공학박사학위논문

임베디드 시스템에서 여러 컨볼루션 뉴럴 네트워크를 위한 하드웨어를 고려하는 소프트웨어 최적화 기법

Hardware-Aware Software Optimization Techniques for Convolutional Neural Networks on Embedded Systems

2021년 2월

서울대학교 대학원
컴퓨터공학부
강 두 석

임베디드 시스템에서 여러 컨볼루션 뉴럴 네트워크를 위한 하드웨어를 고려하는 소프트웨어 최적화 기법

Hardware-Aware Software Optimization Techniques for Convolutional Neural Networks on Embedded Systems

지도교수 하 순 회

이 논문을 공학박사 학위논문으로 제출함

2020년 10월

서울대학교 대학원

컴퓨터공학부

강 두 석

강두석의 공학박사 학위논문을 인준함

2020년 12월

위 원 장 유 승 주 (인)

부위원장 하 순 회 (인)

위 원 이 재 욱 (인)

위 원 버나드 에거 (인)

위 원 이 영 민 (인)

Abstract

Hardware-Aware Software Optimization Techniques for Convolutional Neural Networks on Embedded Systems

Duseok Kang
Department of Computer Science and Engineering
College of Engineering
The Graduate School
Seoul National University

Executing deep learning algorithms on mobile embedded devices is challenging because embedded devices usually have tight constraints on the computational power, memory size, and energy consumption, while the resource requirements of deep learning algorithms achieving high accuracy continue to increase. To cope with increasing computation complexity, it is common to use an energy-efficient accelerator, such as a mobile GPU or digital signal processor (DSP) array, or to develop a customized neural processor chip called neural processing unit (NPU). In the application domain, many optimization techniques have been proposed to change the application algorithm in order to reduce the computational amount and memory usage by developing new deep learning networks or software optimization techniques that take advantage of the statistical nature of deep learning algorithms. Another approach is hardware-ware software optimization, which finds the performance bottleneck first and then distributes the workload evenly by scheduling the workloads.

This dissertation covers hardware-aware software optimization, which is based on a hardware processor or platform. First, we devise a systematic optimization methodology

through the experience of participating in the Low Power Image Recognition Challenge (LPIRC) [1, 2] and build a deep learning framework called C-GOOD (C-code Generation Framework for Optimized On-device Deep Learning) based on the devised methodology. For hardware independence, C-GOOD generates a C code that can be compiled for and run on any embedded device. Also, C-GOOD is facilitated with various options for application domain optimization that can be performed according to the devised methodology. By applying the devised methodology to three hardware platforms, NVIDIA Jetson TX2 [3], Odroid XU4 [4], and the Samsung Reconfigurable Processor (SRP) [5], we demonstrate that the devised methodology is independent of the hardware platforms and application domain optimizations can be performed easily with C-GOOD.

Recently, embedded devices are equipped with heterogeneous processing elements (PEs), and the need for running multiple deep learning applications concurrently in the embedded systems such as self-driving cars and smartphones is increasing at the same time. In those systems, we devise an end-to-end methodology to schedule deep learning applications onto heterogeneous PEs and implement a scheduling framework according to the methodology. It covers from profiling on real embedded devices to verifying the schedule results on the devices. In this methodology, we use a genetic algorithm (GA)-based scheduling technique for scheduling deep learning applications onto heterogeneous PEs and consider several practical issues in the profile step. Furthermore, we schedule multiple applications with different throughput constraints considering the schedulability of mapped tasks on each processor. After implementing a deep learning inference engine that can utilize heterogeneous PEs using a low-level library of the ARM compute library (ACL) [6], we verify the devised methodology by running two widely used convolution neural networks (CNNs) on a Galaxy S9 smartphones [7] and a Hikey970 board [8].

While the previous optimization methods focus on the computation and processing elements, the performance bottleneck of deep learning accelerators is the communication between off-chip and on-chip memory. Moreover, the off-chip DRAM access volume

has a significant effect on the energy consumption of an NPU. To reduce the impact of off-chip DRAM access on the performance and energy of an NPU, we devise compiler techniques for an NPU to manage multi-bank on-chip memory with two different objectives: one is to minimize the off-chip memory access volume, and the other is to minimize the processing delay caused by unhidden DRAM accesses. The main idea is that by organizing on-chip memory into multiple banks, we may hide the off-chip DRAM access delay by prefetching data into unused banks during computation and reduce the off-chip DRAM access volume by storing the output feature map data of each layer to on-chip memory. By running CNN benchmarks on a cycle-level NPU simulator, we demonstrate the trade-off relation between two objectives. The devised multi-bank on-chip memory management (MOMM) techniques are extended to consider layer fusion that aims to reuse feature maps between layers maximally. Since the pure layer fusion technique incurs extra computation overhead and increases DRAM access for filter weights, a hybrid fusion technique is presented between a per-layer processing technique and the pure layer fusion techniques, based on the devised MOMM techniques with two different objectives. Experiment results confirm the superiority of the hybrid fusion technique to the per-layer processing technique and the pure layer fusion technique.

Keywords : convolutional neural network, software optimization, on-device learning, scheduling, genetic algorithm, heterogeneous processor, mobile device, accelerator, neural processing unit, multi-bank memory management, layer fusion, prefetching

Student Number : 2014-21780

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Deep learning is making significant progress in almost all areas of machine learning, including image classification, object detection, and so on. Extensive research efforts are being made to improve the accuracy of deep learning, paying a huge cost of computational, memory, and energy requirements. To apply such artificial intelligence to our daily life, it is necessary to make edge devices intelligent. The current practice to make a device intelligent is to use a cloud service accessed via a mobile network. There are several concerns with this practice of cloud-based intelligence such as privacy, dependence on the network condition, difficulty of personalization, and so on. As a consequence, on-device learning or inference has recently been drawing keen research attention to run deep learning algorithms directly on the device to relieve those concerns.

Executing deep learning algorithms on mobile embedded devices is challenging because embedded devices usually have tight constraints on the computational power, memory size, and energy consumption while the resource requirements of deep learning algorithms achieving high accuracy continue to increase. Therefore, in order to reduce computational and memory usage, many optimization techniques are proposed.

The optimization techniques can be categorized into three optimization areas: application domain optimization, hardware domain optimization, and hardware-aware soft-

ware optimization. Application domain optimization reduces the amount of computation and memory usage by changing the application algorithm. One approach is to develop new deep learning algorithms that seek to balance accuracy, speed, and resource requirements. SqueezeNet [10], Tiny-YOLO [11], and MobileNet [12] are such examples. Another approach is to develop software optimization techniques that take advantage of the statistical nature of deep learning algorithms. Approximate computing can achieve almost the same accuracy with significantly reduced resource requirements and is used in various approximation techniques such as pruning [13], quantization [14], and low-rank approximation [15].

Hardware domain optimization accelerates deep learning computation with a customized chip, called neural processing unit (NPU). Many pieces of research, such as Eyeriss [16], TPU [17], NVDLA [18], and MIDAP [19], have been recently performed in this optimization area to achieve higher performance per watt than GPU. Lastly, hardware-aware software optimization is a method of optimizing a deep learning system by mapping and scheduling workloads onto multiple hardware resources to distribute the workloads evenly. First, to distribute the workload evenly, it is necessary to find the performance bottleneck in the target system. After finding what the problem is, solve it by scheduling the workloads or applying other optimization techniques to fix it.

The scope of this dissertation is hardware-aware software optimization. Since the hardware-aware software optimization is based on a hardware processor or platform, this dissertation contains optimization methods in two cases. One is for a commodity hardware platform and the other is for a domain-specific accelerator. In this dissertation, there are two optimization methods devised for a commodity hardware platform. First, we devise a systematic optimization methodology using application domain optimizations and traditional system optimizations. Based on the devised methodology, we build a novel deep learning framework called C-GOOD (C-code Generation Framework for Optimized On-device Deep Learning). It generates a C code that can be run on any device and is fa-

cilitated with various software optimization options that can be performed according to the optimization methodology devised in this dissertation. We explain how C-GOOD applies optimization techniques to various deep learning networks such as Darknet [20], Darknet19 [20], Tiny-YOLO [11] and YOLOv2 [21] on three different hardware platforms: the Jetson TX2 [3], the Odroid XU4 [4], and the Samsung Reconfigurable Processor (SRP) [5]. Experiments confirm that the devised methodology is independent of the platforms and software optimizations can be performed easily with C-GOOD.

Embedded devices recently tend to be equipped with heterogeneous processors to cope with deep learning applications' high computing demand. Also, there is a growing need to run multiple deep learning applications concurrently in emerging embedded systems such as self-driving cars and smartphones. It is necessary to schedule multiple deep learning applications on the shared heterogeneous processing elements in those systems, which is a challenging problem. To solve this problem, in this dissertation, we devise a methodology to schedule multiple applications onto a heterogeneous embedded system as the other optimization method for a commodity hardware platform. In addition, we implement a scheduling framework that follows the devised methodology. The devised methodology covers from profiling on real embedded devices to verifying the schedule results on the devices. We use a genetic algorithm (GA)-based scheduling technique to effectively schedule deep learning applications onto heterogeneous PEs, exploring both data-parallelism and task-parallelism to find Pareto-optimal schedules in terms of real-time performance and energy consumption. We consider several practical issues in the profiling step, such as dynamic voltage frequency scaling (DVFS) and CPU hot-plug. Moreover, we consider the schedulability of mapped tasks on each processor, which is a critical issue in the parallel scheduling of multiple applications with different throughput constraints. The devised methodology is verified by running two widely used convolutional neural networks (CNNs) on a Galaxy S9 smartphone [7] and a HiKey970 board [8]. For the experimentation, we implemented a deep learning inference engine that can uti-

lize heterogeneous PEs using a low-level library of the ARM compute library (ACL) [6].

Similar to most optimization methods for a commodity hardware platform, most research on an NPU is focused on computation and aim to parallelize the convolution operation with a number of multiplier-accumulator (MAC) units. However, the achievable performance is much lower due to various causes such as resource contention delay, memory access delay, and processing of non-convolution layers. For example, it is reported that DPU-v2 achieves less than 40% utilization for GoogleNet and ResNet-50, which commonly have a lot of memory operation [22]. Moreover, the off-chip DRAM access volume has a significant effect on the energy consumption of an NPU. In the case of DianNao [23] and Cambricon-X [24], it is reported that the energy consumed by DRAM access accounts for 80% of total energy consumption [25, 26]. Thus, to reduce the DRAM access volume, it is beneficial to increase the on-chip memory size. Nonetheless, the memory requirement for feature maps and filter weights is usually beyond the on-chip memory capacity.

Because on-chip memory capacity is not enough to process all operations without off-chip memory access, many NPU studies attempt to reuse data as much as possible in on-chip memory [17, 16, 25, 27, 26]. However, although the output data is the input of the next layer, many studies have focused on reusing only the data needed to process one layer [16, 25, 26, 27, 28, 29]. It is because many NPUs are designed to transfer the output feature data to off-chip memory after all calculations are completed [28, 29, 27] and many studies reusing data in on-chip memory based on these NPUs. However, according to our experiment in Section 4.4.2, data reuse between layers can reduce DRAM access volume by up to 46.8% without bank management. There are some NPUs which store output data in unified on-chip memory [17, 30]. However, they did not study how to reuse the data in on-chip memory maximally. In this paper, we focus on how to efficiently reuse the output data between consecutive layers in on-chip memory.

Although many NPUs have separate memory for input and output feature maps,

it is more efficient to configure the memory for feature map data as unified memory, considering data reuse between two consecutive layers. It is because separating memory for feature map data limits the reused data size. For example, an NPU has two memory of 256 KB: one for input data and the other for output data. In this case, we cannot reuse the output data more than 256 KB. However, if an NPU has one memory of 512 KB for all feature data, we could store the output data more than 256 KB and reuse it when processing the next layer. Therefore, in this work, we organize the memory for feature map data into unified memory.

The goal of the optimization for an NPU in this dissertation is to reduce the impact of off-chip DRAM access and on-chip memory contention. By making a multi-bank structure of on-chip memory, we can hide the DRAM access by prefetching the next input data during computation. Also, off-chip DRAM access can be reduced by storing the output feature map data in on-chip memory since the data is the input feature map data of the next layer. However, managing the multiple on-chip memory banks is a problem, because both the input and output feature maps are stored in unified on-chip memory, and the on-chip memory size for the feature map is limited. This dissertation devises compiler techniques to manage multi-bank on-chip memory with two different objectives to tackle the problem. One technique aims to minimize the off-chip memory access volume, and the other aims to minimize the processing delay caused by unhidden DRAM accesses. Both techniques reduce on-chip memory contention by assigning different feature map banks for computation and DRAM access, generating different bank assignment sequences. There is a trade-off between DRAM access minimization and performance maximization, which will be discussed in Chapter 4. In case a CNN network contains multiple branches, we need to consider sharing the input feature map between multiple branches. How to deal with multiple branches in the multi-bank on-chip memory management (MOMM) problem is also presented in this work.

A layer fusion technique has recently been proposed to maximize the feature map

reuse between consecutive convolution layers [31]. While it reduces the DRAM access for feature maps, it may increase the DRAM access for filter weights as well as the computation cycle due to duplicate computation. We apply the proposed MOMM techniques to the network to which the layer fusion technique is applied and devise a hybrid technique between the default per-layer processing technique and the layer fusion technique [31].

We perform experiments with several CNN benchmarks, using a cycle-accurate simulator of an adder-type CNN accelerator, MIDAP [19]. All the proposed techniques are implemented as an optimization module of the compiler that produces the control code for a given CNN specified in Caffe 2 [19]. Experimental results show the trade-off between energy consumption and performance with two proposed memory management techniques. The proposed hybrid fusion technique is also compared with the per-layer processing technique and the pure layer fusion technique with a ResNet 50 [32] benchmark. Experimental results confirm the goodness of our hybrid fusion technique that includes the other techniques as extreme cases.

1.2 Contribution

The contributions of this dissertation can be summarized as follows:

- We devise a systematic methodology for embedded systems to explore the wide design space of algorithm selection and software optimizations for a given hardware platform.
  - We build a C-code generation framework for embedded systems, which generates C code that can be compiled for and run on any embedded devices.
  - The framework supports multiple profiling options and easy optimization applications to help users find performance bottlenecks and explore software optimization techniques for deep learning applications.
- We devise a methodology for scheduling deep learning applications onto heterogeneous processors.
  - We build a scheduling framework that maps (sub-) layers of a CNN on heterogeneous PEs such as CPU, GPU, and neural processing unit (NPU).
  - The devised methodology considers the practical issues, including dynamic voltage frequency scaling (DVFS) and CPU hot-plug.
  - The devised scheduling framework is the first to schedule multiple deep learning applications on heterogeneous processing elements with processing sharing, considering both data-level parallelism and task-level parallelism.
  - The devised methodology is verified by running CNNs on two different embedded devices: a Galaxy S9 smartphone [7] and a Hikey970 board [8].
- We devise two multi-bank on-chip memory management (MOMM) techniques for NPUs.
  - We define the multi-bank on-chip memory management problem.

- We devise heuristic compile techniques to solve the multi-bank on-chip memory management problem with two different objectives: one is to minimize the off-chip memory access volume, and the other is to minimize the processing delay caused by unhidden DRAM accesses.
- We extend the multi-bank on-chip memory management techniques to consider layer fusion that aims to reuse feature maps between layers.

1.3 Dissertation Organization

The rest of the dissertation is organized as follows: Chapter 2 introduces our target hardware platforms and reviews the optimization methods which is used in this dissertation. We devise two optimization methodology for a commodity hardware platform, which is presented in Chapter 3. Chapter 4 contains our multi-bank on-chip memory management techniques for a deep learning accelerator. Finally, we summarize the devised methodology and techniques and discuss some future works in Chapter 5.

Chapter 2

Background

2.1 Target Hardware

2.1.1 Commodity Hardware Platform

Commodity hardware is a device that is generally compatible with other devices of its type. In other words, it can be replaced with other devices. Some level of abstraction is required in order to enable multiple devices to be interchangeable, and the level of hardware abstraction determines the coverage of the hardware platforms. The higher the level of abstraction, the more hardware platforms are covered, but more restrictive optimizations can be applied. The lower the level of abstraction, the fewer hardware platforms are covered, but more optimizations can be applied.

C language is a general-purpose language and is also the most used language [33]. Several languages such as python are emerging, but there are still many embedded devices that only support the C language. Therefore, in this dissertation, in order to explore several optimization techniques without hardware dependency, the coverage of target commodity hardware platforms is devices that can be programmed with C language. Chapter 3 explains how to optimize the hardware platforms that supports C language.

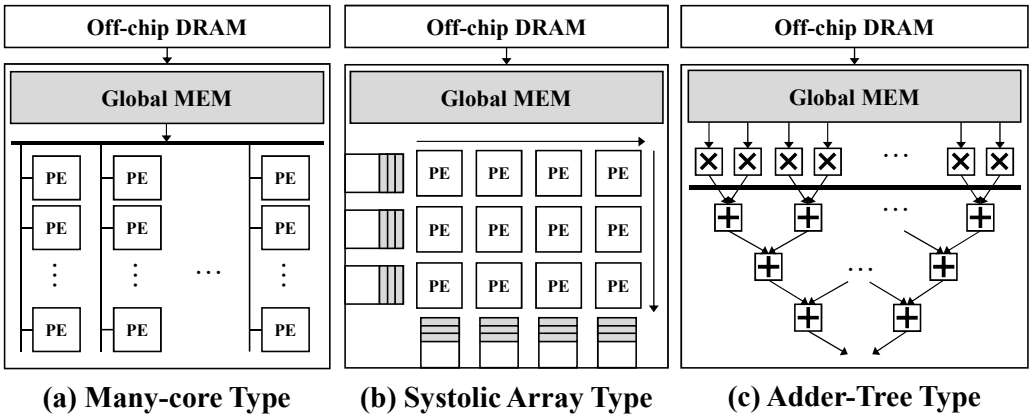


Figure 2.1: Architecture types of CNN accelerators or NPUs

2.1.2 Application-specific Hardware Accelerator

Application-specific hardware accelerators, also known as domain-specific hardware accelerators, are hardware computing engines that are customized for a specific application domain [34]. The application-specific hardware accelerator targeting deep learning applications, which is the target application for this dissertation, is called a neural processing unit (NPU). Recently, Extensive research on NPU has been performed to overcome the high computational demand and energy consumption.

Convolution neural network (CNN) accelerators can be categorized into three types based on the dataflow architecture to parallelize convolution operation as shown in Figure 2.1. An NPU of many-core type uses many processing elements (PEs), and each PE computes a single multiply-accumulate (MAC) operation per cycle with local buffers that store the assigned filter weights and input feature segments [16, 30]. The second type is to use a systolic array architecture of MAC units to accelerate matrix multiplication in a pipelined fashion [17]. Lastly, in the adder-tree type of an NPU [23, 24, 18, 19], each PE contains a set of multipliers and an adder tree to accumulate the multiplication results to produce a partial result of convolution for an output pixel.

In all architecture types, Off-chip access DRAM contention to the global on-chip

memory may occur among three accesses; reading from or writing to the off-chip DRAM, copying input feature segments and filter weights to local buffers in the datapath, storing the output results computed from the datapath. In Chapter 4, we explain how to avoid such access contention.

2.2 Convolutional Neural Network

2.2.1 Convolution

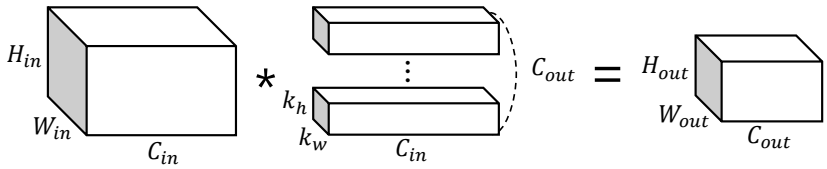


Figure 2.2: A convolution layer in a CNN

A convolution neural network, which is widely used for image applications such as image classification and object detection, consists of a sequence of convolution layers and activation and/or pooling layers between two consecutive convolution layers, as illustrated in Figure 2.2. The input feature map to a convolution layer is represented as 3-D data that consists of C_{in} channels of a 2-D image: each channel corresponds to a specific feature of the 2-D image. The 2-D image size is set to $W_{in} \times H_{in}$. The input feature map is convolved with a set of filters, each of which consists of C_{in} channels of kernel whose size is $k_w \times k_h$. Convolution with a filter produces one channel of output 2-D image whose size is $W_{out} \times H_{out}$, and the total number of channels in the output image is C_{out}.

2.2.2 Optimization Methods for Convolutional Neural Network

The subject of this dissertation is hardware-aware software optimization, which improves the performance of the entire system by scheduling the hardware resources associated with the performance bottleneck and applying other application domain optimiza-

tions. In this section, three optimization areas used in this thesis are introduced.

2.2.2.1 Approximate computing

Low-rank approximation In a software implementation of a convolution neural network (CNN), a 3-D convolution is computed by matrix multiplication. Low-rank approximation aims to minimize a cost function by approximating a given matrix with another matrix with a lower rank. By reducing the rank of a matrix, we can reduce the required computational and memory resources. While this technique has been widely used for mathematical modeling and data compression, it has only recently been adopted for deep learning as an effective approximated computing method, often coupled with matrix decomposition techniques [15].

Quantization Quantization is a technique that reduces the bit width of data representation as much as possible while preserving the accuracy within a given bound. Quantization has been extensively researched for the design of hardware accelerators for DNNs since smaller bit widths lead to simpler computations as well as smaller memory requirements [14]. The key technique is to train the bit-reduced weights carefully in the training stage. A too-large reduction can lead to significant degradation of accuracy; What level of reduction can guarantee a certain accuracy for a given network is still an open problem.

2.2.2.2 Parallelism

A deep learning inference algorithm is usually specified with an acyclic task graph where a node represents a computation task, called a layer, and an arc represents the dependency between two end nodes. Since a task or a layer contains massively parallel computation inside, a task can be mapped to multiple processing cores to exploit *data-level* parallelism of a task or *intra-layer* parallelism. On the other hand, multiple tasks that have no dependency on each other can run in parallel to exploit *task-level* parallelism of

the algorithm or *inter-layer* parallelism.

2.2.2.3 Data Reuse

Although the on-chip memory capacity of most deep learning accelerators is limited, the parameters of deep learning networks are getting bigger and bigger in order to increase the accuracy of deep learning applications. Therefore, even though the off-chip access is slow and gives a significant impact on energy consumption, it is inevitable to access off-chip memory in the process of computing a deep learning network. To reduce the amount of off-chip DRAM, data reuse techniques are recently proposed.

There are two types of data reuse techniques. One is to reuse data in a single layer by applying loop interchange and loop tiling techniques. The other technique is to reuse data between two adjacent layers by storing the previous layer's output in on-chip memory and using it when processing the next layer.

Chapter 3

Optimization for a Commodity Hardware Platform

In addition to the traditional system optimization techniques exploiting several parallelisms in the application, many techniques for optimizing convolution neural networks (CNNs) have recently been studied. In this chapter, we devise two optimization methodologies for how to optimize CNN applications on commodity hardware platforms. The first is a systematic optimization methodology that applies application domain optimizations and traditional system optimizations. To validate our methodology, we implement a C-code generation framework called C-GOOD (C-code generation framework for optimized on-device deep learning) based on the methodology. The other is the study of how to further exploit several parallelisms in CNN applications on the real embedded device. To exploit all parallelisms in CNN applications, we devise an end-to-end scheduling methodology, from profiling on real embedded devices to verifying the schedule results on the devices.

The remainder of this section is organized as follows. After the proposed systematic optimization methodology is described in Section 3.1 based on experience in the Low Power Image Recognition Challenge (LPIRC) [1, 2], Section 3.2 describes the C-code generation framework based on the methodology. The scheduling method for heterogeneous processors is described in Section 3.3. Section 3.4 discusses the related work on

these two methodologies.

3.1 Joint Optimization Method of Multiple Objectives

As there is a need to perform deep learning applications on embedded devices, lightweight deep learning networks and software optimization methods have been proposed. To encourage researchers to tackle the problems, the Low Power Image Recognition Challenge (LPIRC) [1, 2] was started in 2015 as an annual competition. It is designed to evaluate the trade-off between accuracy and energy consumption. The final score of LPIRC is computed as the ratio of the mean average precision (mAP) over the total energy consumption (Wh). The mAP is a widely used accuracy index for object detection problems such as ILSVRC [35].

This section introduces how to build the image recognition system that won the first prize in the LPIRC 2017. Among three conflicting goals of accuracy, speed, and energy consumption, we considered the trade-off between accuracy and speed first to select NVIDIA Jetson TX2 as the hardware platform and Tiny YOLO as the image recognition algorithm. Next, we applied the existent software optimization techniques systematically in sequence. To increase the throughput performance and balance the utilization of processing elements, pipelining is first applied to the network. After pipelining, we applied two optimization methods, Tucker decomposition, and 16-bit quantization, to improve the GPU performance. To speed up the CPU computation, we parallelized the post-processing task with multi-threading. After all software optimizations are performed, we considered the speed and energy consumption trade-off. To achieve the best score, we explored the operating frequencies of CPU and GPU. At last, we tested our implementation with a similar environment as the on-site competition. In this step, additional optimization was performed to reduce the overhead incurred in the test server. As a result, we could achieve an accuracy of 0.24 mAP with energy consumption of 2.08Wh in LPIRC 2017, which corresponds to the score of 0.11931, 2.7 times higher than the

winner of 2017.

3.1.1 Hardware Platform

Since there was no specialized neural processor available to us, we compared commercial embedded platforms in terms sof maximum performance. Table 3.1 compares the peak performance of the GPU subsystem that is the main computing engine in each device.

Table 3.1: Characteristics of candidate embedded devices

Platform	GPU	Chip	Clock	GFlops
Odroid XU4	Mali-T628 MP6	Exynos 5422	533MHz	102.4
Galaxy S8	Mali-G71 MP18	Exynos 8895	546MHz	371.2
Galaxy S8 Active	Adreno 540	Snapdragon 835	710MHz	567
Jetson TX1	Maxwell Cores x 256	Tegra X1	1000MHz	512
Jetson TX2	Pascal Cores x 256	Tegra Parker	1465MHz	750

Odroid XU4 is an embedded device using Exynos5422 that is equipped with a quad-core A15, a quad-core A7, and a Mali-T628 MP6 GPU. Exynos 8895 and Snapdragon 835 are the chipsets used in smartphones, where the Mali-G71 MP18 is the GPU on the Exynos 8895, and the Adreno 540 is the GPU on the Snapdragon 835. Jetson TX2 is an embedded AI computing device made by NVIDIA. It has a 2GHz quad-core ARM A57, a dual-core Denver2, and a Tegra GPU with 256 Pascal CUDA cores running at 1301MHz. It shows 1.46 times better performance than its predecessor, Jetson TX1 that was used in the winner of LPIRC 2016.

Based on the comparison result, as Jetson TX2 shows the best peak performance, it was selected as the hardware platform. Another benefit of using Jetson TX2 is the existence of CUDA and cuDNN library. Since the main kernels are already optimized in the cuDNN library, it is easy to implement deep learning networks on the device. On the other hand, for the other devices in Table 3.1, it is necessary to develop custom

Table 3.2: Performance comparison among object detection models for ImageNet Detection Dataset

Model (Framework)	FPS	mAP	Predicted mAP
SSD (Caffe)	3.5	0.43	0.045
YOLOv2 (Darknet)	5.81	0.51	0.087
Tiny YOLO (Darknet)	17.4	0.32	**0.167**

deep learning kernels with OpenCL since there is no available DNN specific library like cuDNN.

3.1.2 Deep Neural Network and Software Framework

Image recognition, also known as object detection, is a problem that finds all objects and the associated bounding boxes that contain the objects in a given set of test images. There are several neural networks proposed recently for image recognition. They are largely divided into two approaches. One approach separates the detection of the bounding boxes from the image classification, composing the network into two stages. In the first stage, a CNN is used to extract the features and generate region proposals. Image classification and box prediction in each region are performed in the second stage. Faster R-CNN [36] and R-FCN [37] are two examples.

On the other hand, several networks have been recently proposed to take the other approach that unites the region proposal and image classification in a single sequence of layers. YOLO [21], Tiny YOLO [11], and SSD [38] are popular examples of this approach. These networks are selected as the candidate networks because they promise higher performance without accuracy loss than the first approach [21, 38].

We compared three candidate networks in terms of frame per second (FPS) performance and mAP accuracy. The comparison result is summarized in Table 3.2. Since the challenge uses the training data from ImageNet and the test images are "ImageNet-like," the networks were retrained with the ImageNet dataset and evaluation was also conducted

with the ImageNet test dataset [39]. The first column lists the name of each network and the associated software framework that runs the network on Jetson TX2. As shown in the second and third columns of Table 3.2, YOLOv2 dominates SSD in both performance and accuracy. However, there is no dominance relation between YOLOv2 and Tiny YOLO. Since there is a time limit in the challenge, the accuracy was re-computed by considering the time limit, which is listed in the last column, named **Predicted mAP**. For instance, YOLOv2 can process $5.81 \times 60 \times 10$ images in 10 minutes, which corresponds to 17.4% of 20,000 images so that the Predicted mAP becomes 0.51×0.17 as the estimated mAP of each algorithm in given 10 minutes. As a result, Tiny YOLO that gives the highest value of Predicted mAP was chosen as the base network.

While many deep neural networks have one or more fully connected layers, Tiny YOLO is a fully convolutional network (CNN) that consists of 9 convolution layers and 6 max-pooling layers without the fully connected layer. Table 3.3 shows the structure of Tiny YOLO. The output tensor from the last layer includes all predictions of objects and bounding boxes. Since some predictions are not accurate, the post-processing step, called non-maximum suppression (NMS), is supposed to be run on the CPU side, which selects meaningful ones with some threshold values.

Note that the baseline Tiny YOLO is included in the Darknet framework [40]. The mAP accuracy and the FPS performance shown in Table 3.2 were obtained by running the Darknet on Jetson TX2. Since it is an open-source framework written in C, we could modify the Darknet directly to apply the software optimization techniques to the network. For example, we inserted some codes for performance profiling of each layer. To accurately measure the time spent in each layer, *cudaDeviceSynchronize()* API is used to isolate the executions between layers. Also, the time spent was measured using the *clock_gettime()* API provided by the standard C *time* library.

