공학박사학위논문

인공지능 하드웨어 플랫폼에서의 딥 러닝 어플리케이션을 위한 소프트웨어 최적화 기법

Software Optimization Techniques for Deep Learning Applications on AI Hardware Platforms

2023년 8월

서울대학교 대학원
컴퓨터공학부
김 장 률

인공지능 하드웨어 플랫폼에서의 딥 러닝 어플리케이션을 위한 소프트웨어 최적화 기법

Software Optimization Techniques for Deep Learning Applications on AI Hardware Platforms

지도교수 하 순 회

이 논문을 공학박사 학위논문으로 제출함

2023년 5월

서울대학교 대학원

컴퓨터공학부

김 장 률

김장률의 공학박사 학위논문을 인준함

2023년 6월

위 원 장	유 승 주	(인)
부위원장	하 순 회	(인)
위 원	Bernhard Egger	(인)
위 원	이 영 기	(인)
위 원	양 회 석	(인)

Abstract

Software Optimization Techniques for Deep Learning Applications on AI Hardware Platforms

Jangryul Kim
Department of Computer Science and Engineering
College of Engineering
The Graduate School
Seoul National University

To meet the growing demand for deep learning applications in embedded systems, new embedded devices tend to include multiple heterogeneous processors, including a GPU and a deep learning hardware accelerator called a neural processing unit (NPU). In addition, a software development kit (SDK) is provided for fast and efficient development of deep learning applications. The deep learning SDK includes optimizer that delivers low latency and high throughput for deep learning inference applications.

Even the deep learning SDK optimize the inference internally, the SDK assumes that inference is performed on a single processing element, either the GPU or the NPU, but not both. However, running inference on a single processing element does not fully utilize the system. Since the system consists of heterogeneous processors, it is necessary to use these processors simultaneously to run efficiently.

In other words, it is necessary to optimize deep learning applications at the system-level. In this context, we approach the problem from three main topics: optimization of a single deep learning application, optimization of multiple deep learning applications under real-time constraints, and support for deep learning applications in model-based

embedded software design methodology. In this work, we target the NVIDIA Jetson embedded platform with heterogeneous processors, including NPUs, and TensorRT which is a leading deep learning SDK for fast inference.

First, we devise systematic optimization techniques and methodology to increase the throughput of a single deep learning application. We present parallelization techniques for a deep learning application: multi-threading, pipelining, buffer allocation, and network duplication. We also present a framework that supports various optimization parameters to accelerate a deep learning application. The optimization techniques are parameterized and can be applied to a deep learning application by merely adjusting parameters in a configuration file, which is an input to the framework. Since the design space of optimizing parameters is huge, we develop a parameter optimization methodology consisting of a heuristic for balancing pipeline stages among heterogeneous processors and a fine-tuning process for optimizing parameters. This is the first work to partition a deep learning inference which is developed with the TensorRT and improve throughput on the heterogeneous processor system including NPUs. With nine real-life benchmarks, we could achieve $101\% \sim 680\%$ performance improvement and up to 55% energy reduction over the baseline inference using GPU only.

Second, it is becoming popular to run multiple deep learning applications simultaneously to provide various functionalities. In addition, deep learning applications can have real-time constraints that vary at runtime. While extensive studies have been conducted recently to find an efficient mapping of multiple deep learning applications on different hardware platforms, they do not consider the constraints imposed by the NPU and its SDK in a real embedded platform. In this work, we propose a novel energy-aware mapping methodology of multiple deep learning applications on a real embedded system with multiple heterogeneous processors. The objective is to minimize energy consumption while satisfying the real-time constraints of all applications. In the proposed scheme, we first select Pareto-optimal mapping solutions for each application. Then, the mapping

combination is explored considering the scenario that shows the dynamics of the applications while satisfying the constraints. We also reduce energy consumption by tuning the frequency of the processors. This is the first work to consider the concurrent execution of multiple deep learning applications which are developed with the TensorRT on a real hardware platform. We could satisfy up to 40% higher deadline constraints and reduce energy consumption by 22% $\sim$ 31% compared to the static mapping methods with real-life applications and different scenarios on a real platform.

Finally, as deep learning applications become more prevalent in embedded systems, how to support deep learning applications in model-based embedded software design methodologies becomes a challenging problem. One solution so far is to represent each deep learning application with a model. However, it requires considerable effort to translate the specifications and achieve good performance by applying optimization techniques to deep learning applications. In this work, we propose a novel methodology that takes advantage of using a deep learning SDK for performance optimization. In the proposed method, we first obtain the Pareto-optimal mapping solutions of deep learning applications using the SDK associated with the hardware platform. Then, we jointly perform the mapping of dataflow tasks and the selection of mapping solutions for deep learning applications using a genetic algorithm and a heuristic. Experiments with a real-life example and randomly generated graphs show that we could reduce at least 5% of the maximum utilization compared to our previous work that maps deep learning applications and dataflow applications sequentially.

Keywords : Mapping and Scheduling, Design Space Exploration, Deep Learning Applications, Software Optimization, Heterogeneous Processor Systems

Student Number : 2017-22440

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

As deep learning inference applications are increasing in embedded devices, an embedded device tends to equip hardware accelerators in addition to a multi-core CPU and a GPU. To run a trained network on an embedded device, we usually use the software development kit (SDK) provided with the device. The deep learning SDK includes optimizer that delivers low latency and high throughput for deep learning inference applications. Even the deep learning SDK optimize the inference internally, the SDK assumes that inference is performed on a single processing element, either the GPU or the NPU, but not both.

Extensive research have been conducted to accelerate deep learning applications via software optimization on a given hardware platform [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]. Some research have focused on approximate computing such as low precision computation, low-rank approximation, and filter pruning [1, 2], and some others have focused on the exploitation of various levels of parallelism that a deep learning inference network has: data-level parallelism of a convolution layer, task-level parallelism, and pipelining [3, 4, 5, 6, 7, 11]. Also, some studies have extended the problem to multiple networks from a single network while taking real-time constraints into account [8, 9, 10, 12, 13]. We call the last two approaches *system-level* optimization in this paper.

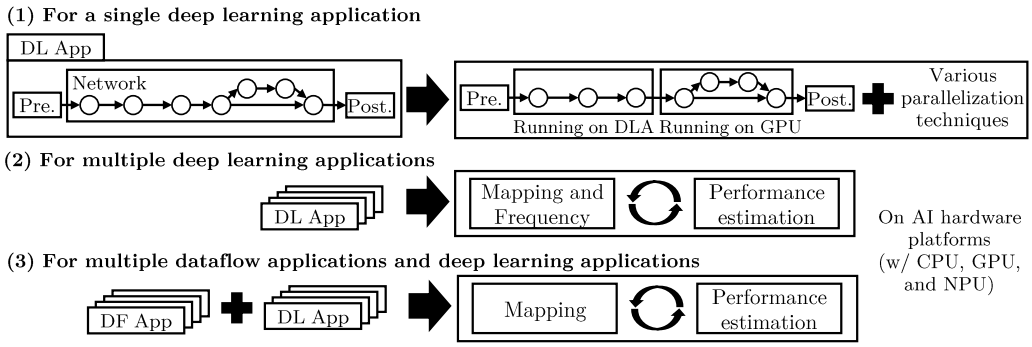


Figure 1.1: Three topics for system-level optimization on AI hardware platforms

In this dissertation, we introduce the system-level optimization methodology on AI hardware platforms that include heterogeneous processors with neural processing units (NPUs) as described in Fig. 1.1. We present the optimization methodology for not only (1) a single deep learning inference but also (2) multiple deep learning applications. Furthermore, we also consider the case where (3) the formal model-based applications and deep learning applications are performed on the AI hardware platform. In this work, we target the NVIDIA Jetson embedded platform, including not only CPU and GPU but also NPUs, and TensorRT which is a leading deep learning SDK for fast inference.

For a single deep learning inference, we present a parallelization techniques that use both GPU and NPUs to maximize the throughput. Multiple threads are used to parallelize pre-/post-processing steps of inference, which is denoted as *Pre.* and *Post.* in Fig. 1.1. Also, accelerators are fully exploited by using multiple streams and pipeline fashion. In addition, we duplicate a part of the network and maps them onto different NPUs while sharing the GPU for the remaining part.

Since applying the techniques manually is a non-trivial job, we present a TensorRT-based framework, called **J**etson-aware **E**mbedded **D**eep learning **I**nference acceleration (JEDI) framework that enables a user to apply various acceleration techniques easily to run a deep learning application by setting a configuration file. We represent techniques as parameters to accelerate a deep learning inference including multi-threading, pipelining,

and network duplication.

Since the design space of allocating layers to diverse processing elements and optimizing other parameters is huge, we devise a parameter optimization methodology that consists of a heuristic for balancing pipeline stages among heterogeneous processors and fine-tuning process for optimizing parameters. With nine real-life benchmarks, we could confirm the throughput improvement and energy reduction over the baseline inference using GPU only.

As for multiple deep learning applications, extensive studies have been conducted recently to find an efficient mapping of applications on various hardware platforms [8, 10, 12, 13]. However, they do not consider the constraints imposed by the NPU and the associated software development kit in a real embedded platform. Therefore, we reveal the challenges when considering the NPU and its SDK, and propose a novel energy-aware mapping methodology of multiple deep learning applications.

There are four major challenges. First, deep learning applications usually have real-time constraints in terms of latency, and the real-time constraints may vary at run time. For instance, as the self-driving car speed increases, the latency constraint for the object detection will be tightened. Second, the set of concurrently running applications may vary. If the car is moving forward, we may want to suspend the object detection from the backside.

The third challenge is to estimate the performance of each mapping candidate. In most previous works, it is assumed that the worst-case execution time (WCET) of each layer in a deep learning application is known, and the performance of mapping can be estimated analytically. But layer-wise profiling is often not appropriate in real systems. For example, a deep learning accelerator (DLA), which is an NPU, in the NVIDIA Jetson platform does not support the layer-wise profiling. Also, TensorRT internally optimizes the inference by applying the techniques and changes structure of kernels depending on the mapping. This makes the problem difficult because the profiled per-layer execution

time may not be available in the analysis and mapping exploration.

The fourth challenge comes from the restrictions imposed by the device and its SDK. TensorRT optimizes the network, saves it as an engine, and loads the engine to the assigned processor. Since the loading takes a long, a few seconds, it may not be done at run time without deadline violation. It means that dynamic task migration is not allowed. And the maximum number of DLA-mapped parts is limited. In addition, since there are only two levels of priority in GPU and no priority level in DLA, the priority assignment is usually not used for GPU and DLAs. Note that the traditional fixed priority-based worst-case response time (WCRT) analysis may incur over-estimated results in case many tasks have the same priority. Hence the mapping techniques based on the assumption that tasks are assigned different priorities are not suitable to our problem.

To tackle these challenges, we propose a novel scenario-based mapping methodology to map multiple deep learning applications onto heterogeneous processors. As described in Fig. 1.1, we find mapping and frequency for multiple deep learning applications. The objective is to minimize energy consumption while satisfying the real-time constraints of all applications. In the proposed scheme, we first select Pareto-optimal mapping solutions for each application. Then mapping combination is explored, considering the scenario that indicates the dynamism of applications while satisfying the constraints. Also, we reduce energy consumption by tuning the frequency of processors. Experimental results confirm the goodness of the proposed methodology for various real-life applications and scenarios on a real platform.

Finally, we also extend the model-based design methodology to support deep learning applications and applications, which are specified by a decidable dataflow model [14], together. For a decidable dataflow, we can determine the mapping and scheduling of tasks at compile time and detect some critical errors in the specification, such as buffer overflow and deadlock [15, 16]. A decidable dataflow model is widely used for the model-based design (MBD) of embedded software on a hardware platform that consists of multiple

processing elements thanks to its properties.

In this regard, how to support deep learning applications in the model-based design methodology has emerged as a challenging problem since the deep learning applications are getting popular in embedded systems. Even though the layer structure of a deep learning application looks similar to a dataflow graph, it is challenging to specify it with a dataflow model. To tackle this problem, previous studies [17, 11] have proposed to specify deep learning networks with a specific dataflow model in order to treat them with other applications in the model-based design framework. The former work [17] extends a dataflow model to specify loop structures explicitly, while the latter [11] transforms a deep learning network into a cyclo-static dataflow (CSDF) graph [16]. However, this approach has the following drawbacks. First, it requires a lot of effort to specify a deep learning application with a dataflow model. The number of data samples produced or consumed per task execution needs to be explicitly specified, and the internal behavior of tasks may need to be redefined. Second, the number of tasks grows significantly to make the design space exploration (DSE) step more difficult as a deep learning network usually consists of numerous layers. Third, the previous studies usually target CPU-GPU heterogeneous processor systems without including a neural processing unit. Since recent hardware platforms tend to include an NPU for accelerating deep learning applications, it is necessary to consider NPUs in the design methodology. Last but not least, it is not possible to apply the optimization techniques that are provided by the deep learning SDK. As a result, the synthesized deep learning application from the MBD framework is likely to perform poorly compared with the conventional deep learning application that runs with the deep learning SDK.

Therefore, we propose a novel methodology that leverages the benefits of using deep learning SDK for performance optimization. Through the proposed methodology, we explore mapping of both dataflow applications and deep learning applications as shown in Fig. 1.1. First, we find the Pareto-optimal mapping candidates for each deep learning

application onto available processing elements with multiple objectives, such as latency, utilization of each processing element, and so on. We explore the design space of partitioning and mapping of a deep learning application for pipelined execution, similar to the first phase of our work for multiple deep learning applications. Next, we explore the mapping candidates of deep learning networks and the mappings of model-based tasks simultaneously. Lastly, we synthesis the interface code between dataflow applications and deep learning applications automatically by the use of well-established model-based design framework, HOPES+ [18]. The viability and efficiency of the proposed scheme are verified with a real-life example and randomly generated graphs.

1.2 Contribution

The contributions of this dissertation can be summarized as follows:

- We introduce the techniques to increase the throughput of a single deep learning application, a framework that supports techniques, and the optimization methodology.
  - We present a parallelization methods that use both GPU and NPUs to maximize the throughput of a single deep learning application: multi-threading, pipelining, and network duplication.
  - To easily accelerate a deep learning application, we develop a JEDI framework. The JEDI framework gets the various optimization parameters as a input, and it accelerates the deep learning application on top of a deep learning SDK. Thus a user can optimize a deep learning application easily without the burden of implementing various acceleration techniques manually. JEDI is publicly released to demonstrate our contributions.
  - The parameter optimization methodology is devised to effectively explore a huge design of space defined by various optimization parameters.
  - The viability of the proposed design environment is evaluated with nine real-life benchmark applications on a real platform.
- We present the problems posed by NPU and its SDK for the mapping exploration of multiple deep learning applications and propose a methodology to address them.
  - We reveal the technical challenges involved in multiple deep learning applications on a real embedded device that consists of heterogeneous processors, including NPU.
  - We propose a novel methodology for mapping multiple deep learning applications onto heterogeneous processors, tackling the technical challenges.

While previous studies assume that the execution time of a layer on each PE is known before a mapping decision is made, such per-layer profiling is not possible for NPUs. And there are several implications imposed by the SDK that need to be considered in making a mapping decision. The proposed methodology considers such limitations imposed by NPU and its SDK.

  - While the mapping decision is made statically, we adjust the frequency of processors dynamically to minimize energy consumption while satisfying the real-time constraints of all deep learning applications.
  - The experiments are carried out on the real platform to show the efficacy of the proposed approach.

- We extend the model-based design methodology to support deep learning applications.
  - We propose a novel technique to support deep learning applications in a model-based design methodology without translating deep learning applications to dataflow models, leveraging the optimization capability of a deep learning SDK.
  - Differently from our previous work that maps the deep learning applications and dataflow applications sequentially, we propose a mapping technique to consider them together, using an evolutionary algorithm.
  - The proposed methodology supports heterogeneous processor systems that include an NPU, considering the characteristics and limitations of the hardware platform and the associated SDK.
  - Experiments with a real-life example and randomly generated graphs show that we could reduce at least 5% of the maximum utilization compared to our previous work that maps deep learning applications and dataflow applications sequentially.

1.3 Dissertation Organization

The rest of the dissertation is organized as follows: Chapter 2 explains the background for main topics. Then, in Chapter 3, the techniques, framework, and optimization method for a single deep learning application is introduced. The methodology for multiple deep learning applications is presented in Chapter 4. Chapter 5 contains our extension to support deep learning applications in a model-based design methodology. Lastly, we summarize the proposed methodologies and discuss future works in Chapter 6.

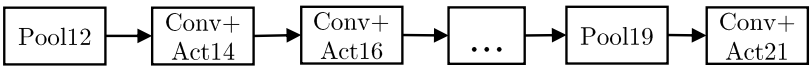
Chapter 2

Background

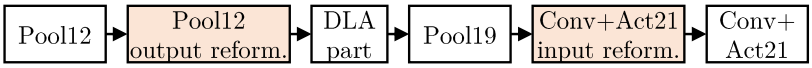
In this chapter we explain the background of the studies described in the following chapters. First, we explain the NVIDIA Jetson platform used in this thesis and TensorRT running on it. Next, we briefly describe genetic algorithms, the design space exploration method used in this research. Next, we introduce compositional performance analysis (CPA), one of the worst-case response time analysis methods used in this study. Finally, we explain the formal model-based embedded software design methodology.

2.1 NVIDIA Jetson AGX Xavier

The NVIDIA Jetson AGX Xavier (Xavier) equips three types of processors: an octa-core ARMv8 CPU, a single Volta GPU, and two NVIDIA DLAs. The DLA shows a power efficiency, but it is slower than GPU, and it does not support all types of layers. For example, some layers, such as *Yolo* layer, cannot be executed on DLA. Also, per-kernel profiling is not supported for DLA. The unit of profiling for DLA is a pipeline stage, which will be explained in the following section. The board has a unified and shared DRAM for CPU and GPU. It means that on-chip communication between CPU and GPU is carried out by DRAM memory access. However, there is an overhead in a DLA to load the data from DRAM to internal memory for the inference. It requires adding extra kernels to be executed for communication. For example, Fig. 2.1 (a) indicates the kernel



(a) When all layers are mapped to the GPU



(b) When layers #13-#18 are mapped to a DLA, and others are mapped to the GPU

Figure 2.1: The example composition of kernels

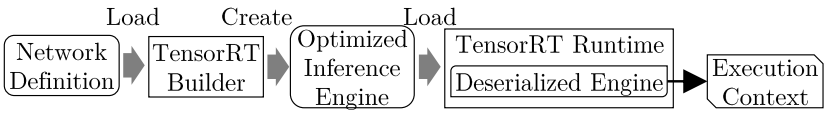


Figure 2.2: The workflow of generating execution context in TensorRT

composition when running *Yolov2* network on the GPU. Fig. 2.1 (b) shows the kernel composition when layers #13 to #18 are mapped to a DLA, and the rest are mapped to GPU. The colored *reformatter* kernel (input/output reform.) is added at the interface between the GPU and the DLA, as displayed in Fig. 2.1 (b). Since the overhead of the added kernel depends on the mapping, it needs to be considered in the mapping step. Also, as shown in the figure, the part mapped to DLA is not profiled layer by layer.

2.2 NVIDIA TensorRT

TensorRT [19] is an SDK for high-performance inference targeted on NVIDIA products. Figure 2.2 displays the workflow of TensorRT. The optimized inference engine is generated by the *Builder* module from a given network definition. In the building process, some optimization techniques such as layer fusion are applied. Then the *Runtime* module loads and deserializes the engine to create an execution context. The *Runtime* module also maps a PE to the deserialized engine. The execution context is assigned to a stream in a GPU or a DLA. It is possible to run the network across multiple PEs in

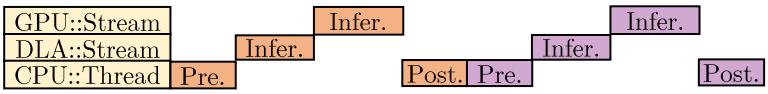


Figure 2.3: An example schedule of deep learning inference

a pipelined manner by dividing a network into multiple pipeline stages. Each pipeline stage needs to be defined by a separate network with which a separate engine and the associated execution context are generated.

