여 최적화된 실행 계획을 생성해주는 컴파일러 프레임워크이다. *e-PlaNNer*는 심층신경망의 각 신경망 계층에 대하여 데이터 이동, 타일링, 그리고 작업 배분을 고려한 성능 최적화된 실행 계획을 결정한다. 또한, 생성된 실행 계획을 실제 컴파일 가능한 코드로 변환함으로써, 서로 다른 다양한 합성곱 신경망과 하드웨어 가속기에 대하여 빠른 개발 주기를 제공한다. 다양한 메모리 구성으로 다섯 가지 합성곱 신경망 응용을 Nvidia의 Jetson TX2 에서 검증하여 기존의 연구와 비교한 결과, *e-PlaNNer*는 평균적으로 6배의 성능 향상과 21.14% 의 오프칩 메모리 데이터 접근량 감소를 보였다. 뿐만 아니라, *e-PlaNNer*는 전체 심층신경망의 실행 관점에서 기존에 잘 알려진 딥러닝 프레임워크와의 비교에서도 의미있는 결과를 보였다.

주요어: 합성곱 신경망, 컴파일러, 실행 계획
학번: 2018-22990