Table 3.3: Tiny YOLO layer information

Type	Size/Stride	# of In/Out Channels	Output Size
Convolution	3×3 / 1	3 / 16	416×416
Maxpool	2×2 / 2	16 / 16	208×208
Convolution	3×3 / 1	16 / 32	208×208
Maxpool	2×2 / 2	32 / 32	104×104
Convolution	3×3 / 1	32 / 64	104×104
Maxpool	2×2 / 2	64 / 64	52×52
Convolution	3×3 / 1	64 / 128	52×52
Maxpool	2×2 / 2	128 / 128	26×26
Convolution	3×3 / 1	128 / 256	26×26
Maxpool	2×2 / 2	256 / 256	13×13
Convolution	3×3 / 1	256 / 512	13×13
Maxpool	2×2 / 1	512 / 512	13×13
Convolution	3×3 / 1	512 / 1024	13×13
Convolution	3×3 / 1	1024 / 1024	13×13
Convolution	1×1 / 1	1024 / 1025	13×13

3.1.3 Software Optimization Techniques

The overall flow of the proposed methodology is shown in Figure 3.1. After the hardware platform and the neural network were selected in the first two steps, we applied a sequence of software optimization techniques from step 3 to step 6.

To improve the throughput performance, we first pipelined the network, aiming to overlap the CPU and GPU operations maximally. Afterward, other optimization tech-

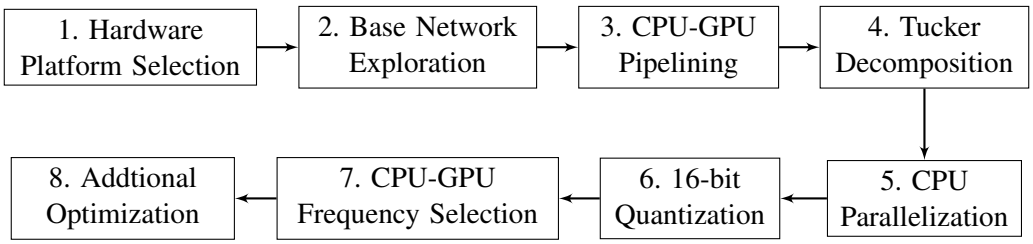


Figure 3.1: Overall flow of the proposed optimization methodology

Table 3.4: Step-by-step Performance Improvement Results

Step	Running Time	FPS	Improvement ratio
Baseline	1150s	17.4	1.0×
Pipelining	660s	30.3	1.74×
Tucker	540s	37.0	
Selective-Tucker	530s	37.7	1.25×
CPU Parallelization	502s	39.8	1.06×
16-bit Quantization	502s	39.8	
Selective-Quantization	460s	43.5	1.09×

niques were performed to reduce the bottleneck time based on the profiled execution times of CPU and GPU. To reduce the GPU execution time, we applied two techniques, Tucker decomposition, and 16-bit quantization. To reduce the CPU execution time, we parallelized the CPU computation by multi-threading.

After software optimization was performed, we explored the CPU and GPU frequencies to minimize the energy consumption of the system in step 7. At last, we emulated the end-to-end test scenario to check if the proposed solution performs as expected. During this final test, we identified some unexpected bottlenecks and so applied an additional optimization to solve the problems.

Table 3.4 summarizes the performance improvement from each optimization step. In the rest of this section, software optimization techniques are explained in detail.

3.1.3.1 CPU-GPU Pipelining

While the CNN structure of Table 3.3 represents the main algorithm, an image recognition system has a pre-processing step that fetches an input image from the disk sequentially, and a post-processing step, non-maximum suppression (NMS) step, as explained in the previous section. Both steps are performed on the CPU side. In the baseline Darknet implementation of Tiny YOLO, the pre-processing step and the other two steps are pipelined as shown in Figure 3.2 (a). As the first step of optimization, we added

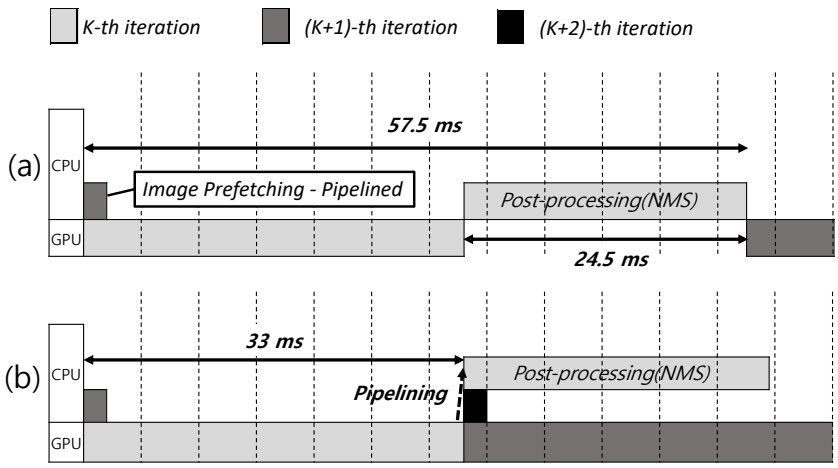


Figure 3.2: Illustration of the CPU-GPU pipelining procedure

another pipeline stage between the second and third steps to overlap the CPU and GPU operations as illustrated in Figure 3.2 (b). To this end, we modified the Darknet by adding a buffer between these two steps.

Table 3.5 displays the profiled information on the execution times on CPU and GPU. By simply pipelining the second and third steps, we could reduce the inference time from 1150 sec to 660 sec (42.6%) since the post-processing step takes a significant amount of execution time.

Table 3.5: Execution time comparison between the baseline and the pipelined network

	Inference Time	**CPU Time**	**GPU Time**
Baseline	1150 sec	488 sec	656 sec
Pipelined	660 sec	488 sec	656 sec

3.1.3.2 Tucker Decomposition

Since GPU is the bottleneck in the pipelined network, it was necessary to reduce the execution time of the CNN. Since convolution layers are the most time-consuming and require large memory space, several approximate computing methods have been developed

to reduce the computation time and memory requirements as explained in Section 2.2.2.1. Tucker decomposition is one of such methods we adopted in the proposed solution [15].

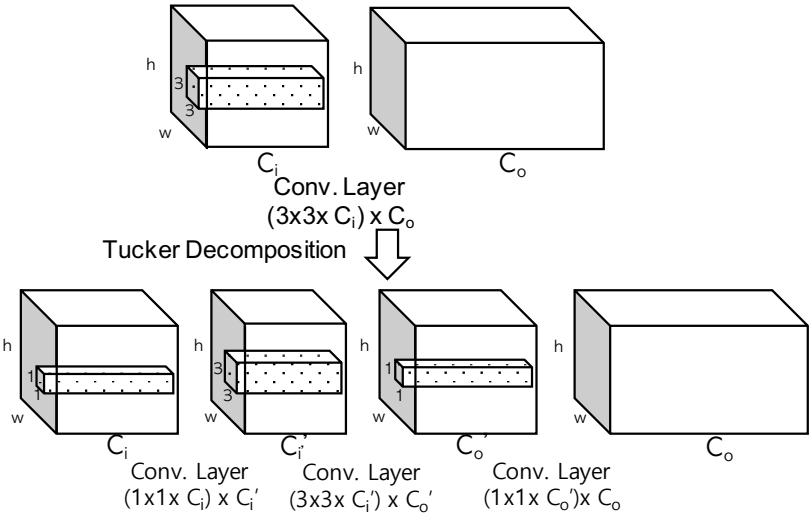


Figure 3.3: Tucker Decomposition

Figure 3.3 shows how one convolutional layer with 3×3 kernel size, C_i input channels, and C_o filters can be decomposed to three small convolution layers that include two 1×1 convolution layers and one 3×3 convolution layer by Tucker decomposition. The top figure represents the original convolution that convolves the 3-D input matrix ($h \times w \times C_i$) and C_o filter matrices of size $3 \times 3 \times C_i$ to produce an output matrix ($h \times w \times C_o$). The number of multiplications involved in this convolution is $3 \times 3 \times C_i \times C_o$ per pixel. As shown in the bottom figure, the weight matrix is decomposed into three [1] matrices with lower ranks and the original convolution is replaced with three smaller convolutions that are cascaded. The matrix size and the number of filters are depicted in the figure where C_i' and C_o' are smaller than C_i and C_o respectively. This decomposition reduces the total number of multiplications to $C_i \times C_i' + 3 \times 3 \times C_i' \times C_o' + C_o' \times C_o$. The memory requirements to save the filter matrices are also reduced by the same ratio.

In this technique, speed and accuracy trade-off can be adjusted by two variables, C_i'

[1] If C_i or C_o is small, the weight matrix is decomposed into two matrices by merging the first or the last with the middle convolution, respectively.

Table 3.6: Parameters for tucker decomposed layers

Layer	C_i	C_o	C_i'	C_o'
Conv2	16	32	16	20
Conv3	32	64	20	32
Conv4	64	128	32	64
Conv5	128	256	64	128
Conv6	256	512	128	256
Conv7	512	1024	256	512
Conv8	1024	1024	512	512

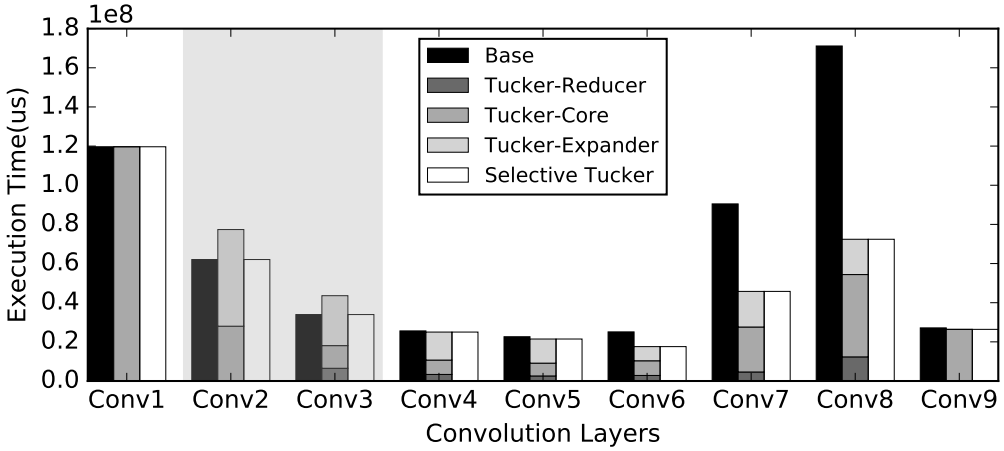


Figure 3.4: Execution time profile of convolution layers before and after Tucker decomposition

and C_o'. As a rule of thumb, we set C_i' and C_o' to the half of C_i and C_o, respectively, as depicted in Table 3.6 except two layers, Conv2 and Conv3. Since the number of input channels is small in the Conv2 layer, we omit the first 1×1 convolution in Tucker decomposition.

With Tucker decomposition, we could reduce the total inference time to process 20,000 images from 660 sec to 540 sec as shown in the fourth row of Table 3.4, which is smaller than the time limit of 10 minutes.

From the layer-level profiling of execution times, it was observed that Tucker decomposition is not always beneficial. As shown in Figure 3.4, three convolution layers

from Conv2 to Conv3 become slower if Tucker decomposition is applied. Thus rather than applying Tucker decomposition to all convolution layers, we selectively applied Tucker decomposition to the layers. As a result, we could reduce the execution time to 530 seconds with the selective-Tucker method as reported in the fifth row of Table 3.4. From the profiling, however, we observed that the post-processing step, the NMS step, became the performance bottleneck that limits the speed improvement only by 10 seconds after the selective Tucker is applied. Thus, we moved to the next optimization step of CPU parallelization.

3.1.3.3 CPU Parallelization

Algorithm 1 Non-Maximum Suppression in Tiny YOLO network

```
1:  procedure NMS(classes, predictions, thrs)
2:      for c in classes do
3:          t ← predictions[c]
4:          for each p in t do
5:              if p.conf < thrs.conf then
6:                  remove p
7:          for each (p1, p2) : p1.conf > p2.conf in t do
8:              if IOU(p1, p2) > thrs.iou then
9:                  remove p2
10:         print each element of t
```

Algorithm 1 shows the pseudo-code of the NMS step where the main loop is applied to each object class (line 2), which gives an opportunity of parallelization. Among predictions made from the CNN, it first invalidates predictions whose confidence value is smaller than a threshold (line 4 to 6). Then redundant predictions are removed (line 7 to 9). The intersection of union (IOU) is a metric to quantify how the proposed bounding box is close to the ground truth bounding box [1]. Finally, the remaining predictions are valid ones.

Since the Jetson TX2 platform has a multi-core CPU, we parallelized the NMS step by using the OpenMP library [41]. In consequence, the CPU execution time of the

NMS step was greatly reduced the total inference time from 530 sec to 502 sec and GPU became the performance bottleneck again.

3.1.3.4 16-bit Quantization

The next optimization method was to reduce the precision of data representation from 32-bit floating-point to 16-bit floating-point, which is denoted as 16-bit quantization. While aggressive quantization has been extensively researched in the design of specialized neural processors, there is no benefit of using smaller bit widths than 16-bit in Jetson TX2 in the computation time.

Jetson TX2 supports 16-bit data type called *fp16* and *fp16* data type should be used with the *fp16* APIs, including the math function APIs and basic operation APIs such as addition, subtraction, multiplication, and division. While the cuDNN library supports 16-bit precision in Jetson TX2, we had to re-implement the GPU kernels that are directly implemented by the Darknet with *fp16* APIs for 16-bit Quantization. Also, the input feature maps and the filter weights must be converted into *fp16* precision.

Since quantization reduces the bit width of data only, it was expected that it is beneficial in all convolution layers. Thus we applied *fp16* operations to all Tiny YOLO layers. Interestingly, as can be seen in the seventh row '16-bit quantization' of Table 3.4, no speed-up was obtained. From layer-level profiling, it was observed that the first three convolution layers became slower after 16-bit quantization as shown in Figure 3.5,

To find the cause of this unexpected result, we examined the profiling data obtained by the *nvprof* profiler provided by NVIDIA for the 1st and the 8th convolution layers which showed the two largest differences between *fp16* and float operations. Table 3.7 summarizes some data obtained from the *nvprof* profiler. The number cache accesses are decreased with *fp16* operations in both layers as expected since the same cache line contains twice more data with *fp16* types. The profiling result revealed that the performance largely depends on the number of instructions. At the first convolution layer, the *fp16*

Table 3.7: Profiling results for two convolution layers before and after 16-bit quantization

	Conv1		Conv8	
	FP16	**Float**	**FP16**	**Float**
L2 Cache Read	419 K	858 K	2,247 K	5,109 K
L2 Cache Write	173 K	346 K	16 K	58 K
Total Number of Instructions	30 M	12 M	48 M	43 M

kernel executes 2.5× more instructions than the 32-bit kernel, which overshadowed the benefit of reduced cache access in the execution time. On the other hand, there is a small difference at the 8th convolution layer. Thus the reduced cache access results in the execution time reduction. It seems that the cuDNN library provides more optimized kernels for 32-bit float data than the kernels for *fp16* data.

Hence, to optimize the overall network, we applied the *fp16* operation selectively from the fourth convolution layer and the inference time was shortened from 502 sec to 460 sec (9.4%).

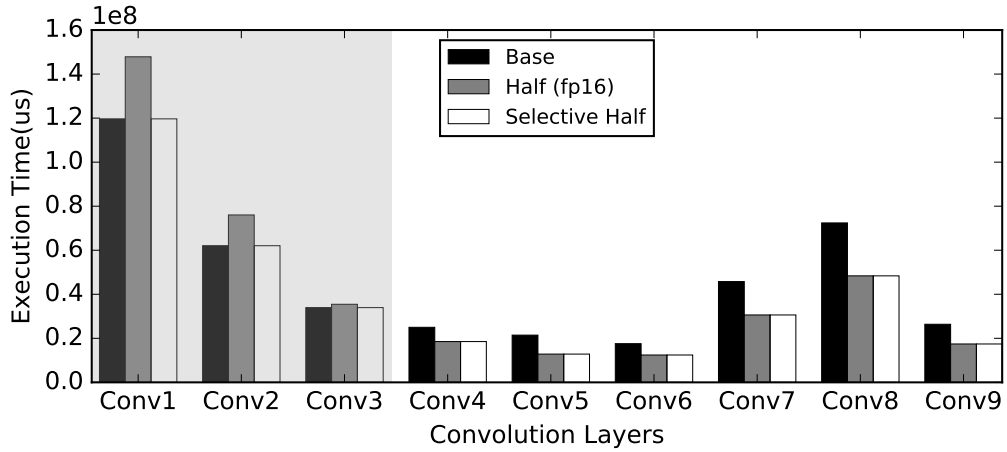


Figure 3.5: Execution time profile of convolution layers before and after 16-bit quantization

Table 3.8: CPU-GPU frequency exploration results

CPU Freq.	GPU Freq.	Exec. Time(s)	Energy Consumption(Wh)
1,114MHz	1,122MHz	58	0.180682
1,728MHz	1,122MHz	52	0.184183
1,728MHz	944MHz	59	0.184728
1,114MHz	1,301MHz	55	0.192547
1,728MHz	1,301MHz	49	0.195594
2,035MHz	1,122MHz	52	0.196636
2,035MHz	944MHz	58	0.196706
2,035MHz	1,301MHz	48	0.205224

3.1.3.5 CPU-GPU Frequency Selection

After a sequence of software optimizations, the total inference time is reduced to 460 seconds, which means that we could finish 140 seconds earlier than the time limit. So we could use the spare time to minimize the energy consumption further by adjusting the operating frequencies of processing elements. If we decrease the frequency, it will increase the execution time, but decrease the power consumption. Since the energy is the integral of power consumption until the completion time, we may reduce the total energy consumption by lowering the frequency of processing elements, particularly if a processing element is under-utilized.

Table 3.8 shows the measurement results on the execution time, and energy consumption consumed to process 2,000 images with various pairs of CPU and GPU frequencies. Note that the table includes the pairs of frequencies only that can meet the time limit (60 sec). Energy consumption was measured with the WT310E power meter in Table 3.8 and inference is performed with local images to exclude the network delay. From this exploration of frequencies, the selected frequencies for CPU and GPU are 1,114MHz and 1,122MHz, respectively.

3.1.3.6 Additional Optimization

After finishing all optimizations, we tested the server-client program used in the challenge. To reduce the communication counts between the device and the server, we sent one packet of detection results for a group of 1000 images at once. It was observed that the server took about 15 seconds to process one packet. To solve this problem, We increased the confidence threshold from 0.01 to 0.1 in the NMS module to reduce the number of valid predictions, which reduced the size of the packet and so the processing time of the server.

3.2 C-code Generation Framework for Optimized On-device Deep Learning

The methodology obtained through the experience of participating in the Low Power Image Recognition Challenge (LPIRC) [1, 2] is described in Section 3.1. In this section, we build a C-code generation framework that helps users build a deep learning system using the methodology explained.

3.2.1 C-GOOD Framework

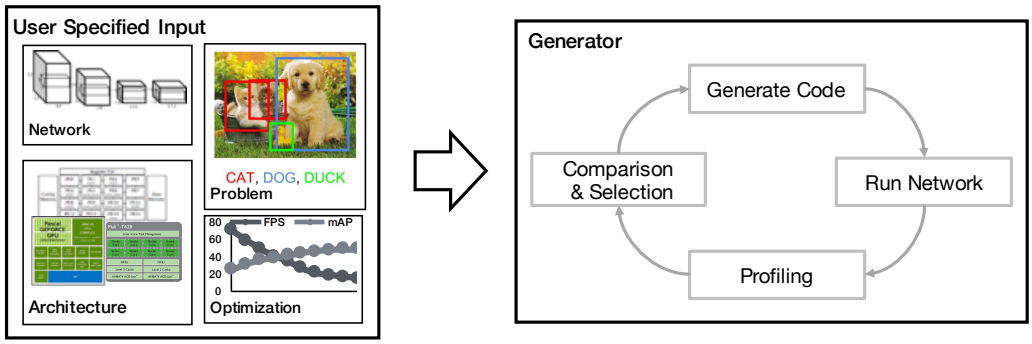


Figure 3.6: Overview of C-GOOD

Figure 3.6 shows the process of optimizing a deep learning network on a given embedded device in the C-GOOD framework. First, a user specifies a network configuration, a problem description, a platform description, and an optional optimization description. Based on this input information, C-GOOD generates a code, compiles the generated code, and performs layer-wise profiling during execution. Then the user determines which optimization technique will be used by comparing the execution times before and after a specific optimization technique is applied. Unless the user specifies the order of optimization techniques to be applied, a default order is followed in the proposed methodology. Finally, it generates an optimized C code for the given network and hardware platform.

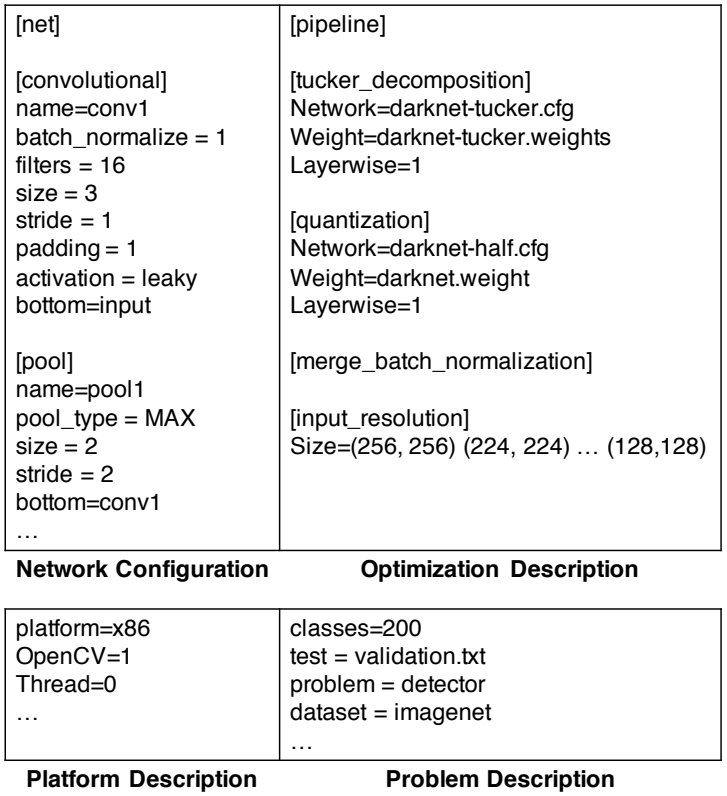


Figure 3.7: An example of input information

3.2.1.1 Input Information

The input information of C-GOOD consists of one optional input file, an optimization description file, and three necessary input files for network configuration, problem description, and platform description. An example is shown in Figure 3.7. Since it is assumed to use the Darknet framework for training a deep neural network, C-GOOD uses almost the same input file format as Darknet for network configuration and problem. However, the platform description file and the optimization description file are additional input files that Darknet does not use.

Network Configuration The network configuration file represents how layers are composed in a deep learning network, and specifies the parameters of each layer such as

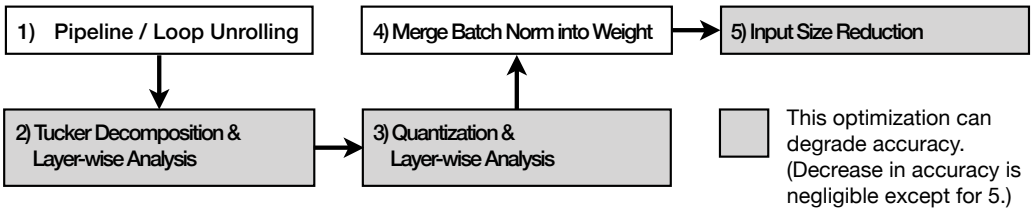


Figure 3.8: Overview of the software optimization flow

kernel size, stride, padding, and a flag indicating whether or not batch normalization technique is used.

Problem Description The problem description file contains information about the problem. It includes the name of a dataset such as ImageNet, VOC, or Cifar, the type of the dataset such as Detection or Classification, and the path of a text file that contains the path of test images.

Platform Description The platform description file describes the HW platform that the user wants to use. This file includes the architecture of the device, whether it supports thread programming and a specific library such as OpenCV and CUDA, and so on.

Optimization Description The optimization description file is an optional file that selects the optimization techniques and specifies the order of application. The optimization techniques supported in the current implementation include loop unrolling, pipelining, Tucker composition, quantization, merging batch normalization into weights, and input size reduction. If the user does not provide this file, C-GOOD applies optimizations in the default order proposed in the optimization methodology depicted in Figure 3.8. In the example of Figure 3.7, C-GOOD proceeds pipelining first and Tucker decomposition, quantization, merge batch normalization, and input resolution reduction in series.

```
1 void main()                  void inference()                  void forward_layers()
2     load_weight();              for each test image               conv1_forward();
3     init_layers();                frontend(image);                pool1_forward();
4     set_workspace();              forward_layers();                     ...
5     inference();                  backend();                      conv8_forward();
6     wrap_up();                                                    pool4_forward();
```

(a) (b) (c)

Figure 3.9: Baseline C-code (pseudo code) that is platform-independent

3.2.1.2 Code Generation & Optimization Methodology

Figure 3.8 shows the default optimization procedure in the proposed software optimization methodology. For a given network, a baseline code without any optimization can be generated as shown in Figure 3.9. After allocating a workspace, or memory space, for the network in *set_workspace()*, *inference()* is called. After inference for all images is completed, *wrap_up()* frees dynamically allocated data structures. In *inference()*, we first read an image in *frontend()*, perform inference through multiple layers in *forward_layers()*, and produce results in *backend()*. Now we explain how the code is modified after each optimization technique is applied.

Pipelining For embedded devices with heterogeneous processing elements, pipelining can improve the throughput performance of the network. The network can be partitioned into three stages: input preprocessing, network inference, and post-processing[2] of the last layer's output feature map. By making the hardware accelerator or GPU take the network inference stage and letting the CPU cores perform the other two stages, we can increase the throughput performance.

To apply pipeline technique to the front-end, we make *frontend_thread* and some changes in *inference*. Figure 3.10 (a) shows how the code is changed in *inference()* to apply pipelining at the front end, which is highlighted by red texts. A separate thread,

[2]In the final output feature map, each proposed answer is encoded as a tensor. The post-processing includes a decoding of such data and selecting the final answers.

```
1  void inference()
2      front_thread_init();
3      for each test image
4          while(front_flag);
5          forward_layers();
6          front_flag= 1;
7          backend();
```

(a)

```
void frontend_thread()
    n = 0;
    while(n < NUM_IMAGE)
        while(!front_flag);
        read_inputs(input_buf);
        front_flag=0;
        n++;
```

(b)

```
1  void inference()
2      front_thread_init();
3      back_thread_init();
4      for each test image
5          while(front_flag);
6          forward_layers();
7          while(back_flag);
8          front_flag = 1;
9          back_flag=1;
```

(c)

```
void backend_thread()
    n = 0;
    while(n < NUM_IMAGE)
        while(!back_flag);
        processing_outputs();
        back_flag=0;
        n++;
```

(d)

Figure 3.10: Code modification by applying pipeline

called *fronend_thread*, is initialized and run in *frontend_thread_init()*. Two concurrent threads are synchronized by an array of shared flags. A similar modification is made for back-end pipelining as illustrated in Figure 3.10 (c) and (d).

Loop Unrolling [42] GEMM (General Matrix Multiplication), which is a popular function in convolution computation, consists of a 3-nested loop inside. In some architectures, loop unrolling is an effective way to increase the utilization of processing elements for effective parallel processing. Figure 3.11 shows how the loop body of *gemm_nn()* can be unrolled when the unrolling factor of each loop is set to 2 and the total unrolling factor is 8. When unrolling factors are set, unrolling code is generated through three processes: making as many temporary variables as the total unrolling factor to store the partial result, generating the unrolled loop body with temporary variables, and adding extra code if the number of iterations is not a multiple of unrolling factor.

Tucker Decomposition As a low-rank approximation technique, we use Tucker decomposition as explained in Section 3.1.3.2. Since the Tucker decomposition transforms

```
1 void gemm_nn (float *A, float *B, float *C)
2    for (i=0; i<I; i++)
3        for (j=0; j<J; j++)
4            for (k=0; k<K; k++)
5                C[J*i + j] += A[K*i + k] * B[J*k + j];
```

```
1 void unrolled_gemm_nn (float *A, float *B, float *C)
2    float c0, …, c7;
3    for (i=0; i<I-1; i+=2)
4        for (j=0; j<J-1; j+=2)
5            for (k=0; k<K-1; k+=2)
6                c0 = A[K*(i+0) + (k+0)] * B[J*(k+0) + (j+0)];
7                c1 = A[K*(i+0) + (k+1)] * B[J*(k+1) + (j+0)];
8                c2 = A[K*(i+0) + (k+0)] * B[J*(k+0) + (j+1)];
9                c3 = A[K*(i+0) + (k+1)] * B[J*(k+1) + (j+1)];
10               c4 = A[K*(i+1) + (k+0)] * B[J*(k+0) + (j+0)];
11               c5 = A[K*(i+1) + (k+1)] * B[J*(k+1) + (j+0)];
12               c6 = A[K*(i+1) + (k+0)] * B[J*(k+0) + (j+1)];
13               c7 = A[K*(i+1) + (k+1)] * B[J*(k+1) + (j+1)];
14               c0 += c1; c2 += c3; c4 += c5; c6 += c7;
15               C[J*(i+0) + (j+0)] += c0;
16               C[J*(i+0) + (j+1)] += c2;
17               C[J*(i+1) + (j+0)] += c4;
18               C[J*(i+1) + (j+1)] += c6;
```

Figure 3.11: Unrolled GEMM Code

a convolution layer into a cascade of two or three convolutional layers, it is necessary to change the network configuration input file but without code change. Since it is reported that Tucker decomposition is not beneficial at all times [43], C-GOOD provides a performance profiling function at each layer. In the current implementation, the user is supposed to apply Tucker decomposition repeatedly with a set of candidate dimension values of decomposed matrices (C_i' and C_o' in Figure 3.3) and find the best dimension values among the candidates for each layer. It will be a future work to automate this process to find the best values of C_i' and C_o'.