Figure 2.3 illustrates an example schedule corresponding to the TensorRT implementation, where a single execution context is mapped to a DLA stream or a GPU stream. TensorRT does not support a CPU to execute the inference. The CPU performs pre/post-processing, marked as *Pre.* and *Post.* in Fig. 2.3. The pre-processing step loads the input data and adjusts the data layout for inference, while the post-processing step performs other processing after inference is done. Due to the execution environment of the Xavier board, the number of deserialized engines mapped to DLAs is limited to four at most at the time of writing this paper. It means that the total number of pipeline stages mapped to two DLAs may not be greater than four. Those properties and restrictions are needed to be considered when using TensorRT on the Xavier board.

2.3 Genetic Algorithm

Algorithm 1 Pseudo code for genetic algorithm

```
Generation of the initial population
Fitness calculation
repeat
    Selecting chromosomes
    Applying crossover
    Applying mutation
    Fitness calculation
    Replacing chromosomes
until Solutions have been converged
```

A genetic algorithm is a type of optimization algorithm inspired by the process of natural selection [20]. The pseudo code of the genetic algorithm is described in Algo-

rithm 1. It works by creating a population of candidate solutions, each called a chromosome. The chromosome indicates the design space to be explored. Then genetic operators such as mutation and crossover are used to evolve the selected chromosomes in the population over time. The algorithm is based on the idea that the fittest individuals in a population are more likely to survive, reproduce and pass on their traits to future generations. This approach is widely used for combination optimization problems such as mapping exploration.

2.4 Compositional Performance Analysis

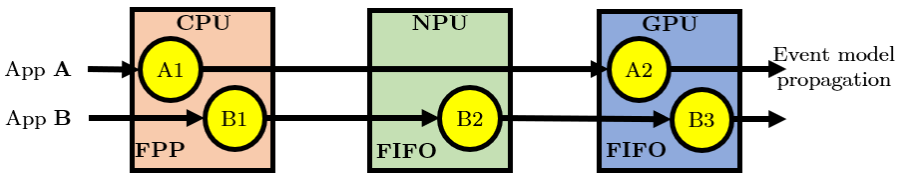


Figure 2.4: Schematic diagram of compositional performance analysis

Compositional Performance Analysis (CPA) [21, 22] is one of the well-established worst-case response time (WCRT) analysis methods used in real-time systems. The CPA method computes the worst-case response time by decomposing the system into smaller components, as shown in Fig. 2.4. This method allows different scheduling policies to be applied to distinct processors and enables the worst-case response time to be computed scalably by propagating event streams between processors. For example, in Fig. 2.4, the CPU determines the output stream considering a fixed priority preemptive (FPP) scheduling policy and passes it to the next component, the NPU (neural processing unit). Similarly, the analysis is applied to the NPU based on first-in first-out (FIFO) scheduling. By propagate the event stream, the worst response time is calculated for each application. To consider the deadline constraint for different processors, the CPA approach would be helpful.

2.5 Model-based Design Methodology

Model-based design (MBD) methodology is widely adopted for embedded software development since it enables us to specify an application behavior independently of the hardware platform that is continually evolving over time. Since appropriate models vary depending on the application domain, various models and methodologies have been proposed. For example, the statechart model is widely used for control-oriented applications [23], and a timed discrete event model is used for power-aware real time scheduling [24]. The works of [25] and [26] adopt the dataflow model for the specification of multimedia or streaming applications. In addition, there exist some works that use more than one model: a combination of the dataflow model and the finite state machine is used in [27, 28, 18], and the work of [29] deploys both the discrete event model and the dataflow model. While appropriate models vary depending on the application domain, a dataflow model of computation is adequate to represent multimedia or streaming applications that are computation intensive. In a dataflow graph, a node represents a computation task, and an arc indicates a data dependency between adjacent tasks. A key benefit of dataflow models is that it is easy to exploit the task-level parallelism of an application by simply mapping nodes to processing elements in a given hardware platform. If the number of data samples that are transferred on each arc is known at compile time, a dataflow model is said to be *decidable* [14]. For a decidable dataflow, we can determine the mapping and scheduling of tasks at compile time and detect some critical errors in the specification, such as buffer overflow and deadlock [15, 16]. In this dissertation, we assume that a decidable dataflow model is used in the model-based design methodology for embedded software development.

Figure 2.5 shows the traditional embedded software design flow based on the dataflow model. Each application is represented by a dataflow graph in which the internal behavior of a task is defined by the task code written in a conventional programming language such as C or C++. The hardware platform information on the available process-

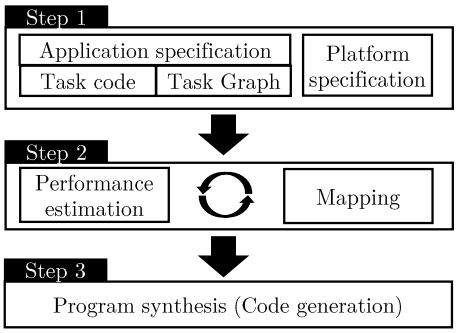


Figure 2.5: Overall flow of the model-based embedded software design

ing elements and communication architecture is given separately from the application specification. For a given hardware platform, we find an optimal mapping of tasks onto processing elements by comparing the estimated performance among various mappings, which is referred to as the design space exploration (DSE) step. Lastly, the application code on each processing element is generated based on the mapping decision made in the DSE step.

In spite of the advantages of the model-based design, it does not support deep learning applications. Therefore, a new methodology to support deep learning application is needed.

Chapter 3

Optimization of a Single Deep Learning Application

3.1 Overview

In this chapter, we aim to improve the throughput performance of a single convolutional neural network (CNN) inference application by utilizing all available processing elements. To accelerate the throughput of an deep learning application, we first introduce parallelization techniques on heterogeneous processor systems including NPUs. Next, We present the JEDI framework to facilitate using the techniques. Based on the framework, we decide how to apply the techniques through the proposed methodology.

The rest of this chapter is organized as follows. We first review the related work to our work in Section 3.2. After the optimization techniques are introduced in Section 3.3, the framework is presented in Section 3.4. Afterward, we explain the proposed methodology in Section 3.5. Lastly, the experiment results are presented in Section 3.6.

3.2 Related Work

This section reviews the related work in the following two subsections involved in the proposed method: deep learning frameworks and optimization for a single deep learning application.

3.2.1 Deep learning Frameworks

The widely used deep learning frameworks TensorFlow [30], Caffe [31], Pytorch [32], Darknet [33], as well as several lite frameworks for embedded systems like TensorRT [19] and TensorFlow Lite [34], provide an environment in which deep learning programs can be run on a single accelerator, GPU or an NPU, but not on multiple accelerators. Although they offer some software optimization features like layer fusion and low precision calculation, they do not consider system-level optimizations.

For allowing hardware-specific optimizations for various embedded systems, some frameworks have been put forth. A Caffe-compatible framework called Caffepresso [35] offers automatic code generation and auto-tuning by specifying setup settings. It is compatible with a range of hardware platforms, including FPGA, DSP, and GPU. A machine learning compiler framework called TVM [36] is designed to address the variety of hardware properties on different devices. It generates hardware-aware optimization code, schedules code segments, and optimizes a computation graph. A C-code generating framework based on Darknet is called C-GOOD [5]. Although it supports approximation-based optimization techniques like tucker decomposition and quantization as well as system-level optimization techniques like multi-threading and pipelining, optimization should be done manually by adjusting the parameters. For system-level optimization, these frameworks do not take heterogeneous accelerators into account.

3.2.2 Optimization For a Single Deep Learning Application

A single deep learning application's optimization methods rely on the hardware platform and the objective. For a multi-core CPU, several techniques have been put forth. In order to maximize resource usage, Tang et al. [7] developed an execution model for deep learning applications on manycore CPUs and suggested a scheduler that allocates the operations to executors at run-time. By pipelining on different CPU cores, a method described in [6] attempts to increase the throughput of convolutional neural network (CNN)

inference. They presented a heuristic based on the profiled execution time of layers because the design space for pipelining is rather large.

For mobile devices, accelerating CNN inferences using GPU has drawn a lot of research interest. In [37], a library for Android is proposed to offload the computationally heacy layers to the GPU. The work of [38] introduced a software accelerator for mobile devices that divides a network into unit blocks that are scheduled to CPU or GPU. In [39], a reinforcement learning-based method is presented that trains policy networks to allocate graph operations into groups and assign the groups to available devices for Tensorflow graphs. Although this method takes into account both CPU and GPU, its primary goal is to decrease latency so that pipelining is not taken into account.

A simple way to boost throughput performance is by using simple pipelining between a multi-core CPU and a GPU. For instance, a 2017 LPIRC (Low Power Image Recognition Challenge) winner used pre/post-processing pipelining; pre/post-processing is carried out on a CPU while the network is run on a GPU in a pipelined manner [40]. However, pipelining a network onto heterogeneous processor systems, including accelerators, is not taken into account. In [11], general system-level optimization on a heterogeneous system is taken into consideration. They could apply any parallel scheduling method that has been presented for a CSDF graph by transforming a CNN model to a CSDF graph. To map the nodes to a CPU/GPU heterogeneous system for experiments, they used a genetic algorithm. The overhead associated with model translation and the challenges of mapping and code generation are this approach's key drawbacks.

A few recent research have studied at heterogeneous processor systems with not only GPU but also NPU for accelerating deep learning applications. Using task-level parallelism of each network, the work of [12] also takes into account multiple DNN instances on a heterogeneous system that comprises both GPU and NPU. However, NPUs are not included in the experimental results, even though throughput optimization is included as an objective function. There is no work using the NPU and its SDK on a real

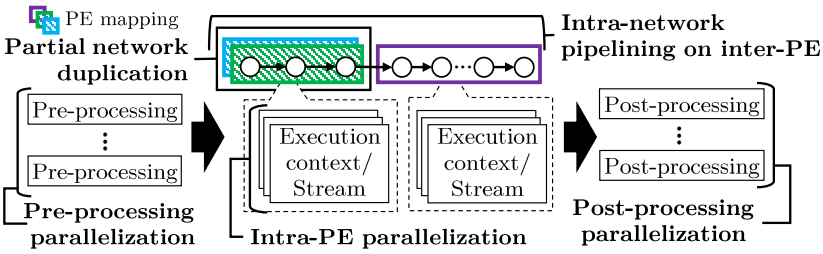
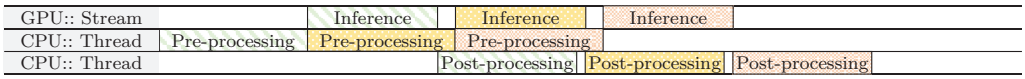
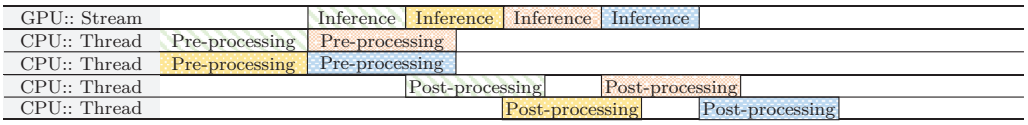


Figure 3.1: Four parallelization techniques



(a) Pre/post-processing pipelining with two buffers



(b) Pre/post-processing parallelization with two threads

Figure 3.2: Schedule diagrams of inference steps with pre/post-processing pipelining

platform, despite the fact that numerous research offer the optimization approach for a single deep learning application.

3.3 Parallelization Techniques

The proposed methodology consists of four main techniques, as outlined in Fig. 3.1.

3.3.1 Pre/Post-Processing Pipelining and Parallelization

Pre- and post-processing are essential parts to run deep learning (DL) applications. In the pre-processing step, we load an input image and re-size the image. After the completion of inference, we perform post-processing for localizing detected objects and storing results. Pipelining the pre-/post-processing part with the main inference body is popularly used to improve the performance [40]. Figure 3.2 (a) is an example diagram of the pre- and post-processing pipelining. Colors express dependency among pipeline

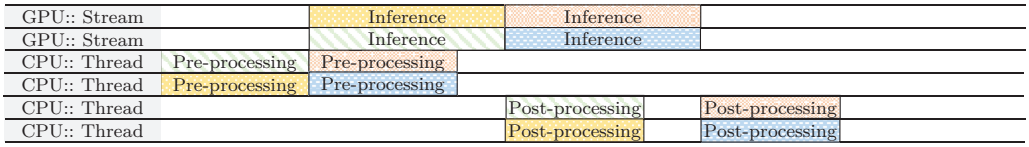
stages. To overlap the execution of adjacent pipeline stages, multiple buffers are needed. In Fig. 3.2 (a), the pre-processing, post-processing, and inference steps can be overlapped together since two buffers are used between pipeline stages.

The number of buffers between pipeline stages affects the throughput performance. A processing element may become idle if there is no free space in the output buffer to store the output data or input data is not available in the input buffer. Thus we need to increase the buffer size until the throughput performance is saturated. If the same memory is shared among different processing elements, a processing element can read input data without copying overhead or write output data directly to the buffer. Since our example embedded device, the Xavier board, provides an API to share pinned CUDA memory among different processing elements, there is no data copy overhead between pipeline stages. This feature is known as *Zero-copy*.

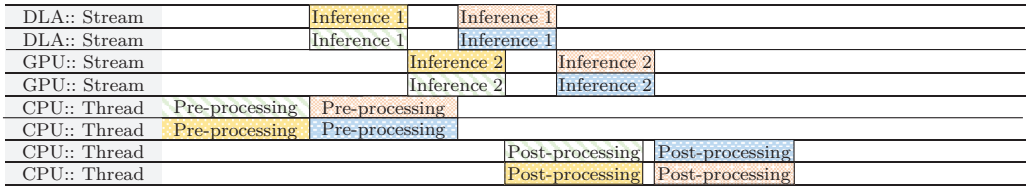
Although the pipelining those parts makes the execution overlap, it is not enough in some cases. *Yolov4-tiny* network, as an example, it is necessary to utilize more than two pre-processing threads concurrently since it may take longer to pre-processing than the time for inference body. Thus, we parallelize the pre- and post-processing parts with multiple threads in case the part becomes the performance bottleneck. Figure 3.2 (b) shows an example schedule where two threads are used in the pre-processing and post-processing step. It is assumed that the number of intermediate buffers is large enough to run the third and fourth pre-processing steps consecutively.

3.3.2 Intra-PE Parallelization

A data-parallel accelerator such as GPU can parallelize multiple instances of the assigned kernel using multiple streams, which increases the utilization further. Independently of the number of threads in the pre-/post-processing steps, we can create more than one stream in GPU and DLA by creating as many buffers as the number of streams at the pipeline-interface. It is observed that if the number of streams exceeds a certain



(a) Multiple streams with a sufficient number of buffers



(b) Network pipelining on heterogeneous processors

Figure 3.3: Schedule diagrams of inference body parallelization

level, the performance is saturated. Thus we set the number of streams as an optimization parameter. In Fig. 3.3 (a), it is assumed that the number of streams in GPU is two, the same as the number of threads in the pre-processing step.

3.3.3 Intra-network Pipelining

To use all available accelerators, we pipeline the inference body. Figure 3.3 (b) shows a simple schedule after applying the intra-network pipelining where the inference network is split into two stages and assign the first stage, *Inference 1*, to a DLA and the second stage, *Inference 2*, to the GPU. Note that we may assign more than one stage to a PE. Therefore, it is necessary to determine how to split the network into stages and how to assign the stages to the PEs. Since the design space of pipelining is huge, how to explore the space is a challenging problem. To tackle this challenge, we devise a heuristic to decide the cut-points for network splitting for a given mapping option which is explained in Section 3.5.

3.3.4 Partial Network Duplication

Lastly, we may duplicate the part of the network, which is called partial network duplication (PND). It is important to balance the execution time among the pipeline stages. Since it is easier to balance one GPU and one DLA than one GPU and two DLAs, we pipeline the network onto one GPU and one DLA in this technique. Then we run two iterations of the network concurrently, mapping the duplicated part of the network to different DLAs and sharing the GPU for the remaining part. In Fig. 3.1, the striped boxes are mapped onto different DLA with the separate inference engine, and each engine is responsible for half of the streams in the stage.

3.3.5 Other Optimization Methods

Besides system-level optimization, the proposed framework supports two popular methods to improve throughput performance. The first is to use low-precision computation that reduces the computation workload as well as memory requirement. Instead of using 32-bit floating-point operations, we may use 16-bit floating-point or 8-bit integer operations. Since the Xavier board supports 8-bit inference, we provide an option to use 8-bit inference. TensorRT supports post training quantization (PTQ) method based on a calibration table. However, this feature is not directly applicable to the pipelined network since TensorRT considers a single network only.

The second is to use batch processing. There are two scenarios we can use batching processing for a single DL application. One is to run it with multiple inputs; An example is an object detection network that receives input images from multiple cameras. The other scenario is to queue incoming input images to given batch size and process them concurrently. Even though queuing may increase the latency, the throughput gain by batching may give a higher benefit than the latency loss.

3.4 JEDI Framework

As was stated in the section above, the throughput of network can be increased by using a number of system-level optimization strategies. A TensorRT-based framework is created to support those techniques due to the lack of public framework that makes it simple for us to use them. The suggested framework's general organization and process are shown in Figure 3.4. The framework requires a configuration file as input that contains the pipelining/mapping information in addition to the following parameters which are related to design space exploration: the number of buffers, the number of threads for pre/post processing, the number of streams per pipeline stage, and mapping to processing elements. Besides, the configuration file allows for the setting of batch size and data precision. Information about the application and test, such as the network settings or image file path, is also included in the configuration file.

A deep neural network is defined by a darknet-based configuration which is adopted in tkDNN. It is necessary to alter tkDNN library [41] so that it generates an engine for each sub-network after pipelining the network, as the original tkDNN library provides a way to build an engine for the entire network using TensorRT only. In addition, we also support an ONNX file format [42] which is widely used to represent deep learning applications. To partition a network presented by ONNX, we use the Polygraphy tool [43]. When using the ONNX file format, it is only supported in Jetpack 5.1 with TensorRT 8.5.2 and later versions. Meanwhile, the darknet-based configuration can be used in Jetpack 4.3 with TensorRT 6 and higher version. As seen in the *Build* section of Fig. 3.4, a distinct engine is produced for each accelerator. Additionally, the modified tkDNN rearranges the engine's outputs and inputs to correspond with those of neighboring pipeline stages in the case of using the darknet-based configuration.

The *Inference* section of Fig. 3.4 illustrates how to run a deep learning application in JEDI. JEDI generates threads based on parameter, including subnetwork threads and pre-/post-processing threads that will perform on a CPU. Each subnetwork thread repre-

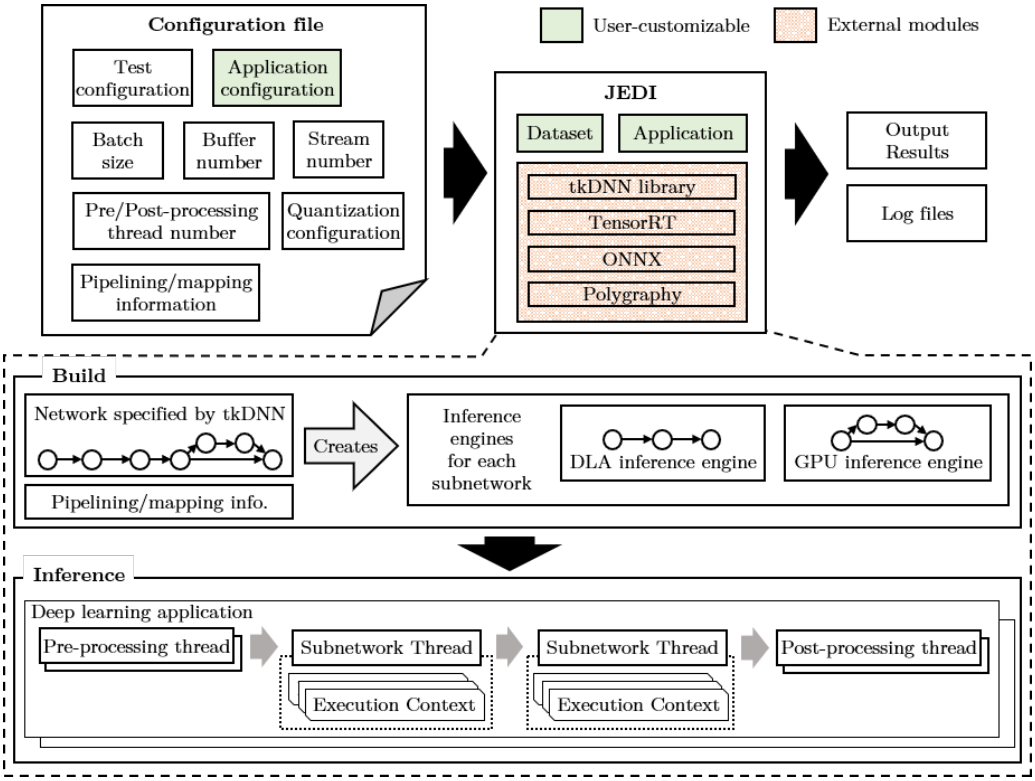


Figure 3.4: Workflow of the proposed inference framework

sents a pipeline stage mapped onto GPU or DLA and handles multiple execution contexts mapped to streams in the designated accelerator. A subnetwork thread synchronizes with the adjacent pipeline stages; inference on a stream is postponed until all input and output buffers are ready. If multiple execution contexts are used, synchronization delay can be disguised by interleaving the execution of streams. If a partial network duplication technique is adopted, each of two different inference engines, corresponding to the identical portion of the network, are mapped to a different processor. Otherwise, only one inference engine is used at each pipeline stage.