Quantization C-GOOD gives users the option to use 16-bit data instead of 32-bit data, which is denoted as quantization optimization. Since quantization does not improve performance in all cases like the Tucker decomposition case, C-GOOD performs layer-wise profiling before and after quantization is applied to decide in which layers quantization is beneficial.

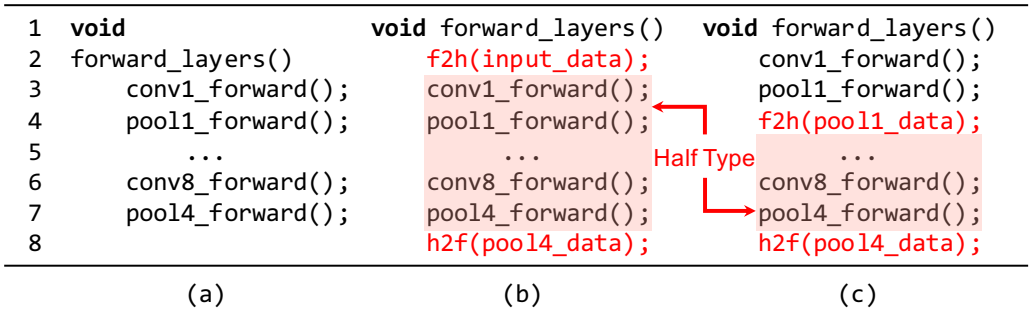


Figure 3.12: Code changes for applying quantization and layer-wise quantization (f2h is a function that converts float data to half and h2f do vise versa.)

Figure 3.12 (b) and (c) show two examples that apply quantization to different regions of layers. For 16-bit quantization, we need to convert the input data to 16-bit before the first convolution layer of the region and convert the result data to 32-bit data after the last convolution layer of the region. Note that we use different libraries or function definitions for 16-bit data at the lower level, which is not shown in the figure.

Merge Batch normalization Batch normalization is a technique to normalize the input data to have unit variance and zero mean. It is known that normalization stabilizes the learning process and accelerates learning. Since batch normalization is a linear transformation, it can be merged with the previous convolution layer by adjusting the filter weights at compile-time. It reduces the overall network execution time.

Input Resolution Reduction Since the computational complexity and the memory space for activation data are proportional to the image size, it is advantageous to reduce the input image size as much as possible while accuracy loss is tolerable. If the objective function is defined as accuracy per power consumption, reducing the image size is advantageous. If the user specifies the input size, the input image size is adjusted accordingly by a front-end thread that performs image resizing.

3.2.2 Experiments

We evaluate the effectiveness and viability of C-GOOD with three widely different hardware platforms: the Jetson TX2 [3], the Odroid XU4 [4], and the Samsung Reconfigurable Processor (SRP) platform [5]. Jetson TX2 is an embedded AI computing device made by NVIDIA. It consists of a quad-core ARM A57 CPU, a dual NVIDIA Denver core, and a Pascal GPU that includes 256 CUDA cores. It is currently the most powerful embedded device supported by the cuDNN libraries. Odroid XU4 consists of a quad-core ARM A15, a quad-core ARM A7, and a Mali-T628 MP6 GPU. Since the platform does not support CUDA, the proposed framework includes an OpenCL-backend to use the Mali GPU.

The SRP is a coarse-grained reconfigurable array processor that has 16 heterogeneous processing elements and supports 32bit floating-point calculations with a peak performance of 4GFlops. It contains a 320KB on-chip data SRAM. Since the array processor can be configured as a single VLIW processor for the processing of sequential programs, it can run the full DNN algorithm. However, the SRP does not run an operating system, and there is no support for libraries. Thus we need to generate a standalone C-code that can be compiled for the array processor by the SRP compiler. Note that this platform cannot run any existent DNN platform such as Caffe2 [44], Tensorflow lite [45], and even Darknet [40].

3.2.2.1 Performance Comparison

Since the Darknet framework can be run directly on Jetson TX2, we first compare the performance between the Darknet framework and the code generated by C-GOOD. For comparison, we use three neural networks supported by the Darknet framework: YOLOv2 [21] for object detection and the Darknet19 [20] and DenseNet [46] for image classification, both with the ImageNet dataset. Table 3.9 displays the speed comparison result and Table 3.10 compares the memory usage of two frameworks.

Table 3.9: Speed comparison with Darknet (Unit: FPS)

Problem	**Networks**	**Darknet**	**Ours**	**Piped**[3]
ImageNet DET	YOLOv2 [21]	5.81	6.39	7.11
ImageNet CLS	Darknet19 [20]	21.26	23.37	32.44
ImageNet CLS	DenseNet201 [46]	12.39	17.45	21.03

Darknet reads multiple images at once by multithreading for object detection, which is similar to pipelining in its effect on the performance while no such technique is used for image classification. The table shows both the FPS measured without pipelining and the FPS with front-end pipelining only in C-GOOD. For object detection, the **Piped** code generated by C-GOOD should be compared with Darknet for a fair comparison. It is observed that C-GOOD shows 22.4% higher performance. For image classification, C-GOOD is 9.9% faster than Darknet without pipelining for Darknet19 and 40.8% faster for DenseNet201. If front-end pipelining is applied, the performance improvement is increased to 52.6% and 69.7%, respectively. Because both approaches use cuDNN for convolution layers, the gain is mainly attributed to network-specific code generation and skipping operations for the reshape layer which changes the shape of the feature map. When concatenating feature maps from two or more previous layers, the code generated by C-GOOD allocates memory more efficiently.

Table 3.10: Memory usage comparison with Darknet (Unit: MB)

Problem	**Networks**	**Darknet**	**Ours**	**Piped**[3]
ImageNet DET	YOLOv2 [21]	861	761	777
ImageNet CLS	Darknet19 [20]	655	649	675
ImageNet CLS	DenseNet201 [46]	734	657	687

Memory footprint is an important metric for an embedded device that has a limited size of memory. Table 3.10 displays the maximum resident set size (RSS) value of the process, measured by using *ps* command. For object detection, the **Piped** code generated

by C-GOOD uses 84MB less memory than Darknet. This gain comes from two reasons. First, C-GOOD optimizes the memory usage for the reshape layer. Second, there is a difference in pipelining implementation between C-GOOD and Darknet: darknet uses four threads to read input, but C-GOOD uses two threads. It is observed that Darknet19 for image classification shows a similar size of memory for both C-GOOD and Darknet since Darknet19 does not include a reshape layer nor does it contain multithreading for reading input. In the case of DenseNet201 that includes 98 reshape layers, C-GOOD uses 77MB less memory than Darknet. Even with pipelining, it uses 47MB less memory.

Table 3.11: Memory usage and speed when using Caffe2

Problem	**Networks**	**Speed (Unit:FPS)**	**Memory (Unit:MB)**
ImageNet CLS	DenseNet121 [46]	13.89	1,061
ImageNet CLS	DenseNet201 [46]	7.69	1,459

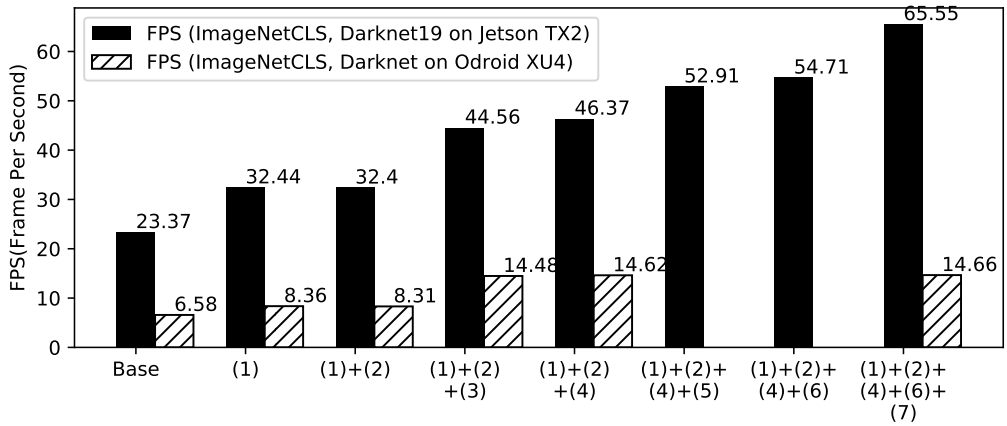
Caffe2 [44] was recently introduced as a deep learning framework for embedded systems and is becoming popular. Table 3.11 shows the memory usage and speed for the Caffe2 framework with two networks, DesneNet201 and DenseNet121, on the same Jetson TX2 board. DenseNet121 model is obtained from the Caffe2 official github [47] and DenseNet201 is manually converted from the Caffe model [48]. For DenseNet201, it is observed from Table 3.9 that C-GOOD performs more than twice better than Caffe2 in terms of speed. Because YOLOv2 and Darknet19 have a layer not supported by Caffe2, a comparison with those networks could not be made.

3.2.2.2 Jetson TX2

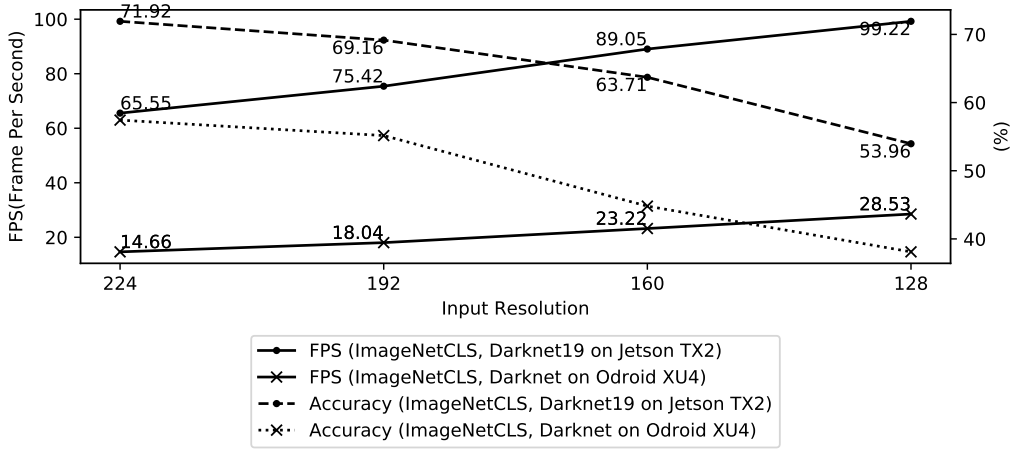
The next set of experiments is conducted by applying the proposed software optimization procedure, using C-GOOD, to a Jetson TX2 platform. Figure 3.13(a) shows how the computation speed improves as the optimization process is applied to the Darknet19 network for image classification. In the figure, the horizontal axis indicates the

Table 3.12: Applied optimization techniques

No.	Technique	No.	Technique
(1)	Frontend Pipelining	(5)	Quantization
(2)	Backend Pipelining	(6)	Layer-wise Quantization
(3)	Tucker Decomposition	(7)	Merge BatchNorm
(4)	Layer-wise Tucker	(8)	Loop Unrolling



(a) Performance change with application of optimization methods



(b) Performance and accuracy change with decreasing input's resolution

Figure 3.13: Restuls of methodology application in two different devices results for image classification using the C-GOOD framework

optimization methods added from the origin and the vertical axis indicates the FPS performance. Each number represents an optimization technique shown in Table 3.12. As an optimization method is added, the performance is improved, and after all optimizations are applied, the performance reaches a 2.8 $\times$ improvement compared to the baseline implementation that is identical to that shown in Table 3.9.

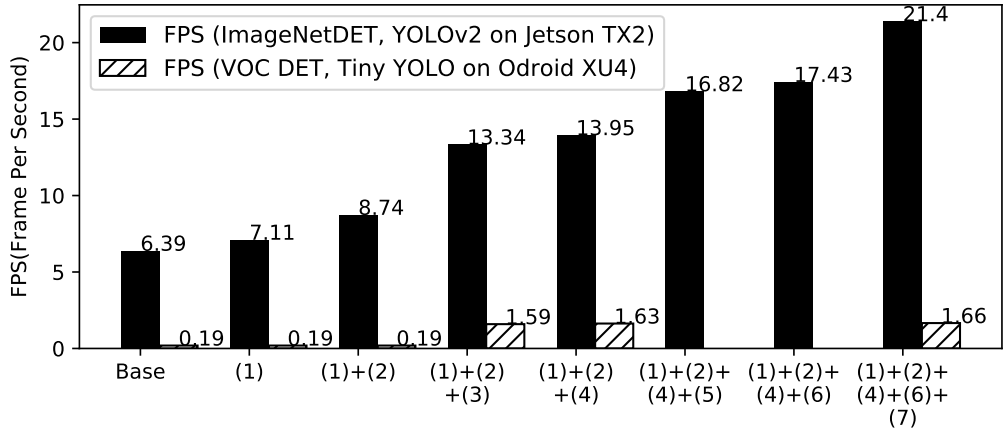
It can be observed that the Tucker decomposition yields the largest improvement and layer-wise exploration produces additional gains for both Tucker decomposition and quantization. There is no gain with backend pipelining because the workload of postprocessing is negligible. Figure 3.13(b) shows the trade-off between accuracy and speed as the input image size varies. The speed increases almost linearly with the input size while the accuracy drops rather abruptly when the input size decreases too much.

A similar experiment is conducted with YOLOv2 for object detection. As shown in Figure 3.14(a), we were able to achieve significant performance improvements by 3.35 times compared with the baseline implementation. Figure 3.14(b) shows the trade-off between speed and accuracy as the input size is changed. We observe an outlier consistently when the input size decreases from 384 to 352, which needs further investigation in the future.

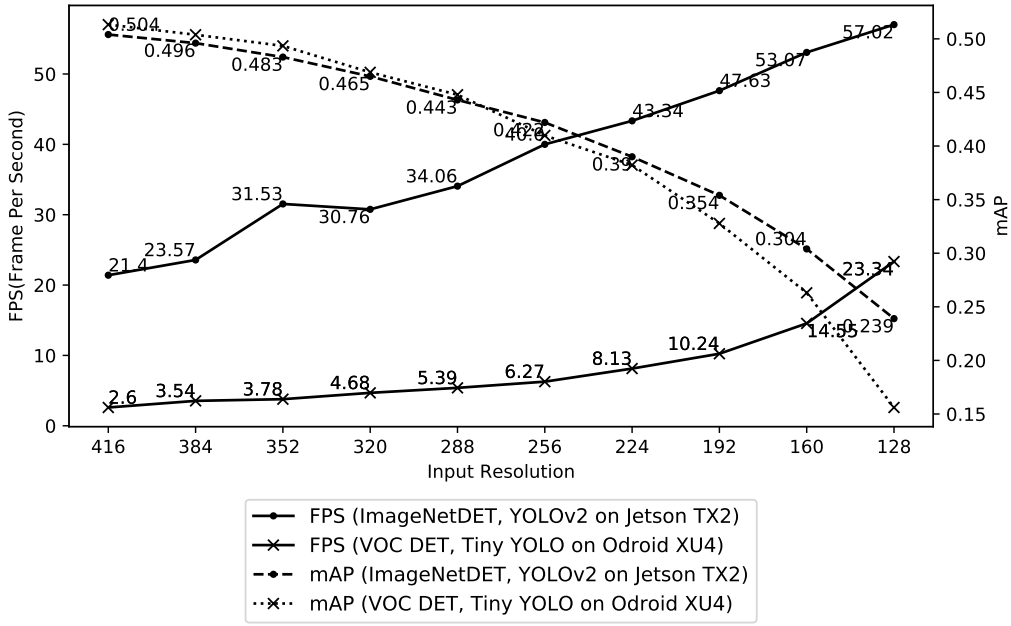
3.2.2.3 Odroid XU4

For Odroid XU4 that does not support CUDA, C-GOOD generates an OpenCL code, based on our own OpenCL kernels for the DNN layers. [4] Since the Odroid XU4 platform has much lower computation power than Jetson TX2, experiments are conducted with smaller networks: *Darknet* with the ImageNet dataset for image classification and *Tiny-YOLO* with the VOC dataset for object detection. The optimization flow presented in Section 3.2.1.2 is applied except quantization since the MALI-T628 GPU does not support 16-bit half-precision computations. Experimental results are displayed in the same

[4] Since the OpenCL kernels are not fully optimized, there is still room for improvement.



(a) Performance change with application of optimization methods



(b) Performance and accuracy change with decreasing input's resolution

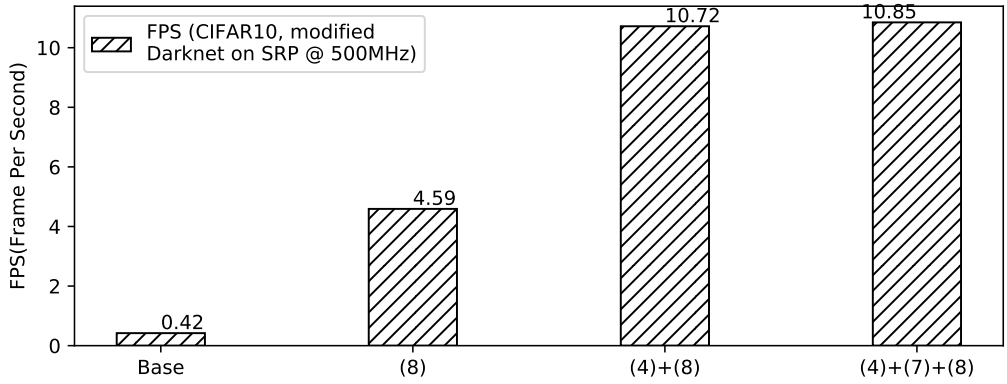
Figure 3.14: Restuls of methodology application in two different devices results for object detection using the C-GOOD framework

figures as those of the Jetson TX2 experiments from Figure 3.13(a) to Figure 3.14(b). Software optimization results in a 2.2× performance improvement for Darknet and an 8.7× improvement for Tiny-YOLO. Unlike the Jetson TX2 case, there is no performance gain achieved with pipelining since the CPU workload is negligible compared with the GPU workload for inferencing. The steep accuracy drop is observed at a larger input size than the Jetson TX2 case for image classification, but no difference is observed for object detection.

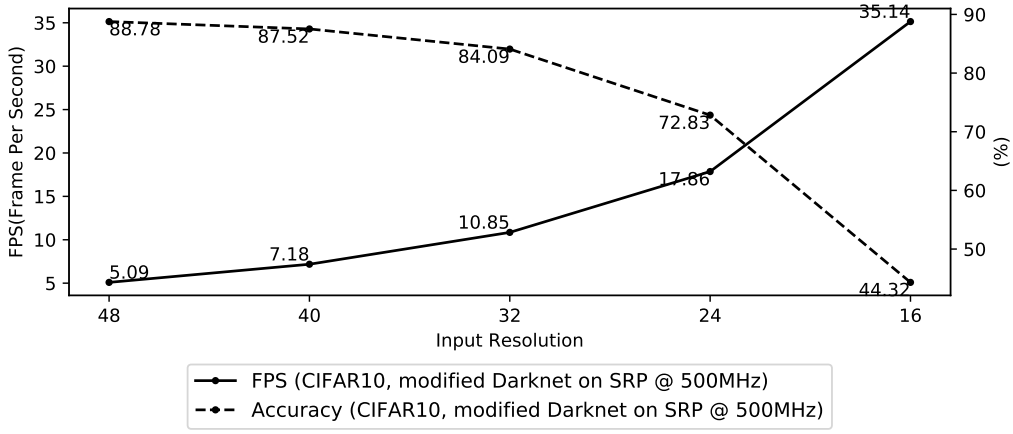
3.2.2.4 The Samsung Reconfigurable Processor

Since the SRP processor has limited computing power and on-chip SRAM size, we applied our methodology to Darknet [20] only for image classification. From the original Darknet network, three max-pooling layers are removed since the dataset is changed to CIFAR that assumes a smaller image size than ImageNet. Experiments are conducted on a cycle-accurate simulator instead of a real device since access to a development board was not available. Note that there exists no DNN framework that supports the SRP processor except the proposed framework.

Figure 3.15(a) shows performance improvements on the SRP when our methodology is applied. Because SRP does not support thread or 16-bit data type, we cannot apply the optimization techniques using threads and the quantization technique. The hardware-specific optimization step is applied as the first step to generate loop-unrolled C-code to ease compiler-optimization. These loop-unrolling compiler optimizations achieve a 10.93× performance improvement. Similarly to the other platforms, the speed of inference is improved by more than a factor of two by the Tucker decomposition in all layers, while the Merge-BN optimization does not raise performance noticeably. Experimental results for varying input sizes are shown in Figure 3.15(b). Again, we have to find a good compromise between accuracy and performance (or power) for the given objective function.



(a) Performance change with application of optimization methods for DarkNet



(b) Performance and accuracy change with decreasing input's resolution

Figure 3.15: Results of methodology application in the Samsung Reconfigurable Processor (SRP) using the C-GOOD framework

3.3 Scheduling Deep Learning Applications Onto Heterogeneous Processors

As the need for on-device machine learning is increasing recently, embedded devices tend to be equipped with heterogeneous processors that include a multi-core CPU, a GPU, and/or a DNN accelerator called a neural processing unit (NPU). In the scheduling of multiple deep learning applications in such embedded devices, there are several technical challenges. First, a task can be mapped onto a single core or any number of available cores. So we need to consider various possible configurations of CPU cores. Second, embedded devices usually apply dynamic voltage and frequency scaling (DVFS) to reduce energy consumption at run-time. We need to consider the effect of DVFS in the profiling of task execution times. Third, to avoid overheat condition, it is recommended to limit the core utilization. Lastly, some cores will be shut-down at run-time if core utilization is not high enough, in case the hot-plugging option is turned on. In this paper, we propose a scheduling technique based on Genetic Algorithm to run deep learning applications on heterogeneous processors, considering all those issues. First, we aim to optimize the throughput of a single deep learning application. Next, we aim to find the Pareto optimal scheduling of multiple deep learning applications in terms of the response time of each deep learningh application and overall energy consumption under the given throughput constraints of deep learning applications. The proposed technique is verified with real deep learning networks running on two embedded devices, Galaxy S9 and HiKey970.

The remainder of this section is organized as follows. First, in Section 3.3.1, we calculate the size of the search space for the problem we are dealing with in this scheduling work. The hardware platform and system model used are described in Section 3.3.2. In Section 3.3.3, we describe the overall flow and how to profile execution time and communication time of each layer on different PEs. After the scheduling technique for a single application is explained in Section 3.3.4, Section 3.3.5 describes how to extend our

scheduling framework to multi-application scheduling. Section 3.3.6 verifies our scheduling method by comparing the scheduling results with the results obtain from the actual implementation.

3.3.1 Search Space Size

Table 3.13: The number of possible solutions when scheduling a convolutional neural network to a device with 4 CPUs, 1 GPU, and 1 NPU

Network	# of Layers	# of all possible solution
SqueezeNet [10]	40	$4^2 \times 6^{38} \approx 5.9 \times 10^{30}$
MobileNet v1 [12]	31	$4^2 \times 6^{29} \approx 5.9 \times 10^{23}$
MobileNet v2 [49]	67	$4^2 \times 6^{65} \approx 6.0 \times 10^{51}$
DenseNet 40 [50]	64	$4^2 \times 6^{62} \approx 2.8 \times 10^{49}$

Let K be the number of layers in a deep learning network, and let the number of CPU, GPU, and NPU in an embedded device be N_c, N_g, and N_n. Then, the total number of solution is $2^{N_c} \times (K-2)^{(N_c+N_g+N_n)}$. The first term is the number of cases where the first and the last tasks are mapped to the CPU, and the second term is the number of cases where the other tasks are mapped to all PEs including CPU, GPU, and NPU. Table 3.13 shows the number of all possible solutions when scheduling a single CNN network to a device with 4 CPUs, 1 GPU, and 1 NPU. It can be seen that the results in this table are very large even though they are the number of possible solutions when scheduling a CNN network. In the case of scheduling multiple networks, it can be obtained by multiplying the number of possible solutions when scheduling each network. Therefore, there are so many possible solutions that we cannot perform an exhaustive search to find the optimal Pareto solution.

3.3.2 Hardware Platform and System Model

Since the proposed framework is applied to embedded devices, we first explain the characteristics of hardware platforms used in this work: Galaxy S9 smartphone and

HiKey970 board. Galaxy S9 is a heterogeneous system that consists of a Mali-G72 MP18 GPU and big.LITTLE CPUs with a quad-core M3 CPU running at 2.7GHz and a quad-core Cortex-A55 CPU at 1.79GHz. HiKey970 is also a heterogeneous system that consists of a Mali-G72 MP12 GPU and big.LITTLE CPUs with a quad-core A73 running at 2.36GHz and a quad-core Cortex-A53 at 1.8GHz. Besides, it has an NPU that can accelerate deep learning applications. We believe that our approach can be applied to other hardware platforms since it uses a black box model for each processing element (PE) with the profiled execution time and communication time at the task level without assuming a specific hardware architecture.

Since reducing the energy consumption is critical in mobile embedded devices, Galaxy S9 adopts an aggressive DVFS policy, called *schedutil*, which lowers the frequency and the voltage level of CPU cores if the average utilization of cores is below a pre-specified threshold. No DVFS policy is used in HiKey970. Galaxy S9 even shutdowns some cores dynamically using a CPU hot-plug feature supported by Linux. Since the overheating induces unexpected slow-down of an application, the maximum CPU core utilization is usually set to avoid such an unpleasant situation. We profile deep learning networks considering DVFS and hot-plug, and propose a mapping technique that can limit CPU utilization.

For software implementation of deep learning applications on ARM processors, ARM provides an open-source software development kit, called ARM NN [51]. Unfortunately, it does not support Galaxy S9. Moreover, it does not support the parallel execution of a deep learning network on a heterogeneous system. Thus partitioned deep learning applications need to be written manually using ACL [52] that contains OpenCL implementation of deep learning operations for GPUs and NEON [5] implementation for CPU. APIs for utilizing the NPU are more limited. Even though HiKey970 has an NPU inside, the software development environment is not open to the public so that the NPU could

[5]NEON is a SIMD architecture extension for ARM Cortex-A processors.

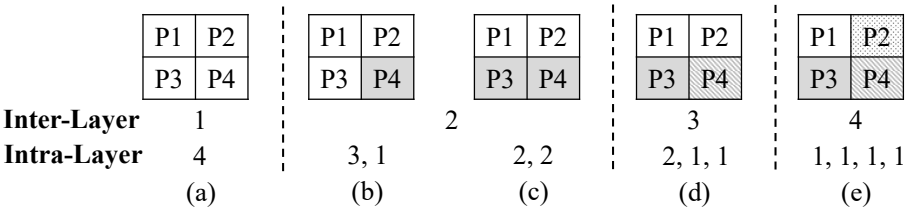


Figure 3.16: Five different CPU core configurations considering inter-layer parallelism and intra-layer parallelism

not be used to verify our schedule results in Section 3.3.6.

Most deep learning libraries and previous works consider a multi-core CPU as a single processor and exploit the data-parallelism of the mapped layer or task [6] using multi-threading. For instance, ACL executes a convolution layer either on a multi-core CPU or a GPU exploiting the data-parallelism in the layer to reduce overall inference time. In contrast, we model the quad-core CPU as a set of logical processing elements since multiple layers may run concurrently on different cores in the CPU. Figure 3.16 illustrates five different configurations that we can choose with four CPU cores. Each figure in Figure 3.16 shows that four CPU cores can be utilized differently with different combinations of *inter-layer* and *intra-layer* parallelism degree that is indicated by numbers below the figures. The degree of inter-layer parallelism indicates how many layers are executed concurrently, and the degree of intra-layer parallelism is given as a tuple that represents how many cores are assigned to the layers running concurrently. For example, Figure 3.16 (b) illustrates the case in which the degree of inter-layer parallelism is 2 and that of intra-layer parallelism is (3, 1), meaning that two layers are mapped on the CPU of which one layer is executed by 3 cores while the other layer by 1 core.

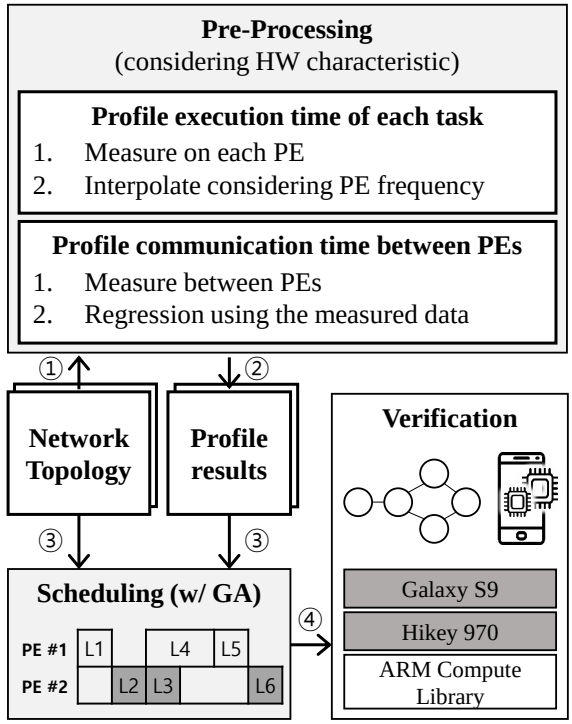


Figure 3.17: The proposed deep learning applications scheduling flow (The numbers in the small circles indicate the order of the scheduling flow.)

3.3.3 Proposed Scheduling Framework and Profiling

Figure 3.17 displays an overview of the proposed scheduling framework of deep learning applications on an embedded device. While the proposed methodology is applied to two specific embedded devices in this work, it is applicable to other embedded devices. Before a scheduling decision is made, it should be known how much time is taken to execute a task on each processing element (PE) and the communication overhead between two dependent tasks, which would vary if they are mapped to different PEs. Such profiles are obtained by running the deep learning application on each PE in the preprocessing step.