In Fig. 3.4, the green boxes represent the user-customizable JEDI modules. A user can add any application-specific parameters like an application configuration, new data sets, and a deep learning application. Log files and output results are produced after JEDI runs the program on the Xavier board. These files are used to record resource utilization

```
1  configs = {
2      instance_num = "1";
3      instances = ( {
4          ...
5          # Common/test configurations
6          app_type = "YoloApplication";
7          model_type = "tkDNNApplication";
8          sample_size = "4952";
9          ...
10         # Application-specific configurations
11         ## Darknet-based configuration
12         cfg_path = "yolo4.cfg";
13         image_path = "all_images.txt";
14         name_path = "coco.names";
15         ## ONNX configuration
16         onnx_file_path = "yolo4.onnx"
17         ...
18         # Optimization parameters
19         batch = "1";
20         device_num = "3";
21         pre_thread_num = "1";
22         post_thread_num = "1";
23         buffer_num = "8";
24         cut_points = "52,134,268";
25         streams = "1,4,3";
26         devices = "GPU,DLA,GPU";
27         dla_cores = "0,2,0";
28         data_type = "INT8";
29 } ) }
```

Figure 3.5: An example segment of JEDI configuration file

and evaluate the accuracy and performance of a deep learning application.

3.4.1 Configuration Parameters

An example segment of the configuration file is shown in Fig. 3.5. The configuration file format is adopted from the *libconfig* library, C/C++ configuration library. First of all, *instances* and *instance_num* (line 2-3) are used for specifying one or multiple deep learning applications. If *instance_num* is one, only a single deep learning application is executed in JEDI. Each element of *instances* shows the independent settings of each deep learning application. The options in each instance can be classified into three types: common/test configuration (line 6-8), application-specific configuration (line 10-13), and

optimization parameters (line 15-24).

Common/test configurations are used for all types of deep learning applications. *app_type* is used to specify the application type that is given by the user, which will be explained in the next subsection. *model_type* is used to indicate the input format of the application. In the example, the darknet-based configuration, *tkDNNApplication*, is used. Another option is the *ONNXApplication* which represents that the ONNX file is adopted. The number of inputs to test inference is set by *sample_size*. There are other common configuration options such as weight file directory path, engine building directory path, and calibration table path.

Application-specific configurations depend on the application type (*app_type*); A user can define new options which are needed for creating a network or loading a data set. Since the application type is given as *YoloApplication*, related options are listed as application-specific options in Fig. 3.5. If the model type is *tkDNNApplication*, then network information is read from *Darknet*-based configuration file [33], so *cfg_path* is used for getting the path of the network configuration file. While the model type is *ONNXApplication*, then *onnx_path* is used to read the onnx file. The input image path and the path to the labeled data set are also specified as application-specific configurations.

Figure 3.5 displays the optimization parameters in the last segment. Note that these optimization parameters are independent of deep learning applications. The numbers of pre-/post-processing threads are decided by the parameters *pre_thread_num* and *post_thread_num*, respectively. The number of buffers between pipeline stages is represented by the *buffer_num*.

For intra-network pipelining, four parameters are defined: *device_num*, *devices*, *dla_cores*, and *cut_points*. The number of pipeline stages is denoted by the *device_num*, and *devices* indicates the assigned processor to each pipeline stage. For instance, the first and the last stage are mapped to GPU while the second stage is mapped to a DLA. The third parameter, *dla_cores*, indicates the mapped core ID in case the assigned processor

is DLA. So the second element, which is 2, has a meaning while the first and the third elements can be ignored. If the value is 0 or 1, it indicates the core ID of two DLAs. If the value is greater than or equal to the number of cores, then the PND technique is applied. In the example of Fig. 3.5, we duplicate the second stage and map two copies onto two DLAs to run concurrently.

The number of layers allocated to each pipeline stage is specified by the *cut_points* option which indicates the last layer of each pipeline stage. For instance, in line 20 of Fig. 3.5, the first stage consists of layers #0 to #52, the next stage includes layers #53 to #134, and the rest are allocated to the last stage. The number of streams for each pipeline stage is denoted by the *streams*. In line 21, the first and last stages take one and three GPU streams, respectively. The second stage uses four streams, two streams on each DLA, since the PND technique is adopted.

Other parameters discussed in Section 3.3.5 are also included in the JEDI configuration file: *batch* is the batch size that is used to run an inference with multiple inputs, and *data_type* indicates the precision of the inference. JEDI supports 32-bit (FP32), 16-bit (FP16) floating-point, and 8-bit integer (INT8) inference. Since TensorRT generates a calibration table from a single inference engine for INT8 inference, JEDI uses the calibration table generated from the whole neural network. Based on cut-points information, JEDI automatically divides the calibration table to generate a separate calibration table for each pipeline stage and uses the divided table for generating the associated INT8 inference engine.

3.4.2 Application Development

As a general deep learning framework, JEDI provides a generic interface to support various deep learning applications. JEDI defines a virtual class named *IInferenceApplication* which includes virtual methods to implement a deep learning application as listed in Table 3.1. The *readCustomOptions* method is used to

Table 3.1: Main virtual methods for user-implemented deep learning applications in JEDI

Method Name	Description
readCustomOptions	Read custom options which is going to be used for the application. (e.g. Network file path, data set path, etc.)
createNetwork	Create a network.
initializePreprocessing	Initialize data which are used during pre-processing. (e.g. Data set initialization)
initializePostprocessing	Initialize data which are used during post-processing.
preprocessing	Run pre-processing (e.g. image loading, image resizing)
postprocessing	Run post-processing.

read application-specific options from the JEDI configuration file. The *createNetwork* method creates a network and retrieves a TensorRT-specified network object: A User may specify the network with a C++ code or add a conversion code from other formats to TensorRT-based network object. Two methods, *initializePreprocessing* and *initializePostprocessing*, are used for initializing data or allocating buffers that are going to be used during pre-processing or post-processing, respectively. Actual processing codes are implemented in *preprocessing* and *postprocessing* methods, respectively. Registering an application is made by the registry pattern [44], and the registered application name is used as the *app_type* option in the JEDI configuration file.

YoloApplication shown in Fig. 3.5 is an example of a deep learning application development. *YoloApplication* is an inherited class from *IInferenceApplication* that we implemented virtual methods listed in Table 3.1. The *readCustomOptions* method is implemented to read *cfg_path*, *image_path*, and *name_path*. In the *createNetwork* method, we call a conversion function from a *Darknet* configuration file or *ONNX* file to TensorRT specification. The darknet configuration file is translated by the tkDNN library. And the onnx file is read by the *parseFromFile* method provided by TensorRT. In the *initializePreprocessing* method, a data set is initialized to read the image path from *image_path*, and *initializePostprocessing* allocates buffers for storing detection results. Finally, the *preprocesssing* method is implemented to load and resize input images, and *postprocesssing* calls functions related to localizing objects. With our proposed frame-

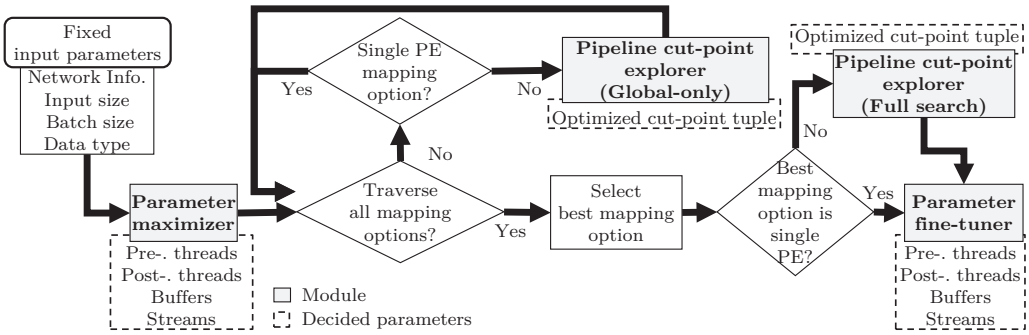


Figure 3.6: The proposed optimization process with three design space exploration modules

Table 3.2: Options for mapping on the target platform

Option	# of pipeline stages	Composition of PEs
A	2	DLA - GPU
B	3	GPU - DLA - GPU
C	3	DLA - DLA - GPU
D	4	GPU - DLA - DLA - GPU
E	1	GPU
PND-A	2	DLA (two DLAs) - GPU
PND-B	3	GPU - DLA (two DLAs) - GPU

work, a user may focus only on the implementation of an application itself, without concern about acceleration techniques.

3.5 Design Space Exploration

Since JEDI supports many optimization parameters, exploring the design space to find optimal parameter values is challenging. Rather than searching the entire design space at once, we use a divide-and-conquer approach to reduce the search complexity. Figure 3.6 depicts the proposed optimization process that consists of three key modules, each of which uses JEDI to explore a subset of parameters. First of all, a group of parameters is explored manually by setting some fixed values. The group includes network information, input data size, batch size, data type, and a mapping option. Since the Xavier board has one GPU and two DLAs, the number of possible mapping options is limited

and finite. Table 3.2 shows the explored mapping options in the proposed methodology. The second column indicates the number of pipeline stages, and the last column shows to which accelerators the pipeline stages are mapped. Unlike the GPU, a DLA cannot execute some layer types. If a layer that cannot be run is mapped to a DLA, the layer is remapped to the GPU automatically by TensorRT, which is called *GPU fallback*. Since such layers exist at the bottom part of all benchmark networks used in the experiments, we map the last pipeline stage to GPU in all options. Except for options *A*, *B*, and *E*, all options use two DLAs. For options *PND-A* and *PND-B*, the pipeline stage assigned to DLA is duplicated and mapped to two DLAs by the PND technique. In options *B*, *D*, and *PND-B*, GPU is assigned to two pipeline stages, the first and the last, since the GPU has better computation power than DLA. If our methodology is applied to a different hardware platform, we will define a different set of mapping options, considering the characteristics of the processing elements in the hardware platform.

After fixed input parameters are set manually, the parameter maximizer module determines the upper bounds of the following parameters to explore: the number of pre-/post-processing threads, the number of buffers, and the number of streams. In this module, we use mapping option *E* and increase the parameter values incrementally until the performance is not improved anymore: we change the configuration file of JEDI and run JEDI to obtain the performance and repeat this process automatically.

After the upper bound values of those parameters are determined, the next step is to determine the pipeline cut-points by the pipeline cut-point explorer module for a given mapping option unless the mapping option *E* is taken. Two versions of pipeline cut-point explorer are used with a different purpose. First, the pipeline cut-point explorer (global-only) is used to find the best mapping option among seven given options. In this case, only the sampled cut-points are explored to find the mapping option with the best performance. Then, the pipeline cut-point explorer (full search) considers all feasible cut-points. The pipeline cut-point explorer module will be explained in the next subsection in detail.

If the mapping option E is selected, the pipeline cut-point explorer is skipped since no network pipelining is made in this option. After pipeline cut-points are determined, the parameter fine-tuner module performs parameter fine-tuning that is going to be explained in Section 3.5.2.

Note that all three modules in Fig. 3.6 measure the performance by actually running JEDI on the target Xavier board. While it incurs the significant overhead of repeated execution of JEDI on the Xavier board, the measured performance is an ideal performance metric that is necessary for design space exploration (DSE): communication time between pipeline stages and other unknown overheads are all considered in the measured performance.

3.5.1 Pipeline Cut-point Explorer

Layer allocation to each pipeline stage can be represented by a cut-point tuple which indicates the last layers mapped to each pipeline stage. All options which use more than one accelerator require running the pipeline cut-point explorer module that searches a set of sub-optimal cut-point tuples. The size of possible combinations of cut-points is ${}_{N-1}C_c$ to select c cut-points from N layers. To avoid the exponential complexity of exhaustive search, a 2-phase heuristic is devised to find a sub-optimal tuple of cut-points. The first phase is a global search over a sampled set of cut-points: we prune the search space by sampling the cut-points regularly. The other cut-points are explored in the second phase, local search. Algorithms 2 and 3 show the pseudo-code of the global search heuristic and local search heuristic, respectively. If the pipeline cut-point explorer is used for searching the best mapping option, the local search heuristic is skipped to reduce the search time as shown in lines 10-13 of Algorithm 2. At the beginning of the global search phase, we set the following three: an initial cut-point tuple, a candidate cut-point set, and a stream number tuple. An initial cut-point tuple can be set as a sequence of ordered random numbers. Suppose the profiling information of layer execution times on GPU or NPU is

Algorithm 2 Pseudo code for global search heuristic

Input : T: Threshold value of GPU utilization
Input : K: The number of cut-point tuples with top K FPS
▷ For fast search, K = 1. Otherwise, K = 10
Input : $cut_{init/cur/prev}$: an initial/current/previous cut-point tuple
Input : str_{cur} : a stream number tuple of cut_{cur}
Input : $conf_{best}$: (cut-points tuple, streams tuple) with the best FPS
Input : $fps_{cur/prev}$: FPS of the run with $cut_{cur/prev}$
Input : $gpuUtil_{cur}$: GPU utilization with current tuple of cut-points
Input : Top_K: the list of top K cut-point tuples in FPS
Input : P: The set of move policies to explore the cut-point tuples

```
cut_cur = cut_init, cut_prev =None, fps_prev = MAX
fps_cur, gpuUtil_cur = Run JEDI with (cut_cur, str_cur)
while New cut-point tuple is selected do
    Select P with gpuUtil_cur
    for each P_i in P do
        cut_prev = cut_cur, fps_prev = fps_cur
        cut_cur = Select a new cut-point tuple with (cut_cur, P_i)
        fps_cur, gpuUtil_cur = Run JEDI with (cut_cur, str_cur)
        Update Top_K and conf_best with (cut_cur, fps_cur, str_cur)
        if NOT  Global-only and
        gpuUtil_cur ≥ T and cut_cur ∈ Top_K then
            Perform local search with (cut_cur, str_cur, fps_cur)
        end if
        if fps_prev < fps_cur then
            continue with P_i again
        end if
    end for
end while
```

available. In that case, an initial cut-point can be decided based on the profiling execution time, aiming to balance the pipeline stage length. Since layer-wise profiling for DLA is not available on the Xavier board, we used a random initial cut-point tuple in the experiments. If a cut-point tuple is already searched with different fixed input parameters, that cut-point tuple could be reused as the initial cut-point tuple.

A candidate cut-point set is a sampled set of cut-points to prune the search space for global search. A candidate cut-point set is basically determined by sampling the cut-points regularly, but additional cut-points can be inserted to include the cut-points of the initial cut-point tuple. Figure 3.7 (a) shows an example of the sampled cut-point set. The sampled cut-point set, which is shown as the solid green lines, is created by sampling after every three layers, so 5 out of 15 cut-points are used during the global search.

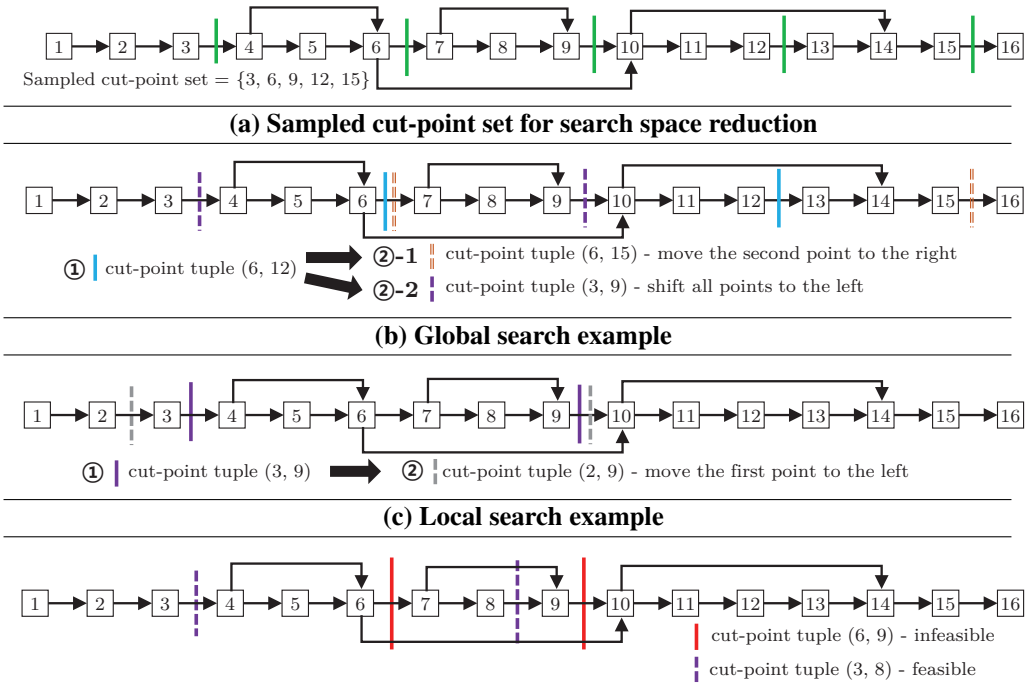


Figure 3.7: Illustration of the proposed heuristic with an example

The last is a stream number tuple which contains the number of streams for each pipeline stage. We simply set the number of streams to two for the pipeline stage mapped to a single DLA. For options *PND-A* and *PND-B*, we set the number of streams to four to assign two streams to each DLA. The number of streams for GPU is set to the maximum number of streams that is decided by the parameter maximizer module. If two pipeline stages are mapped to a single GPU, such as options *B*, *D*, and *PND-B*, each GPU pipeline stage uses half of the maximum number of streams.

The proposed global search heuristic requires two internal variables, T and K: T is a certain threshold of GPU utilization and K is the number of tuples to maintain during the iterative process of global search. As shown in Algorithm 2, the global search starts with the initial cut-point tuple, cut_{init} (line 1). We obtain the FPS, fps_{cur}, and GPU utilization, $gpuUtil_{cur}$ by running JEDI with the initial cut-point tuple and the number of streams

tuple (line 2). Then, the searching area of cut-points is expanded by defining several cut-moving policies from the current tuple of cut-points. If $gpuUtil_{cur}$ is greater than T, the cut-points around the current cut-point tuple are moved by the following policies: One is to move a single cut-point by one, and the other is to shift all cut-points in the left or right direction. Otherwise, the cut-points are moved in the direction of increasing the GPU utilization, assigning more layers to GPU. After the policy set P is selected based on $gpuUtil_{cur}$ (line 4), we obtain a new candidate tuple of cut-points according to each moving policy $P_i \in P$ (line 7). Two new cut-point tuples created this way are shown in Fig. 3.7 (b). From the original cut-point tuple depicted with sky-blue lines (①), the double-dashed brown lines (②-1) indicate a new candidate cut-point tuple by moving the second cut-point to the right direction, while the single-dashed violet lines (②-2) represent another candidate tuple by shifting all cut-points to the left direction. As shown in the figure, moving is performed with the sampled cut-point set.

After we measure the FPS and GPU utilization by running JEDI (line 8), we update the list of cut-point tuples with top K FPS performance, Top_K. And, we update the $conf_{best}$ which contains the cut-point tuple and the stream number tuple with the best FPS result (line 9). The local search heuristic is executed when the $gpuUtil_{cur}$ is not smaller than T and cut_{cur} is newly added to Top_K (lines 10-13 in Algorithm 2) in the full search version.

Note that variable T affects the search area of the design space and the speed of the global search heuristic. If T is too large, the explorer is terminated fast after exploring a small volume of candidate cut-points, and the local search heuristic may never be executed. To determine a proper T value, we run the pipelining cut-point explorer several times by reducing the T value in a greedy fashion until the local search heuristic is executed for a certain number of cut-point tuples explored during the global search phase. The number is determined empirically, for instance 10. Depending on K value, the searching area can be extended or shrunken. If the initial cut-point tuple is a sub-optimal

Algorithm 3 Pseudo for local search heuristic

Input : cut_{global}: a cut-point tuple passed from the global search
Input : cut_{best} : a cut-point tuple with the best FPS
Input : $str_{cur/global}$: a stream number tuple of $cut_{cur/global}$
Input : $cut_{cur/prev}$: an current/previous cut-point tuple
Input : $fps_{cur/prev/global}$: FPS of the run with $cut_{cur/prev/global}$
Input : P_{local}: The local search move policies to move a cut-point

1: $cut_{cur} = cut_{global}$, $str_{cur} = str_{global}$, fps_{prev} = MAX
2: **if** The first pipeline stage is mapped to GPU **then**
3: Update the first pipeline stage of str_{cur} to 1
4: fps_{cur} = Run JEDI with (cut_{cur}, str_{cur})
5: Update $conf_{best}$ with (cut_{cur}, fps_{cur}, str_{cur})
6: **if** fps_{cur} ¿ fps_{global} **then**
7: Rollback str_{cur} and fps_{cur}
8: **end if**
9: **end if**
10: **while** fps_{cur} is improved **do**
11: **for** each cut-point of cut_{cur} **do**
12: **for** P_i in P_{local} **do**
13: $cut_{prev} = cut_{cur}$, $fps_{prev} = fps_{cur}$
14: cut_{cur} = Move a cut-point of cut_{cur} with P_i
15: fps_{cur} = Run JEDI with (cut_{cur}, str_{cur})
16: Update $conf_{best}$ with (cut_{cur}, fps_{cur}, str_{cur})
17: **if** $fps_{cur} > fps_{prev}$ **then**
18: continue with P_i again
19: **else**
20: $cut_{cur} = cut_{prev}$, $fps_{cur} = fps_{prev}$
21: **end if**
22: **end for**
23: **end for**
24: **end while**

tuple found from different fixed parameters (e.g., different image size or batch size), the fast search can be performed by setting K value to 1. Otherwise, K was set to 10 in the experiments.

The local search heuristic is shown in Algorithm 3. The local search heuristic is called by the global search for the current cut-point tuple, cut_{cur}, which becomes cut_{global} and is regarded as the initial cut-point tuple in the local search heuristic. The fps_{global} is initialized with the FPS performance of cut_{global}. If the local search heuristic is run with the mapping options which map two pipeline stages to GPU such as *B*, *D*, and *PND-B*, we examine if it is beneficial to change str_{cur} by allocating one stream to the first pipeline stage and the other streams to the last pipeline stage (lines 2-7). If the FPS result is better

than fps_{global}, the number of streams is changed to the newer one. Otherwise, str_{cur} is restored to str_{global}. This step is added after we observe that assigning more streams to the last stage often produces better performance in our experiments.

While the overall process of the local search heuristic is similar to the global search heuristic, the cut-point selection policy and the cut-point selection range are different from the global search heuristic. The local search heuristic always uses P_{local} as a moving policy that only moves a single cut-point one by one to the left or right direction. While the global search heuristic only chooses the cut-points from the sampled candidate cut-point set, the local search selects the cut-points from all cut-points. Figure 3.7 (c) shows an example of moving a cut-point by the local search heuristic. The cut-point tuple with the dashed gray line (②) is generated from the tuple with the solid violet line (①) by moving the first cut-point from 3 to 2. Note that the local search only updates the $conf_{best}$ to store the best solution (line 16). Top_K is not affected by the local search. If the FPS performance is improved, the same policy is tried again (lines 17-18). Otherwise, the cut-point tuple is restored (lines 19-20), and the next policy is applied until there is no performance improvement.

When selecting a new cut-point tuple in both the global and local search heuristic, the pipeline cut-point explorer checks the feasibility of the selected cut-point tuple. If more than one cut-point cuts an arc between two layers, it is considered infeasible. We perform synchronization between two adjacent pipeline stages only for simple implementation. If we allow more than one cut-point to cut the same arc, it is needed to synchronize two non-adjacent pipeline stages. Since it incurs the extra overhead of synchronization and buffer management, we decided to disallow it. Figure 3.7 (d) shows an example of a feasible and infeasible cut-point tuple. The cut-point tuple with the solid red line is infeasible since the output of the first pipeline stage is directly passed to the last pipeline stage, not the second. On the other hand, the cut-point tuple with the dashed violet lines is feasible.

Note that the pipeline cut-point explorer actually builds the TensorRT engines for each pipeline stage and runs the application multiple times, so the run-time of the pipelining cut-point explorer depends on the number of cut-point tuples searched during execution. Suppose pipelining cut-points are the same, but the other parameters are different. In that case, the engines are reused when performing different mapping options in the experiments to avoid the redundant overhead of building engines.

Note that the pipeline cut-point explorer uses the utilization value of GPU as a metric to narrow down the search space. If a hardware platform provides a way to monitor the processor utilization, the proposed heuristic can be adapted to the hardware platform accordingly. Since we measure the performance by running the network on the hardware platform directly, the same method can be applied.

3.5.2 Parameter Fine-tuner

The final step of parameter optimization is running the parameter fine-tuner module. It finds the minimum number of pre-/post-processing threads, streams for each pipeline stage, and buffers to reduce the hardware overhead while not hurting the performance. Since the number of possible combinations of those parameters is huge, a greedy heuristic is devised to explore the parameters effectively. We use a constraint that the number of buffers must be greater than or equal to other optimization parameters.

The greedy heuristic consists of three steps. First, it increases the parameters one by one to check whether the performance is improved or not by running JEDI with the changed configurations. Even though we use the maximum parameter values obtained in the parameter maximizer module that assumes mapping option E, we may improve the performance by using larger values if a different mapping option is selected. If the performance is improved, the fine-tuner sets the performance value as the best value. In the second step, the fine-tuner reduces the values of all parameters one by one to get the minimum parameter values while maintaining the performance as much as possible.

The percentage of allowable performance degradation is given as an input. Finally, the number of streams assigned to GPU is adjusted while maintaining the total number of streams in case two pipeline stages are mapped to GPU.

3.6 Experiments

3.6.1 Set-Up

All experiments were conducted on a Jetson AGX Xavier board with Jetpack 4.3 and TensorRT 6. We used the tkDNN [41] library that makes TensorRT easy to use. Since it does not support pipelining, however, we modified the library to create a separate inference engine for each pipeline stage. In addition, the experiments are conducted by using a darknet-based network format except for the Section 3.6.6.3 which uses the ONNX file format. Table 3.3 lists the benchmark networks supported by tkDNN; They are all object detection networks. Since the DLA does not support *leaky relu* or *mish* activation currently, we replaced those with *relu* activation and retrained the networks. If there is any layer that the DLA does not support among the layers mapped to the DLA, the layer is actually executed on the GPU, which is called *GPU fallback*. We set the maximum frequency on the *MAXN* power mode, which does not limit the power budget. In the pipelining heuristic, we use 5,000 test images to estimate the FPS of each candidate set of cut-points. After the final configuration is determined, we perform each experiment five times and get the average value.

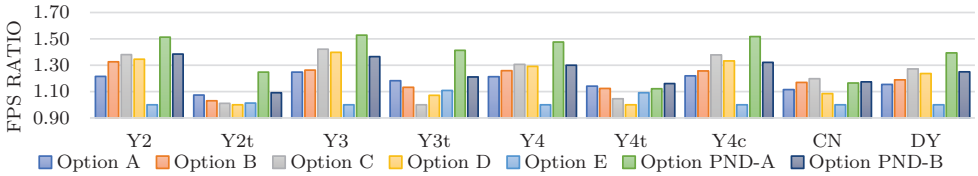
Table 3.3 lists the object detection benchmark applications. Although most of the networks are based on Yolo, the networks vary in accuracy and the number of layers. Since the DLA does not support *leaky relu* or *mish* activation, we replace those with *relu* activation and retrain the networks. All networks except *Yolov4csp* read and resize images to 416x416 during pre-processing. For *Yolov4csp*, input images are converted to letter box images rather than resizing that may distort the image.

Table 3.3: The labels and the number of layers of benchmark applications.

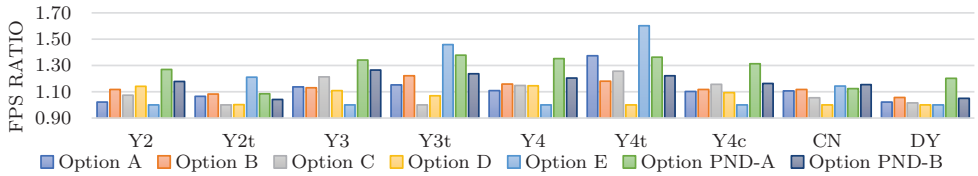
Network	Label	# of layers	mAP (AP)				mAP (AP50)			
			GPU		DLA		GPU		DLA	
			FP16	INT8	FP16	INT8	FP16	INT8	FP16	INT8
Yolov2 [45]	Y2	54	0.224	0.222	0.224	0.216	0.436	0.435	0.437	0.432
Yolov2tiny [45]	Y2t	24	0.096	0.096	0.096	0.093	0.246	0.245	0.246	0.244
Yolov3 [46]	Y3	179	0.286	0.289	0.286	0.270	0.506	0.515	0.506	0.511
Yolov3tiny [46]	Y3t	35	0.133	0.133	0.133	0.127	0.312	0.310	0.312	0.308
Yolov4 [47]	Y4	269	0.399	0.395	0.399	0.342	0.612	0.611	0.612	0.597
Yolov4tiny [47]	Y4t	57	0.203	0.200	0.203	0.183	0.392	0.389	0.392	0.380
Yolvo4csp [48]	Y4c	290	0.427	0.421	0.427	0.401	0.608	0.604	0.608	0.598
CSPNet [49]	CN	228	0.359	0.359	0.359	0.332	0.586	0.588	0.586	0.585
Densenet+Yolo [50]	DY	508	0.190	0.205	0.190	0.137	0.380	0.412	0.380	0.318

We used *COCO2014 trainval* data set and 416x416-size images for training all the networks except *Yolov4csp*. *Yolov4csp* is trained with *COCO2017 train* data set and 512x512-size letter box images. The input image size for inference is 416x416, and *COCO2017 val* is used for measuring the FPS performance and energy consumption. We use *CodaLab* [51] to obtain the mean average precision (mAP) of networks with *COCO2017 test* data set and check the accuracy of the networks which are converted from their original activation to *relu* activation.

In Table 3.3, the *AP50* is the mean average precision when the intersection over union (IoU) threshold is 0.5, and the *AP* is the mean from *AP50* to *AP95*. Even though the mAP values are measured without pipelining on each processor, they are not affected after pipelining is applied. As shown in Table 3.3, there is no significant mAP difference between 16-bit floating-point (FP16) and 8-bit integer (INT8) precision except for one case; In the case of *Densenet+Yolo* with INT8 precision on DLA, we could achieve noticeably lower accuracy with INT8 precision. Nonetheless, we include this benchmark to evaluate the complexity of the proposed technique since it is the largest benchmark available.



(a) FP16



(b) INT8

Figure 3.8: FPS comparison among options on FP16 and INT8 precision

3.6.2 Design Space Exploration Results

The proposed optimization methodology is applied to each benchmark network. First, we run the pipeline cut-point explorer with the global search for all mapping options to find the best mapping option. Figure 3.8 displays the throughput ratio obtained by the global search among difference mapping options for FP16 precision (Figure 3.8 (a)) and INT8 precision (Figure 3.8 (b)). We compute the relative FPS ratio over the lowest FPS for each network. Option *E* shows the least performance in large networks. However, for light-weight networks which have a small number of layers with a short inference time, option *E* outcomes the decent result or the best result even when using INT8 precision. This is because the overhead caused by pipelining overshadows the benefits in such networks. The result of using INT8 precision shows this tendency more clearly since using GPU only gives the best performance in light-weight networks.

It is observed that *PND-A* outperforms the other options for large networks except for *CSPNet*. For *CSPNet* that uses grouped convolution in early layers, it is difficult to balance the pipeline stages mapped to GPU and DLA because the execution time of a grouped convolution layer is moved from the GPU to a DLA. To make matters worse,

Table 3.4: The search time and range of the network pipelining heuristic

Network	FP16			INT8		
	Search Time (Hours)	Searched tuples	Searched design space (%)	Search Time (Hours)	Searched tuples	Searched design space (%)
Y2	4.3	249	0.900	4.3	243	0.878
Y2t	2.1	178	6.910	1.8	172	6.677
Y3	8.2	223	0.023	7.0	260	0.027
Y3t	2.1	183	2.366	1.8	174	2.250
Y4	9.9	283	0.009	10.1	344	0.010
Y4t	4.0	294	0.906	2.6	214	0.659
Y4c	11.0	319	0.008	12.2	320	0.008
CN	12.1	308	0.015	16.0	424	0.021
DY	38.1	386	0.002	29.0	325	0.001

CSPNet has a very long residual path from one-third point to the end of the network, which incurs a severe limitation to acquire feasible cut-point tuples. As a result, the performance gain of *CSPNet* is relatively small compared with other large networks, and the obtained throughput is not dependent on the number of DLAs used. Since option *D* has four pipeline stages and selecting a competitive and feasible cut-point tuple is very difficult, it gives a poor performance, unlike other large networks.

We examine the exploration complexity of the proposed methodology counting the total number of explored tuples in the pipeline cut-point explorer module. Table 3.4 shows the results with FP16 and INT8 precision, summing up the tuples explored during the global search step and the tuples explored in the full search step for the best mapping option. Note that each tuple is unique since we reuse the results when the same cut-point tuple is explored in the search process. The table also shows the percentage of the searched design space by the proposed heuristic. Since the design space of possible cut-point tuples is huge, the proposed heuristic prunes the design space drastically so that the searched design space takes a very small portion. For example, for *Densenet+Yolo* that has 508 layers, we explore only $0.001\% \sim 0.002\%$ of the total search space. Note that the number of searched tuples is not exploding as the network size increases.

Note that there is no single best option for all networks and precision, and the FPS performance varies up to 53% on FP16 precision and 60% on INT8 precision, depending

Table 3.5: Fine-tuned configurations of the selected cut-points from our methodology

Label	FP16						INT8					
	Option	Cuts.	Pre.	Post.	Buf.	Streams	Option	Cuts.	Pre.	Post.	Buf.	Streams
Y2	PND-A	23	2	1	5	4,2	PND-A	16	4	1	7	4,2
Y2t	PND-A	11	5	2	10	4,2	E	-	6	3	18	2
Y3	PND-A	57	1	1	4	4,2	PND-A	59	2	1	8	4,2
Y3t	PND-A	5	5	1	11	4,4	E	-	6	2	20	5
Y4	PND-A	82	1	1	5	4,2	PND-A	87	2	1	8	4,2
Y4t	PND-B	30,45	4	1	7	2,4,1	E	-	6	1	12	4
Y4c	PND-A	80	1	1	16	4,3	PND-A	88	2	2	18	4,4
CN	C	30,35	1	1	5	2,2,3	PND-B	34,106	1	1	4	1,4,2
DY	PND-A	95	1	1	4	4,2	PND-A	98	2	1	4	4,2

on which mapping option is used.

3.6.3 Parameter Fine-tuning Results

In this experiment, we fine-tuned the system-level optimization parameters after we selected the best mapping option of each network. The best configuration of each network for FP16 and INT8 precision is shown in Table 3.5. *Option* represents the mapping option in Table 3.2. *Cuts.*, *Pre.*, *Post.*, *Buf.* and *Streams* indicate the cut-point tuple, the number of pre/post processing threads, and the number of buffer and streams, respectively. For example, the best option for *Yolov2* with FP16 precision is using the *PND-A* mapping option that consists of two stages. The first stage is mapped to a DLA from layers #0 to #19, and the second stage is mapped to the GPU from layers #20 to the last layer. It is noteworthy that the number of pre-processing threads and the number of buffers are larger for INT8 precision than for FP16 precision. It is because that the inference time becomes smaller if INT8 quantization is used. Also, light-weight networks use many pre-processing threads in both precision types since the inference time is much shorter than the pre-processing time.

3.6.4 Comparison with Other Methods

3.6.4.1 Comparison of Performance Results

We compared the performance results of the proposed methodology with the other three methods. Two schemes, denoted as *Base(D)* and *Base(G)*, are the default TensorRT implementation where a single execution context is mapped to a DLA or GPU stream without the CPU-GPU pipelining (pre-/post-processing pipelining), respectively. In the case of *Base(D)*, the layers that cannot be executed on the DLA are mapped to GPU automatically by the *GPU fallback*. The profile-based method, denoted as *Profiled*, is a similar method to the state-of-the-art approach of [12], which uses a genetic algorithm (GA) to make the pipelining decision based on the per-layer profiled information. The execution time of each layer on GPU could be profiled by using the TensorRT IProfiler. In the work of [12], they estimated the layer-by-layer basis execution time on an NPU by multiplying some multiple to the per-layer execution time on a GPU. Similarly, we estimated the execution time on a DLA by weighting the GPU execution by the average performance ratio between DLA and GPU based on the total inference time since layer-wise profiling is not available on a DLA. We implemented the GA in which a chromosome consists of genes that represent the mapping of layers to the processing elements. While the original GA method of [12] does not limit the number of pipeline stages, we limit the number of pipeline stages to 4. The profile-based method determines the pipeline cut-points only. Hence we fine-tuned the optimization parameters similar to the proposed method.

Figures 3.9 and 3.10 show the performance comparison among four methods for FP16 and INT8 precision, respectively. As shown in Fig. 3.9 (a) and Fig. 3.10 (a), the profile-based method and the proposed method achieve significantly higher performance than the baseline method since the pipelining and parallelization techniques are effective for throughput improvement. The proposed method achieves the FPS performance improvement by 101% $\sim$ 680% over the baseline GPU. The proposed method gives higher

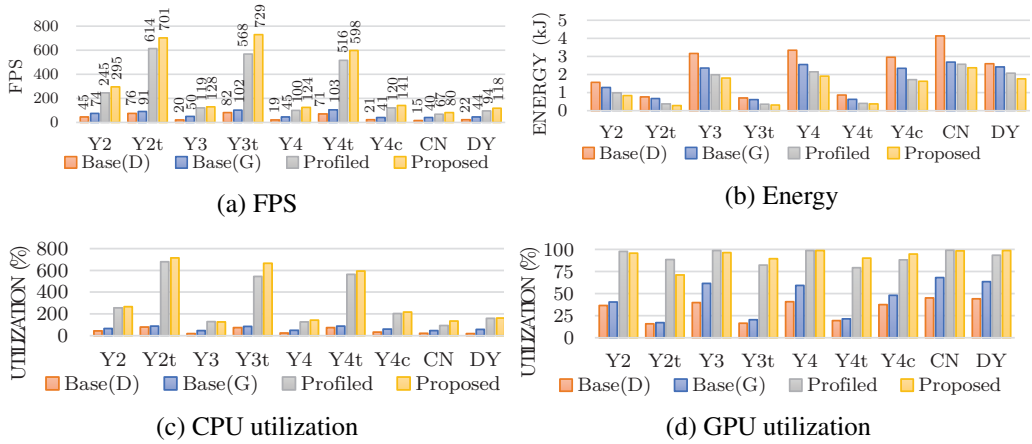


Figure 3.9: FPS, energy comparison, and CPU/GPU utilization among four methods with FP16 precision

FPS performance than the profile-based method by up to 32%. It confirms the superiority of the proposed technique over the profiling-based method that is popularly taken in the previous work. The profile-based method and the proposed method give the same result for *Yolov2tiny* and *Yolov3tiny* networks with INT8 precision since both methods select the same mapping option *E* which uses the GPU only.