With the profiled task information, we perform static scheduling of deep learning applications on the heterogeneous processors in the given hardware platform to optimize

[6]In this scheduling work, we use the terms, task and layer, interchangeably.

a given objective function under the constraint on the CPU utilization. For a single deep learning application, the objective is to increase the throughput and to minimize energy consumption. The output of the framework is a set of Pareto optimal schedules, which includes mapping information of layers onto PEs.

When scheduling multiple deep learning applications, we assume that the period of each application is given as the throughput constraint. While we can use other objectives, the scheduling objective assumed in this paper is to minimize the response time of each application and the total energy consumption of the system. The deadline is set to be the same as the period. The framework finds a set of Pareto optimal schedules that meet the deadline of each application. Before we explain the proposed scheduling techniques in subsequent sections, we elaborate on the profiling and performance estimation technique in this section.

3.3.3.1 Task Profiling and Estimation

Profiling is performed with an in-house deep learning framework [53] that generates an OpenCL code for the Mali GPU and multi-threaded NEON code for the multi-core ARM CPU in the device using ACL. By adding a time-stamping code at each task (or layer) boundary, the elapsed time between any two points of interest can be measured. For the GPU, the kernel time can be measured by using OpenCL profiling APIs.

The GPU execution time is easy to obtain since OpenCL utilizes all GPU cores, and the task execution time does not vary in a deep learning application. On the other hand, CPU profiling of a task is tricky, with many issues that should be taken into account. First, the five different CPU configurations should be considered as discussed in Figure 3.16: a task should be profiled with a varying number of cores from 1 to 4, as shown in Table 3.14. For profiling with the different number of CPU cores, a Linux command *taskset* is used to assign specific CPU cores to a process.

Second, the CPU execution time should be adjusted since the embedded device may

Table 3.14: An example of profiling (network: SqueezeNet)

Layer	4 core	3 core	2 core	1 core	1 core*	GPU
c1	3,951	6,958	8,178	14,569	13,358	1,628
c9	734	1,205	1,362	2,253	1,755	536
c14	324	400	477	589	417	590

1 core* means that no ACL thread is created. (Unit: μs)

run a Dynamic Voltage Frequency Scaling (DVFS) governor that changes the CPU frequency depending on the utilization. Moreover, the Linux kernel may turn on and off a CPU core at run-time, supporting the CPU hot-plug feature. For example, if a task is executed on a single core and no other applications or processes are running on the other cores, Galaxy S9 would turn off the other cores to reduce the energy consumption and increase the CPU frequency up to the maximum of 2.7 GHz. If a task is mapped to all four cores, on the other hand, the DVFS governor would lower the CPU frequency to 1.79GHz to reduce the heat and power consumption. With two cores assigned, a task is run at the frequency of 2.31GHz. Since other processes may be running while the target deep learning applications are running, it is safe to assume that all CPU cores will be busy in reality. Thus the profiled CPU execution time is calibrated based on the observed CPU frequency, assuming that all cores are busy even though a task is not mapped onto four cores. For instance, the execution time profiled on a PE with two CPU cores has to be increased by $\frac{2.31}{1.79}$. This simple interpolation is based on the assumption that computation time will increase inversely proportional to the CPU frequency, which is a source of error between our profiling results and the actual measurements.

More detailed profiling of convolution layers is performed to examine the cause of varying speedup ratio. Table 3.15 shows the measured execution time of four kernels involved in each convolution layer for c1, c9, and c14. Note that a convolution operation is computed by GEMM (general matrix multiplication) after converting the input image to a suitable matrix. If a target device has four cores, the ACL creates three threads in addi-

Table 3.15: Intra-Layer Parallelism Analysis

Layers	Kernel	Computation Time	Sync Overhead
c1	Im2Col	182,341	18,449
	GEMM	623,818	215,803
	Col2Im	2,567,736	310,425
	ReLU	198,487	51,220
c9	Im2Col	25,097	32,840
	GEMM	95,607	15,418
	Col2Im	307,815	52,362
	ReLU	59,717	38,172
c14	Im2Col	30,455	30,312
	GEMM	111,703	6,195
	Col2Im	23,583	38,762
	ReLU	7,328	29,714

Unit: ns

tion to the main thread. The main thread first distributes the workload to the threads and computes the remaining workload. Then, it synchronizes with other threads waiting for them to be joined. In the table, *Sync Overhead* is this waiting time including the conditional wait API overhead. In c1 and c9, the synchronization overhead is small compared to the computation time. In c14, however, the overhead is comparable to or even larger than the computation time. It makes the CPU cores idle and hinders the performance improvement of c14 when intra-layer parallelism is four, as shown in Table 3.14.

It is noteworthy that, in some layers, *Col2Im* time is greater than the GEMM time. This is because the memory access pattern of *Col2Im* has very poor locality: the kernel mainly consists of memory operations, but the stride of the write operations is the 2-D output tensor size (width $\times$ height), and the size per access is only 4 Bytes.

3.3.3.2 Communication Overhead Profiling and Estimation

Communication overhead between two tasks should be considered in making a scheduling decision. In ACL, the input and output tensor buffers of each layer (or task)

should be defined statically. If two adjacent tasks are mapped onto the same PE, the output tensor buffer of a task can be shared with the input tensor buffer of its successor task, resulting in no communication overhead between them. If they are mapped onto different PEs, however, a buffer cannot be shared, but separate buffers are required for each tensor in order to run tasks in a pipelined fashion. Likewise, the two adjacent tasks that are mapped to different *logical* PEs among four cores in a CPU cannot share a tensor buffer. Data should be copied between separate buffers using *memcpy*. For the communication between CPU and GPU, *map* and *unmap* OpenCL APIs are used. The *map* API is used to access the data in the GPU memory address space from the CPU, while the *unmap* releases the mapping so that the mapped data can then be computed by the GPU.

Thus communication overhead is estimated differently depending on the types of communicating PEs. Communication time between different CPU PEs is equal to *memcpy* time, and the time from GPU to CPU is set to $(map + memcpy)$ time since the OpenCL maps GPU memory to the host and then copies the data to CPU. The time from CPU to GPU is set to $(map + memcpy + unmap)$ time since it has to map GPU memory to the host (CPU) in order to send data to GPU and then unmap it for the next layer's GPU processing.

To measure the overhead of those APIs, we made a micro-bench and ran it 1000 times, from which the averaged overhead was obtained. By changing the data size in the APIs, the overhead of *memcpy* on Galaxy S9 is approximated as $1.06 \times 10^{-2} \times (DataSize)^2 + 2.75 \times 10^{-1} \times (DataSize) + 1.70$, *unmap* as $9.97 \times (DataSize) + 526.39$, and *map* as $21.75 \times (DataSize) + 569.25$.

3.3.3.3 NPU Profiling and Estimation

Accurate profiling on an NPU could not be made since the API for executing a layer on an NPU is not available, which makes direct profiling impossible. Instead, performance estimation on the NPU in HiKey970 is made indirectly by assuming that the

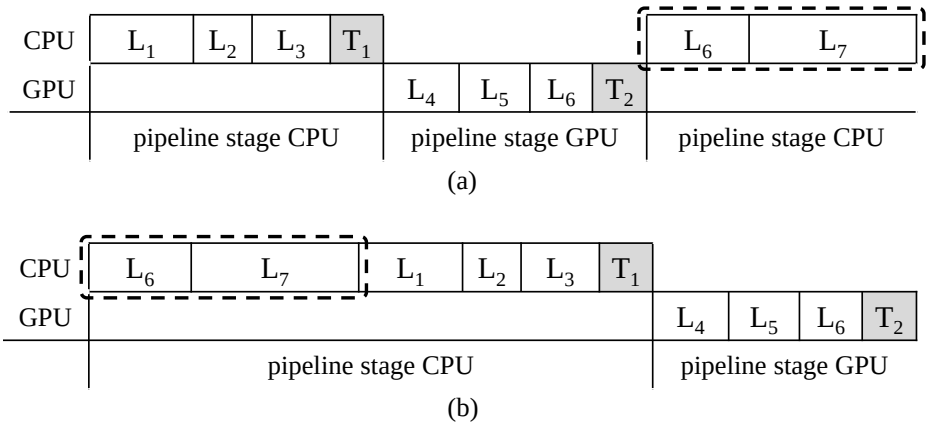


Figure 3.18: Task-clustering mapping of an application

NPU is about 6.5 times faster than that of the quad-core CPU based on the performance comparison reports in [54, 55]. Also, the communication time between CPU and NPU is assumed to be the same as the time between CPU and GPU. For energy consumption, a similar estimation is made with the comparison reported in [54, 55] since it is not possible to measure the power consumption of a task on each PE. As a result, the power consumption of CPU, GPU, and NPU is assumed to be 3.5W, 4W, and 2W, respectively.

3.3.4 Scheduling a Single Deep Learning Application

3.3.4.1 Baseline Task-clustering Scheduler

As a baseline scheduling method, we partition the DNN into a set of sub-networks, map them onto processors, and run them in a pipelined fashion. The intuition for this is to minimize the communication overhead, also keeping the code complexity low. The mapping problem is defined as the task-clustering problem in which the input, a deep learning application, is partitioned into a number of PEs. A processing element (PE) is assigned a single task-cluster only. One exception is allowed for a logical PE of CPU, as illustrated in Figure 3.18 (a) where the CPU is assigned two task-clusters: the first cluster and the last cluster. If we attach the scheduler of the second iteration and move

the cluster of one period, however, the schedule becomes as shown in Figure 3.18 (b) so that the restriction still holds. We implement this scheduler using ILP to find the best solution in this restricted solution space.

Let L = $\{L_1, L_2, \ldots, L_n\}$ be a set of layers, or tasks, in a deep learning application sorted in the topology order, and PE = $\{PE_1, PE_2, \ldots, PE_m\}$ be a set of logical PEs in the device[7]. E_i^j is the computation time of L_i on PE_j and $T_i^{j,k}$ is the communication time taken when transferring L_i's output from PE_j to PE_k.

There are three sets of constraints in the ILP formulation. First, a task is mapped onto only one PE. When M_i^j is a binary decision variable that indicates whether L_i is mapped to PE_j, the following constraint, expressed by Eq. (3.1), makes each task mapped onto only one PE. Let $pred(L_i)$ and $succ(L_i)$ be the set of preceding and succeeding layers of layer L_i, respectively.

$$\forall L_i \in L, \sum_{j=1}^{m} M_i^j = 1 \tag{3.1}$$

The second set of constraints is related to task-clustering mapping. We introduce two additional binary variables, dep_i^j and $Pipe_i^j$. The former indicates whether any predecessor of L_i is mapped on the PE_j. The $Pipe_i^j$ variable indicates whether a cluster or pipeline stage starts from layer L_i on PE_j. The definitions of dep_i^j and $Pipe_i^j$ are presented in Eq. (3.2) and Eq. (3.3), respectively:

$$dep_i^j = \begin{cases} 1, & \exists L_k \in pred(L_i),\ M_k^j = 1 \\ 0, & \text{otherwise} \end{cases} \tag{3.2}$$

$$Pipe_i^j = \begin{cases} M_i^j, & i = 1 \\ M_i^j \wedge \neg dep_i^j, & \text{otherwise} \end{cases} \tag{3.3}$$

[7] n is the number of layers in the deep learning application, and m is the number of PEs in the device.

With these two variables, Eq. (3.4) tells that only one pipeline stage is allowed for each PE, except the PE onto which the first and last stages are mapped.

$$\forall PE_j \in PE,\ \sum_{i=1}^{n} Pipe_i^j - (M_n^j \wedge M_1^j) = 1 \tag{3.4}$$

Lastly, we need to specify dependency constraints. Let $Start(L_i)$ be the start time of a task or layer L_i, $End(L_i)$ be the end time of L_i, and $m(L_i)$ be the index of the PE onto which layer L_i is mapped. Then, the following two constraints of Eq. (3.5) and Eq. (3.6) enforce all dependencies between tasks to be satisfied.

$\forall L_i \in L,\ \forall L_j \in succ(L_i),$

$$Start(L_j) \geq End(L_i) + T_i^{m(L_i),m(L_j)} \tag{3.5}$$

$\forall L_i \in L,\ \forall L_j \notin succ(L_i),\ j > i,$

$$Start(L_j) \geq End(L_i) - ((1 - M_i^{m(L_j)}) \times \infty) \tag{3.6}$$

Note that we aim to maximize the throughput of a deep learning application, which is determined by the longest cluster in the pipelined execution. Thus we define the cluster execution time, $CT(PE_j)$, as follows:

$$CT(PE_j) = \sum_{i=1}^{n} M_i^j \times (E_i^j + \sum_{L_k \in succ(L_i)} T_i^{j,m(L_k)}) \tag{3.7}$$

Then the objective function is to minimize the longest cluster execution time, which is presented as follows:

$$minimize(\max_{PE_j \in PE}(CT(PE_j)) \tag{3.8}$$

3.3.4.2 Proposed GA-based Scheduler

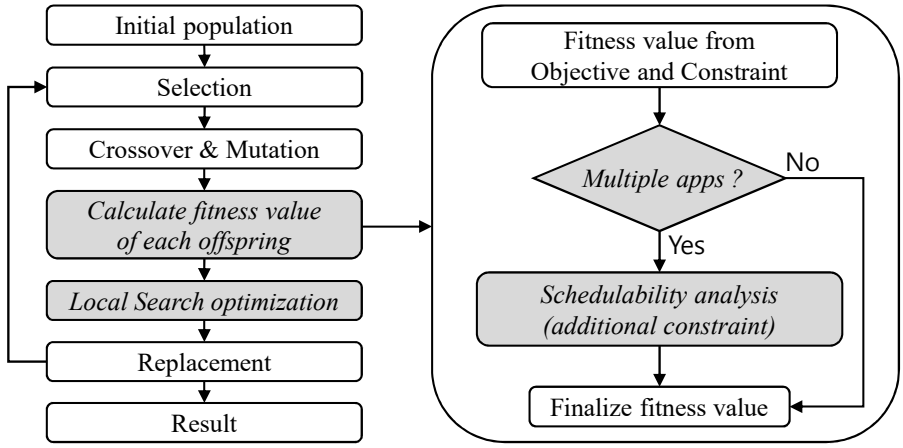


Figure 3.19: Overview of the proposed GA scheduler

Genetic algorithm (GA) is a widely used meta-heuristic inspired by evolutionary processes in nature, where a solution of the problem is encoded as a chromosome, and the fitter survives to the next generation, populating new chromosomes with operations such as crossover and mutation on their chromosomes. The proposed scheduling algorithm is displayed in Figure 3.19. The overall flow and population generation procedure is not much different from the standard GA algorithm.

Figure 3.20 shows an example of a chromosome configured to solve the problem using the proposed GA scheduler. A chromosome consists of an array, where each element represents a layer, and the number in the array indicates the PE onto which the layer is mapped. For example, the chromosome shown in Figure 3.20 indicates that the first layer is mapped on PE 0 (CPU PE) and the third layer is mapped on PE 4 (GPU PE),

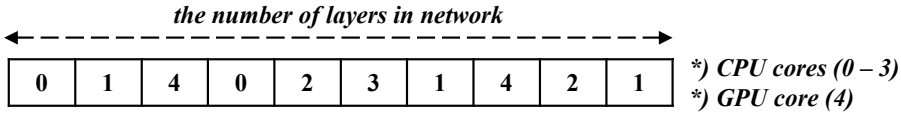


Figure 3.20: GA chromosome structure with PE configuration (1, 1, 1, 1)

and so on. Since encoding varies depending on the configuration of the multi-core CPU, we iterate this encoding with a different number of PEs (i.e., different maximum number in the array) for each possible configuration.

The objective of the proposed scheduling is to maximize throughput and to minimize energy consumption. The CPU utilization should not be higher than some threshold to avoid overheating. Thus we define the fitness function with the following three terms and let GA find the solutions with the minimal fitness value. The first term is the inverse of the throughput that is obtained by Eq. (3.9) [56].

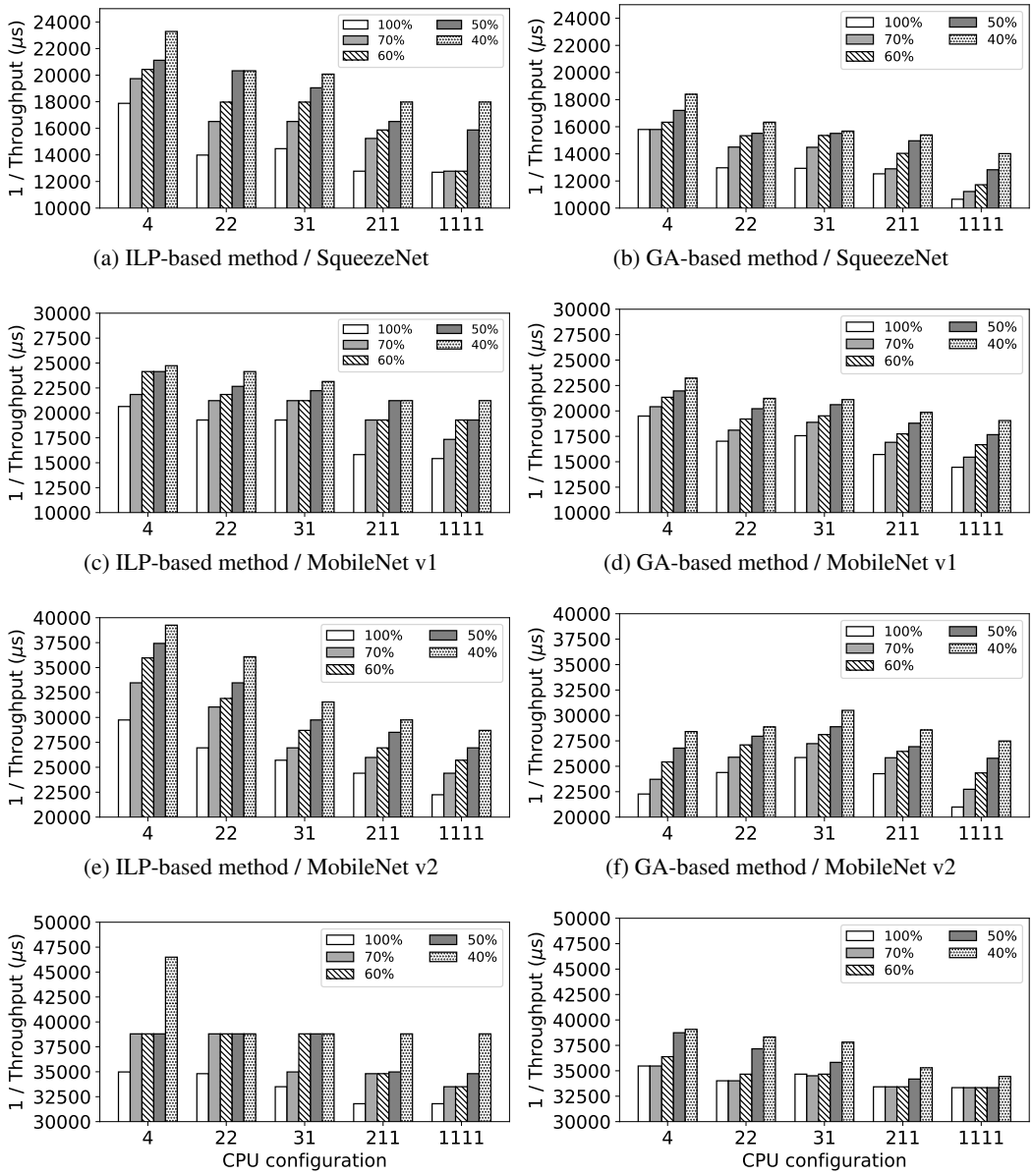
$$Throughput(graph) = \lim_{n \to \infty} \frac{n}{time\ to\ finish\ n\ iterations} \tag{3.9}$$

The second term is the total energy consumption, which is computed by multiplying the execution time of the mapped tasks on each PE and the power consumption of the PE. The third term is the penalty in case the CPU utilization is greater than a given utilization constraint.

3.3.4.3 Experimental Results

The proposed scheduler is implemented with DEAP [57] and SPEA2 [58] is used as the selection algorithm, which is known to perform well for multi-objective problems. MobileNet v1 [12], MobileNet v2 [49], SqueezeNet [10], and DenseNet-40 (k=32) [50] are selected as benchmark applications in this work.

Throughput Performance We first set the optimization criteria of single application scheduling to be the throughput performance ignoring energy consumption. Figure 3.21 shows the scheduling results of the GA-based scheduler and the ILP-based task-clustering scheduler. The x-axis represents the CPU configurations that determine how to exploit *intra-* and *inter-layer* parallelism on a CPU. The y-axis represents the average inference time of a single input image in μs. The throughput performance is the reciprocal of the



(a) ILP-based method / SqueezeNet

(b) GA-based method / SqueezeNet

(c) ILP-based method / MobileNet v1

(d) GA-based method / MobileNet v1

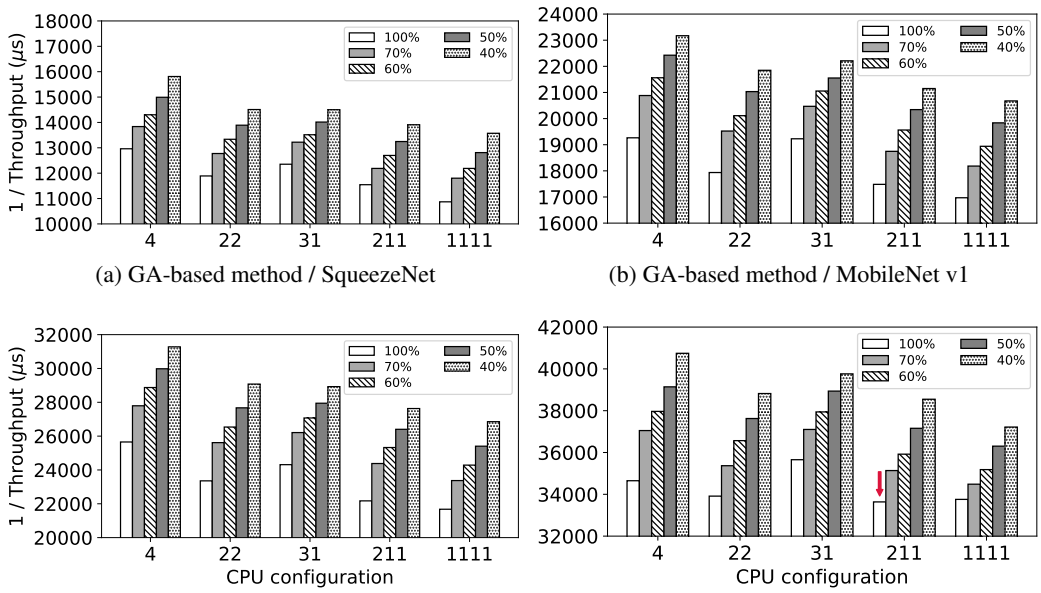
(e) ILP-based method / MobileNet v2

(f) GA-based method / MobileNet v2

(g) ILP-based method / DenseNet-40

(h) GA-based method / DenseNet-40

Figure 3.21: Comparison of the GA-based method and the ILP-based method (Device: Galaxy S9 / Unit: μs)



(a) GA-based method / SqueezeNet

(b) GA-based method / MobileNet v1

(c) GA-based method / MobileNet v2

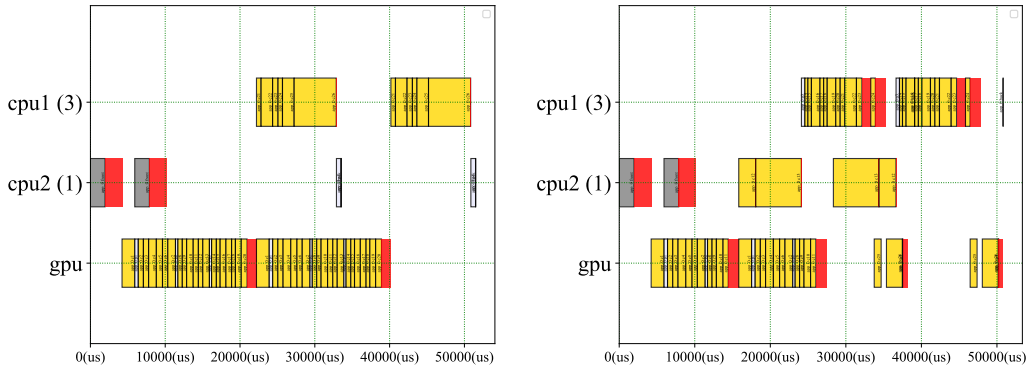
(d) GA-based method / DenseNet-40

Figure 3.22: Scheduling results from Hikey 970 (Unit: μs)

average inference time. Since the typical CPU utilization limit to avoid overheating is within the range of [40%, 70%], we set the constraint within the range and compare the result with the one without any constraint (100%). Note that the CPU utilization is the average utilization across the whole CPU cores.

From Figure 3.21, we make three observations. First, as the CPU utilization constraint gets tighter, the inference time increases and the throughput decreases, as expected. In Figure 3.21(g), the inference time does not decrease with a larger CPU utilization constraint, which is indicating that the GPU execution is the performance bottleneck.

Second, the throughput performance is the best with (1,1,1,1) PE configuration in most cases, which implies that exploiting *inter-layer* parallelism is more beneficial than exploiting *intra-layer* parallelism. From this observation, we decided to use (1,1,1,1) PE configuration for scheduling multiple applications, as will be explained in the later section. Figure 3.22 shows that (1,1,1,1) configuration also gives the best throughput performance for HiKey970. There is one exception in the proposed GA-based schedul-



(a) ILP-based mapping method

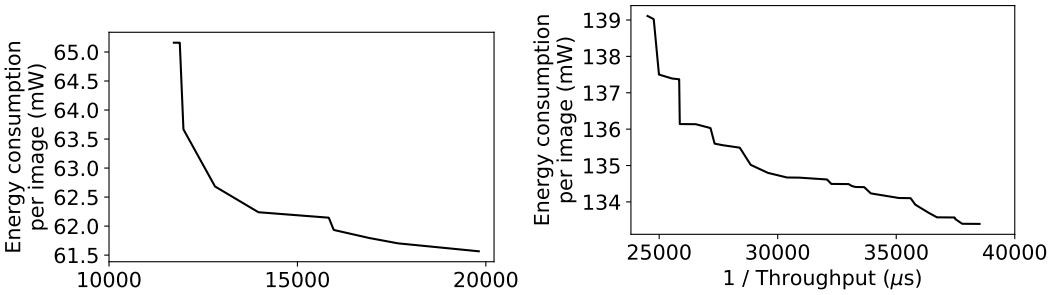
(b) Proposed mapping method

Figure 3.23: Scheduling results from two different methods

ing result: the inference time of (2,1,1) PE configuration, indicated by a red arrow, is smaller than (1,1,1,1) configuration when the utilization constraint is 100% as shown in Figure 3.22(d).

The third observation is that the GA-based scheduling outperforms the task-clustering scheduling in almost all cases. It is because the partitioning restriction imposed on the task-clustering method prunes the solution space too much. To understand the difference between the two methods, Figure 3.23 compares the results of scheduling methods for a single problem instance (SqueezeNet with (3,1) CPU configuration under 60% CPU utilization constraint). The figure shows that the GA-based method can better utilize the CPU cores than the task-clustering method.

Multi-Objective Scheduling As shown in Figure 3.24, the proposed GA scheduler generates Pareto optimal solutions in terms of throughput and energy consumption. Each Pareto optimal solution is associated with a different scheduling result. Thus a user can find the best trade-off from the results.



(a) SqueezeNet (b) MobileNet v2

Figure 3.24: Multi-objective scheduling results

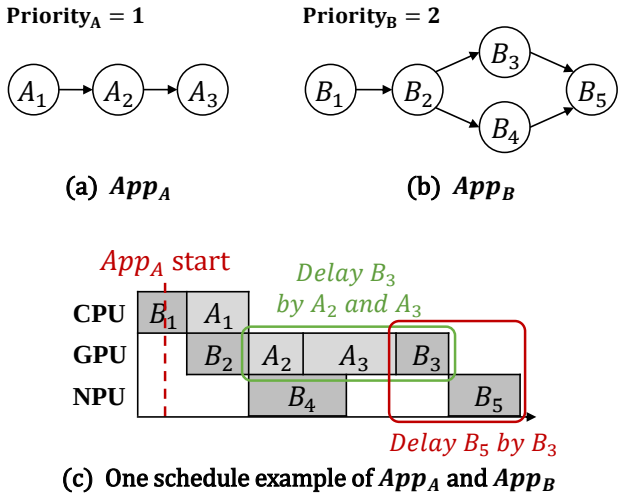


Figure 3.25: Multiple application scheduling example

3.3.5 Scheduling Multiple Deep Learning Applications

3.3.5.1 Schedulability Analysis

Since the schedule of an application affects the other applications, schedulability analysis can be performed only after the mapping and scheduling decisions are made. On the other hand, we need to consider schedulability when making mapping and scheduling decisions. The proposed GA-based scheduler solves this cyclic-dependency naturally in an iterative fashion.

Suppose that we map two applications on three heterogeneous processors as shown

in Figure 3.25 and App_A has a higher priority than App_B.[8] Figure 3.25(c) shows a scheduling example of App_A and App_B. The green box in Figure 3.25(c) illustrates that task B_3 in application App_B is delayed by the tasks A_2 and A_3 of application App_A which has higher priority. As we do not know the relative offset of App_A to App_B, we have to consider the worst-case scenario of interference from App_A in calculating the delay of task B_3. In addition, task B_5 on another PE is delayed due to the dependency with task B_3 as shown in the red box in Figure 3.25(c).

There are two methods to check the schedulability of multiple task graphs. One is to convert each task graph into a set of independent real-time tasks with relative deadlines and starting offsets. Dependency between tasks is expressed by the relative starting offsets of the tasks. The other method is the schedule-based analysis method proposed in [59].