Figure 3.9 (b) and Fig. 3.10 (b) show the comparison results among the methods in terms of energy consumption. Because the throughput obtained from the proposed method is higher than the baseline, the energy consumption is reduced noticeably, up to 55%, even though it consumes more power by using all processing elements. The proposed method is also better than the profile-based method by up to 18%.

The CPU and GPU utilization results among four methods are shown in (c) and (d) of Fig. 3.9 and Fig. 3.10. The profiled-based and proposed methods exploit both processors more effectively thanks to pipelining and parallelization than the baseline methods. Note that the profile-based method of *Yolov2* and *Yolov3* with FP16 precision, and *CSP-Net* with FP16 and INT8 precision show higher GPU utilization compared to the proposed method because more layers are allocated to the GPU. In the case of *Yolov2tiny*

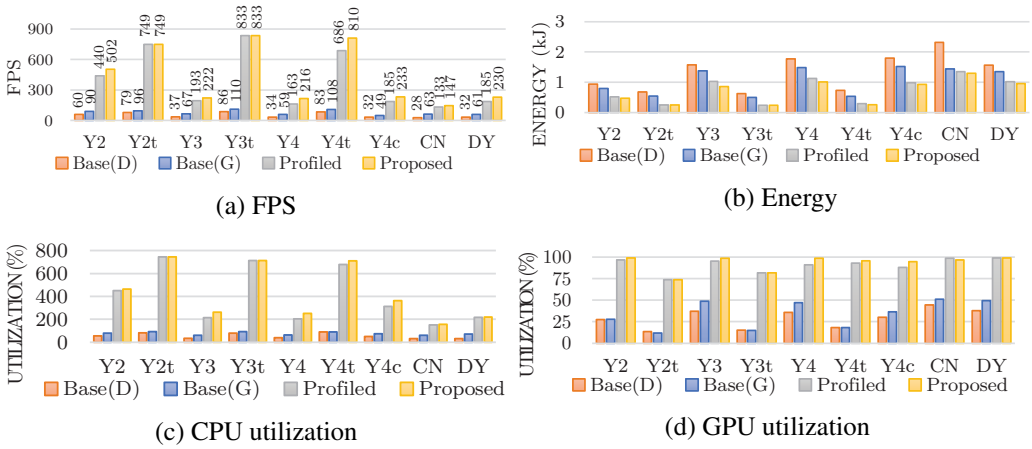


Figure 3.10: FPS, energy comparison, and CPU/GPU utilization among four methods with INT8 precision

with FP16 precision, the selected mapping option of the profile-based method only uses a single DLA even though two DLAs are available in the Xavier. So the FPS of the profile-based method is lower than the FPS of the proposed one even though the GPU utilization of the profile-based method is much higher than the proposed method. *Yolov2tiny* with FP16 precision, *Yolov2tiny*, *Yolov3tiny*, and *Yolov4tiny* with INT8 precision cannot fully utilize the GPU since the CPU processing parts such as pre-/post-processing threads become the bottleneck among the pipeline stages.

3.6.4.2 Comparison with an interleaved execution on different processors without the pipelining

To use multiple processors simultaneously without partitioning the inference, it is possible to run the image on different processors in an interleaved fashion. For each processor, once the inference is complete, data is read from the buffer and the next image is processed, increasing parallelism without pipelining. We compare the interleaved method and the proposed method with INT8 precision for three networks: *Yolov4*, *Yolov4csp*, and *Densenet+Yolo*.

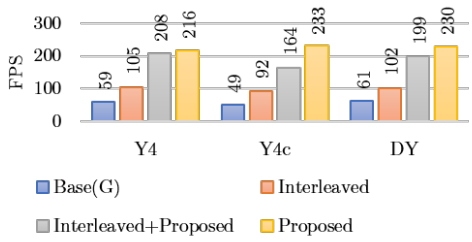


Figure 3.11: Comparison with interleaved execution on different processors

Figure 3.11 shows the results of the experiment. The second item, *Interleaved*, is the result when running the application in the interleaved fashion with three buffers and three streams. This is because different processors need to have at least one stream. The third one, *Interleaved+Proposed*, is the result of applying the pre- and post-processing pipelining, multi-threading, and multiple execution contexts techniques as the proposed method. The results are obtained by reducing the parameters from their maximum values until there is no performance degradation.

The experimental results show that the *interleaved* method is more effective than the baseline method. In addition, the experimental results of the *Interleaved+Proposed* method are comparable to the proposed method that applies the proposed techniques including the pipelining. However, the *Interleaved+proposed* method is not as effective as the proposed method because not all layers could be executed in DLA due to GPU fallback.

3.6.4.3 Comparison of using the solution obtained by the profile-based method as the initial solution

Although the profile-based method does not show the best results, the solution obtained by the profile-based method can be used as an initial solution of the proposed heuristic. However, the profile-based method does not explore the solution by mapping options, but rather determines the mapping layer by layer. Therefore, we experiment with

Table 3.6: FPS comparison of using the solution obtained by the profile-based method as the initial solution

Network	Mapping option	FPS	
		Using a random initial solutions	Using the initial solution by Profiled
Y4	A	101	101
Y4c	C	134	134
DY	B	103	100

the same mapping options as the mapping found by the profile-based method. We compare the results of using the initial solution and the results of the heuristic starting with a random solution. In this experiment, we use FP16 precision because the profile-based method shows various mapping configurations in FP16 precision than in INT8 precision.

Table 3.6 shows the FPS comparison when the solution obtained by the profile-based method is used as the initial solution. Both methods give similar results. Indeed, both methods find the same mappings for the *Yolov4* and the *Yolov4csp*. This confirms that the heuristic is not much influenced by the initial solution.

3.6.4.4 Comparison with Other Exploration Methods

In this section, the proposed parameter optimization method is compared with two other GA-based meta-heuristic methods among the four latest networks with INT8 precision. The first method is the *comprehensive-GA* method which searches all the parameters proposed in this paper through a genetic algorithm at once. The range of the parameter values such as pre-/post-processing thread numbers, the number of buffers, and stream numbers are determined by the parameter maximizer that is shown in Fig. 3.6. The other method is *Fine-tuning-GA* which uses a meta-heuristic for fine-tuning. The latter method uses our heuristic to find the best mapping option and the cut-point tuples. Then, a GA-based meta-heuristic is performed to find optimal fine-tuning parameters.

Comparison results are shown in Table 3.7. We compare the FPS, the number of searched points, and the number of built engines. The FPS of all the methods is sim-

Table 3.7: Comparison of exploration methods

Label	Comp.-GA				FT.-GA			Proposed				
	FPS	Searched points #	Built engines #		FPS	Searched points #		FPS	Searched points #		Built engines #	
			GPU	DLA		Mapping	Param.		Mapping	Param.	GPU	DLA
Y4	212	497	253	329	213	344	162	216	344	19	61	276
Y4t	808	1261	173	495	822	214	559	810	214	35	24	74
Y4c	232	1204	409	779	232	320	143	233	320	14	69	250
CN	148	335	171	220	147	424	161	147	424	15	85	270

ilar, meaning that all methods find near-optimal solutions successfully. The number of searched points of *fine-tuning-GA* and the proposed method are divided into two parts: *Mapping*, and *Param*. *Mapping* indicates the number of explored cut-point tuples during the execution of the pipeline cut-point explorer, and *Param* is the number of tested parameter combinations to minimize the resource usage in the fine-tuning step. The proposed method runs 88% $\sim$ 94% less searched points for parameter fine-tuning compared to the *fine-tuning-GA* method. The total number of searched points is 27% $\sim$ 80% smaller than that of the *comprehensive-GA* method except *CSPNet* network.

Built engines # is the number of built engines during exploration. The engine building time takes a significant portion of the overall exploration time. Since GPU engine building time is much longer than DLA engine building time, building a small number of GPU engines is helpful to reduce the exploration time. As shown in Table 3.7, our proposed method builds fewer engines than the *comprehensive-GA* method. Because our proposed method builds engines on sampled cut-points and limited cut-point locations based on utilization, built engines are easily reused compared to the *comprehensive-GA* method. In summary, the proposed method is efficient in exploring optimization parameters compared to other meta-heuristic techniques.

3.6.5 Experiments with Varying Configurations

In the experiment above, the proposed methodology is applied for a given input size and the batch size. In this experiment, we vary the input size and batch size. Four

Table 3.8: Comparison of the best result of reused cut-points from 416x416 and the re-explored cut-points with our fast heuristic search

(a) Input size with 512 x 512 (Energy unit: J)

Label	Baseline (512x512)		Reused cut-points from 416x416		Re-explored cut-points with fast heuristic search				Searched tuples	Built engines #	
	FPS	Energy	FPS	Energy	Option	Cuts.	FPS	Energy		GPU	DLA
Y4	49	1897	143	1343	PND-A	72	155	1361	78	45	80
Y4t	91	658	576	346	E	-	576	346	54	21	29
Y4c	40	1951	160	1225	PND-A	72	169	1250	54	34	58
CN	54	1806	110	1707	E	-	113	1521	79	30	79

(b) Input size with 608 x 608 (Energy unit: J)

Label	Baseline (608x608)		Reused cut-points from 416x416		Re-explored cut-points with fast heuristic search				Searched tuples	Built engines #	
	FPS	Energy	FPS	Energy	Option	Cuts.	FPS	Energy		GPU	DLA
Y4	42	2544	100	1971	PND-A	72	106	1998	65	40	64
Y4t	78	826	415	482	E	-	415	482	55	21	29
Y4c	34	2572	111	1794	PND-A	72	121	1795	54	34	58
CN	42	2544	77	2517	E	-	78	2254	64	29	65

networks with INT8 precision are used for benchmarks: *CSPNet*, *Yolov4*, *Yolov4tiny*, and *Yolov4csp*. We already obtained sub-optimal pipeline cut-point tuples of all mapping options from Section 3.6.2. Instead of finding a sub-optimal cut-point tuple from scratch, we reuse the obtained tuples as an initial cut-point tuple as explained in Section 3.5.1 and set the internal variable K to 1 in Algorithm 2.

3.6.5.1 Scaling up Input Size

Since our proposed method provides a high FPS performance of a deep learning application, a user may want to increase mAP by scaling up the image size. We conducted the experiments while scaling up the image size to 512x512 and 608x608, as shown in Table 3.8. The third main column shows the result when we reuse the best cut-points of the previous experiment. The fine-tuning of other parameters except for the cut-points is newly done. The results with reused cut-points still overwhelm the baseline in both FPS and energy consumption metrics. The fourth main column shows the results with the newly found cut-points from the proposed method with a fast search heuristic. We obtain

Table 3.9: Comparison of the best result of reused cut-points from batch 1 and the re-explored cut-points with our fast heuristic search

(a) Batch size 4 (Energy unit: J)

Label	Baseline (batch 4)		Reused cut-points from batch 1		Re-explored cut-points with fast heuristic search				Searched tuples	Built engines #	
	FPS	Energy	FPS	Energy	Option	Cuts.	FPS	Energy		GPU	DLA
Y4	95	1196	221	906	PND-A	67	239	922	61	45	61
Y4t	255	341	894	236	E	-	894	236	55	21	29
Y4c	85	1160	242	834	PND-A	72	268	827	53	35	57
CN	112	1103	170	1140	E	-	175	1017	74	28	76

(b) Batch size 8 (Energy unit: J)

Label	Baseline (batch 8)		Reused cut-points from batch 1		Re-explored cut-points with fast heuristic search				Searched tuples	Built engines #	
	FPS	Energy	FPS	Energy	Option	Cuts.	FPS	Energy		GPU	DLA
Y4	115	1109	221	891	PND-A	65	241	918	85	51	82
Y4t	341	303	908	234	E	-	908	234	55	21	29
Y4c	100	1089	245	820	PND-A	72	269	828	58	38	61
CN	127	1052	178	1107	E	-	179	996	77	30	78

the higher FPS and similar energy consumption by finding new sub-optimal cut-points. In addition, the number of searched tuples for finding sub-optimal cut-points is reduced by 74% $\sim$ 85% compared to the first exploration case, and the number of built engines is also reduced by 63% $\sim$ 85%. This experiment proves that the changing image size may affect the resultant configuration, and it is confirmed that reusing the pre-explored cut-points as initial cut-points effectively reduces the search time in the proposed methodology.

3.6.5.2 Scaling up Batch Size

When multiple images come from multiple cameras, adjusting the batch size is a popular way to improve the throughput performance. Table 3.9 shows the results with increased batch sizes. Similar to the experiment of increasing the input size, we reused the selected cut-points from batch 1 and conducted a re-exploration of cut-points with a fast search heuristic. As displayed in the table, we could obtain higher FPS performance and similar energy consumption by increasing the batch size. In addition, the number of searched tuples for finding sub-optimal cut-points is reduced by 74% $\sim$ 83% compared

Table 3.10: Inference time comparison between baseline method and the found mapping by the proposed method

Network	Base(G)	Proposed		
	Inference time (us)	Inference time (us)	Communication overhead (us)	Overhead ratio (%)
Y4	8055	14570	392	2.69
Y4c	7550	13297	373	2.81
DY	8235	14108	203	1.44

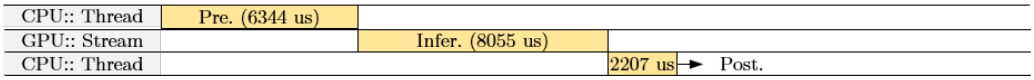
to the first exploration case, and the number of built engines is also reduced by 60% $\sim$ 85%.

3.6.6 Analysis and Discussion

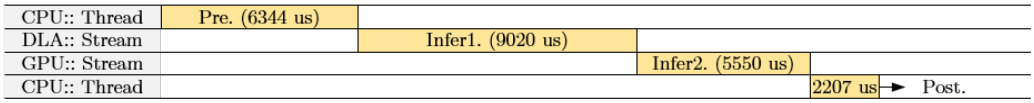
3.6.6.1 Overhead of Partitioning inference

We examine the communication overhead of partitioning deep learning inference. Table 3.10 shows the inference time comparison between the proposed method and the baseline method with INT8 precision. The communication overhead is newly added time due to the partitioning of the network. Since the overhead is relatively small compared to the inference time, the partitioning inference is a reasonable way to improve the performance.

Figure 3.12 shows gantt charts for two cases of the Yolov4 network. The first one is the case when running on a single GPU without the pipelining, and another one is the case when running with a found mapping by the proposed method. Since the DLA is much slow processor than the GPU, the inference part on DLA (Infer1.) in Fig. 3.12 (b) is longer than the entire inference of the baseline scheme in Fig. 3.12 (a). Nonetheless the bottleneck is resolved as confirmed in the experimental results by various techniques, such as PND, multi-threading, multiple execution context, and so on.



(a) When running on a single GPU without the pipelining



(b) When running with a found mapping by the proposed method

Figure 3.12: Gantt charts for different mappings of the Yolov4 network

Table 3.11: Inference time comparison between baseline and the found mapping by the proposed method

Network	Multiple	Profiled (us)	Measured (us)	Difference ratio (%)
Y4	2.95	13436	14570	7.79
Y4c	2.76	12216	13297	8.13
DY	2.72	12855	14108	8.88

3.6.6.2 Difference from the Profile-based Method

Since per-layer profiling is not available for DLA, the profile-based method estimates the layer-wise execution time on DLA by multiplying the ratio of GPU time to DLA time by the per-layer time obtained for GPU. Although the experimental results show that the profile-based method is worse than the proposed method that evaluates the mapping by running on a device, it is necessary to check the difference with the profile-based method.

Table 3.11 shows the comparison of the inference time between baseline and the mapping found by the proposed method with INT8 precision. The multiple in the table is the ratio to estimate per-layer execution time on DLA. The difference ratio is a ratio of difference over measured time. The estimated inference time is shorter than the measured inference time. This can be a problem in the worst-case response time analysis since the schedulability can be overestimated. In addition, Figure 3.13 displays the gantt chart based on the estimated layer-wise execution time for a mapping found by the pro-

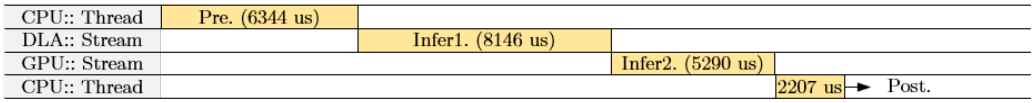


Figure 3.13: Gantt chart based on estimated layer-wise execution time for a found mapping by the proposed method

posed method. Compared to Fig. 3.12 (b), the difference on DLA is not negligible. This shows the estimated layer-wise execution time is not accurate, and can be misleading for mapping decision and analysis.

3.6.6.3 Supporting a Transformer Network

Transformer [52] is an emerging network with a complex structure. Transformer has multiple matrix multiplication layers. Currently, commercially available embedded boards have NPUs targeting CNNs, and these NPUs cannot perform matrix multiplication layers. Therefore, the only processor that can run the transformer network is the GPU in practical. From this point of view, it is difficult to apply the proposed techniques for transformers in practice.

Since transformers are quite computationally intensive, networks are also being developed that reduce the computation. EfficientFormer [53] is a vision transformer that replaces some layers with latency-friendly layers to run faster. This reduces the number of layersm, such as matrix multiplication, so that parts of the network can be performed in DLA. We try to check the viability of our proposed method for the transformer using an EfficientFormer written in ONNX file format.

However, there are also other difficulties in applying proposed techniques to the transformer. First, there is an internal error in TensorRT when trying to run a partitioned network at arbitrary points. As TensorRT is a closed code, this error cannot currently be resolved. The second issue is related to the GeLU activation in the EfficientFormer. The GeLU operator is currently not a primitive operator on ONNX, the GeLU operator is

replaced by a combination of layers in ONNX. Some of these layers fall back to the GPU when running through TensorRT. Since the number of DLA parts in TensorRT is limited, this also implies that there are difficulties in applying proposed techniques.

In conclusion, the proposed techniques cannot be applied to transformer networks due to limitations associated with NPUs. Nevertheless, the proposed techniques will still be effective when NPUs that support operations of transformer become available, we believe. We leave it as a future work.

Chapter 4

Optimization of Multiple Deep Learning Applications under Real-time Constraints

4.1 Overview

In this chapter, we find mappings and frequencies of multiple deep learning applications while keeping deadline constraints and reducing energy consumption. NPUs and SDKs introduce additional challenges that are not considered in systems consisting of CPUs and GPUs only. To address these issues, we propose a three-step methodology. First, we select Pareto-optimal mappings for each deep learning application. Then, we find mapping combination for multiple applications among the mapping candidates selected in the previous step. Finally, the frequency is tuned to reduce energy consumption.

The rest of this chapter is organized as follows. We review the related work to our work in the next section. Afterward, we introduce the system model with a motivational example and notation to clarify the problem in Section 4.3. After the proposed methodology is presented in Section 4.4, the experiment results are displayed in Section 4.5.

4.2 Related Work

This section reviews the related work in the following two subsections involved in the proposed method: mapping/scheduling multiple applications and running multiple

DL applications.

4.2.1 Mapping and Scheduling Multiple Applications

Since the mapping and scheduling problem of multiple applications on multiple processing elements (PEs) is known to be NP-hard [54], numerous sub-optimal techniques have been proposed. They are classified into a design-time approach, run-time approach, and a hybrid approach that combines the design-time analysis and run-time adaption [55].

In the design-time approach, it is assumed that all applications are running concurrently. If there is a dynamic variation of workload, the worst-case scenario is considered. Earlier works used list scheduling heuristics, named BIL [56] and HEFT [57], to schedule multiple applications onto heterogeneous processors. They assume that each application is specified by a DAG, and the execution time of a node on each processor is known.

Kang et al. [58] proposed a two-phase optimization scheme for the mappings of multiple applications represented by SDF (Synchronous Dataflow) graphs [15]. They find a Pareto-optimal set of static schedules for each graph via a genetic algorithm in the first step, then explore the combination of per-graph schedules with the schedulability analysis with another genetic algorithm. Their two-step approach is similar to our proposed technique, while their assumed hardware platforms are homogeneous processors.

If the workload varies at run time, the run-time approach is usually taken, aiming to maximize the load balancing if there are no real-time constraints. Niknafs et al. [59] proposed a mapping and scheduling method for incoming tasks. They optimize the energy consumption with the mixed-integer linear programming formulation and the heuristic while satisfying the deadline of tasks. Also, they investigated the effect of the workload prediction in terms of keeping the deadline and minimizing the energy consumption while not discussing how workload prediction can be made. Khasanov et al. [60] presented a run-time management method for multi-threaded applications to reduce energy consumption without deadline violation on a homogeneous multiprocessor system. They

introduced the multidimensional knapsack problem-based heuristic for fast scheduling. Run-time techniques usually assume that the execution time of each application is known as a priori. Donyanavard et al. [61] collect data through periodic sensing and predict the performance and power based on the data. And they introduced a fast heuristic of run time task allocation to reduce the energy consumption while retaining the throughput. However, they did not consider real-time constraints.

To utilize the benefits of design-time approaches and run-time approaches, hybrid methods have been studied extensively. Some researchers assume that the dynamic behavior of a system can be modeled as a finite set of scenarios and make a static schedule for each scenario. Gheorgita et al. [62] introduced the concept of system scenarios and proposed the methodology to reduce the energy consumption. They identified the scenario and optimized the system per scenario at design time. At the run time, the scenario is predicted, and the system is arranged with a pre-optimized setting. Schor et al. [63] proposed a design flow for mapping applications on manycore systems representing the dynamic scenario variation with an FSM which has a finite number of scenarios. Assuming that the mappings are resident, they found a sub-optimal mapping for each scenario by an evolutionary algorithm. It aims to minimize the maximum core utilization.

The other researchers find a set of static mappings for each application and find an appropriate static mapping at run time depending on the scenario. Jung et al. [64] specified the dynamic behavior of applications by the dataflow model and FSM. They performed design-time analysis for each application subject to the number of processors. At run time, the number of used processors is changed according to the state to satisfy the throughput constraint. Quan et al. [65] proposed a scenario-based run-time mapping algorithm on an MPSoC-based embedded system. After finding a set of mappings of each application, they cluster workload scenarios and find the static mapping for each scenario. Their method finds the critical task that hinders achieving the goal throughput of each application and re-maps the task to fulfill the objective. Also, they expand their approach

Table 4.1: Comparison with the related works of running multiple deep learning applications

Works	Optimization approach	Pipeline	Using NPU	Schedulability check	Experiment on a real platform	Energy awareness
NestDNN [67]	Hybrid				✓	✓
DeepEye [4]	Hybrid	✓			✓	✓
S^3DNN [8]	Run			△	✓	
DART [10]	Design	✓		✓	✓	
LaLaRAND [13]	Run	✓		✓	✓	
Pujol et al. [9]	Design		✓	✓	✓	
Kang et al. [12]	Design	✓	✓	✓		✓
Proposed	Hybrid	✓	✓	✓	✓	✓

by adding a run-time throttling step that estimates the overhead of reconfiguration [66]. However, they evaluated the proposed technique with an in-house system level simulator, not a real hardware platform.

To the best of our knowledge, there is no previous work that considered all technical challenges identified in this work. Our proposed technique belongs to the hybrid approach based on the finite set of scenarios similar to [63].

4.2.2 Running Multiple Deep Learning Applications

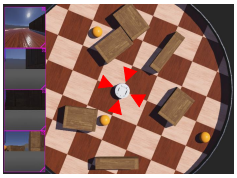
Research on the execution of multiple DL applications in embedded systems has recently become active. Table 4.1 summarizes the comparison of some related works and positions the proposed method.

NestDNN [67] presented a framework that schedules multiple DL applications considering run-time resource requirements. At design time, they created a multi-capacity model consisting of models that provide a resource-accuracy trade-off for each application. Then, they monitor the resource and select the proper model for each application at run time. However, they did not exploit model-parallelism nor pipelining of each application. DeepEye [4] proposed a wearable camera that is capable of running multiple DL applications. They segregated layers into the computation- and memory-heavy layers at design time. At run time, they interleaved the execution of layers of multiple applications

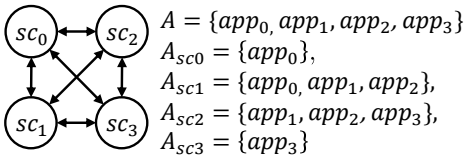
on a CPU-GPU heterogeneous system in a pipeline manner. It aims to reduce the latency of inference while prolonging the battery lifetime. Although both hybrid schemes, NestDNN [67] and DeepEye [4], considered the trade-off between energy and performance, they did not consider the real-time requirements of applications.

S^3DNN [8] introduced a run-time methodology that optimizes real-time correctness while increasing the throughput for multiple deep neural networks (DNNs). Their method selectively fuses multiple images to use fewer DNN instances and schedule instances to GPUs with a least-slack-first (LSF) policy. In spite of using the deadline-aware scheduling policy, it does not guarantee to keep the deadline. So, we put a triangle in the table to mean the partial consideration of the schedulability. They did not exploit the model parallelism of applications. DART [10] proposed a framework for multiple DNNs on a CPU-GPU heterogeneous system. They mapped the layers of DNNs onto CPU and GPU via a heuristic at design time. They ensured the schedulability of real-time applications and increased the throughput of best-effort applications. Albeit they applied the admission control for new tasks at run time, it does not change the mappings. LaLaRand [13] is a framework targeting a CPU-GPU heterogeneous system similar to DART. It allocates layers to processors at run time, supporting dynamic layer-level quantization. The main goal is to improve the schedulability of the system while minimizing the accuracy loss by layer-level mapping and quantization. All these methods [4, 10, 13] exploit the temporal parallelism of inference by pipelining in a CPU-GPU heterogeneous system. However, their target platform does not contain an NPU that incurs several limitations in its usage.

Pujol et al. [9] and Kang et al. [12] presented the design-time method on a heterogeneous platform including NPU. The former implemented DL applications in Apollo [68] on the Xavier board. They executed multiple networks while keeping the schedulability. However, each neural network instance runs on a single type of processor in their method without exploiting the model parallelism or pipelining. The latter, Kang et al. [12], uses a GA technique to find per-layer mapping of multiple DL applications, aiming at mini-



(a) The patrol robot with front, rear, and side-view cameras



(b) FSM-A: An example finite state machine of the patrol robot

Figure 4.1: Motivational example: patrol robot

mizing the WCRT for each application and energy consumption. They assume that layer-wise profiling information is given for all processing elements. Since it is not possible for NPU, they estimated the execution time of a layer on NPU by multiplying the GPU execution time and the average performance ratio between GPU and NPU. Unlike the proposed technique, they do not consider the scenarios, assuming that all DL applications run concurrently. Moreover, NPU is not used on a real platform in their experiments even though they considered the NPU in the analysis. With these limitations, they could find the mapping of all applications together at once. On the contrary, we propose a three-step approach to consider scenarios and the technical challenges imposed by the NPU and its SDK.

As displayed in Table 4.1, the mapping problem addressed in this work has not been tackled by the previous work, to the best of our knowledge.

4.3 System Model

4.3.1 Motivational Example

Let us consider a patrol robot that explores a certain space, as shown in Fig. 4.1 (a). The robot moves around and searches for objects around it. The robot has cameras on all four sides: back and forth, left and right. It executes multiple object detection networks to detect objects while moving. We assume that objects within a certain distance can be

detected by other means such as ultrasonic sensors. If an object is detected, we run an object detection network to identify the object. It means that the number of object networks may vary at run time. Such system behavior can be described by an FSM, as illustrated in Fig. 4.1 (b). Each state has the applications to be executed. In the figure, each application app_0, app_1, app_2, and app_3 denotes the application corresponding to each camera at the front, left, right, and back, respectively. For instance, state sc_1 indicates the state that consists of applications app_0, app_1, and app_2 running at the same time. In addition, each inference has a deadline because its result is used as an input to determine the action of the robot. Furthermore, the deadline constraint can vary depending on the speed of the robot. If the robot moves faster, then the object detection should be finished earlier. It is also important to reduce energy consumption since the robot is battery-powered.

In this paper, we propose a design methodology for running multiple deep learning (DL) applications with real-time constraints on a heterogeneous processor system while minimizing the energy consumption, to support this motivational example. Even though we use an FSM to describe the dynamic system behavior, what is needed in the proposed approach is how long the system stays and which applications are running in each state. Thus we omit the explanation of how state transition occurs and which state is the initial state. In this paper, we assume that the residence time of all states is the same for simplicity. If the state transition probability is given, however, we will be able to compute the relative residence time of all states. Since this computation is out of the scope of this work, it is left as future work.

4.3.2 Notation

Notations used for system model and the problem definition are summarized in Table 4.2.

Architecture Specification: The target platform has a set of heterogeneous PEs, $\mathcal{PE}$, and processor type *proc* can be one of CPU, GPU, and DLA. Each processor has a

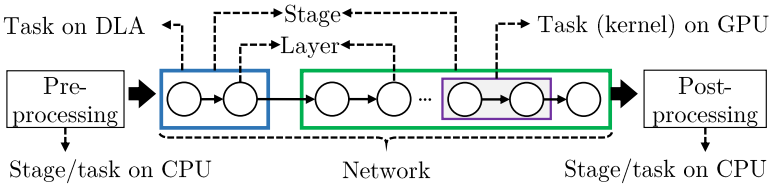


Figure 4.2: Pipelining of the DL application

Table 4.2: Notations for system model and problem definition

Sign	Description
$\mathcal{PE}$	A set of heterogeneous PEs
$freq_{proc}$	A set of frequencies for processor type *proc*
$\|freq\|$	The sum of all $\|freq_{proc}\|$
$\mathcal{A}$	A set of multiple DL applications
app_i	An application in $\mathcal{A}$
S_i	A set of stages of app_i
p_i	A period and relative deadline of app_i
s^i_j	A jth stage of app_i
$\tau^{i,j}_k$	A kth task of s^i_j
$\|s^i_j\|$	The number of tasks in s^i_j
$P(s^i_j)$ and $P(\tau^{i,j}_k)$	A period of s^i_j and $\tau^{i,j}_k$, respectively
$C_{s^i_j}$ and $C_{\tau^{i,j}_k}$	The WCET of s^i_j and $\tau^{i,j}_k$, respectively
SC	A set of scenarios
sc_i	A scenario in SC
$\mathcal{A}_{sc_i}$	A set of applications to be run on sc_i

set of discrete frequencies: $freq_{CPU}$, $freq_{GPU}$, $freq_{DLA}$. We denote $|freq|$ as the sum of $|freq_{CPU}|$, $|freq_{GPU}|$, and $|freq_{DLA}|$.

Application Specification: We denote a given set of multiple DL applications as $\mathcal{A}$, and each application app_i consists of layers. After pipelining decision is made, an application app_i can be represented by a tuple $\langle S_i, p_i \rangle$, where S_i and p_i are the set of stages and the invocation period of app_i, respectively. Stage $s^i_j \in S_i$ is a pipelining stage of app_i corresponding to the pre- or post-processing step or an inference part partitioned to GPU or a DLA. The blue and green box in Fig. 4.2 illustrates two stages mapped to a DLA and the GPU, respectively. The applications run periodically with the implicit deadline assumption under which the period becomes the relative deadline.

Mapping and Scheduling Specification: A task is a unit used in the analysis. Since

the profiling granularity and the scheduling policy are different among processors, tasks are defined differently depending on the mapped processor. For GPU, a task corresponds to a kernel that may include one or more layers since TensorRT fuses layers to a single kernel via layer fusion, as displayed with a purple box in Fig. 4.2. While per-kernel profiling on the GPU is viable and each kernel runs in a non-preemptive way [69, 70], per-kernel profiling is not provided for DLA. Hence, the pipeline stage mapped to a DLA becomes a single task in the DLA (blue box in Fig. 4.2). The pre- or post-processing of the inference is a preemptable task that is mapped on the CPU. Stage s^i_j consists of one or more tasks, each of which is denoted by $\tau^{i,j}_k$, and the number of tasks in the stage is denoted by $|s^i_j|$. Note that the inference is sequentially executed due to the data dependency between tasks [10], and the order of task execution in the GPU is set by TensorRT [19]. Note that a DL application has a chain structure of tasks, which is also assumed in [10, 13].

The period of a stage s^i_j or a task $\tau^{i,j}_k$, denoted by $P(s^i_j)$ or $P(\tau^{i,j}_k)$ respectively, is the same as the period of an application, p_i. In addition, $C_{s^i_j}$ and $C_{\tau^{i,j}_k}$ represent the WCET of stage s^i_j and task $\tau^{i,j}_k$, respectively. We assume that the execution time of a task is inversely proportional to the frequency of the mapped processor. The mapping decision of app_i is denoted as a function $Map : S_i \rightarrow \mathcal{PE}$. A stage is a mapping unit, and it can be mapped to a single PE. The tasks in the same stage are mapped to the same PE. Data transfer overhead is considered since we measure the latency through direct measurement, including the extra kernels added by TensorRT as depicted in Fig. 2.1. Since the migration overhead is huge, the mapping decision is made statically and not changed at run time.

Scenario Specification: A set of scenarios is specified as an FSM where a state corresponds to a scenario. The set of states (scenarios) is represented as SC. Each scenario $sc \in SC$ consists of a set of applications to be run, A_{sc}, which is a subset of $\mathcal{A}$. For example, in Fig. 4.1 (b), A_{sc1} consists of three applications, app_0, app_1, and app_2. As

each scenario has a different behavior, the exploration is needed to be done per scenario.

4.3.3 Problem Formulation

The problem addressed in this paper can be defined as follows.

Input: All applications in $\mathcal{A}$ and the scenarios, represented by an FSM, are given as input. The available frequency range of processors is already known at design time.

Constraints: The range of throughput constraints is given. For instance, it may be needed to process 15 or 30 image frames per second(FPS), depending on the robot speed. Since we assume the implicit deadline, the deadline constraint is inverse of the throughput constraint[1]. There exist other constraints imposed by the NPU and its SDK, as described in Chapter 1.

Objective: We aim to reduce the energy consumption of the system. If we do not know how long it stays in each scenario at runtime, it is not feasible to estimate the actual energy consumption. Thus, the average energy consumption over scenarios is used as the objective function in the experiments. Note that the proposed methodology can be applied with a different formula for expected energy consumption.

Output: The output of the design-time analysis is the mapping of stages in applications onto processors and the frequency of processors for each scenario and each deadline constraint. In a real deployment, the output includes the measured energy consumption.

[1]We use throughput constraint (FPS) and deadline constraint interchangeably to indicate a real-time constraint in this paper. For example, the throughput constraint of 30 FPS (frame per second) indicates the deadline constraints of $33,333$us.

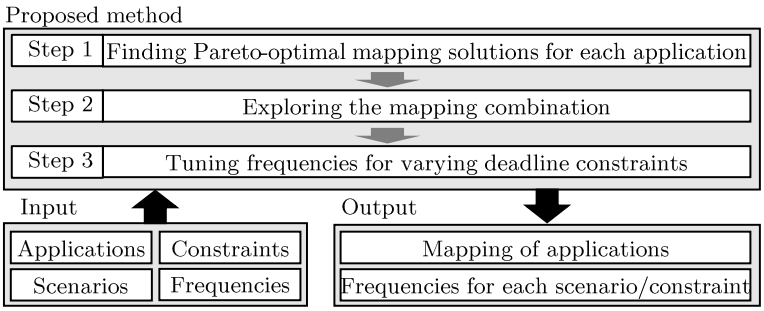


Figure 4.3: Overall flow of the proposed mapping methodology

4.4 Proposed Optimization Methodology

As shown in Fig. 4.3, the proposed methodology consists of three steps explained in this section.

4.4.1 Step 1: Finding Pareto-optimal Mapping Solutions for Each Application

Because of the challenges imposed by the NPU and its SDK, we first determine the Pareto-optimal mappings for each application instead of making the mapping decision for all applications at once. We aim to simultaneously minimize the average end-to-end latency and average power consumption of each processor via a genetic algorithm (GA). Genetic algorithm is a widely used meta-heuristic inspired by evolutionary processes in nature, where sub-optimal solutions are found iteratively, starting with randomly generated initial solutions. A solution of the problem is encoded as a chromosome, and the objective function value, called *fitness*, of each solution is evaluated and compared in each generation. Since there exist several GA solvers publicly available, it is used in the current implementation of the proposed methodology. Other meta-heuristics can be used as long as multiple objectives are supported.

Figure 4.4 illustrates the overview of this step. The GA gets the network as the input and initializes multiple chromosomes or generates a given number of initial ran-

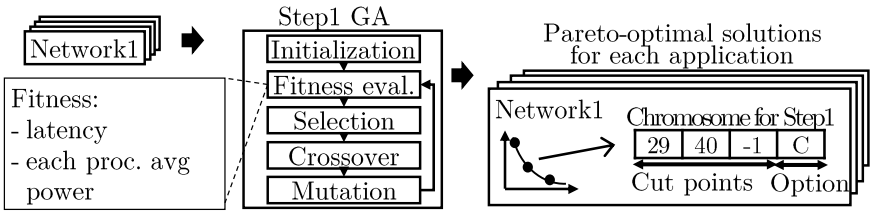


Figure 4.4: The overview of step 1

Table 4.3: Options for intra-network pipelining

Options	# of pipeline stages	Composition of PEs
A	2	DLA0 - GPU
B	3	GPU - DLA0 - GPU
C	3	DLA0 - DLA1 - GPU
D	4	GPU - DLA0 - DLA1 - GPU
E	1	GPU

dom solutions. After fitness evaluation, we select a set of chromosomes and create new chromosomes (offspring) through evolutionary operations such as crossover and mutation. This step is performed on the real board since the fitness evaluation is performed by direct measurement of latency and average power consumption on the real hardware platform.

The chromosome structure is shown on the right side of Fig. 4.4. The first three genes indicate pipeline cut points, and the last gene specifies the mapping option. A cut-point indicates the index of the last layer partitioned into a stage. For instance, the first pipeline stage consists of the 0th layer through the 29th layer. We compare five pipelining options that are listed in Table 4.3. Since the networks we used in this paper have some layers that cannot be performed on a DLA at the last part of the network, the last stage is assigned to GPU in all options. In options *C* and *D*, we distinguish two DLAs with *DLA0* and *DLA1*. In options *B* and *D*, the GPU is allocated two stages since the GPU has more computation power than DLA. In the example, option C is chosen in which three pipeline stages are mapped to DLA0, DLA1, and GPU, respectively, and two cut-points are specified: the second pipeline stage covers from the 30th layer to 40th layer, and the

last pipeline stage starts from the 41st layer to the end of the network. A cut point value of -1 means that there is no corresponding cut point. The maximum number of cut-points is three for mapping option D. For mapping option E, no cut-point needs to be explored. It is noteworthy that DLA0 and DLA1 can be switched in the mapping options depicted in Table 4.3.

The proposed chromosome structure can be applied to other systems. For a given hardware platform, it is needed to define the possible mapping options first and increase the number of genes for pipeline cut-points up to the maximum number of pipeline stages in the mapping options.

The number of possible cut-point combinations depends on the number of layers and the number of pipeline stages. Since exhaustive exploration is very time-consuming, we reduce the number of cut-points to explore by filtering out some cut points. We exclude the cut points between the convolutional layer and its activation because the convolutional layer and the next activation layer are fused in the TensorRT. If there are consecutive layers that cannot be mapped to DLA, cut-points between them are also excluded.

For each Pareto-optimal solution, we estimate the WCET of each task on each processing element with a measure-based method. For the CPU task (pre/post-processing task), we measure the execution time using the POSIX time library. As for DLA, the assigned stage is regarded as a single task, and its execution time is measured by augmenting the code segment at the start and the end of the DLA task. As explained above, a GPU task corresponds to a kernel in the associated pipeline stage. The execution time of a task (kernel) on GPU is measured by the TensorRT IProfiler. To estimate the WCET of a task from measurements, we find the bigger value between the measured biggest value and the value obtained by adding six times the standard deviation to the mean. The estimated WCET of each task is saved in a database and used for the schedulability check in the next step. In addition, the average power of processors and average-case execution time (ACET) for each mapping solution are also measured. This information is used

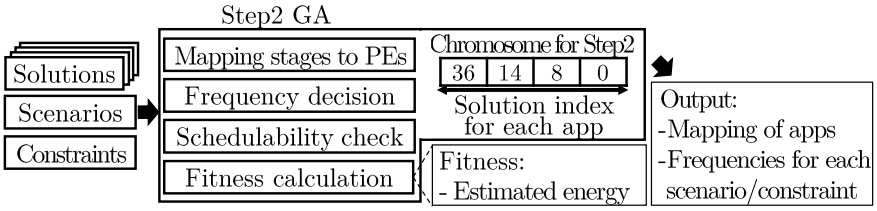


Figure 4.5: The overview of step 2

to calculate the fitness value in the next step. We use the maximum frequencies for all processors in this step.