The schedule-based analysis is based on the scheduling results of each application. For this analysis, the maximum resource demand (MRD) and maximum allowable interference (MAI) are computed. The former, MRD, means the maximum duration for which a high priority application can delay lower priority applications, while the latter (MAI) means the maximum duration for which a task can be delayed by higher priority tasks. MRD can be obtained by using the demand bound function (DBF) that returns the maximum processor execution time during a time interval given as an argument [60]. MAI is derived from the mobility concept of behavioral synthesis by subtracting the As-Soon-As-Possible (ASAP) schedule offset from the As-Late-As-Possible (ALAP) schedule offset [61].

For each task T_i, let P_i be the period of the task T_i, M(T_i) be the processor onto which T_i is mapped, and H(T_i) be the application set whose priority is higher than T_i. Also, let $dbf_{pe}(A,t)$ be the demand bound value of the tasks in an application A which are mapped onto pe during time interval t. Then, the application is schedulable if each task T_i satisfies

[8] A lower number means a higher priority.

the following equation: Eq. (3.10)[9].

$$MAI_{T_i} - \sum_{hp \in H(T_i)} dbf_{M(T_i)}(hp, P_i) \geq 0 \tag{3.10}$$

Since we assume non-preemptive scheduling as GPU and NPU cannot support preemptive scheduling, we modify Eq. (3.10) such that it considers the maximum possible interference by a lower priority task. If we denote a set of applications whose priority is lower than T_i as L(T_i) and the computation time of task lp as C_{lp}, then the final equation becomes Eq. (3.11).

$$MAI_{T_i} - \sum_{hp \in H(T_i)} dbf_{M(T_i)}(hp, P_i) - \max_{lp \in L(T_i)} (C_{lp}) \geq 0 \tag{3.11}$$

3.3.5.2 GA-based scheduler

The scheduler shown in Figure 3.19 is applied to the scheduling of multiple applications with a similar chromosome structure. For the scheduling of multiple deep learning applications, the fitness function is redefined since the objective and the constraints are changed. We set multiple objectives, minimizing the response time of each application, and minimizing the total energy consumption. Thus the fitness function has as many terms as the number of applications, each of which is the response time of the corresponding deep learning application, plus the term for the total energy. If the response time of an application is greater than the throughput constraint of the application, a large penalty is added to the term so that it cannot be selected.

[9]In the general case, after subtracting the interference of one higher priority application from the MAI, the MAI value should be updated iteratively, but this equation does not include that part for simplicity.

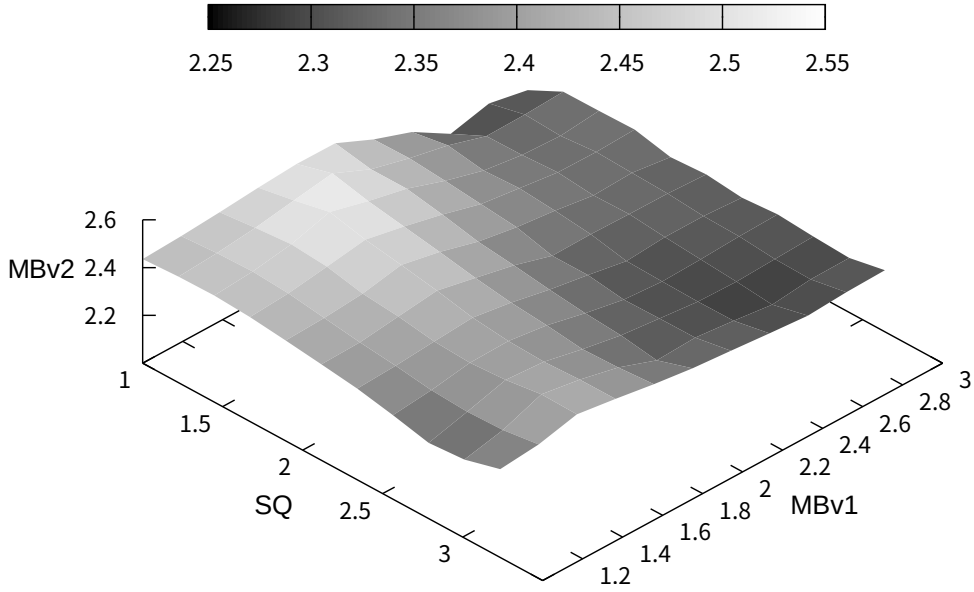


Figure 3.26: Pareto-optimal solutions in terms of relative response time when scheduling SqueezeNet (SQ), MobileNet v1 (MBv1), and MobileNet v2 (MBv2)

3.3.5.3 Experimental Results

The same configurations are used as in Section 3.3.4.3 except that the three CNNs are scheduled together and the objectives are different: to minimize response times of three applications and total energy consumption of the device. The CPU configuration is fixed to (1,1,1,1), as this configuration has resulted in better solutions for a single deep learning application in most cases.

Figure 3.26 and Figure 3.27 show the scheduling result of three benchmark CNNs with different periods; SqueezeNet, MobileNet v1, and MobileNet v2. Each application has periods of 33, 40, and 50 *ms*, respectively. In this experiment, the highest priority is assigned to SqueezeNet while the lowest one to MobileNet v2.

Figure 3.26 shows 288 Pareto-optimal solutions in terms of the relative response times of the three applications. We do not include energy information on this chart for a simple illustration. The grayscale indicates the response time of MobileNet v2: the

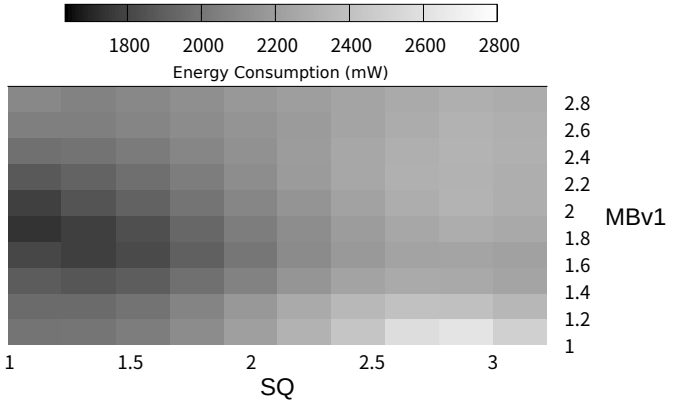
darker is the color, the smaller is the response time. The x-axis indicates the relative response time of SqueezeNet to the fastest response time of SqueezeNet on the device. The minimum response times of SqueezeNet, MobileNet v1, and MobileNet v2 are 10.3 *ms*, 13.7 *ms*, and 16.9 *ms*, respectively. The y-axis and z-axis represent the relative response time of MobileNet v1 and MobileNet v2 respectively. We could observe that the application with the higher priority has the smaller response time, as expected: the minimal relative response times of SqueezeNet, MobileNet V1, and MobileNet v2 are 1, 1, and 1.43, respectively.

Figure 3.27 shows the change in energy consumption as the response time of two different application changes. The darker color indicates less energy consumption. In Figure 3.27(a), we can observe that the energy consumption is low when the response times of SqueezeNet and MobileNet v1 are 1 and 1.8, respectively.

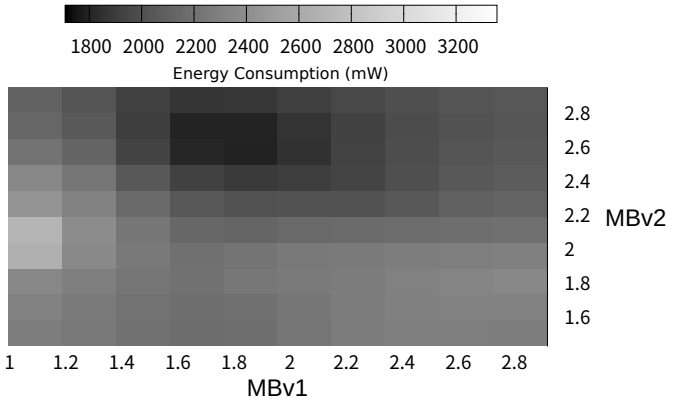
3.3.6 Verification with Real Hardware Platforms

Based on the scheduling results of a single deep learning application, we parallelize two benchmark networks, MobileNet v1 and MobileNet v2, on two hardware platforms: Galaxy S9 and HiKey970. Since there is no existent software framework to generate the parallelized code, we extended our own deep learning software framework [53]. For pipelined execution, we make a separate thread for each logical PE in CPU and implement double buffering for the tensor of which the source layer and the destination layer are mapped onto different PEs. And we use the conditional wait API for synchronization.

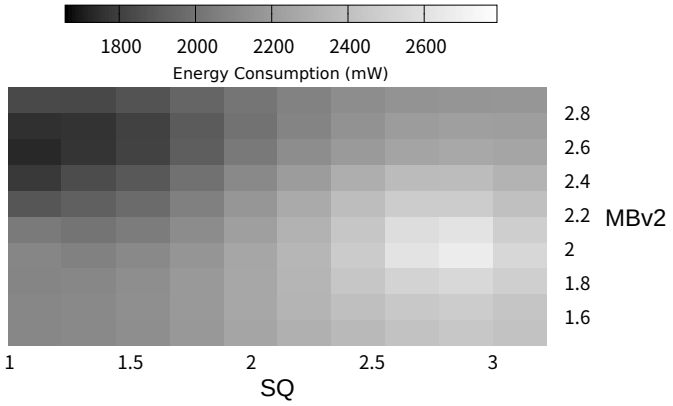
Due to the limitation imposed by ACL, however, we could not implement all PE configurations of the multi-core CPU, and we could not verify the scheduling of multiple deep learning applications. The current ACL implementation does not allow more than one task to use multi-threading for data-parallel execution simultaneously. For example, (2,2) configuration is not allowed since it would have two tasks, each of which in turn would create two threads. In (1,1,1,1) configuration, we disable the threading option for



(a) SqeezeNet (SQ) and MobileNet v1 (MBv1)



(b) MobileNet v1 (MBv1) and MobileNet v2 (MBv2)



(c) SqeezeNet (SQ) and MobileNet v2 (MBv2)

Figure 3.27: Pareto-optimal solutions in terms of energy when scheduling SqueezeNet (SQ), MobileNet v1 (MBv1), and MobileNet v2 (MBv2)

Table 3.16: Verification results on two hardware platforms

HW	Conf.	Net	Schedule	Actual	Error
HiKey970	C1	MN v1	18.5	19.7	-6.1%
HiKey970	C2	MN v1	21.1	19.8	6.6%
HiKey970	C1	MN v2	23.9	28.8	-17.0%
HiKey970	C2	MN v2	27.5	27.4	0.4%
Galaxy S9	C1	MN v1	15.4	16.5	-6.7%
Galaxy S9	C2	MN v1	23.5	23.2	1.3%
Galaxy S9	C1	MN v2	24.8	25.5	-2.7%
Galaxy S9	C2	MN v2	31.1 (31.2)	31.2	-0.3 (0)%

MN: MobileNet / Unit: sec

data-parallelism. Consequently, we could verify the scheduling results of a single deep learning application for the following two cases:

- (C1): With ACL scheduler disabled, scheduling with (1, 1, 1, 1) PE configuration
- (C2): With ACL scheduler enabled, scheduling with a quad-core CPU: (4) configuration

In this experiment, the scheduling objective is to maximize the throughput under no CPU utilization constraint, and comparison is made in terms of the processing time for 1000 images. Since we can change the DVFS governor and GPU frequency of HiKey970, unlike Galaxy S9, we chose the *performance* CPU governor and 767MHz GPU frequency for HiKey970. Neural Processing Unit (NPU) is not used even though HiKey970 has it.

Table 3.16 shows the verification results. It can be observed that the performance difference of the parallelized MobileNet v1 between the scheduling result and the measured one is less than 7% on both hardware platforms for two PE configurations. For MobileNet v2, the performance gap is relatively larger, particularly on HiKey970 for (C1) configuration. The main reason is that we underestimate the communication cost between processors in HiKey970. We found that Galaxy S9 uses only two CPU cores instead of four for (C2) configuration with MobileNet v2, turning off two CPU cores as the

CPU utilization becomes lower than the given threshold that is not known to us. Thus we re-run the GA-based scheduler with profiling results with two CPU cores for the layers. Then, we could obtain no performance difference as shown with parentheses in the last row of Table 3.16.

3.4 Related Work

3.4.1 Deep Learning Framework

New algorithms are being developed every day, and many software optimization techniques exist. It is thus necessary to develop a systematic methodology to explore the wide design space of algorithm selection and software optimizations for a given hardware platform. On the other hand, many convolution neural network (CNN) algorithms are developed on a deep learning framework such as Caffe [62], Torch [63], or Tensorflow [64] that is assumed to run directly on the target hardware. Since the framework itself is performed on the target hardware to perform from building a CNN graph to optimizing and inferencing the graph, it is slow and consumes more hardware resources. Also, most CNN applications built with these frameworks are implemented in scripting languages such as Python, and most frameworks use many libraries that are not supported by embedded systems. This makes it difficult to apply to embedded devices.

Darknet [40] is another neural network framework written in C language. It provides options to use various functions for image processing in the OpenCV library, and to optimize a CNN algorithm on NVIDIA GPUs using CUDA or cuDNN. To improve efficiency and portability, three major extensions are made in C-GOOD. One is to make new function definitions or provide a different set of libraries if a library is not supported in the target platform. For the Odroid platform [4] that does not support CUDA, for instance, C-GOOD provides and generates OpenCL kernels for GPU computation. The second extension is to generate a target-specific C code. For the Samsung Reconfigurable Processor (SRP) platform [5], loop unrolling techniques are applied in the code generation step to help the SRP compiler optimize the deep learning operations efficiently. The last extension is to optimize some operations for inferencing. When batch size is 1, for example, a reshape layer, which changes the shape of the feature map while merging two or more previous feature maps into one feature map, can be skipped by allocating the

previous layers' output memory continuously. It also reduces memory allocation.

3.4.2 Deep Learning Compiler

While there is a trend to perform deep learning applications on a variety of hardware devices, current frameworks rely on specific libraries, and deploying workloads to new hardware platforms requires considerable manual work. To reduce the effort to deploy workloads to every new hardware, several deep learning compilers have been proposed [65, 66]. TVM [65] is an end-to-end compiler that allows the deployment of deep learning workloads specified in high-level frameworks to diverse hardware backends. It introduces scheduling primitives and proposes a machine learning based optimization system to automatically explore and search for optimized tensor operations. Tiramisu [66] is a polyhedral compiler for dense and spare deep learning and data-parallel algorithms.

These two compiler works are similar to ours in that they take deep learning networks as inputs and generate optimized code. However, while these two studies optimize and schedule in one convolution operation, our methodology optimizes multiple layers or a whole network by applying several software optimization techniques such as heterogeneous PE mapping and layer decomposition. Also, the devised methodology uses libraries, such as cuDNN and ARM compute library (ACL), to perform the optimizations that the two compilers do.

3.4.3 Scheduling Deep Learning Application

There are many recent studies for executing deep learning applications on embedded devices. While they mostly consider a single deep learning application, we also consider the scheduling of multiple applications that share the processors. CNNDroid [67] is a library that accelerates a DNN by dividing layers into GPU and CPU. It simply maps the convolution and fully-connected layers to GPU and the other layers to CPU. RSTen-

sorflow [68] is a framework for users to use heterogeneous processors such as CPU and GPU easily. It uses RenderScript [69] to manually parallelize the computation workloads in a layer, such as matrix multiplication and convolution, across CPU cores and GPUs.

Mirhoseini et al. [70] proposed a hierarchical DNN model for the efficient placement of a neural network graph onto hardware devices. The *Grouper* groups graph operations and the *Placer* maps them to the devices. Even if they also try to utilize not only GPUs but also CPUs in the system, unlike our approach, they do not consider the various configurations with multi-cores in a CPU, nor the pipelining scenario: only task-parallelism with parallel edges is considered.

μLayer [71] is the latest work to accelerate a DNN on heterogeneous processors on Samsung Galaxy Note 5 and Galaxy A5. It aims to exploit only data-level parallelism of each layer using the heterogeneous PEs, unlike ours which also exploits task-level parallelism. Since all PEs need to be synchronized and data communication between PEs is necessary at the end of each layer, the communication and synchronization overhead is significant. Also, it did not consider a DVFS policy and CPU utilization constraints of the smartphone. Nonetheless, experimentation with the real smartphone is laudable since practical issues need to be resolved like this work.

DeepX [72] considers data-level parallelism in the mapping and scheduling of deep learning applications on heterogeneous multi-processor platforms. DeepX performs layer-wise partitioning first and divides the workload of a layer into a group of unit blocks that are defined as the computation requirements to update a single output node in a layer. The authors propose an integer linear programming (ILP) formulation to find a trade-off between energy consumption and latency by allocating the unit blocks to PEs layer by layer. While they show the experimental results of layer-wise partitioning onto the heterogeneous PEs, no performance comparison is reported between the simple layer-wise partitioning and the ILP-based block-level partitioning. Their work differs from ours in that they do not consider task-level parallelism and their objective function does not con-

sider throughput constraint.

3.4.4 Scheduling Multiple Applications on Heterogeneous Processors

3.4.4.1 Scheduling techniques on heterogeneous processors

Since scheduling acyclic task graphs on a heterogeneous system is a well-known NP-hard problem, several heuristics have been proposed to solve this problem. Topcuoglu et al. [73] proposed a Heterogeneous Earliest Finish Time (HEFT) algorithm and a Critical Path On a Processor (CPOP) algorithm, which both greedily reduce task finish time. HEFT uses the concept of *rank*, which calculates the execution time of a critical path from one task to the last task, to prioritize each task, and schedules tasks by mapping them to the processor in a way that minimizes the finish time. CPOP, on the other hand, reduces the overall latency of the application by first mapping all tasks in the critical path to one fastest processor. When HEFT and CPOP schedule tasks on heterogeneous PEs, the rank of a task is determined by the average execution time on all processors. On the other hand, the Predict Earliest Finish Time (PEFT) algorithm proposed by Arabnejad et al. [74] uses the Optimistic Cost Table (OCT), which has different task execution times for each processor. These heuristics are concerned about the scheduling of a single task graph. Roy et al. [75] proposed an Integer Linear Programming (ILP) algorithm to find the optimal schedule result in terms of a makespan, or response time, of an application. This work differs from other works in that it finds the optimal schedule. While this work considers only one application, our method can schedule multiple applications. Also, our method maximizes throughput when scheduling a single application, while other studies aim to reduce the makespan.

Zhao et al. [76] have addressed the scheduling problem of multiple applications. Their solution is to simply merge multiple applications into one large task graph. Hence it does not allow applications to have different periods and random starting offsets. Xie

et al. [77] proposed two static heuristic algorithms, F_MHEFT and D_MHEFT, which are global scheduling algorithms that schedule multiple applications with mixed-criticality levels on heterogeneous PEs. They present two scheduling algorithms; F_MHEFT aims to improve the system performance based on the fairness policy while D_MHEFT aims to meet the deadline of high-criticality applications. D_MHEFT is similar to our method in that it schedules the target applications with deadlines of tasks in mind. However, we schedule multiple applications with different periods and starting offsets, and also consider the worst interference by other application tasks. To apply D_MHEFT to multiple applications with different periods and starting offsets, it would have to simulate all possible cases.

3.4.4.2 Schedulability when scheduling multiple applications

When we schedule multiple applications onto a multiprocessor system, a popular solution is to partition the processors to the applications spatially and/or temporarily. In other words, each PE is assigned exclusively to a single application. Then, each application can run on the assigned set of processors exclusively without worrying about schedulability. If we allow a processor to be shared among multiple task graphs, however, any parallel scheduling should check if the mapped tasks are schedulable on each processor.

There are two approaches to tackle this scheduling problem. One is to transform the task graph into a set of independent tasks that have different starting offsets and relative deadlines [78]. In this approach, the starting offsets and deadlines should be conservatively assigned, considering the dependency between tasks. After transformation, the conventional schedulability analysis method for independent tasks is applied.

The second approach is to compute the time range in which each task should be scheduled to guarantee the satisfaction of the deadline and to check if the possible interference from the other applications is smaller than the range [59]. Since the former

approach, the transformation approach, checks schedulability pessimistically [59], we choose the latter approach, the schedule-based approach.

3.4.4.3 Scheduling techniques considering data parallelism inside a task

Even though there exist numerous scheduling techniques that have been proposed for homogeneous or heterogeneous multi-core systems, they mostly assumed that a task is run on a processor, and seldom considered data parallelism inside a task. Some recent studies considered data parallelism of tasks as well as task parallelism. Liu et al. [79] proposed heuristic algorithms to minimize the scheduling length of a task graph with data-parallel tasks. Yang et al. [80] proposed an evolutionary algorithm to schedule a task graph, considering task parallelism, data parallelism, and pipelining, with an objective to maximize the throughput performance. The same authors proposed an ILP based technique for minimizing the total processor cost while satisfying the time constraints.

The previous works are usually based on a static model of a hardware platform. They do not consider the characteristics of the actual hardware platform on which the scheduling algorithm will run. Thus it is simply assumed that the execution time of a task on each PE is given and the processor configuration is also fixed. On the other hand, we integrate the characteristics of the hardware platform into problem formulation by profiling each task considering various processor configurations. This makes our approach distinguished from the existent ones. Moreover, the scheduling results are verified with actual implementations, which has not been carried out in most of the previous works.

Chapter 4

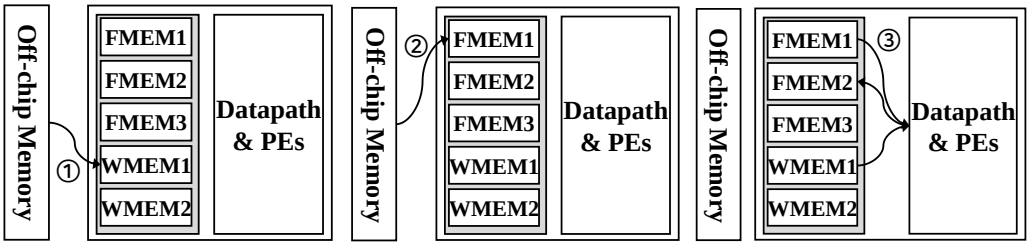
Optimization for an Application-specific Hardware Accelerator

The main factors affecting the performance and energy consumption of a neural processing unit (NPU) are off-chip memory access and on-chip memory contention. For example, in the case of DianNao [23] and Cambricon-X [24], it is reported that the energy consumed by the off-chip DRAM access accounts for 80% of total energy consumption [25, 26]. Besides, DPU-v2 [22] is reported to be under 40% utilization for GoogleNet and ResNet-50, which commonly have a lot of memory operation. Thus, we reduce the impact of off-chip memory access by reusing the feature map data between consecutive layers. To avoid on-chip memory contention, we organize the on-chip memory to multiple banks and manage them.

4.1 Multi-Bank On-chip Memory Management Problem

4.1.1 Main Idea

This work's key idea is to manage multi-bank on-chip memory to reduce performance and energy consumption due to off-chip memory access. This section explains the key idea with the help of Figure 4.1 and explains why multi-bank on-chip memory



(a) Load weight (b) Load feature map (c) Computation

Figure 4.1: NPU operations with multi-bank on-chip memory

management is required.

Suppose we have multiple memory banks to store filter weights and feature maps, as illustrated in Figure 4.1. After loading the filter weights and an input feature map from the off-chip DRAM, computation is performed in the datapath or processing elements (PEs) by accessing three memory banks without access conflict. By prefetching the filter weights and input feature map to available memory banks that are not accessed by the datapath, we may hide the DRAM access delay. Thus multi-bank structure is usually adopted to reduce the access contention in neural processing units (NPUs) [16, 19, 81]. Note that the output feature map had better be stored in the on-chip memory to reduce the DRAM access since it will be the input feature map of the next convolution layer. By doing so, if the sum of the input feature map and output feature map is smaller than the total size of feature map banks, we can avoid the DRAM access completely between two consecutive convolution layers. Otherwise, we need to manage the memory banks judiciously for storing the input and output feature maps.

4.1.2 Assumed Dataflow

Dataflow is important as it is a major factor in determining the amount of data that can be reused in on-chip memory, which is the subject of this paper. A convolution operation can be represented by six nested **for** statements. From the six nested **for** statements, the dataflow of each accelerator is determined by three optimization terms: loop order,

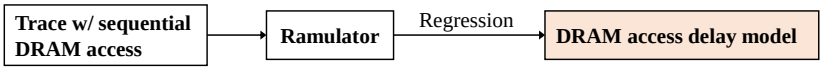
partitioning loop iteration, and hardware mapping of each partition [16, 28, 82]. Figure 2 shows the assumed dataflow in this paper. To efficiently reuse data between layers, we reduce the time that partial sum data occupies on-chip memory by moving **for** statements related to one output pixel into the on-chip computation. Also, we prioritize input reuse over weight reuse. Based on this dataflow, the proposed compiler generates a sequence of the instructions for when and how much to load or process input.

Algorithm 2 Dataflow of the convolution operation assumed in this paper using variables in Figure 2.2

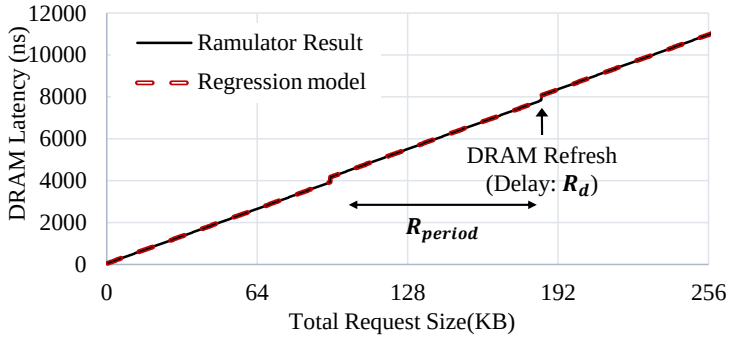
```
1:  Variables
2:      O       Output Tensor
3:      I       Input Tensor
4:      K       Weight Filter
5:  for w = 0; w < W_out; w++ do
6:      for h = 0; h < H_out; h++ do
7:          // Load inputs
8:          for f = 0; f < C_out; f++ do
9:              // Load weights
10:             // On-chip computation
11:             for c = 0; c < C_in; c++ do
12:                 for kw = 0; kw < K_w; kw++ do
13:                     for kh = 0; kh < K_h; kh++ do
14:                         MAC(O, I, K, w, h, c, f, kw, kh)
```

4.1.2.1 Delay Estimation

Since a key motivation of using a multi-bank structure is to hide the off-chip DRAM access delay by prefetching as much as possible, it is necessary to compare the DRAM access delay and the computation time. As a part of input information to the MOMM problem, we need to estimate the DRAM access delay and the processing delay of an input FMEM bank. As shown in Figure 4.2(a), the DRAM access delay model is obtained by regression of the DRAM simulation results with Ramulator [83]. We assume that LPDDR4 3200Mbps with 2 channels and 1 rank is used. Figure 4.2(b) shows the read latency obtained by Ramulator and the regression result by varying the data size.



(a) DRAM access delay model by regression



(b) Ramulator results and the proposed model for $D_m(B)$

Figure 4.2: DRAM access delay model

From the simulation result, we derive the DRAM access delay, $D_m(B)$ where B is the data size, as follows:

$$D_m(B) = \frac{B}{BW_m} + L_m(B) \tag{4.1}$$

$$L_m(B) = (D_{init} + P_d \times \lfloor \frac{B}{P_{sz}} \rfloor + R_d \times \lceil \frac{B}{R_{period}} \rceil) \tag{4.2}$$

The first term of eq. (4.1) is the data transmission time from the row buffer, which corresponds to the slope in Figure 4.2(b). The second term, $L_m(B)$, represents the other delay that corresponds to sporadic jumps in the figure, which is as a function of B. In eq. (4.2), D_{init} is the initial read delay, P_d is the delay to read a new row whose size is P_{sz}, and R_d is the delay caused by DRAM refresh whose period is R_{period}. With three delay parameters obtained by regression, the estimated delay by eq. (4.1) becomes almost identical to the simulation result, as shown in Figure 4.2(b).

To verify the DRAM delay model's accuracy, we perform a preliminary experiment to compare two different simulation results. In the first experiment, the Ramulator is connected to the NPU simulator; the DRAM access trace generated by the NPU simulator is passed to the Ramulator, and the simulated delay is fed back to the simulator whenever a

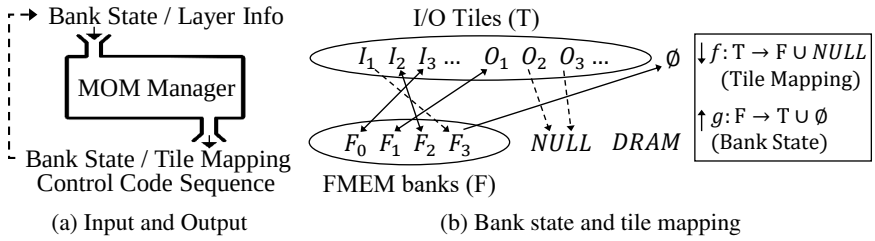


Figure 4.3: MOMM problem definition

DRAM is accessed. In the second experiment, we use the DRAM delay model in the NPU simulator instead of running the Ramulator. The delay error between the experiments is at most 3.8%, on average 2.8%.

For computation time estimation, we compute the total number of cycles to process a convolution layer, assuming that MACs are fully utilized, no dynamic behavior exists due to resource contention inside the data path, and no zero skipping is used. Therefore, once the target NPU is determined, the execution time can be estimated with a specific formula.

4.1.3 Multi-bank On-chip Memory Management Problem

As shown in Figure 4.3(a), the inputs of this problem are the layer information and the current bank state. Let I and O be the input feature map and output feature map of a given layer, respectively. Then, $I = \cup\{I_j\}$ where I_j is the j-th input tile and $O = \cup\{O_k\}$ where O_k is the k-th tile of the output feature map. Let T be the set of input and output tiles, then T is obtained as $T = I \cup O$. Also, we could define FMEM, F, as $F = \cup\{F_k\}$ where F_k is the k-th FMEM bank where $k \in \{1, 2, \ldots, N\}$ and N is the number of FMEM banks. The size of each tile is limited to the size of an FMEM bank.