4.4.2 Step 2: Exploring the Mapping Combination

After Pareto-optimal mapping solutions for each application are determined, we find a sub-optimal combination of individual mapping solutions of all applications and clock frequencies of processors for scenarios. To find a sub-optimal combination, a GA and two heuristics are devised, aiming to minimize the expected energy consumption.

The workflow of this step is described in Fig. 4.5, and the chromosome structure of the GA is shown in the center of the figure. Each gene represents a mapping solution index. In the example of Fig. 4.5, the mapping of the first application is the 36th mapping among all Pareto-optimal mappings of the corresponding application. Since we cannot change the mapping of an application dynamically due to the limitation of the SDK, the selected mapping defined by a gene is fixed in all scenarios.

How to compute the fitness value of a chromosome is depicted in Algorithm 4. First, it checks whether the number of tasks (stages) assigned to DLAs is more than four (line 2). If it exceeds four, the maximum fitness value is assigned to indicate that this chromosome cannot be a feasible solution. For a feasible solution, mapping stages to PEs is determined by a heuristic (line 5). Note that the mapping solution of an application does not say which DLA is used between two DLAs. Thus we devise a simple heuristic to determine which DLA will be used for each application.

Algorithm 4 Pseudo code for chromosome evaluation

```
procedure EVALUATECHROMOSOME(chromosome)
    Check the number of DLA mapped stages
    if not executable on DLAs then
        return MaxInt
    end if
    mappings = MapStageToPE(chromosome)
    totalFitness = 0
    for each scenario do
        schedulable, fitness = SetFreq(scenario,mappings)
    ▷ Schedulability check and fitness calculation are done in this function
        if not schedulable then
            return MaxInt
        end if
        totalFitness += fitness
    end for
    return totalFitness / the number of scenarios
end procedure
```

Next, the schedulability test for each scenario is performed on each processor. How to perform the schedulability test will be explained below in this section. Since the clock frequency affects the execution time of each task on a mapped processor, we select the frequencies of processors to minimize the energy consumption for each scenario while satisfying the schedulability constraint (line 8). Note that this exploration is done per scenario to cope with the different behavior of the scenarios. If no feasible schedule is found in any scenario, the maximum fitness value is immediately returned to mark the solution as infeasible. Otherwise, the average fitness value over all scenarios is returned.

4.4.2.1 Mapping Stages to PEs

The proposed heuristic for mapping stages to PEs is depicted in Algorithm 5. At first, it parses the mapping option for each selected mapping of application (line 3). Then, the neighbor application list is made for each application (line 4). The list of neighbor applications of the target application consists of the applications that will be executed simultaneously in some scenarios. We sort the applications in the decreasing order of the maximum length of the DLA stage (task). Mapping is done in a round-robin fashion using the mapping index on each processor (lines 5-16). If a stage is mapped to the CPU,

Algorithm 5 Pseudo code for mapping stages to PE

```
   Input : AppList: the list of applications
     ▷ In the decreasing order of the max length of DLA stage (task)
   Input : mappings: the table to save mapping info
 1: procedure MapStageToPE(chromosome)
 2:     CPUIdx, DLAIdx = 0, 0
 3:     Parse mapping option of app                          ▷ Based on chromosome
 4:     nearAppList = MakeNeighborAppList()
 5:     for each app in AppList do
 6:         if app is already mapped then
 7:             continue
 8:         end if
 9:         for each stage do
10:             if stage maps on CPU then
11:                 mappings[app][stage] = CPUIdx
12:                 CPUIdx = (CPUIdx + 1) % CPUCoreNum
13:             else if stage maps on GPU then
14:                 mappings[app][stage] = 0                 ▷ One GPU on the board
15:             else stage maps on DLA
16:                 mappings[app][stage] = DLAIdx
17:                 DLAIdx = (DLAIdx + 1) % DLANum
18:             end if
19:         end for
20:         for nearApp in nearAppList[App] do
21:             Repeat the line 6 to 16 for nearApp
22:         end for
23:     end for
24:     return mappings
25: end procedure
```

it is mapped to the indexed core of the CPU, and the index is incremented. In the case of GPU, the mapping index is always zero since there is only one GPU in the target board. If a stage is mapped to a DLA, two DLAs are alternatively mapped. Since the applications are sorted based on the DLA workload, the heuristic aims to balance the workloads of two DLAs available in the hardware platform. After the mapping for the current application is complete, we map the neighbor applications in the same fashion (lines 17-18).

4.4.2.2 Frequency Decision

After the mapping of stages is settled, we determine the frequency of each processor. Since the frequency affects the execution time, we perform the schedulability test for each frequency combination. Since the number of frequency combinations is too large to be

Algorithm 6 Pseudo code for frequencies decision

```
   Input : ProcList: the list of processors                    ▷ In the order of GPU, CPU, and DLA
 1: procedure SETFREQ(scen,maps)
 2:     set frequencies for all processors (freqs) as maximum
 3:     i, schedulable, minSchedulable, minFitness = 0, False, False, MaxInt
 4:     Save curr freqs as the best freqs
 5:     for two iterations do
 6:         for each processor in ProcList do
 7:             set curr processor frequency as maximum
 8:             while True do
 9:                 schedulable = checkScheduability(scen,maps,freqs)
10:                 if schedulable then
11:                     fitness = calculateFitness(scen,maps,freqs)
12:                     if fitness < minFitness then
13:                         save curr freqs as the best freqs
14:                         minFitness, minSchedulable = fitness, True
15:                     end if
16:                 else
17:                     if curr frequency is not the maximum then
18:                         Revert to the previous frequency
19:                     end if
20:                     break
21:                 end if
22:                 if curr frequency is the minimum frequency then
23:                     break
24:                 else
25:                     Set curr processor frequency one level down
26:                 end if
27:             end while
28:         end for
29:     end for
30:     return minSchedulable, minFitness
31: end procedure
```

explored exhaustively inside a GA, a heuristic is devised to explore the frequencies in a greedy fashion as depicted in Algorithm 6. The initial frequency combination is set by using the maximum frequency of each processor (line 2). Since the algorithm reduces frequencies in a greedy way, we examine the reduction amount of power consumption per each frequency change and determine the order based on the reduction amount. The order turns out to be GPU, CPU, and DLA in the process of fitness calculation covered in the next paragraph. We select the processor whose frequency is to be changed (line 6), starting from the maximum frequency (line 7), and check the schedulability with the current frequency combination (line 9). If it passes the schedulability test, we calculate

the fitness value that is the expected energy consumption and save it if it is the minimum fitness (lines 11-14). And we reduce the current processor's frequency one level down (line 22). If the schedulability test fails, we go back to the previous frequency (line 17). We repeat this process until the frequency becomes the lowest one (lines 19-20). Then, we select the next processor whose frequency is to be altered (line 6), starting from the last schedulable frequency combination found so far. In the first iteration, we find the lowest schedulable frequency in the order of GPU, CPU, and DLA in a greedy fashion. In the next iteration, it searches the frequency range between the highest frequency and the found frequency to check if the energy can be lowered with a higher frequency, again in a greedy fashion.

4.4.2.3 Fitness Calculation

Fitness is defined as the average energy consumption for all scenarios in a given period. We use the average value for all scenarios since we do not know which scenario to stay how long. It can be applied differently if we have the probability information. To estimate energy consumption, we first identify the relationship between the frequency and power consumption for each processor by the regression analysis based on the measurement. We consider both static and dynamic power consumption.

For the estimation of static power consumption, we change the frequencies over an available range as displayed in Table 4.7 [2]. We estimate the static power on a fixed frequency when no application runs, which is similar to the work of [72][3]. So we figure out the relation between frequency and static power by measuring the power after setting the minimum and maximum frequencies to be equal.

For dynamic power estimation, we measure the average power consumption of each application by varying the frequency of each processor. For the CPU and GPU, we use

[2]While the Xavier board allows a user to change the processor frequencies [71], the voltage is under the direct control of the board power management firmware. Hence, we tune the processor frequencies only.

[3]The static power in this paper means the power when no application runs in the P-state. So, the static power can be affected by the frequency.

the mapping option E. On the other hand, option A is used for DLAs with all mappable layers for DLA. When changing the frequency of one processor, we set the frequencies of the other processors to be maximum. Then, we obtain the relationship between the frequency and dynamic power by subtracting the estimated static power from the measured total power. Such regression analysis is conducted individually for four benchmark applications that are described in Table 4.4.

With the obtained relationship between the frequency and dynamic power, we estimate the average dynamic power consumption of the given mapping at a given frequency. The average power consumption of each processor on each mapping solution is measured in step 1 with the maximum frequencies as explained in Section 4.4.1. We apply the relationship to estimate the dynamic power consumption of processors at the given frequencies for a given mapping. The dynamic energy consumption is computed by multiplying the computed dynamic power consumption and the average execution time. The average execution time is estimated to be inversely proportional to the frequency from the profiled ACET. For static energy consumption, we multiply the estimated static power consumption and the period. The estimated energy consumption is the sum of the static and dynamic energy consumption.

4.4.2.4 Schedulability Analysis

For schedulability analysis, we estimate the worst-case response time (WCRT) for each chain-structured DL application by tailoring the compositional performance analysis [21]. If the sum of the WCRTs for each part of an application considering the jitter is no greater than the deadline constraint, the deadline constraint is satisfied. We can assign the distinct priority levels to the tasks mapped on the CPU, and the tasks are scheduled by preemptive scheduling with a fixed priority. However, the tasks mapped on GPU or DLA have the same priority.

The worst-case response time of a CPU task, $\tau_k^{i,j} \in s_j^i$ of app_i, is bounded by equa-

tion (4.1) which is well-known for fixed-priority preemptive scheduling [73, 21]. The WCRT of the task is the converged value of r^m. The $hp(\tau_k^{i,j})$ is a set of higher or equal priority tasks that are mapped to the same PE with $\tau_k^{i,j}$ and belong to the other applications. For simple notation, τ_h indicates a task in $hp(\tau_k^{i,j})$. J_{τ_h} is the jitter of the τ_h.

$$r^{m+1} = C_{\tau_k^{i,j}} + \sum_{\tau_h \in hp(\tau_k^{i,j})} \lceil \frac{r^m + J_{\tau_h}}{P(\tau_h)} \rceil \cdot C_{\tau_h} \qquad where\ r^0 = C_{\tau_k^{i,j}} \tag{4.1}$$

In the case of GPU, the kernels within a stream are executed in a FIFO order, and kernels are executed in a non-preemptive way [69, 70]. Since the priority of all GPU tasks is equal, the WCRT may be over-estimated if we consider worst-case interference from all tasks. Thanks to the chain structure of an deep learning application and the FIFO scheduling policy of GPU, we can reduce the pessimism of the analysis. The worst-case response time of stages mapped to the GPU can be calculated by the formula as shown in equation (4.2).

$$r = C_{s_j^i} + \sum_{s_e \in ep(s_j^i)} B_{s_j^i, s_e} \tag{4.2}$$

In this formula, $ep(s_j^i)$ means a set of stages that are mapped to the same PE with s_j^i of app_i and belong to the other applications. $B_{s_j^i, s_e}$ represents the maximum interference from stage s_e to s_j^i where s_e indicates a stage in $ep(s_j^i)$. It is the maximum sum of WCETs of as many successive tasks in s_e as $min(|s_j^i|, (\lceil \frac{P(s_j^i)}{P(s_e)} \rceil + 1) \cdot |s_e|)$, which is likely to be smaller than the entire WCET of stage s_e, C_{s_e}. Suppose that stage s_j^i consists of 7 kernels (tasks), and the interfering stage s_e consists of 3 kernels. Since kernels in a stage are queued sequentially, a kernel of an application can be interfered by at most one kernel of

the other application, thanks to the FIFO scheduling policy. Thus the maximum number of interfering kernels cannot be more than the number of kernels in s^i_j, $|s^i_j|$. In case the number of kernels in the interfering stage is smaller than $|s^i_j|$, we compute the maximum possible number of interfering kernels by multiplying the number of kernels in the stage and the number of interference, which is bounded by $\lceil \frac{P(s^i_j)}{P(s_e)} \rceil + 1$. If stage s_e that has three kernels can interrupt twice maximally, the six kernels may interfere the target stage s^i_j at most. Then, the minimum value between $|s^i_j|$ $(=7)$, and $2 \cdot |s_e|$ $(=6)$ becomes the number of kernels that may interfere.

Since the stage mapped to a DLA consists of a single task as explained in Section 4.3, the schedulability test for a DLA also can be performed with equation (4.2). The estimated worst-case response time of an application is the sum of the response time of the stages considering the jitter. Since we assume implicit deadline constraints, the estimated WCRT should be no greater than the period for each application.

4.4.2.5 Complexity Analysis

Since Algorithm 1 calls Algorithm 2 and Algorithm 3, we first investigate the complexity of Algorithm 2 and Algorithm 3. First of all, we assume that the schedulability analysis has a given time complexity, $T(\mathcal{PE}, \mathcal{A}_{sc_i})$, which is a function of processors and applications. Even though the response time analysis is known as NP-Hard [74], it is usually computed fast for the problem size assumed in this paper. In Algorithm 5, the amount of computation is proportional to the number of all stages because the mapping is determined in a round-robin manner so that its time complexity is $O(\Sigma_i |S_i|)$. Since it uses a space for *nearAppList*, the space complexity is $O(|\mathcal{A}| \cdot |\mathcal{A}| + \Sigma_i |S_i|)$.

Algorithm 3 has a nested loop of two levels. Since each loop runs at most the summation of the number of frequencies for each processor, the time complexity is $O(|freqs| \cdot T(\mathcal{PE}, \mathcal{A}_{sc_i}))$. The space complexity of Algorithm 3 is $O(|freq| + \Sigma_i |S_i| + |\mathcal{PE}|)$, including the space for frequencies and processors. Since Algorithm 1 calls Algorithms 2 and

3 for each scenario, Algorithm 1 has the time complexity of $O(|SC| \cdot (\Sigma_i |S_i| + |freqs| \cdot T(\mathcal{PE}, \mathcal{A}_{sc})))$ and the space complexity of $O(|freq| + \Sigma_i |S_i| + |\mathcal{PE}| + |\mathcal{A}| \cdot |\mathcal{A}|)$.

4.4.3 Step 3: Tuning Frequencies for Varying Deadline Constraints

While we consider the tightest deadline constraint in the previous step, the deadline constraint can be loosened at run time if the robot moves slowly. It means that there is a chance to use lower frequencies while satisfying the deadline constraint when the deadline constraint is loosened. In the last step of the proposed methodology, we explore the frequency combinations, varying the deadline constraints. We use the same heuristic as Algorithm 6 for each deadline constraint. Since running the heuristic at run time incurs non-negligible overhead, we sample the deadline constraints and construct a table that contains the frequency combination for the sample deadline constraint. For example, if the maximum deadline constraint is given as 30 FPS[4], we define a set of discrete deadline constraints and perform the heuristic at design time. At run time, we refer to the table to get the frequency combination of the closest tighter deadline constraint than the actually required deadline.

4.5 Experiments

4.5.1 Set-Up

The hardware platform used in the experiments is a Jetson AGX Xavier board with Jetpack 4.3 and TensorRT 6. We use the tkDNN [41] library and JEDI framework to deploy DL applications in a pipeline way on TensorRT easily. We use four deep learning networks described in Table 4.4 as benchmark applications that are specified with tkDNN and JEDI. Since the *leaky relu* activation is not supported on the DLA, we used the

[4]Since we assume the implicit deadline, the deadline constraint is inverse of the throughput constraint. The higher FPS indicates the tighter deadline constraint.

networks, provided in JEDI, that replace *leaky relu* with *relu*. The *COCO2017 val* is used for inference with an image size of 256x256 with FP16 precision, and the batch size is set to one. In the target board, the power consumption is measured by the *tegrastats* command.

For GA implementation, the DEAP [75] framework is used. In the first step of the proposed method, the maximum evolution count is set to $20,000$ and the population size to 65. In each generation, two chromosomes are created as offspring of each survivor through genetic operations such as uniform crossover, one-point mutation, and random selection. We execute the GA on the target board since it requires running the network to obtain the performance value by measurement. In the second step, we proceed up to $10,000$ generations with 128 chromosomes in the population. The same operations in the previous step are used except that Lexicase [76] is used for the selection operation. To check the maximum achievable deadline constraint, we find the mappings from the lower constraint, then use the found mapping as an initial solution for exploring the higher constraints. In this step, the GA is run on a host computer that consists of Ubuntu 18.04.6 LTS, AMD Ryzen 9 3950X 16-Core Processor with 64GB RAM.

4.5.2 Finding Pareto-optimal Mappings of Each Application

As mentioned in Section 4.4.1, we filter out the cut points to reduce the design space. Table 4.4 shows the number of layers and selected cut points for each network. The number of candidate cut points is reduced to fewer than a half of the total number of layers. The total number of cut-point combinations is calculated based on the mapping options of Table 4.3. For example, in the case of *Yolov2*, it is calculated as follows: $\binom{22}{1} + \binom{22}{2} \cdot 2 + \binom{22}{3} + \binom{22}{0} = 2025$. The last column shows how many Pareto-optimal solutions could be obtained by the proposed GA. The number of Pareto-optimal solutions is 0.2% - 8.6% of total candidates.

Table 4.4: The benchmark networks and the volume of design space for step 1 to find the Pareto-optimal mappings.

Network	Layer #	Selected cut-points #	Total cand. #	Pareto sol. #
Yolov2 [45]	54	22	2025	112
Yolov3 [46]	179	78	82161	158
Yolov2tiny [45]	24	10	221	19
Yolov3tiny [46]	35	10	221	14

Table 4.5: Information on three different system behaviors: which applications are performed in each state

(a) FSM-B

State	Applications ($\mathcal{A}_{sc}$)
sc_0	app_0
sc_1	app_3
sc_2	app_0, app_1
sc_3	app_1, app_3
sc_4	app_0, app_2
sc_5	app_2, app_3
sc_6	app_0, app_1, app_2
sc_7	app_1, app_2, app_3

(b) FSM-C

State	Applications ($\mathcal{A}_{sc}$)
sc_0	app_0, app_3
sc_1	app_0
sc_2	app_3
sc_3	app_0, app_1
sc_4	app_1, app_3
sc_5	app_0, app_2
sc_6	app_2, app_3
sc_7	app_1, app_2

(c) FSM-D

State	Applications ($\mathcal{A}_{sc}$)
sc_0	app_0
sc_1	app_0, app_1
sc_2	app_0, app_1, app_2
sc_3	$app_0, app_1, app_2, app_3$

4.5.3 Exploring Mapping Combination and Tuning Frequencies

In this experiment, we evaluate the second step of the proposed scheme with four different system behaviors that are represented by FSMs. In addition to our motivational example of Fig. 4.1 (b) denoted by *FSM-A*, Table 4.5 shows which applications are performed in each state for other three FSMs. Since the transition of states does not affect the result of the mapping, we omit the transition information between state.

FSM-B is an extended version of *FSM-A* with twice more scenarios. In *FSM-C*, the maximum number of applications running concurrently is limited to two. Meanwhile, the number of applications is increasing in *FSM-D*. The most important factor is the

maximum number of applications running concurrently since the schedulability test is conducted in this step. We also conduct the experiments with two different combinations of applications as denoted by cases α and β shown in Table 4.6. In the case of α, all applications from app_0 to app_3 are the same network, *Yolov2*. We assign the priority of tasks mapped to CPU in the following order: app_0, app_3, app_1, app_2. On the other hand, the tasks mapped onto GPU or DLA have an equal priority. Although all applications are assumed to have the same period in all experiments, the proposed method allows them to have distinct periods.

We compare the proposed method with the following three methods.

- Local(LC): It allows dynamic task migration across the scenarios. We find the sub-optimal mapping of applications for each scenario by a GA. However, it is not feasible in practice since dynamic task migration is too costly in the target platform. Nonetheless, this method serves as a decent indicator to check how close is the proposed method to the ideal case.