Bank state information g is a function that receives a bank as an input and gives which data is currently stored in that bank. Therefore, the domain and the codomain of g are the bank domain F and feature map tile domain including an empty state $(T \cup \{\emptyset\})$, respectively. Then, $g(F_k) = I_j$ means that the j-th input tile is mapped onto the

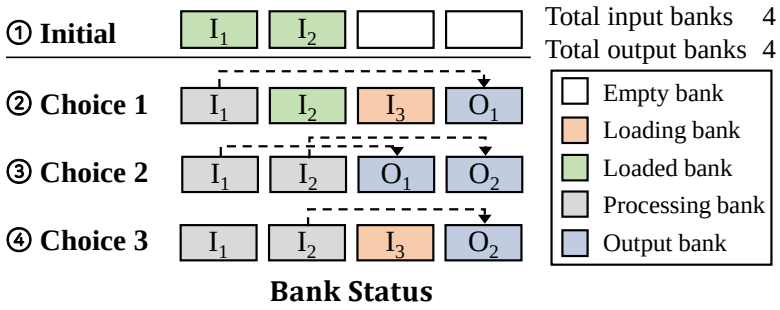


Figure 4.4: An example of MOM Manager

k-th FMEM bank, and $g(F_k) = \emptyset$ means no feature map tile is mapped on k-th bank. In Figure 4.3(b), the bank state indicates that F_0, F_1, and F_2 store I_3, O_1, and I_2, respectively. Since there is currently no tiles mapped to F_3, the bank state of F_3 is $\emptyset$ as shown in the figure. The layer information includes the input size, filter kernel size, and the number of filters.

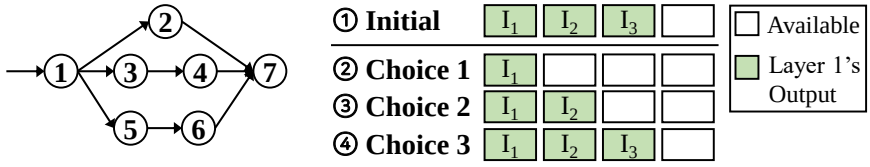
As shown in Figure 4.3(a), the output of the proposed Multi-bank On-chip Memory Manager (MOM Manager) includes control code sequence, tile mapping information, and updated bank state. The control code sequence determines the NPU behavior by a sequence of control codes. Tile mapping information provides a mapping location for each input and output tile during computation. That is, the tile mapping information is a function that receives an input or output tile and provides in which banks the tile is stored. Therefore, tile mapping information is represented by the function f from I/O tile domain T to the FMEM bank domain including NULL and DRAM $(F \cup \{NULL, DRAM\})$. In Figure 4.3(b), the tile mapping information indicates that I_1, I_2, and I_3 is mapped to F_3, F_2, and F_0, respectively. Also, it informs that O_1 is mapped to F_1. Note that F_3 is currently empty as explained in the previous paragraph, but I_1 is mapped to the bank. That means that I_1 was previously mapped to F_3 and then processed, so F_3 is empty now.

Figure 4.4 shows an example MOMM step during the convolution computation. In this example, we assume that there are four FMEM banks while the required number of banks for the input feature map and the output feature map is four each. For a simple

illustration, we further assume that one input tile produces one output tile. Note that these assumptions are for ease of explanation, and processing one input bank can produce more than or less than one bank in this work. The first row (①) shows the input information to this step: the first two input tiles are already mapped on two banks, and the remaining two banks are empty. We can make several choices when processing the input feature map, and the figure illustrates three choices from the second (②) to the fourth row (④). Each choice corresponds to a pictorial representation of the associated control code. There are more than one possible control codes we can generate. The state of each FMEM bank is categorized into five states that are distinguished by different colors in the figure.

The first choice (②) is to use an empty bank to store the output feature map and prefetch the third input segment to a remaining empty bank. This choice allows us to hide the DRAM access for the next input tile by prefetching. Nevertheless, the filter weights are used for one input tile only. The second choice is to process two input banks consecutively and use two banks to store the produced output tiles. In case the size of filter weights is larger than the size of WMEM, the loaded filter weights can be reused to process two FMEM banks. Thus the second choice (③) reduces the number of DRAM access for weight loading than the first choice. However, it disallows prefetching of the next input feature segment. After finishing processing two FMEM banks, we need to wait until the next input tile is loaded from the off-chip DRAM. Remind that that the loading time of filter weights can be hidden by prefetching into WMEM banks.

The third choice (④) is to reserve only one bank to store the output feature map and use the other empty bank for prefetching while processing two input banks together. It seems to retain the benefits of the previous two choices, allowing prefetching to hide the DRAM access delay and reusing the loaded filter weights to process two FMEM banks together. However, it incurs additional overhead to access DRAM to store an output feature map segment. Surely there may be other choices not shown in the figure. For instance, we may use two empty banks both to prefetch the next input segments while



(a) An example layer block (b) Initial bank status and remaining options

Figure 4.5: An example of MOMM problem in multiple paths

storing the first two output feature map segments to the off-chip DRAM to minimize the loading count of filter weights. Since there are trade-offs among various choices, selection should be made carefully depending on the objective function of MOM management.

In case a CNN has parallel branches, the problem becomes more complicated. Figure 4.5 shows a segment of a CNN network with multiple paths; a circle in Figure 4.5(a) represents a layer, and the numbers in the circles represent the layer number and the processing order. Note that three layers (layers 2, 3, and 5) share the output of layer 1 as their input. Therefore, if the output of layer 1 is kept in the on-chip FMEM after processing layer 2, there is no need to load the data from the off-chip memory when processing layer 3. Figure 4.5(b) shows the possible choices in the processing of multiple paths, particularly after layer 3. The first choice (②) is to reserve one bank to keep one output tile of layer 1. This choice will allow layer 4 to use the remaining three banks to reduce off-chip memory access. However, when processing layer 5, the other output tiles of layer 1 need to be loaded from the off-chip memory. The second choice (③) is to keep two FMEM banks for layer 1's output. Then layer 4 can use only two other banks for its input and output feature maps, which would incur more DRAM accesses during computation than the first choice. If we keep all three banks of layer 1's output to reuse them for layer 5, layer 4 has only a single bank for computation (④). This example shows that we need to consider how many input banks need to be kept for parallel branches, considering the memory usage of all layers and bank state carefully.

There are two objective functions considered in this paper for the MOMM problem.

One is to minimize the processing time, and the other is to minimize the DRAM access volume for energy minimization. We consider both feature maps and filter weights in the computation of DRAM access overhead.

4.2 Proposed Multi-bank On-chip Memory Management Techniques

How to efficiently use the multiple on-chip memory banks defines the Multi-bank On-chip Memory Management (MOMM) problem. More specifically, it is about how to allocate the on-chip memory banks. There are three decisions to solve this problem: what data is stored, which banks to store the data, and how many banks to store. In this work, we decide which banks to store the data by simply allocating one by one from the queue that contains empty banks. Therefore, two decisions remain.

Suppose the on-chip memory is big enough to store all feature maps necessary during processing. In that case, the MOMM problem can be solved easily by assigning available banks to the feature maps, avoiding access contention. Otherwise, it is necessary to access the off-chip DRAM while processing a convolution layer. Therefore, we first consider when the output feature map is too large to store all output in on-chip FMEM. We propose to store the overflowed output feature map to the off-chip DRAM first called 'DRAM-first Storing' (DFS) policy, which will be discussed below. Next, we propose two bank management techniques with different objectives based on the DFS policy. One is to minimize the DRAM access size, and the other is to maximize the performance by hiding the DRAM access as much as possible. Lastly, we discuss how to apply the proposed techniques to parallel branches.

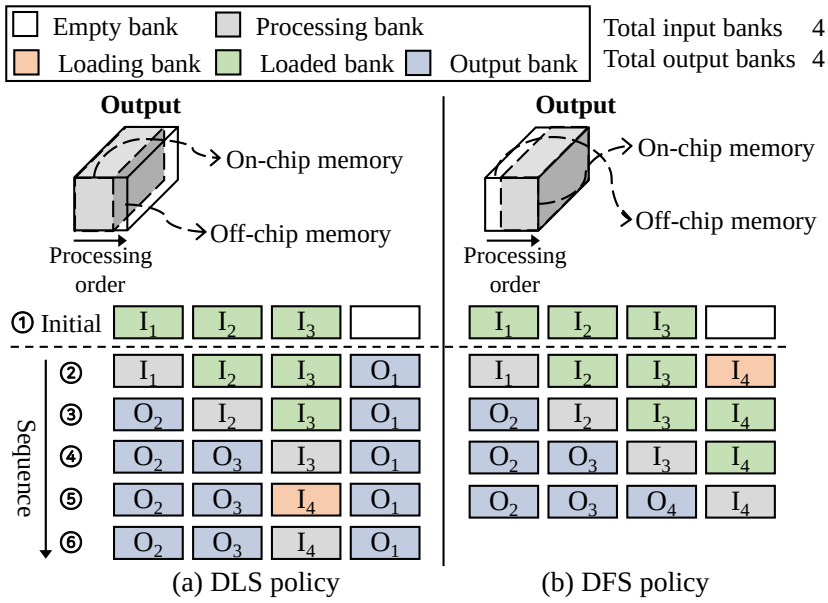


Figure 4.6: Comparison between DLS and DFS policy

4.2.1 DRAM-first Storing Policy

To minimize the DRAM access, we aim to store the output feature map to the on-chip memory as much as possible since it will be used as the input feature map in the subsequent layer. Since the FMEM banks are shared between the input feature map and the output feature map, we need to decide how to allocate the available banks to input and output feature maps. Note that the number of remaining input tiles decreases while the number of output tiles increases as the convolution computation proceeds. In case we cannot accommodate the entire output feature map in the on-chip FMEM banks, the proposed technique, called DRAM-first storing (DFS) policy, stores the overflowed output feature map to DRAM first to minimize the number of banks to store the output feature segments during computation.

Figure 4.6 illustrates the difference between the proposed DFS policy and the naïve DRAM-last storing (DLS) policy that accesses DRAM only after there is no available bank in the on-chip memory. This example assumes that the input feature map and the

output feature map require 4 FMEM banks each, and three banks store the input tiles initially. Since there is one available bank for the output feature map, the DLS policy assigns it to the output tile, as shown in Figure 4.6 (a). After finishing the processing of *Bank 0* (②), the bank becomes available for the second output tile that corresponds to the input tile stored in *Bank 1* (③). After finishing all three input tiles, it is necessary to load the last input tile from the off-chip DRAM, as shown in the fifth row (⑤) of Figure 4.6 (a), which incurs visible DRAM access delay during computation. After loading is completed, the last output tile is stored to the off-chip DRAM since there is no available bank in the on-chip memory.

On the other hand, in the proposed DFS policy, we store the first output tile to DRAM and use one available bank, *Bank 3*, for prefetching of the remaining input tile (①). After finishing the processing of *Bank 0*, the bank becomes available for the second output tile that corresponds to the input feature map segment stored in *Bank 1* (②). By reusing the FMEM bank of the just-finished input tile for the next output tile, we can store the remaining three output tiles into the on-chip memory without DRAM access. The DFS policy allows efficient use of bus resources by loading input data from the DRAM earlier than DLS and reduces processing time without visible DRAM access delay. We always use the DFS policy because it gives better performance than the DLS policy.

4.2.2 DRAM Access Minimization Policy (MIN policy)

The *MIN* policy aims to minimize the DRAM access volume in the processing of a convolution layer. In this policy, we need to consider three DRAM access sources: input feature map load, filter weights load, and output feature map store. As an example, Figure 4.7 shows the compilation result for the 25th convolution layer of WideResNet50 [9] for an NPU that has 4 FMEM banks of 128KiB each. The layer is 1×1 convolution layer in which the size of the input feature map is $32 \times 32 \times 512$, and that of the output feature map is $32 \times 32 \times 512$. Thus, the weight's total size is 512×512(= 256KiB), which is

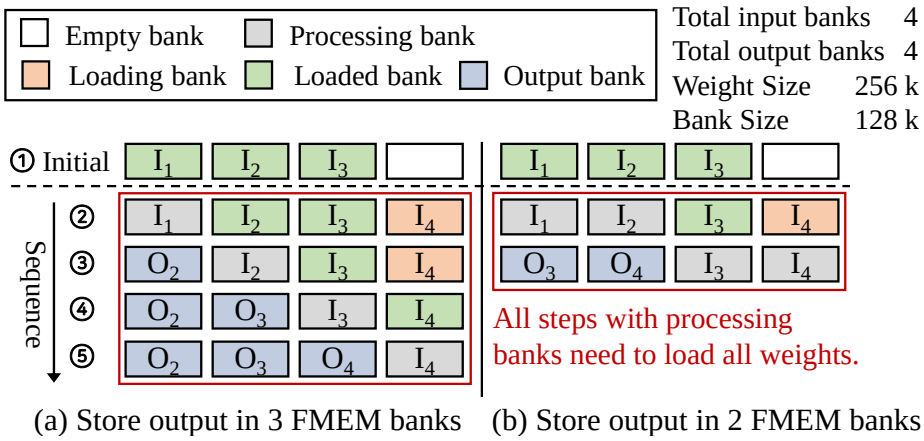
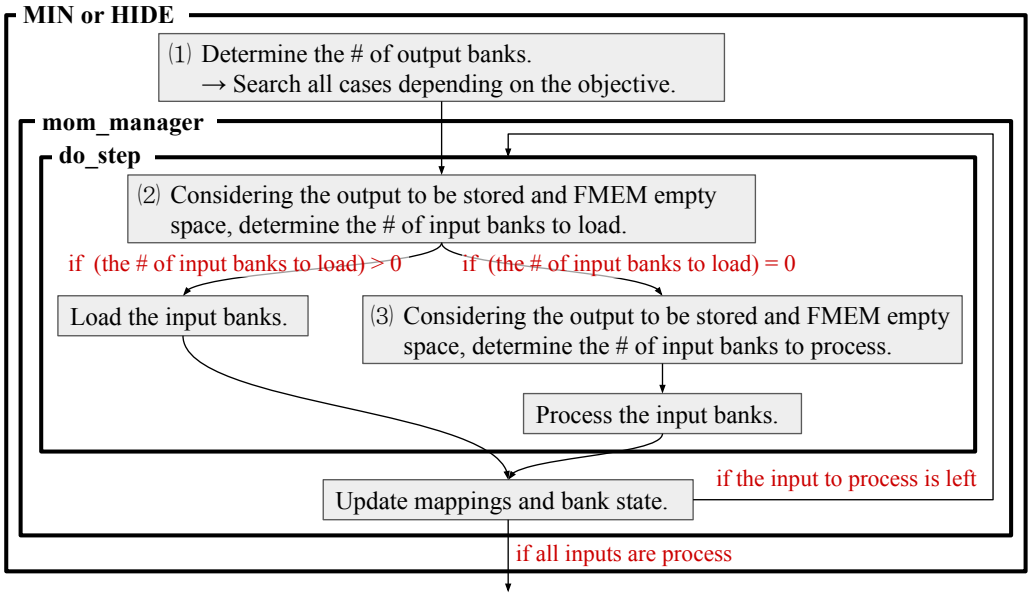


Figure 4.7: Effect of the number of FMEM banks assigned to the output feature map with the 25th convolution layer of WideResNet 50 [9]

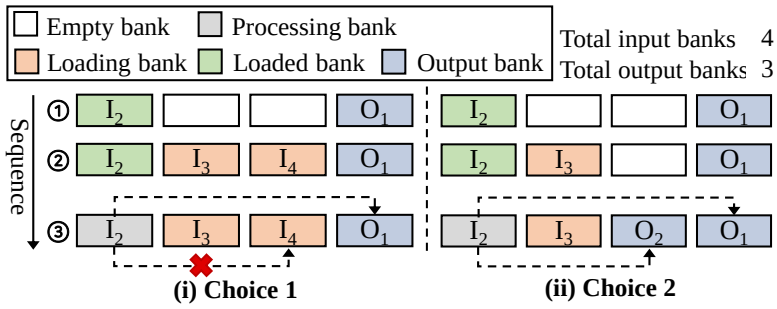
assumed larger than the WMEM size in this example.

In order to maximize the feature map reuse between two consecutive layers, we may want to assign 3 FMEM banks to the output feature map, as shown in Figure 4.7(a). In this case, we need four control steps, producing one output tile at each step. Since we assume that the WMEM size is smaller than the total volume of filter weights, we need to load the filter weights from the off-chip DRAM every step. Suppose that we assign 2 FMEM banks to the output feature map. Then, only two control steps are sufficient, as shown in Figure 4.7(b) where two input tiles are processed together, sharing the filter weights. As a result, the filter weights need to be loaded only twice from the off-chip DRAM. Between these two options, which one is better in terms of DRAM access volume? If we assign 3 FMEM banks, we reduce 2 FMEM bank accesses, one for storing and the other for loading an input tile in the next layer. However, the filter weights of 256KiB need to be loaded twice more. Thus assigning 2 FMEM banks is better in this example scenario. It is noteworthy that if the WMEM size is big enough to accommodate all filter weights, in both cases, we only load the filter weights once to process the layer. Therefore, assigning 3 FMEM banks is better.

Figure 4.8(a) shows an overall flow of the proposed techniques. Since the control



(a) Overall flow of the proposed techniques



(b) An example of considering output

Figure 4.8: The proposed techniques

sequence and the DRAM access volume depend on the number of FMEM banks to be assigned to the output feature map, each policy first determines the number of banks to store output feature maps according to its objective (Figure 4.8(a)(1)). In the case of the MIN policy, it decides the number of output banks which reduces the DRAM access volume most. The **do_step** generates a control code for each step in **mom_manager** according to the number of output FMEM banks. In **do_step**, it first checks if there are input tiles to prefetch and decides how many input tiles to load, considering FMEM bank state and the output to be stored (Figure 4.8(a)(2)). If there are input tiles to load, it loads

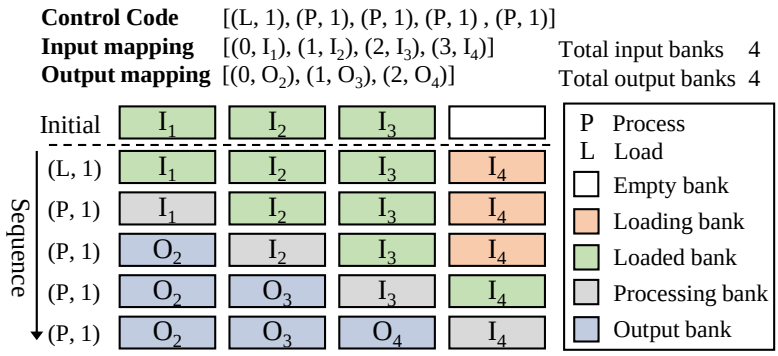


Figure 4.9: Control code example

all input tiles to empty banks. Otherwise, it determines the set of input tiles to process (Figure 4.8(a)(3)). It processes all possible input tiles for more reuse of filter weight data. After the command is determined, it updates input and output mapping and bank state. This process is repeated until all layers have been computed.

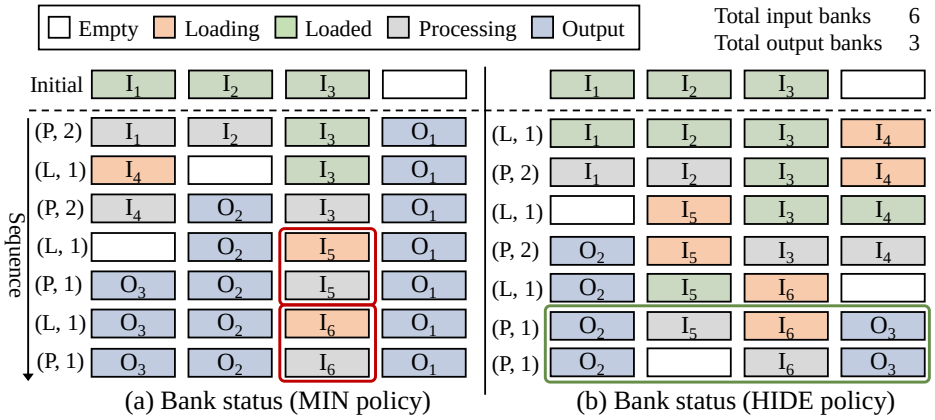
Figure 4.8(b) shows an example where the output should be considered when determining the number of input banks to load (Figure 4.8(a)(2)). In this example, let us assume that the total number of banks for the input feature map and output feature map are four and three, respectively. In other words, processing one input bank produces a three-quarters output bank. Figure 4.8(b) shows two choices. The first is to prefetch two input tiles to the remaining two banks as shown in Figure 4.8(b)(i). The other is to prefetch only one input tile (Figure 4.8(b)(ii)). In both cases, as shown in Figure 4.8(b), the FMEM stores the second input tile and the first output tile (①). Note that processing the second input bank produces data to be stored in the first and the second output bank because three-quarters of the first output bank store the output data after processing the first input bank. Therefore, in the case of (i), we cannot store the second output tile because of the lack of FMEM bank. To solve this case, the proposed compiler check if there are banks for the outputs to be stored in advance, and if there is no empty bank, it reduces the input bank to load. Similar logic is required when determining the number of banks to process (Figure 4.8(a)(3)), which is described in Appendix A.2.

Figure 4.9 shows an example of a control code sequence generation by the MIN policy to produce three output tiles in the FMEM as the example of Figure 4.7(a). To simplify our control code, we define an abbreviated control instruction as $(\{L,P\},num)$ where L and P mean loading and processing, respectively, and *num* is the number of tiles to which the instruction is applied; for instance, $(P,2)$ means "processing 2 input tiles" with the fetched weights. A mapping is defined as $(ix,\{I/O\}_{num})$ where *ix* means which bank to store the input or output tile indexed by *num*. Initially, three input tiles are loaded, as shown in the figure's first row of the bank state. When the first control code, *(L,1)*, is applied, the fourth input tile is loaded to the last FMEM bank. Note that the load is a *non-blocking* operation, meaning that the next control code may start its execution before the load operation is completed. While loading is processed, the next control code, *(P,1)*, is applied to process one input tile and write the first output tile to the off-chip DRAM by the DFS policy. The first two lines merged into a single row in Figure 4.7(a) since two control codes are running concurrently. The next three control codes are the same, processing one input tile to write one output tile into the FMEM bank that becomes available after the previous step completes.

4.2.3 DRAM Access Hiding Policy (HIDE policy)

The DRAM access hiding policy, called *HIDE* policy, aims to hide the DRAM access as much as possible to minimize the execution time. If the processing delay of an FMEM bank is larger than the loading delay of an FMEM bank and filter weights from DRAM, we can hide the DRAM access delay by prefetching. The DRAM access delay that cannot be hidden by prefetching incurs an additional delay called *PE delay*, and the HIDE policy aims to minimize the total PE delay. To guarantee the existence of input banks to be processed next, the HIDE policy maintains at least one input bank at each control step if there exist unprocessed input tiles.

Figure 4.10 demonstrates the difference between the MIN policy and the HIDE pol-



(a) Bank status (MIN policy) (b) Bank status (HIDE policy)

Figure 4.10: An example with the MIN and the HIDE policies

icy with a convolution layer that has 6 input tiles and produces 3 output tiles; processing 2 input tiles produces 1 output tile. Starting with the same initial bank status with 3 loaded input tiles, each policy shows different final bank status. While the MIN policy assigns three FMEM banks to the output feature map, the HIDE policy assigns two FMEM banks only to make one bank for prefetching during the computation. The control sequence of each policy is depicted in the figure. Even though the loading operation is non-blocking, the next processing operation has to wait until the load operation is completed in some cases that are highlighted by a red box in Figure 4.10. Such waiting incurs the extra processing delay due to DRAM access. While the MIN policy has the processing delay twice, the HIDE policy has no delay. In the MIN policy, we process as many input tiles as possible to reuse the weight filters. On the other hand, the HIDE policy may use a smaller number of input tiles in the processing step, as indicated by a green box in the seventh control step of the HIDE policy; only one bank is processed even though there are two input tiles.

To determine the number of output banks, the HIDE policy also performs an exhaustive search by varying the number of input tiles as depicted in Figure 4.8(a)(1). However, unlike MIN policy, HIDE policy searches the number of output banks in terms of the processing cycle. In other words, it decides the number of output banks which reduce the

processing cycle most. The processing cycle can be estimated by fast timing simulation using the estimated cycle explained in Section 4.1.2.1. Then, it checks whether there is any input tile to load considering outputs as described in Section 4.2.2. If there is no input tile to load, it determines the number of input banks to process (Figure 4.8(a)(3)). One of the main differences between MIN policy and HIDE policy is how many input tiles to process. As explained in Section 4.2.2, the MIN policy processes all possible input tiles. However, the HIDE policy can reduce the number of input banks to process to avoid PE delays due to the lack of input banks to process next. The algorithm to reduce the number of input banks to process is included in the Appendix A.3.

4.2.4 Multiple Path Consideration

While the proposed techniques are explained with consecutive convolution layers above, special care should be taken if there are multiple paths of layers. In this case, we have to make two decisions that are related to each other. One is to determine how many shared input tiles to be stored in the FMEM banks, as discussed in Section 4.1.3 with an example of Figure 4.5. The other is to determine the order of path processing. Note that the number of input tiles to leave in the FMEM for the next path can be determined independently of the path processing order while it affects the memory access overhead and the computation time of the current path. Suppose that the number of FMEM banks is N and the number of parallel paths is B. The total number of cases to be compared is $B! \times (N-1)^{B-1}$ where the first term is the number of all possible processing orders, and the second term is the number of total scenarios of how many inputs to be left from each path. Since at least one on-chip memory bank is required to process layers in a path, the maximum number of tiles to be left is $(N-1)$, and the minimum number is set to one in order to avoid the unhidden DRAM access delay in the start of the next path. In the case of Inception v3 [84], however, it takes too much time to compare all possible cases. If the number of on-chip memory banks is eight and there are four paths like Inception block A,

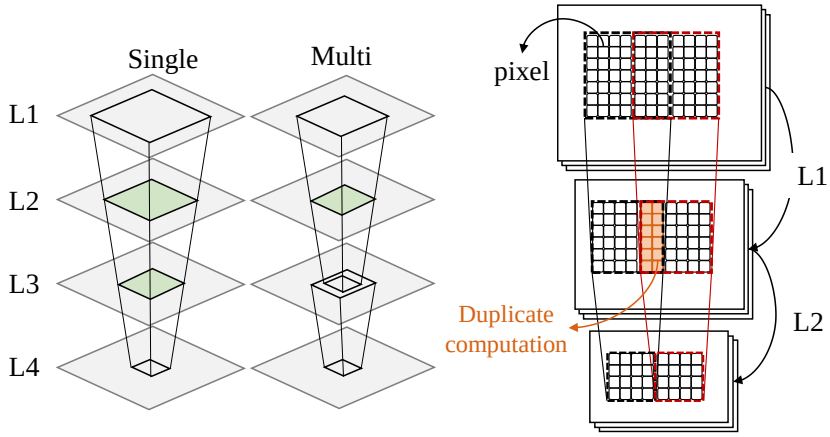
the total number of cases is $4! \times 7^3 = 8,232$. To reduce the complexity, a greedy heuristic is devised to reduce the number of cases to compare; For a given path ordering, we determine the number of input tiles to leave sequentially from the first path. We compare all the selected cases to find the best decision about the path processing order and the number of input tiles to leave from each path.

4.3 Layer Fusion Technique

In the proposed multi-bank on-chip memory management techniques, as explained in previous sections, we allow the output feature map to be stored in the on-chip FMEM in order to reduce the DRAM access volume between two consecutive convolution layers. Such feature map reuse is extended over consecutive convolution layers in the recently proposed layer fusion technique [31]. The layer fusion technique transforms a deep learning network into a new one by splitting one layer path into multiple paths. Therefore, after applying the layer fusion technique to make a new deep learning network that gives the same calculation result, the proposed technique can be applied. In this section, we first review the layer fusion technique and point out that it may increase the off-chip memory access for filter weights as well as computation delay. Next, we present a hybrid method between the layer fusion and the per-layer processing that has been assumed so far, in order to obtain better performance than them.

4.3.1 Layer Fusion Technique

Layer fusion is a method of fusing the processing of multiple layers to minimize off-chip feature map data transfer. Basically, a *computation pyramid* is formed for an output pixel at the bottom layer, as shown in Figure 4.11(a). The number of input pixels on an upper layer increases as the layer goes higher. The key idea is that the feature map segments in the computation pyramid are processed without DRAM access, storing all intermediate feature maps (green color in the figure) in the on-chip memory. We may



(a) Layer Fusing Technique (b) Computation Overhead

Figure 4.11: Layer fusion techniques and computation overhead

expand the computation pyramid by having more than one output pixel at the bottom layer. Layer fusion parameters, including how many layers can be fused and how many output pixels to include, need to be determined considering the on-chip memory size. In case the on-chip FMEM is large enough to store the entire input feature map, layer fusion becomes identical to per-layer processing that is assumed so far. Thus layer fusion is meaningful when the input feature map size is larger than the on-chip FMEM size. Figure 4.11(a) shows two possible fusing candidates; Four layers can be fused in a single pyramid or in two pyramids. The fusion technique proposed in [31] explores the design space of fusing exhaustively to make the best fusing decision.

While the fusing technique maximizes the feature map reuse across multiple layers, it brings about penalties in computation and extra DRAM access. Figure 4.11(b) shows the source of the computation penalty. In the figure, the black box and the red box represent two computation pyramids, sharing a set of pixels in the intermediate layers, colored orange in the figure. Convolution computation of two pyramids produces those pixels twice, which corresponds to the computation penalty. If the pixels in the intermediate layer are stored in on-chip memory, the redundant computation may be avoided. But it will reduce the number of available memory banks used to process other layers and incur

additional DRAM accesses. Thus, we assume that pure layer fusion causes delay overhead due to duplicate computations. On the other hand, in case all filter weights in the computation pyramid cannot be stored in the on-chip memory, we need to load the filter weights from the DRAM for each computation pyramid. Thus we need to compare the gain and the penalty to determine if the fusing technique is beneficial.

4.3.2 Hybrid Fusion Technique

In search of a good balance between feature map reuse by fusing and filter weight reuse by per-layer processing, we propose a hybrid method that performs fusing layers, relaxing the constraint that all feature maps involved in the computation pyramid should be stored in the on-chip FMEM. In the proposed hybrid method, called *hybrid fusion*, the feature map size can be larger than the FMEM size in a computation pyramid. We apply the proposed *MIN* or *HIDE* policy for the computation pyramid in the hybrid fusion method.

Figure 4.12 shows how three techniques differ in terms of computation overhead, off-chip memory access for feature maps, and off-chip memory access for filter weights: per-layer processing without fusion, pure layer fusion, and the proposed hybrid fusion. It is assumed that the input feature map of $Layer_n$ is divided into four computation pyramids in the pure layer fusion technique. The first column shows the computation overhead due to duplicate computation of the overlapped pixels between two computation pyramids. If two pyramids have an overlapped region (colored orange) in $Layer_n$, the red-colored portion of the feature map in the next layer, $Layer_{n+1}$, is computed twice. Thus the pure fusion technique incurs duplicate computations for the overlapped region three times, as illustrated in the second row, while there is no such penalty in the per-layer processing. If we form two computation pyramids only in the hybrid method, such a computation penalty is paid only once.

The second column of the figure compares the off-chip memory access volume re-

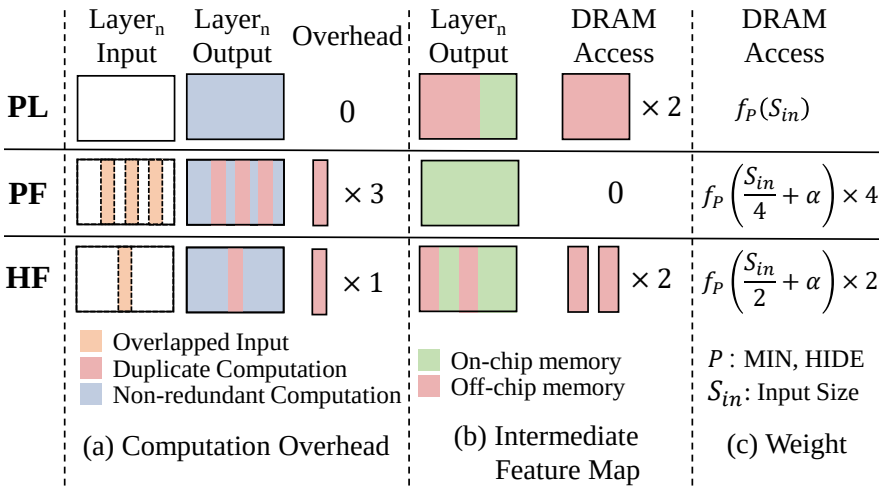


Figure 4.12: Comparison among three different techniques: per-layer processing (PL), layer fusion (PF), hybrid fusion (HF)

quired for feature maps. The portion of the feature map, colored green, can be stored in the on-chip FMEM, and the remaining portion needs to be stored to the off-chip memory after processing $Layer_n$ and read back to process in $Layer_{n+1}$. In the per-layer processing technique, only a part of the feature map can be stored in the on-chip memory, while all feature maps can be stored in the pure fusion technique. On the other hand, the hybrid fusion technique increases the green part of the feature map even though it incurs off-chip DRAM memory access for the remaining part.

The third column of Figure 4.12 compares the volume of the filter weights that need to be loaded from the off-chip DRAM in the processing of $Layer_n$ in case the on-chip WMEM cannot accommodate all filter weights. Since the exact value cannot be formulated analytically but obtained by simulation with the assumed MIN or HIDE policy, we denote it as a function of the size of the input feature map segment, $f_P(S_{in}), policy \in \{MIN, HIDE\}$. As mentioned in Section 4.3.1, when the layer fusion technique is applied, the overlapping feature map may occur in the intermediate layer. In Figure 4.12(c), the size of the additional feature map is represented by α. If the increasing slope of the function f_P over the input size is less than 1, which is usually the case, the per-layer

processing technique requires the smallest DRAM access volume.

In summary, the proposed hybrid method is located in the middle in all three comparison metrics between the per-layer processing technique and the pure fusion technique. In the proposed technique, we vary the computation pyramid size and choose the best size, considering all three metrics. Note that the per-layer processing and the pure fusion technique are two extreme cases of the proposed hybrid fusion technique; the former is obtained by making the pyramid include the entire feature map and the latter by making the pyramid include the maximum portion of the feature map that can be stored in the on-chip FMEM.

4.4 Experiments

4.4.1 Setup

4.4.1.1 Target Neural Processing Unit

We implement the proposed technique as the compiler for MIDAP [19], whose abstract model is illustrated in Figure 4.13. There are N_f adder trees to produce the partial result for N_f output pixels. Each adder tree is associated with a separate filter buffer and a shared input feature map buffer. If the width of the adder tree is N_{pe}, the total number of MAC units becomes $N_f \times N_{pe}$. A pixel is assumed to be an 8-bit fixed point. In MIDAP, different banks are assigned for feature maps and filter weights to avoid read access conflicts; there are N banks to store feature maps (FMEM) and M banks for filter weights (WMEM). Separation of FMEM banks and WMEM banks can be realized by hardware or software that assigns different address ranges for feature maps and filter weights. Note that the unit of read access from FMEM and WMEM to buffers is N_{pe} pixels in the channel direction while the unit of write access is N_f output pixels produced simultaneously from the datapath. As we assume in this paper, MIDAP has one unified memory for the feature map, and a compiler controls the NPU behavior. Moreover, we

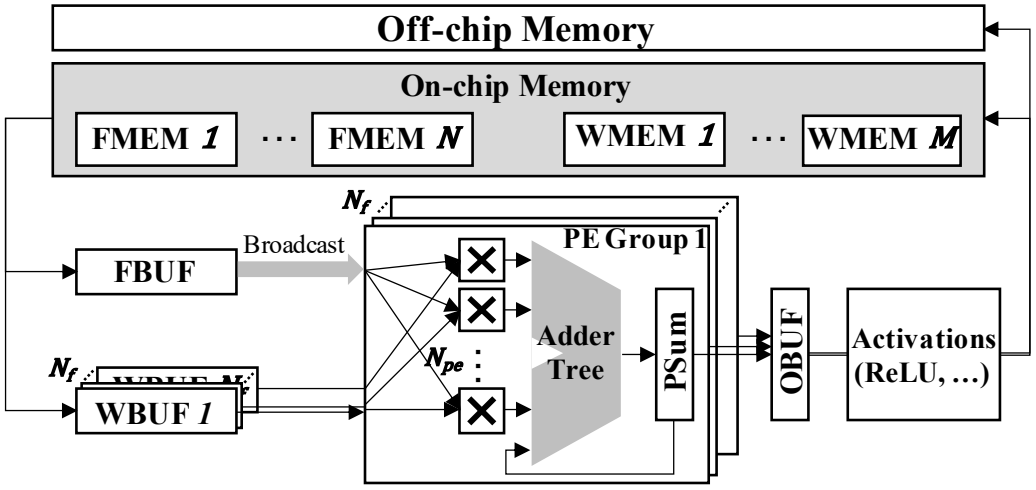


Figure 4.13: MIDAP architecture

can get the cycle-accurate results from the simulator [1], because MIDAP has no dynamic resource contention in the datapath as well as between on-chip memory and buffers.

The architecture parameters we can vary are shown in Table 4.1. The other NPU parameters, N_f, N_{pe}, and the number of WMEM banks, are set to 16, 64, and 32, respectively. Note that the number of WMEM banks is fixed to 2 for each PE group to support double buffering while we vary the number of FMEM banks.

Table 4.1: Hardware architecture parameters used in the experiments

Parameters	Values
On-chip FMEM size (KiB)	256, 512, 768, 1024
# of FMEM bank	4, 8
On-chip WMEM size (KiB)	288
Off-chip DRAM	LPDDR4*
DRAM data rate (GT/s)	1.6, 2.4, 3.2

*: 32bit per channel / 2 channels / 1 rank

[1] https://github.com/cap-lab/MidapSim

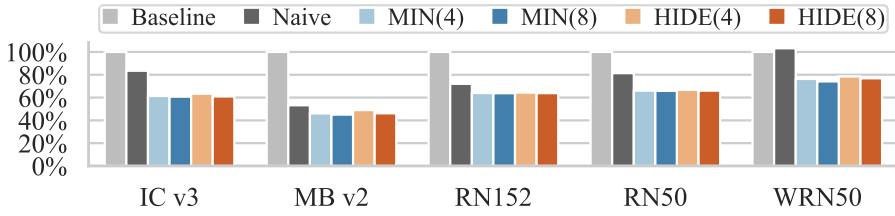
4.4.1.2 Comparison Targets

For comparison purpose, we implement two comparison techniques: baseline and 'naive reuse.' The baseline technique is a technique that does not reuse data between consecutive layers. Therefore, it uses all FMEM banks for input. To reduce the PE delay due to off-chip DRAM access, we implement the double buffering technique.

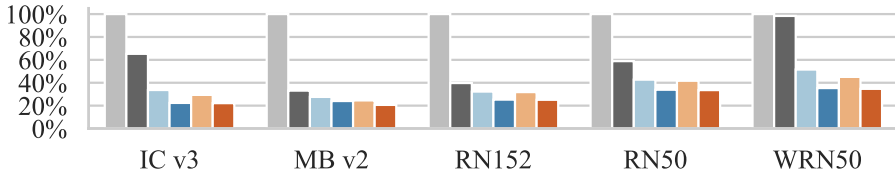
The only difference between the baseline and the 'naive reuse' is that the 'naive reuse' reuses data between consecutive layers. To implement the 'naive reuse' technique, we assign two banks each for the input and the output and implement the double buffering technique in the two input banks to reduce delay caused by the off-chip memory access. Also, to reuse data between consecutive layers, the roles of input and output memory are switched each time processing a layer is completed.

4.4.2 Performance Comparison of MOMM Techniques

In the first set of experiments, we vary the number of banks for feature maps and compare the policies with the widely used CNNs as follows: Inception v3 (IC v3) [84], MobileNet v2 (MB v2) [49], ResNet 152, 50 (RN152, RN50) [32], and WideResNet 50 (WRN50) [9]. In all experiments, the input image size is 224×224 except for Inception v3, whose input size is 299×299. The input size is set to each network's default input size when inferencing the ImageNet dataset [39]. Since the on-chip memory size is fixed to 512KiB, as the number of banks increases, the size of each bank becomes smaller. Figure 4.14(a) shows the relative volume of DRAM access during execution, compared with the baseline technique. It can be observed that the baseline technique accesses off-chip memory a lot. Especially for the MobileNet v2, the volume of off-chip memory access is about 1.88 times larger than 'naive reuse.' The performance gain comes from data reuse between layers because other configurations, such as on-chip memory size and the number of banks, are the same. However, in the case of WRN50, the 'naive reuse' technique has more DRAM access than the baseline because the WRN50's filter weight



(a) DRAM access size comparison between bank management policies



(b) PE delay comparison between bank management policies

Figure 4.14: Performance comparison: The number of banks is shown in the parenthesis

is large, and the baseline technique, which uses all FMEM banks for input, can reuse more filter weights.

Although the 'naive reuse' reduces off-chip DRAM access by reusing data between consecutive layers, the proposed policies further reduce off-chip memory access. Since the 'naive reuse' technique also uses feature map data reuse between layers, the performance gain is attributed to multi-bank management by judiciously sharing the FMEM banks for input and output feature maps. As expected, the MIN policy produces the minimum DRAM traffic, up to 28.1% gain over the 'naive reuse' technique. Policies *MIN(4), MIN(8), HIDE(4), and HIDE(8)* reduce DRAM access by an average of 19.2%, 20.3%, 16.9%, and 19.2%, respectively, compared to the 'naive reuse'.

The second comparison is made on the PE delay caused by unhidden off-chip DRAM accesses. Firstly, we could observe that the PE delay takes a significant portion of the end-to-end computation time of applications in the 'naive reuse' technique: 6.7, 19.1, 8.8, 16.0, and 10.2% for Inception v3, MobileNet v2, ResNet 152, ResNet 50, and WideResNet 50, respectively. As shown in Figure 4.14(b), in the case of the baseline, it can be observed that a lot of the PE delay occurs because the input of the next layer has to be

loaded from the off-chip memory. As expected, the HIDE policy reduces the PE delay most. Since the 'naive reuse' also uses the double buffering technique, the PE delay reduction comes from multi-bank management. The largest gain, about 66.0% over the 'naive reuse,' is obtained for the Inception v3 network that confirms the importance of careful on-chip bank management. Since it has many parallel branches, the number of active feature maps varies widely depending on the layer processing order and bank assignment. Policies *MIN(4), MIN(8), HIDE(4), and HIDE(8)* reduce the PE delay by an average of 31.8%, 47.2%, 36.8%, and 49.7%, respectively, compared to the 'naive reuse'.

From Figure 4.14, we can notice the trade-off between DRAM access size and the PE delay time between the MIN policy and the HIDE policy in three benchmarks, Inception v3, MobileNet v2, and WideResNet 50. On the other hand, for two ResNet benchmarks, it is observed that the performance gap between the MIN policy and the HIDE policy is not significant. It is because minimizing the DRAM access size also reduces the PE delay in most convolutional layers, and the majority of the PE delay is caused by the residual connection, which cannot be hidden by the HIDE policy. For example, when compiling ResNet 152 with the MIN policy with 4 FMEM banks, 73.5% of the PE delay is attributed to the residual connections, and the only 26.5% comes from convolutional layers.

4.4.3 Multiple Path

As explained in Section 4.2.4, we need to determine the processing order of multiple paths and the number of banks for reusing shared inputs. Since Inception v3 has five different types of blocks that all consist of multiple paths, we evaluate the proposed technique to handle multiple paths in this experiment. Table 4.2 shows the experimental results for five different types of Inception blocks [84] using the MIN policy. The second row represents the proposed method that searches the best processing order by exhaustive search and uses a greedy heuristic to determine the number of reused banks for each path.

The reused banks are banks that store shared inputs that will be used for the next layer path. The four rows in the next group show the results when we fix the number of reused banks after finding the best processing order for each case, and the last row shows the results obtained by exhaustive search to determine the number of reused banks. In the table, four cells are blank, meaning infeasible scenarios. It is noteworthy that the proposed heuristic could find the best solution in all Inception blocks.

Table 4.2: Comparison of DRAM access size with different methods to determine the number of banks for reusing shared inputs between paths in Inception v3 blocks (FMEM: 128KiB×4, unit: KiB)

# of reused banks	A	B	C	D	E
Greedy Search	1,696	1,924	2,656	1,807	6,195
No reuse	2,760	2,971	3,293	2,458	6,803
1 bank	2,291	2,622	3,013	2,075	6,195
2 banks	1,822	2,273	2,656	1,807	-
3 banks	1,870	1,924	-	-	-
Best	1,696	1,924	2,656	1,807	6,195

4.4.4 Design Space Exploration of NPU Architecture

In the next set of experiments, we vary the other architecture parameters as presented in Table 4.1, fixing the number of feature map banks to four. Figure 4.15 shows the off-chip DRAM access size, varying the overall size of on-chip memory and off-chip DRAM data rate. Figure 4.16 shows the PE delay. In this experiment, we only compare the proposed techniques with the 'naive reuse' technique because the baseline shows too bad results compared to others in PE delay comparison. Three policies are distinguished by colors: grey for 'naive reuse,' blue for MIN, and orange for HIDE policy. In all figures, the numbers below the x-axis represent the data rate of LPDDR4 and the total size of the on-chip memory in the second row.

Three observations can be made from this experiment. First, the DRAM access size

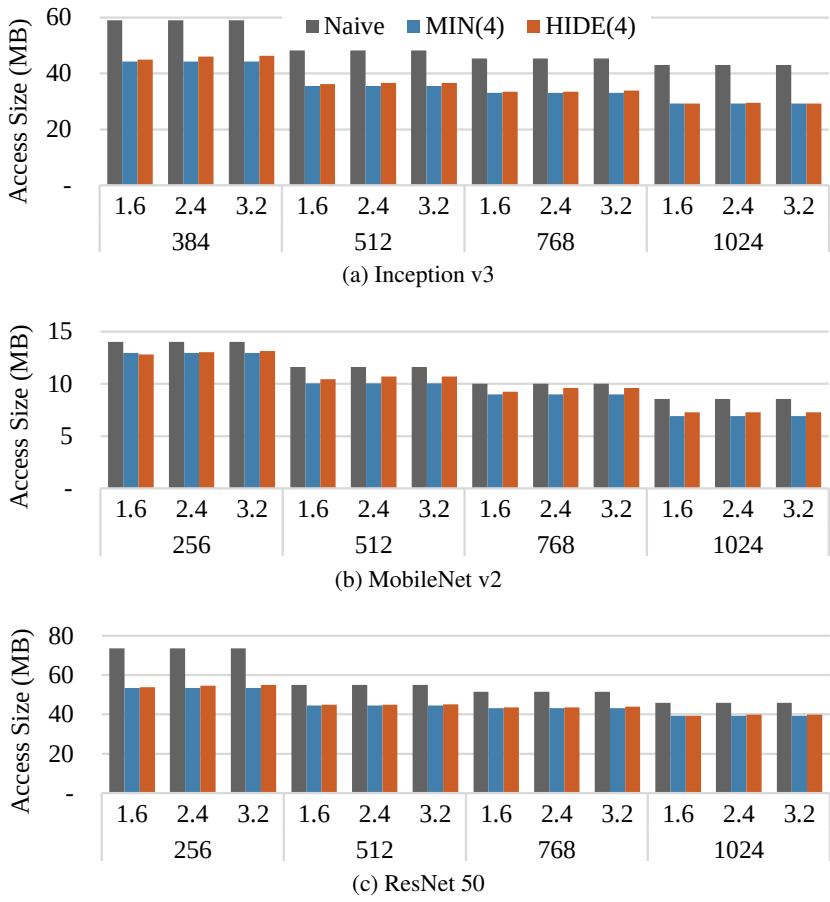


Figure 4.15: DRAM access size comparison by varying the on-chip FMEM size and off-chip DRAM data rate

is only affected by the total on-chip size in the MIN policy because it just reduces the memory access volume ignoring the DRAM data rate. On the other hand, the DRAM access size varies in the HIDE policy. As the on-chip memory size increases, the gap between the HIDE policy and the MIN policy tends to decrease in all benchmarks. Second, as the size of on-chip memory reduces, the bank management becomes more critical in the performance so that the HIDE policy outperforms the MIN policy more in terms of PE delay. Third, as discussed in Section 4.4.2, the difference between the MIN policy and HIDE policy is not significant in ResNet 50. Lastly, as the on-chip memory size increases, the off-chip DRAM access size and the PE delay tend to decrease as expected.

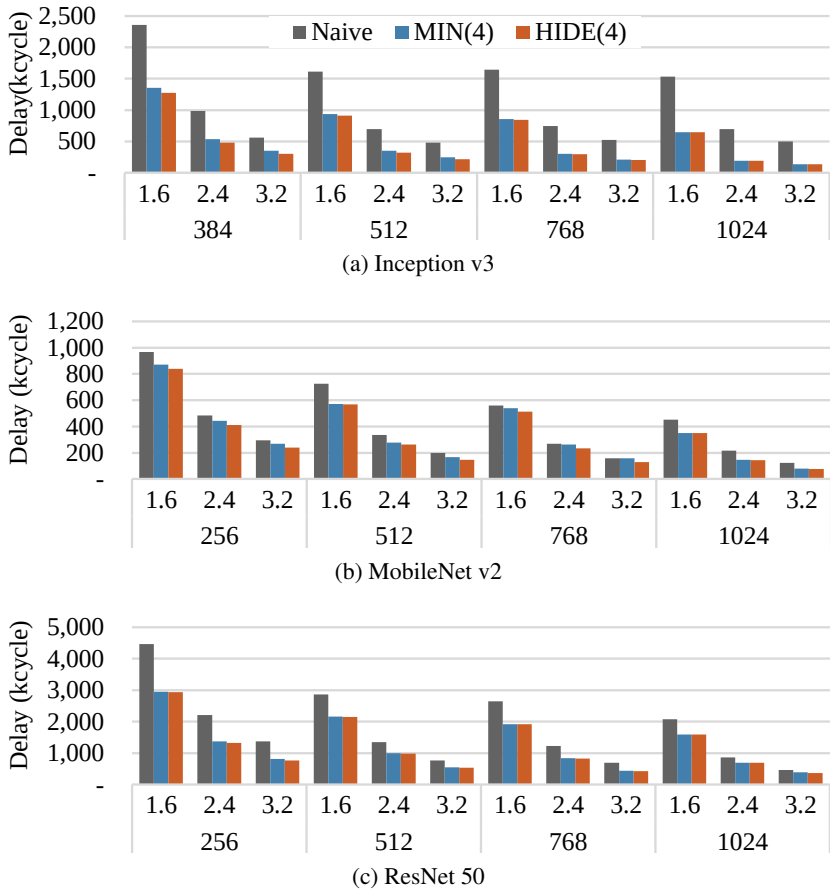
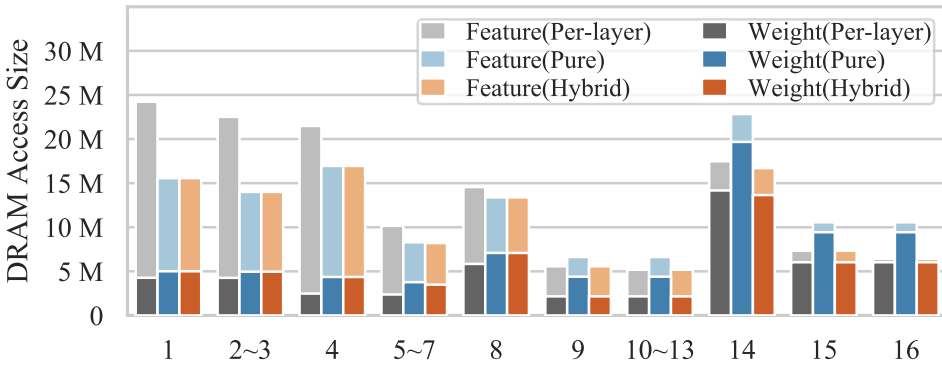


Figure 4.16: PE delay comparison by varying the on-chip FMEM size and off-chip DRAM data rate

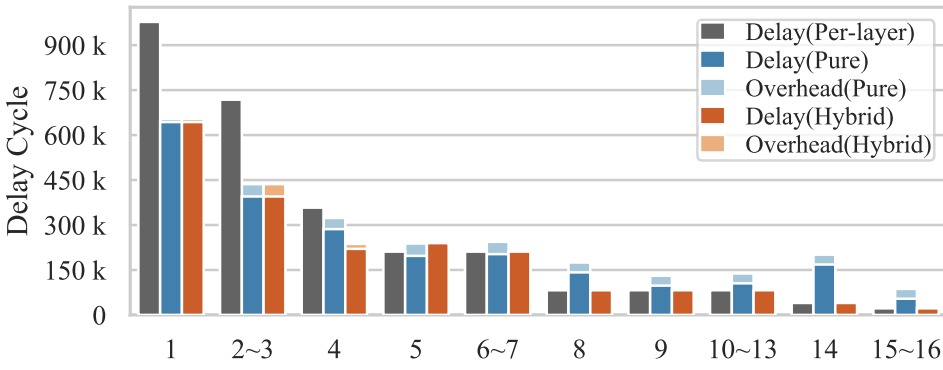
In all benchmarks, we could observe that the trade-off between the HIDE policy and the MIN policy. Compared to the MIN policy, the HIDE policy reduces the PE delay by up to 18.0, 17.0, and 6.2% for Inception v3, MobileNet v2, and ResNet 50, respectively. On the other hand, the MIN policy reduces the DRAM access size for Inception v3, MobileNet v2, and ResNet 50 by up to 4.5, 6.5, and 2.7%, respectively, compared to the HIDE policy. For Inception v3 and ResNet 50, the architecture parameters that show the greatest differences between the MIN policy and the HIDE policy are 256KB total FMEM size and 3.2 GT/s DRAM data rate. For MobileNet v2, the architecture parameters are the FMEM size of 768 KB and DRAM data rate of 3.2 GT/s.

4.4.5 Hybrid Fusion Technique

Lastly, we compare the proposed hybrid fusion technique with the per-layer processing technique and the pure layer fusion technique with ResNet 50 [32] that consists of 16 blocks that have three or four convolution layers inside. Refer to [32] for the network structure. ResNet 50 is used as a backbone network in various image segmentation networks that take a large size of the input image; the input image size is 512 × 512. We assume that the total FMEM size is 512KiB and the FMEM has four banks of 128KiB each.



(a) DRAM access size using the MIN policy (Y-axis unit: byte)



(b) PE delay time and computation overhead (Y-axis unit: cycle)

Figure 4.17: Comparison between three fusing techniques (network: ResNet 50, input Size: 512×512, FMEM: 128KiB×4)

Figure 4.17 displays two experimental results with different objective functions; One

is to minimize the DRAM access volume with the MIN policy, and the other is to minimize the latency with the HIDE policy. Figure 4.17(a) compares the DRAM access size among three techniques, separating the requirements for feature maps and filter weights. Since the DRAM access size depends on the initial bank state, we make three techniques have the same initial bank state that is obtained after the hybrid technique is assumed to be used for the previous block. For instance, the initial bank state for the second block is assumed to be the same as the output bank state of the first block with the hybrid technique. The experimental results show that the pure fusing technique is better than the per-layer processing technique in the initial blocks, from *Block 1* to *Block 8*, while the reserve is true from *Block 9* to *16* since it increases the DRAM access volume for filter weights significantly as discussed in [31]. Note that the proposed hybrid fusion method is no worse than the other two methods in all blocks since it includes the other methods as extreme cases, searching for the best size of the computation pyramid exhaustively. By applying the hybrid method, 16.8% of off-chip access is reduced compared to the per-layer processing method.

The processing delay due to DRAM access and duplicate computation are compared among the three techniques in Figure 4.17(b). It is obtained by running the entire networks on the cycle-level simulator with each technique separately, considering the extra delay due to unhidden DRAM access for feature maps and filter weights as well as resource contention. The per-layer processing is par with the hybrid method and faster than the pure fusion technique in the latter blocks; it means that most of the DRAM access can be hidden by our HIDE policy in the per-layer processing technique. The hybrid method is faster than the per-layer processing technique in *Block 1*, *Block 2*$\sim$*3*, and *Block 4* by 20.6%, 23.2%, and 9.8%, respectively.

In summary, if we compare the end-to-end results of the given network among the three techniques, the hybrid fusion technique gives noticeably better performance, reducing 10.6% memory access volume than the pure layer fusion technique and 25.1% PE

delay than the per-layer processing technique.

4.5 Related Work

Since reducing DRAM access is well recognized as an important objective in the NPU design, all NPU architectures aim to maximize the data reuse in the on-chip memory. There are two approaches to maximize data reuse in on-chip memory. The first one is to maximize data reuse within one convolution operation in on-chip memory [25, 26]. SmartShuttle [25] and ROMANet [26] propose to switch different data reuse schemes and the corresponding tiling factor for each convolution layer. However, they do not consider sharing feature maps between adjacent layers, which is a key issue in the proposed techniques.

The other approach is to reuse feature map data between adjacent layers in the on-chip [31, 85, 86, 87]. FusedCNN [31] fused multiple CNN layers to share the feature maps between two adjacent layers, which is already reviewed in Section 4.3.1. As described in Section 4.3, we apply the proposed techniques to the network to which the layer fusion technique was applied and improve the layer fusion technique in a hybrid manner. Tangram [85] also attempted to reuse the output feature map in the accelerator through the inter-layer pipelining. MEM-OPT [86] used the input cache scheduling algorithm, which prioritizes the reuse of input and output, on the secondary cache system architecture for feature map data reuse. The other work proposed by Kim et al. [87] decides the processing order of layers when the network has parallel branches, considering the feature map sharing between connected layers. However, all these works differ from ours in that they did not consider the DRAM access delay and multi-bank management.

Similar to the proposed method, Shortcut Mining [88] reused the output feature map in the on-chip buffers and increased the utilization of the on-chip buffer by allocating data per bank. However, they separated the buffers for each role, such as buffers for computing input, buffers for a residual connection, and buffer for prefetching. Also, they fixed the

number of banks for each role to make bank management easier. On the other hand, in the proposed method, banks can be used more freely because banks' role is not specified, and we optimize DRAM access size and processing delay further by changing the number of banks to be processed even during processing one layer.

Chapter 5

Conclusion

In this dissertation, we devise three methods for hardware-aware optimization of an embedded deep learning system. First, a systematic optimization methodology is devised to apply software optimizations for deep learning and traditional system optimizations through the experience of participating in the Low Power Image Recognition Challenge (LPIRC) [1, 2]. Based on the methodology, we implement a novel deep learning framework, called C-GOOD, that generates a C code that can be run on any embedded platform. It is an optimization-aware framework in which a user can specify which optimization techniques will be applied to a given network. Then the optimized C-code is automatically generated, which helps the user to explore the design space of network selection and software optimization. A software optimization methodology is also introduced to apply optimization techniques systematically: pipelining, layer-wise Tucker decomposition, layer-wise quantization, merging batch normalization into weights, and input size reduction. The proposed methodology has been applied to three different hardware platforms: the Jetson TX2 [3], the Odroid XU4 [4], and the Samsung Reconfigurable Processor [5]. Experimental results show that the baseline C code generated from C-GOOD is better than the Darknet framework in terms of fps performance. In addition, the proposed optimization methodology improves the performance by 2.2 to 25.83 times over the baseline unoptimized code for a given hardware platform.

Next, we devise a scheduling framework of deep learning applications for embedded devices with heterogeneous processors. We exploit both task-level and data-level parallelism in the scheduling of a deep learning application. In particular, we propose to consider five different PE configurations for a multi-core CPU to exploit both types of parallelism in various ways. We also consider the DVFS policy and the CPU utilization constraints to avoid thermal throttling in the profiling of tasks on each processing element. We use a GA-based method for scheduling a single deep learning application. The GA-based scheduler is also applied to the scheduling of multiple deep learning applications. We modify the schedule-based schedulability analysis to apply to non-preemptive tasks. We verified the proposed scheduling methods by comparing the results with the measured ones in two real hardware platforms: Galaxy S9 and HiKey970. By considering several practical issues in the real implementation, we could achieve quite an accurate estimation, with the error smaller than 7% except for one case. The primary source of error is the inaccuracy of task and communication profiling. Nonetheless, we believe the comparison with the actual implementation is a meaningful contribution.

Finally, for a deep learning accelerator, we first define the multi-bank on-chip memory management (MOMM) problem to minimize the DRAM access overhead in the processing of convolution neural networks (CNNs) on a CNN accelerator with a multi-bank on-chip memory. By estimating the processing time and the memory access delay, and considering the memory contention, we devise three on-chip memory management policies. The DRAM first Storing (DFS) policy stores the overflowed output feature map to DRAM, to reduce the bank occupation by the output feature map during execution. The DRAM access minimization policy and the DRAM access hiding policy determine the assignment of feature map banks to the input and output feature maps considering the objective function, minimizing the DRAM access volume and minimizing the processing delay, respectively. In addition, we propose a method of determining the processing order and the number of banks for reusing shared input when a CNN has parallel paths.

Furthermore, we extend the MOMM problem to layer fusion and propose a hybrid fusion technique that searches for an optimal computation pyramid structure by varying the input feature map size of the top layer. Experiments have been conducted by running benchmark CNNs on a cycle-accurate adder-type NPU simulator. Experimental results confirm the superior performance of the proposed techniques over the baseline technique and 'naive resue' technique. The proposed methods reduce the off-chip DRAM access and the processing delay up to 55.0% and 79.4%, respectively, compared to the baseline. For layer fusion, we demonstrate the superiority of the proposed hybrid fusion technique through ResNet 50 simulation with an input size of 512×512. The proposed hybrid fusion technique reduces 10.6% DRAM access volume than the pure layer fusion technique and 25.1% PE delay than the per-layer processing technique.

Since many deep learning applications and accelerators are still being researched, there are many future research topics for optimizing embedded deep learning systems. In particular, since there is a need for a multi-NPU system by bundling multiple NPUs, it is future work to expand the MOMM method devised in this dissertation to optimize multi deep learning applications in a multiple NPU system.

Bibliography

[1] Kent Gauen, Rohit Rangan, Anup Mohan, Yung-Hsiang Lu, Wei Liu, and Alexander C Berg. Low-power image recognition challenge. In *Proceedings of ASP-DAC*, pages 99–104. IEEE, 2017.

[2] Sergei Alyamkin, Matthew Ardi, Alexander C Berg, Achille Brighton, Bo Chen, Yiran Chen, Hsin-Pai Cheng, Zichen Fan, Chen Feng, Bo Fu, et al. Low-power computer vision: Status, challenges, and opportunities. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):411–421, 2019.

[3] Nvidia. Nvidia jetson. `http://www.nvidia.com/object/embedded-systems-dev-kits-modules.html`, 2017.

[4] hardkernel. Odroud-xu4. `http://www.hardkernel.com/main/products/prdt_info.php`, 2017.

[5] Dongkwan Suh, Kiseok Kwon, Sukjin Kim, Soojung Ryu, and Jeongwook Kim. Design space exploration and implementation of a high performance and low area coarse grained reconfigurable processor. In *Proceedings of ICFPFT*, pages 67–70. IEEE, 2012.

[6] ARM. Arm compute library. `https://www.arm.com/why-arm/technologies/compute-library`, 2017.

[7] Samsung Electronics. Galaxy s9. `https://www.samsung.com/global/galaxy/galaxy-s9`, 2018.

[8] Kirin. Hikey970. `https://www.96boards.org/product/hikey970/`, 2018.

[9] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[10] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[11] Joseph Redmon. Yolo: Real-time object detection. http://pjreddie.com/darknet/yolo/, 2013–2016.

[12] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[13] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[14] Eunhyeok Park, Junwhan Ahn, and Sungjoo Yoo. Weighted-entropy-based quantization for deep neural networks. In *Proceedings of CVPR*, pages 5456–5464. IEEE, 2017.

[15] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530*, 2015.

[16] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of ISCA*. ACM, 2016.

[17] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of ISCA*. ACM, 2017.

[18] F. Sijstermans. The nvidia deep learning accelerator. In *HotChips, http://nvdla.org*, 2018.

[19] Donghyun Kang, Jintaek Kang, Hyungdal Kwon, Hyunsik Park, and Soonhoi Ha. A novel convolutional neural network accelerator that enables fully-pipelined execution of layers. In *Proceedings of ICCD*. IEEE, 2019.

[20] Joseph Redmon. Imagenet classification. https://pjreddie.com/darknet/imagenet/, 2013–2016.

[21] Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger. pages 7263–7271, 2017.

[22] Song Yao, Shuang Liang, Junbin Wang, Zhongmin Chen, Shaoxia Fang, Lingzhi Sui, Qian Yu, Dongliang Xie, Xiaoming Sun, Song Han, Yi Shan, and Yu Wang. The evolution of deep learning accelerators upon the evolution of deep learning algorithms. In *Hot Chips 30 Symposium*. IEEE, 2018.

[23] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. volume 42, pages 269–284. ACM New York, NY, USA, 2014.

[24] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-x: An accelerator for sparse neural networks. In *Proceedings of Micro*, pages 1–12. IEEE, 2016.

[25] Jiajun Li, Guihai Yan, Wenyan Lu, Shuhao Jiang, Shijun Gong, Jingya Wu, and Xiaowei Li. Smartshuttle: Optimizing off-chip memory accesses for deep learning accelerators. In *Proceedings of DATE*, pages 343–348. IEEE, 2018.

[26] Rachmad Vidya Wicaksana Putra et al. Romanet: Fine-grained reuse-driven data organization and off-chip memory access management for deep neural network accelerators. *arXiv preprint arXiv:1902.10222*, 2019.

[27] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2019.

[28] J. Li et al. Squeezeflow: A sparse cnn accelerator exploiting concise convolution rules. 68(11):1663–1677, 2019.

[29] Huiyu Mo, Leibo Liu, Wenjing Hu, Wenping Zhu, Qiang Li, Ang Li, Shouyi Yin, Jian Chen, Xiaowei Jiang, and Shaojun Wei. Tfe: Energy-efficient transferred filter-based engine to compress and accelerate convolutional neural networks. In *Proceedings of MICRO*, pages 751–765. IEEE, 2020.

[30] Dongyoung Kim, Junwhan Ahn, and Sungjoo Yoo. Zena: Zero-aware neural network accelerator. *IEEE Design Test*, Feb 2018.

[31] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer cnn accelerators. In *Proceedings of Micro*. IEEE, 2016.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of CVPR*, pages 770–778. IEEE, 2016.

[33] TIOBE. Tiobe index for november 2020. https://www.tiobe.com/tiobe-index/, 2020.

[34] William J. Dally, Yatish Turakhia, and Song Han. Domain-specific hardware accelerators. *Commun. ACM*, 63(7):48–57, June 2020.

[35] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[36] Ross Girshick. Fast r-cnn. In *Proceedings of ICCV*, pages 1440–1448. IEEE, 2015.

[37] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-fcn: Object detection via region-based fully convolutional networks. In *Advances in neural information processing systems*, pages 379–387, 2016.

[38] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *Proceedings of ECCV*, pages 21–37. Springer, 2016.

[39] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proceedings of CVPR*, pages 248–255. IEEE, 2009.

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

[41] Openmp website.

[42] Inpyo Bae, Barend Harris, Hyemi Min, and Bernhard Egger. Auto-tuning cnns for coarse-grained reconfigurable array-based accelerators. In *Proceedings of CASES*. IEEE, 2018.

[43] Duseok Kang, DongHyun Kang, Jintaek Kang, Sungjoo Yoo, and Soonhoi Ha. Joint optimization of speed, accuracy, and energy for embedded image recognition systems. In *Proceedings of DATE*, pages 715–720. IEEE, 2018.

[44] Facebook Open Source. Caffe2. https://caffe2.ai, 2017–2018.

[45] Tensorflow. Tensorflow lite. https://www.tensorflow.org/mobile/tflite/, 2017–2018.

[46] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. pages 4700–4708, 2017.

[47] caffe2. models. https://github.com/caffe2/models, 2017-2018.

[48] shicai. Densenet-caffe. https://github.com/shicai/DenseNet-Caffe, 2017.

[49] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of CVPR*, pages 4510–4520. IEEE, 2018.

[50] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of CVPR*, volume 1, page 3. IEEE, 2017.

[51] ARM. Arm nn. https://www.arm.com/products/silicon-ip-cpu/machine-learning/arm-nn, 2018.

[52] ARM. Arm compute library. https://developer.arm.com/technologies/compute-library, 2017.

[53] Duseok Kang, Euiseok Kim, Inpyo Bae, Bernhard Egger, and Soonhoi Ha. C-good: C-code generation framework for optimized on-device deep learning. In *Proceedings of ICCAD*, pages 1–8. IEEE, 2018.

[54] Andrei Frumusanu. The samsung galaxy s9 and s9+ review. https://www.anandtech.com/show/12520/the-galaxy-s9-review/, 2018.

[55] Andrei Frumusanu. Hisilicon kirin 970 - android soc power & performance overview. https://www.anandtech.com/show/12195/hisilicon-kirin-970-power-performance-overview/, 2018.

[56] Tae-ho Shin, Hyunok Oh, and Soonhoi Ha. Minimizing buffer requirements for throughput constrained parallel execution of synchronous dataflow graph. In *Proceedings of ASP-DAC*, pages 165–170. IEEE, 2011.

[57] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

[58] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. Spea2: Improving the strength pareto evolutionary algorithm. *TIK-report*, 103, 2001.

[59] Shin-haeng Kang, Duseok Kang, Hoeseok Yang, and Soonhoi Ha. Real-time co-scheduling of multiple dataflow graphs on multi-processor systems. In *Proceedings of DAC*, page 159. ACM, 2016.

[60] Sanjoy K Baruah, Aloysius K Mok, and Louis E Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of RTSS*, pages 182–190. IEEE, 1990.

[61] Barry M Pangrle and Daniel D Gajski. Slicer: A state synthesizer for intelligent silicon compilation. In *Proceedings of ICCAD*, pages 42–45, 1987.

[62] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of Multimedia*, pages 675–678. ACM, 2014.

[63] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *Proceedings of NIPS Workshop*, 2011.

[64] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016. Software available from tensorflow.org.

[65] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 578–594, 2018.

[66] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205. IEEE, 2019.

[67] Seyyed Salar Latifi Oskouei, Hossein Golestani, Matin Hashemi, and Soheil Ghiasi. Cnndroid: Gpu-accelerated execution of trained deep convolutional neural networks on android. In *Proceedings of Multimedia*, MM '16, pages 1201–1205, 2016.

[68] Moustafa Alzantot, Yingnan Wang, Zhengshuang Ren, and Mani B Srivastava. Rstensorflow: Gpu enabled tensorflow for deep learning on commodity android devices. In *Proceedings of EMDL*, pages 7–12. ACM, 2017.

[69] Google. Renderscript. `https://developer.android.com/guide/topics/renderscript/compute`.

[70] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V Le, and Jeff Dean. A hierarchical model for device placement. In *Proceedings of ICLR*, 2018.

[71] Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. μlayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization. In *Proceedings of EuroSys*, pages 1–15, 2019.

[72] Nicholas D Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *Proceedings of IPSN*, pages 1–12. ACM/IEEE, April 2016.

[73] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, 13(3):260–274, 2002.

[74] Hamid Arabnejad and Jorge G Barbosa. List scheduling algorithm for heterogeneous systems by an optimistic cost table. *IEEE Transactions on Parallel and Distributed Systems*, 25(3):682–694, 2013.

[75] Sanjit Kumar Roy, Rajesh Devaraj, Arnab Sarkar, Sayani Sinha, and Kankana Maji. Optimal scheduling of precedence-constrained task graphs on heterogeneous distributed systems with shared buses. In *2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*, pages 185–192. IEEE, 2019.

[76] Henan Zhao and Rizos Sakellariou. Scheduling multiple dags onto heterogeneous systems. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 14–pp. IEEE, 2006.

[77] Guoqi Xie, Gang Zeng, Liangjiao Liu, Renfa Li, and Keqin Li. High performance real-time scheduling of multiple mixed-criticality functions in heterogeneous distributed embedded systems. *Journal of Systems Architecture*, 70:3–14, 2016.

[78] Hazem Ismail Ali, Benny Akesson, and Luís Miguel Pinho. Generalized extraction of real-time parameters for homogeneous synchronous dataflow graphs. In *Proceedings of PDP*, pages 701–710. IEEE, 2015.

[79] Yang Liu, Lin Meng, Ittetsu Taniguchi, and Hiroyuki Tomiyama. A dual-mode scheduling approach for task graphs with data parallelism. *International Journal of Embedded Systems*, 9(2):147–156, 2017.

[80] Hoeseok Yang and Soonhoi Ha. Pipelined data parallel task mapping/scheduling technique for mpsoc. In *Proceedings of DATE*, pages 69–74. IEEE, 2009.

[81] Jinook Song, Yunkyo Cho, Jun-Seok Park, Jun-Woo Jang, Sehwan Lee, Joon-Ho Song, Jae-Gon Lee, and Inyup Kang. An 11.5 tops/w 1024-mac butterfly structure dual-core sparsity-aware neural processing unit in 8nm flagship mobile soc. In *Proceedings of ISSCC*, pages 130–132. IEEE, 2019.

[82] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. *ACM SIGPLAN Notices*, 53(2):461–475, 2018.

[83] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer architecture letters*, 15(1):45–49, 2015.

[84] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of CVPR*. IEEE, 2016.

[85] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. Tangram: Optimized coarse-grained dataflow for scalable nn accelerators. In *Proceedings of ASPLOS*, pages 807–820, 2019.

[86] Gianmarco Dinelli, Gabriele Meoni, Emilio Rapuano, Tommaso Pacini, and Luca Fanucci. Mem-opt: A scheduling and data re-use system to optimize on-chip memory usage for cnns on-board fpgas. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 10(3):335–347, 2020.

[87] Doyun Kim, Kyoung-Young Kim, Sangsoo Ko, and Sanghyuck Ha. A simple method to reduce off-chip memory accesses on convolutional neural networks. *arXiv preprint arXiv:1901.09614*, 2019.

[88] Arash Azizimazreah and Lizhong Chen. Shortcut mining: Exploiting cross-layer shortcut reuse in dcnn accelerators. In *Proceedings of HPCA*, pages 94–105. IEEE, 2019.

Appendix

A Proposed Multi-bank On-chip Memory Management Algorithm

A.1 Multi-bank On-chip Memory (MOM) Manager

Algorithm 3 shows the proposed compiler's pseudocode. First, the function COMPILE_NETWORK determines the layer processing order (line 10) and initializes the bank state information as all empty (line 11). How to determine the layer processing order is determined by searching all paths as described in Section 4.2.4. After determining the processing order, it generates the control code for each layer in the processing order (line 12 $\sim$ line 14). The LAYER_COMPILER first determines the number of banks to store output activation or feature map according to its objective. As explained in Section 4.2.2 and Section 4.2.3, MIN policy selects the number of output banks which reduces the off-chip DRAM access most, and HIDE policy selects the number of output banks which reduces the processing delay most.

After determining the number of output banks, the MOM_MANAGER generates other control instruction information such as 'Process' and 'Load' commands and tile mapping information. We first set up the information on input, output, and operation with layer information, l, in line 19. When the manager processes all input tiles, the iteration is halted in line 22. Tile mapping information is a function that receives an input or output tile and provides in which banks the tile is stored. Therefore, the tile mapping information is represented by the function f from I/O tile domain T to the FMEM bank domain

Algorithm 3 Proposed compiler technqiue pseudocode

```
1:  Variables
2:      l        Layer information
3:      g        Current FMEM bank state information
4:      num_o      Number of output banks
5:      C        Control code list
6:      PI       A set of the processed input tiles
7:      MI       A set of input tiles currently mapped on FMEM
8:      f        Tile mapping information
9:  procedure COMPILE_NETWORK
10:     order ← determine the layer processing order.
11:     g ← empty FMEM bank information
12:     for each l from order do
13:         C, f, g ←LAYER_COMPILER(l, g)
14:         APPEND_CODE(C, f)    ▷ Append the control code
15: procedure LAYER_COMPILER(l, g)
16:     num_o ← the number of banks to store output activation
17:     return MOM_MANAGER(l, g, num_o)
18: procedure MOM_MANAGER(l, g, num_o)
19:     SETUP_LAYER(l)
20:     C ← []; PI ← ∅
21:     f, MI ← GET_INITIAL_STATE(g)
22:     while PI ≠ I do
23:         L_s, P_s ← DO_STEP(f, g, MI, PI, num_o)
24:         C.append((L_s, P_s))    ▷ Add a control code
25:         MI ← MI ∪ L_s − P_s    ▷ Update input mapping
26:         PI ← PI ∪ P_s    ▷ Update processed input tiles
27:     return C, f, g
```

including NULL and DRAM ($F \cup \{NULL, DRAM\}$). Bank state information is a function that receives a bank as an input and gives which data is currently stored in the bank. Therefore, the domain and the codomain of g are the bank domain F and feature map tile domain including an empty state ($T \cup \{\emptyset\}$), respectively. The MOM Manager generates a control code at each iteration step, represented as a tuple of input tile sets (L_s, P_s), as shown in line 23. L_s and P_s indicate input tile sets to be loaded on FMEM and processed at the current step, respectively. According to the decision made at DO_STEP, PI and MI are updated for each step, in line 25 and 26.

A.2 MIN policy

Algorithm 4 shows the pseudocode of the MIN policy. As described in Section 4.2.2, the MIN policy compares the total DRAM access volume by varying the number of on-chip output feature map and selects the best one (line 13 in MIN_POLICY). The DO_STEP generates a control code for each step in MOM_MANAGER according to the given number of output FMEM banks. In each step, it first checks whether there are input tiles to prefetch and determines the input tiles to load (GET_AVAILABLE_LOAD). If there is no input tile to load, it determines the set of input tiles to process (PROCESS). After the command is determined, it updates input and output mapping (UPDATE_IO_MAPPING).

Since the number of FMEM banks is limited, it is necessary to reduce the number of input banks to process by considering the bank state, g, and the number of output banks, num_o. Suppose that three out of four FMEM banks store the input feature map and one input bank produces one output bank. In this case, if three input banks are processed, only one bank can be used to store the output feature map. If we want to store more output banks in FMEM, the number of input banks to be processed must be reduced. Let O be a set of all output tiles to be stored on FMEM, and MO be a set of output tiles currently mapped on FMEM. If there remain output banks to be stored in FMEM (line 29), update the mapped output tile set, MO, by adding available FMEM banks (line 30). Afterward, we determine how many input banks to process by checking the corresponding output banks are mapped to DRAM or included in MO (lines 32-37). Remind that DO_STEP not only generates a control code but also update f and g as shown in line 24.

Algorithm 4 DRAM Access Minimization Policy

Variables
- l Layer information
- g Current FMEM bank state information
- min_o Number of output banks with least DRAM access
- L_s Input tiles to be loaded
- P_s Input tiles to be processed
- PI A set of the processed input tiles
- MI A set of input tiles currently mapped on FMEM
- UI A set of input tiles unloaded to FMEM
- O A set of output tiles to be stored in FMEM
- MO A set of output tiles currently mapped on FMEM

procedure MIN_POLICY
  $min_o \leftarrow \text{argmin}_{num_o}$(GET_DRAM_ACCESS(MOM_MANAGER(l, g, num_o))
    ▷ *Determine the # of output banks to remain on FMEM.*
  return MOM_MANAGER(l, g, min_o)

procedure DO_STEP(f, g, MI, PI, num_o)
  $L_s, P_s \leftarrow \emptyset$
  $UI \leftarrow I - MI - PI$
  if $UI \neq \emptyset$ **then** ▷ If there are banks to load, load inputs.
    $L_s \leftarrow$ GET_AVAILABLE_LOAD(UI, g, num_o)
    ▷ Leave the banks to store the outputs and load the inputs into the remaining banks.
  if $L_s = \emptyset$ **then** ▷ If there is no bank to load, process the inputs.
    $P_s \leftarrow$ PROCESS(g, MI, num_o)
  UPDATE_IO_MAPPING(f, g, L_s, P_s)
  return L_s, P_s

procedure PROCESS(g, MI, num_o)
  $O \leftarrow$ GET_OUT_TILES(num_o)
  $MO \leftarrow$ GET_OUT_TILES_ON_FMEM(g)
  if len(MO) $< num_o$ **then** ▷ If there are output banks to be mapped on FMEM,
    $MO \leftarrow MO \cup$ MAP_OUTPUT_TO_FMEM(g, O, MI)
    $num_p \leftarrow 0$
    for i_p **in** MI **do** ▷ For all input banks mapped to FMEM,
      if OUTPUT(i_p) $\not\subset O$ **or** OUTPUT(i_p) $\subseteq MO$ **then**
        ▷ If the outputs corresponding to i_p are mapped on DRAM or belong to MO,
          num_p++
      else
        break
  else
    $num_p \leftarrow$ len(MI)
  return ($MI[:num_p]$)

A.3 HIDE policy

Algorithm 5 Pseudo-code on how to determine the number of input tiles to process in the HIDE policy

```
 1: Variables
 2:     l          Layer information
 3:     g          Current FMEM bank state information
 4:     min_d      Number of output banks with least PE delay
 5:     d, delay   PE delay estimated by fast timing simulation
 6:     l_next     Layer to be processed next
 7:     max_o      Maximum number of banks which can store outputs
 8: procedure HIDE_POLICY
 9:     min_d ← argmin_{num_o}(GET_PE_DELAY(l, g, num_o))
10:                                  ▷ Determine the # of output banks to remain on FMEM.
11:     return MOM_MANAGER(l, g, min_d)
12: procedure GET_PE_DELAY(l, g, num_o)
13:     d ← ESTIMATE_DELAY(MOM_MANAGER(l, g, num_o))
14:     l_next ← GET_NEXT_LAYER(l)
15:     d ← d + ESTIMATE_DELAY(MOM_MANAGER(l_next, g, max_o))
16:     return d
17: procedure REDUCE_TILE_NUM(MI, num_p)
18:     min_delay ← int_max
19:     for num_i in range(1, num_p) do   ▷ For all input banks that can be processed
20:         delay ← PREDICT_DELAY(MI, num_i)   ▷ Get the estimated DRAM delay
21:         if min_delay ≥ delay then
22:             min_delay ← delay
23:             reduced_num_p ← num_i
24:     return reduced_num_p
```

The key challenge in the HIDE policy is to determine the number of input tiles to process. The proposed solution is to perform an exhaustive search by varying the number of input tiles as depicted in line 19 in Algorithm 5. The maximum number of input tiles, num_p, and a set of the input tiles mapped on FMEM, MI, are given as the arguments of the REDUCE_TILE_NUM. The PREDICT_DELAY function at the 20-th line provides the PE delay due to off-chip memory access through fast timing simulation with estimated delays. The REDUCE_TILE_NUM function is applied between lines 39 and 40 in Algorithm 4 to reduce the number of input tiles to process. The other difference from the MIN policy

is that when deciding how many output tiles to leave in the on-chip memory, it considers the effect on the next layer since the number of input tiles is critical to the PE delay if the layer is memory-bound. Therefore, in function GET_PE_DELAY, it estimates the PE delay of the next layer as well as the delay of the current layer using fast timing simulation, and it selects the number of output tiles that minimizes the overall delay in line 9. When the PE delay of the next layer is estimated, the number of output banks in FMEM is fixed to max_o, which is (the number of FMEM banks) $-$ 1.

요 약

임베디드 기기는 대개 계산량, 메모리 크기, 에너지 소모량 등의 제약 사항이 있기 때문에, 딥 러닝 응용을 임베디드 기기에서 수행하는 것은 쉽지 않다. 딥 러닝 응용의 계산량 증가를 해결하기 위해서 에너지 효율적인 모바일 GPU, 디지털 신호 처리 프로세서을 사용하거나, 또는 새로운 뉴럴 프로세서 칩을 만드려는 하드웨어 영역의 최적화 방법이 있다. 반면에 딥 러닝 응용 영역에서는 새로운 딥 러닝 응용을 만들거나, 딥 러닝의 통계적인 특성을 이용한 근사 계산 방법을 이용하여 최적화 방법을 제안하고 있다. 그리고 또 다른 최적화 방법으로는 먼저 하드웨어 플랫폼의 성능 병목 부분을 찾고, 일을 동등하게 여러 계산 자원에 분배하여 최적화하는 하드웨어를 고려한 최적화 방법이 있다.

본 논문에서는 하드웨어를 고려한 소프트웨어 최적화 방법들을 고안하였다. 먼저, LPIRC [1, 2] 대회에 참가한 경험을 바탕으로 임베디드 딥 러닝 시스템을 최적화하는 체계적인 방법론을 고안하고, 그 방법론에 따른 C-GOOD이라는 딥 러닝 프레임워크를 구현하였다. C-GOOD은 하드웨어 플랫폼에 독립적으로 작동하기 위해 대부분의 임베디드 기기에서 컴파일, 수행이 가능한 C 코드를 생성한다. 또한 여러 가지 딥 러닝 응용 영역의 최적화 방법을 적용할 수 있는 옵션과 시스템 성능을 측정할 수 있는 기능을 제공하였다. 이 방법론을 Jetson TX2 [3], Odroid XU4 [4], SRP [5] 등의 서로 다른 3개의 기기에 적용해 봄으로써, 고안된 방법론이 하드웨어 플랫폼에 독립적이며 C-GOOD을 통해 쉽게 여러 딥 러닝 응용 최적화 방법을 적용할 수 있음을 확인하였다.

최근 임베디드 기기에 이종 프로세서들이 많이 탑재되고 있고, 동시에 자율 주행 자동차와 스마트폰 등의 하나의 임베디드 기기에서 여러 개의 딥 러닝 응용을 동시에 수행하는 것이 필요해지고 있다. 본 논문에서는 여러 딥 러닝 응용을 이종 프로세서들을 탑재한 임베디드 기기에 스케줄하는 방법을 고안하고, 스케줄링 프레임워크를 구현하였다. 이 방법론은 실제 기기에서의 프로파일링부터 스케줄 결과를 실제 기기에서 확인하는 과정까지 포함하며, 실제 기기에서 발생하는 이슈들인 DVFS, CPU Hot-plug 등을 고려하였다. 이종 프로세서로의 스케줄링 기법으로는 많이 사용되는 메타 휴리스틱 알고리즘은 유전 알고리즘을 사용하였다. 특히, 서로 다른 주기와 상대 오프셋을

가지고 있는 여러 응용을 동시에 스케줄하기 위해서 모든 태스크들의 스케줄 가능성을 고려하여 스케줄하였다. 스케줄 결과를 검증하기 위해서, ACL의 코어 라이브러리를 이용하여 딥 러닝 추론 응용을 구현하였으며, 스케줄 결과와 같이 각 레이어들을 실제 하드웨어의 서로 다른 프로세서 매핑하도록 구현하였다. 갤럭시 S9 스마트폰 [7]과 Hikey 970 보드 [8]에서 서로 다른 두개의 딥 러닝 네트워크를 수행하고, 스케줄 결과와 비교하여 방법론을 검증할 수 있었다.

이전 최적화 방법들이 딥 러닝 응용의 계산량과 프로세서들에 집중하였는데, 딥 러닝 가속기 또는 NPU의 성능 병목이 생기는 원인은 오프 칩 메모리와 온 칩 사이의 통신이다. 더욱이 오프 칩 메모리 DRAM 접근은 NPU의 전력소모의 많은 부분을 차지한다고 알려져있다. 따라서 이와 같은 오프 칩 DRAM 접근으로 인한 NPU의 성능과 에너지 영향을 줄이고자 본 논문에서는 온 칩 메모리 뱅크를 관리하는 컴파일러 기법을 고안하였다. 온 칩 메모리를 여러 개의 뱅크로 구성하고 연산 도중에 인풋 데이터를 미리 로드함으로써 연산 지연 시간을 줄일 수 있다는 점과 레이어의 아웃풋을 온 칩 메모리에서 재사용하여 오프 칩 메모리 접근을 줄일 수 있다는 점을 이용하여 서로 다른 두 가지의 목적 함수를 가진 두 가지 기법을 고안하였다. 목적 함수는 각각 오프 칩 메모리 접근을 최소화하는 것과 오프 칩 메모리 접근으로 인한 프로세서들의 처리 지연시간을 줄이는 것이다. 서로 다른 5개의 딥 러닝 네트워크를 사이클 레벨 NPU 시뮬레이터에서 수행하여 두 목적 함수에 따른 절충 (Trade-off) 관계 를 확인하였다. 또한 온 칩 메모리 뱅크 관리 기법을 레이어 간 피처 데이터를 최대한 재사용하는 레이어 융합 방법으로 확장하였다. 기존의 순수한 레이어 융합 방법의 경우에는 중복 계산하는 오버헤드와 추가적인 필터 웨이트 로드가 생긴다. 따라서 본 논문에서는 기존의 레이어 별로 처리하는 방법과 순수한 레이어 융합 방법 사이의 하이브리드 레이어 융합 방법을 고안하였다. 두 온 칩 메모리 뱅크 관리 기법을 기반으로 하이브리드 레이어 융합 방법이 기존의 레이어 별 처리하는 기법과 순수한 레이어 융합 방법보다 좋은 성능을 보임을 확인할 수 있었다.

주요어 : 컨볼루션 신경망, 소프트웨어 최적화, 온 디바이스 딥 러닝, 스케줄링, 유전

알고리즘, 이종 프로세서, 모바일 기기, 가속기, 뉴럴 프로세서, 멀티 뱅크 관리, 레이어 융합, 프리패치

학번 : 2014-21780