- Baseline(BS): It is a typical case of running each deep learning application on a single processing element, GPU. Since all applications have some layers that are not supported by DLA, it uses only GPU in the inference, which corresponds to option E in Table 4.3.

- Global(GB): It uses a GA to find a sub-optimal static mapping of all applications, assuming that all applications are running concurrently, ignoring the scenarios. It produces the same mapping for all system behaviors from *FSM-A* to *FSM-D*. The frequency for each processor is set to the maximum without applying step 3.

- Work of [12] (SOTA): It also explores static mapping with maximum frequencies ignoring scenarios similar to GB, but it uses the estimated layer-wise execution times. After obtaining the mapping result from the SOTA method, we recalculate the fitness and schedulability analysis with the actual execution times for fair com-

Table 4.6: Cases for a combination of applications in FSMs

Case	app_0	app_1	app_2	app_3
α	Yolov2	Yolov2	Yolov2	Yolov2
β	Yolov2	Yolov2tiny	Yolov3tiny	Yolov3

Table 4.7: The frequency range used in the exploration

Processor	Frequency range (MHz)
DLA	550.4, 640, 729.6, 806.4, 896, 972.8, 1062.4, 1152, 1228.8, 1395.2
GPU	675.75, 828.75, 905.25, 1032.75, 1198.5, 1236.75, 1338.75, 1377
CPU	1190.4, 1267.2, 1344, 1420.8, 1497.6, 1574.4, 1651.2, 1728, 1804.8, 1881.6, 2035.2, 2112, 2188.8, 2265.6

parison.

- Proposed(PR): It finds a static mapping combination tailored for each system behavior. It explores the design space with the objective to reduce the average energy consumption by considering scenarios and constraints.

In this experiment, we make three comparisons. Firstly, we compare the tightest deadline constraint that each method can support without schedulability violation. Also, the comparison is made in terms of the fitness value. Then we evaluate the effect of frequency tuning in terms of energy reduction. Lastly, we confirm the necessity of the proposed heuristics compared with the exhaustive search in terms of speed.

4.5.3.1 Comparison of the maximum achievable FPS

Table 4.8 shows the maximum FPS that each method can achieve. In all FSMs except for *FSM-D*, the PR method can satisfy the tighter deadline constraint (higher FPS) than both BS and GB methods. The PR could satisfy 15% $\sim$ 86% tighter constraint compared to the BS method, and up to 40% tighter constraint than the GB method. Since the static mapping is determined by the worst-case scenario in the PR method and *FSM-D* runs all applications in state *SC4*, PR and GB have the same FPS constraint for *FSM-D*.

Table 4.8: Comparison of the maximum achievable FPS among methods

Methods	Case α - FSMs				Case β - FSMs			
	A	B	C	D	A	B	C	D
LS	47	47	71	40	32	32	38	28
BS	32	32	44	24	21	21	33	15
GB	40				28			
SOTA [12]	39				26			
PR	47	47	56	40	32	32	38	28

Table 4.9: Mapping and frequency tuning result for FSM-A from the proposed method

(a) Mapping result for FSM-A when FPS constraints are 47 and 32 for cases α and β, respectively

Case	α	β
app_0	D[0-29],G[30-53]	D[0-29],D[30-40],G[41-53]
app_1	D[0-29],G[30-53]	D[0-16],G[17-23]
app_2	D[0-29],G[30-53]	D[0-16],G[17-34]
app_3	D[0-29],G[30-53]	G[0-178]

(b) Frequency (MHz) tuning example on SC3 of FSM-A for varying FPS constraints with the same static mapping as (a)

Processor	Case α			Case β		
	40 FPS	45 FPS	47 FPS	25 FPS	30 FPS	32 FPS
CPU	1574.4	2112.0	2265.6	1958.4	1651.2	1728.0
GPU	1032.75	1236.75	1338.75	1032.75	1338.75	1377.0

While comparison with BS confirms the advantage of utilizing heterogeneous processors, comparison with GB confirms the advantage of considering the scenarios in the mapping decision. The work of [12] shows slightly worse results than the GB method. When using estimated time, the mappings obtained by [12] could satisfy 50 FPS and 32 FPS for each case, respectively, but recalculation with the actual execution times shows different results. This is because the worst-case execution time could be underestimated when using the estimated time so that an infeasible solution can be falsely classified as feasible.

The proposed method shows a similar result as LC except for *FSM-C* of case α. In the LC method, DLA mapping can be changed for each scenario. Hence we can assign a different DLA to each application when two applications are running in all scenarios in *FSM-C*. On the other hand, two applications are mapped to the same DLA in some

scenarios in the PR method. In case β, the entire *Yolov3* is mapped GPU since its execution time is much larger than the other networks and GPU is faster than DLAs. Since GPU becomes the bottleneck in the schedulability check, the PR yields the same result to the ideal one, LC. Comparison with LC proves the effectiveness of the proposed static mapping technique.

Table 4.9 (a) shows a mapping decision example that the proposed method produces for *FSM-A*. In the table, *D[0-40], G[41-53]* means that pipelining option A is taken, layers #0 to #40 are mapped to DLA, and the rest are mapped to GPU. In case α, all applications use the same mapping result since they are all *Yolov2*. In case β, however, the selected options are different for applications: option C is used for *app0* and option E is used for *app3* while option A is used for the others. It indicates that we need to explore different mapping options for applications. Table 4.9 (b) describes how frequencies can be changed as the deadline constraint varies at run-time after mapping decision is made to achieve the maximum FPS as shown in Table 4.9 (a). The DLA frequency, which is not shown in the table, turns out to be maximum at all times. It means that reducing the GPU or the CPU frequency is better than reducing the DLA frequency in terms of average energy consumption. Since the GPU consumes more power than the CPU, the proposed heuristic tries to reduce the GPU frequency as much as possible while satisfying the schedulability constraint. Since the CPU performs pre-processing and post-processing tasks that have a linear dependency on the inference part, we may need to reduce the CPU execution time by increasing the CPU frequency under a lower FPS constraint in order to satisfy the constraint on the end-to-end worst-case response time. While the GPU frequency decreases as the FPS constraint is loosened, there is no such tendency in the CPU frequency. It explains why the CPU frequency increases as the FPS constraint decreases from 30 FPS to 25 FPS for case β in Table 4.9 (b).

To evaluate the effectiveness of the proposed GA-based heuristic for the second step of the proposed scheme, we find an optimal mapping of the GB method by exploring

Table 4.10: Throughput constraint (FPS) variation

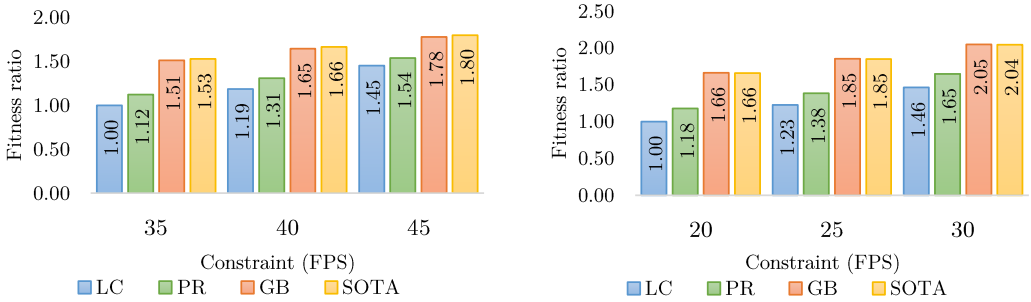
Case	FSM-A	FSM-B	FSM-C	FSM-D
α	35, 40, 45	35, 40, 45	40, 45, 50	30, 35, 40
β	20, 25, 30	20, 25, 30	25, 30, 35	15, 20, 25

exhaustively the design space that is vast (112^4 mappings for the case α). It is observed that the resultant mapping is the same as found by the proposed GA heuristic. Since only a few mappings can satisfy the deadline constraint, the proposed GA-based scheme could find the optimal solution in our experiments, we believe. The proposed GA-based heuristic explored much less design space. This experiment confirms the efficacy of the proposed GA-based exploration method as the second step of the proposed methodology.

4.5.3.2 Frequency tuning effect

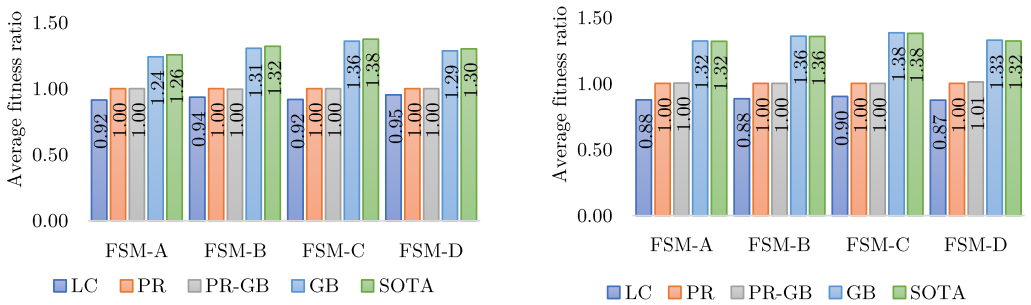
In this experiment, we examine the effect of frequency tuning as the deadline constraint varies at run time. For a given mapping, we visit each scenario to find the best frequency combination of processors in terms of estimated energy consumption. Note that we can obtain a higher FPS performance than that of Table 4.8 for the GB method in case there is no scenario that runs all applications. With the given mapping obtained from the GB method, we increase the throughput constraint until there is any scenario that cannot meet the constraint. The modified maximum achievable FPS from the GB method for each FSM turns out to be 45, 45, 51, and 40 FPS for case α, while it is the same as PR in Table 4.8 for case β. It is still lower than the maximum achievable FPS obtained from the PR method. We vary the deadline constraint as described in Table 4.10, where the tightest constraint is determined by the GB method.

Figure 4.6 shows how the fitness ratio varies as the deadline constraint varies for the *FSM-A* benchmark with cases α and β, respectively. The fitness value represents the estimated energy consumption during the same time duration. The fitness ratio is the relative fitness over the lowest value. The figure demonstrates the energy can be reduced



(a) case α (b) case β

Figure 4.6: Fitness ratio as the FPS constraint varies for FSM-A



(a) case α (b) case β

Figure 4.7: Average fitness ratio over the FPS constraint variation for all benchmarks

while satisfying the constraint by manipulating the frequencies. In addition, the difference between the PR and both GB and SOTA methods is large, while the gap between PR and LC methods is small. In this comparison, the GB method does not tune the frequency as the constraint varies.

To study the effectiveness of frequency tuning, we propose an alternative method which is a variant of the PR method. We apply the frequency tuning algorithm to the resultant mapping obtained by the GB method, calling this method as PR-GB. Figure 4.7 shows the average fitness ratio of methods over the FPS constraint variation. In this figure, the fitness ratio is the relative fitness over the fitness value of the PR method on each bundle. While the difference between methods PR and GB is noticeable, PR and PR-GB show almost no difference. Also, GB and SOTA methods give similar results. It means

Table 4.11: Average fitness evaluation time and end-to-end execution time during 50 generations in GA

Case		α		β	
Method		heuristic	brute-force	heuristic	brute-force
Average fitness evaluation time (sec)	FSM-A	1.41	46.13	2.28	61.00
	FSM-B	3.44	99.76	5.07	131.78
	FSM-C	3.08	87.30	3.98	118.78
	FSM-D	1.68	57.08	3.30	87.21
Average total time (sec)	FSM-A	317.66	13390.05	519.03	17405.57
	FSM-B	1119.19	35457.24	1223.27	34586.12
	FSM-C	1268.51	36095.17	726.65	23861.59
	FSM-D	439.59	17972.25	867.07	27643.66

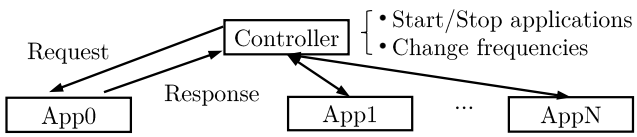
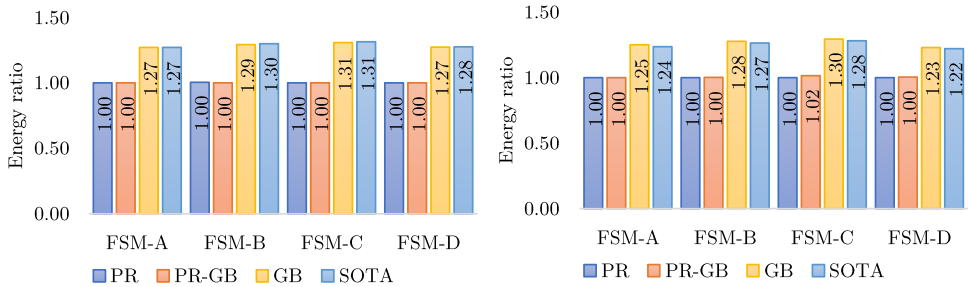


Figure 4.8: The run time management following scenarios

that the frequency adjustment is more important than the mapping decision for energy reduction. In addition, the difference between the two proposed methods (PR and PR-GB methods) and the ideal case (LC method) is smaller than the gap between the proposed methods and both GB and SOTA methods.

4.5.3.3 Execution time for fitness evaluation

Since frequency tuning is a key issue for energy reduction, we explore the frequency combination of processors in the fitness evaluation of each candidate mapping in step 2, as explained in Section 4.4.2.2. Table 4.11 shows the average execution time for fitness evaluation and the end-to-end time of the proposed method during 50 generations in the proposed GA-based heuristic, compared with a brute-force exhaustive method. The speedup by the heuristic is more than at least 26 times for fitness evaluation and 28 times for end-to-end execution than the brute-force method. It is confirmed by experiments that both schemes produce the same fitness value for the resultant mappings. It proves the effectiveness of the proposed frequency selection heuristic.



(a) case α (b) case β

Figure 4.9: Average energy ratio over the FPS constraint variation, measured after real deployment

4.5.4 Real Deployment

We apply the proposed methodology to run benchmark applications on the target board. A run-time controller is implemented to control the execution status of applications according to the scenario, as illustrated in Figure 4.8. It consists of multiple threads to communicate with the applications. In the experiment, a thread requests execution of the associated application to process each input periodically and suspends itself until the completion of the processing is informed. The communication overhead between the controller and applications is considered in the latency estimation at the design time by subtracting the overhead from the deadline at the design-time optimization. The communication overhead is set by the bigger value between the measured value and the value that the mean plus six times the standard deviation. The frequencies are managed by the controller for each scenario and each constraint. While we change the processor frequencies, we set the frequency of the other components to the maximum. The controller runs on a single dedicated core which is not used for DL applications. Note that the run-time management itself is not a contribution of this paper.

In this experiment, we run applications on the target board during the same duration, 10 seconds, varying the scenarios for each mapping and frequency tuning result for each deadline constraint. The energy consumption is calculated by multiplying the running

time and the average power consumption of the system that includes not only processors but also the other components. We compare the PR, PR-GB, GB, and SOTA methods, excluding the LC method that is infeasible on a real platform. We perform each experiment three times and get the average value. Figure 4.9 shows the measured average energy consumption over the FPS variation for two cases α and β. The energy ratio indicates the relative energy consumption over the lowest value on each bundle. In both cases, the gap between PR and PR-GB is not noticeable, as expected by our estimation result. The proposed methods could save the energy by $22\% \sim 31\%$, compared with the GB and SOTA methods. Albeit the estimated energy saving (Fig. 4.7) is slightly higher than the actual saving shown in Fig. 4.9, the difference is not significant. It confirms that the proposed method effectively reduces energy consumption in a real hardware platform.

Chapter 5

Supporting Deep Learning Applications in a Model-based Design Methodology

5.1 Overview

In this chapter, we decide mappings for both dataflow applications and deep learning applications. Since it is difficult to consider deep learning applications in traditional model-based embedded software design methodology, we extend the model-based design flow. First, we find Pareto-optimal mappings for each deep learning application, similar to the previous chapter. Next, we find mappings for dataflow applications and deep learning applications together by the heuristic and meta-heuristic.

The rest of the chapter is organized as follows. In Section 5.2, we review the related work. After presenting a system model to clarify the problem in Section 5.3, the proposed methodology is explained in Section 5.4. The experiment results are discussed in Section 5.5.

5.2 Related work

We review the previous studies in the following three main subjects related to the proposed methodology: mapping of multiple dataflow applications, mapping of deep learning applications, and integrating deep learning applications into the model-based

design.

5.2.1 Mapping of Multiple Dataflow Applications

Since the mapping and scheduling of a dataflow graph onto a multiprocessor system is an NP-hard problem [54], many approximate methods for finding the optimal solution have been proposed such as heuristics based on list scheduling [56, 57, 56] explored mappings by heuristics, meta-heuristics using a genetic algorithm (GA) [77]. While most of works focus on the mapping and scheduling of a single dataflow graph, only a few works deal with the mapping of multiple dataflow graphs onto multiple processing elements, to the best of our knowledge. In case there exist real-time constraints on dataflow applications, we need to check a mapping solution satisfies those constraints by schedulability analysis for throughput constraint or worst-case response time analysis for latency constraint, which is recognized as a very challenging problem in real-time community [78]. A simple solution to avoid this difficulty is to map each dataflow graph onto a disjoint set of processors, avoiding interference between applications on the same processor. Then, we can map each dataflow separately onto the assigned processors.

Schor et al. [63] proposed an evolutionary algorithm to find a mapping to minimize the maximum utilization. Kang et al. [58] proposed a two-step approach. In the first step, they find a set of Pareto-optimal parallel schedules of each individual dataflow graph using a multi-objective evolutionary algorithm. In the second step, another evolutionary algorithm is to used to find the best combination of Pareto-optimal solutions of all dataflow graphs, aiming to minimize resource usage. For each mapping candidate, worst-case response time analysis is performed to check if the deadline constraint is satisfied for each dataflow graph.

5.2.2 Mapping of Multiple Deep Learning Applications

As deep learning applications are getting popular in embedded systems, extensive studies have been conducted recently to find the optimal mapping of DL applications specified by the associated SDK on a heterogeneous hardware platform. Xiang et el. [10] partition deep neural networks (DNNs) into pipeline stages, and map stages to processing elements by a heuristic. They focus on increasing the schedulability of multiple DNNs by balancing the utilization among PEs. While early works, including [10], considered CPU-GPU heterogeneous systems as the target hardware platform, recent works consider the DL hardware accelerators, also called NPU (neural processing unit) [9, 12]. Pujol et al. [9] consider an entire network as the mapping unit to a single PE without partitioning to leverage the associated SDK with each PE, assuming that the number of DL applications is more than the processing elements. Kang et al. [12] explore the per-layer mapping of DNNs with a GA, assuming that the layer-wise profiling information is given for each DL application. Even though they considered NPUs in the mapping step, no experiment with the NPU on a real hardware platform was made. Note that all works [9, 10, 12] consider the mapping of multiple DL applications only without considering other tasks running concurrently.

5.2.3 Integrating Deep Learning Applications into the Model-based Design

Since supporting DL applications in the model-based design methodology is a recent demand, there exist only a few previous studies that tackle this problem in the model-based design methodology. The work of [17] introduces an extended SDF model, called the SDF/L model, that specifies two types of loop structures explicitly and shows how to specify a DL application with the SDF/L model. They leave it as future work on how to perform task-mapping of the SDF/L graph, exploiting the data-level parallelism. Minakova et al. [11] transform a CNN (convolutional neural network) to an SDF graph and

find the mapping of SDF by using the genetic algorithm (GA). After the mapping decision is made, they translate the CNN network to a CSDF graph that is partitioned into sub-graphs that are run on each processing element. In this work, a CNN network has to be translated into two different models, the SDF model for mapping, and the CSDF model for code generation and execution. Such translation requires considerable effort. Since the translated SDF model has a wide range of sample rates, finding an optimal mapping onto multiple processors itself is a challenging problem. Both works [17, 11] do not consider the mapping exploration of multiple applications.

Recently, Jeong et al. [79] proposed a methodology based on the genetic algorithm to explore the mappings for multiple dataflow graphs on CPU-GPU heterogeneous system with a deep learning application. In their work, they assume that the mapping for the deep learning application is fixed and find the mapping of dataflow graphs, aiming to reduce the worst-case response time for each dataflow application. To the best of our knowledge, the proposed method is the first approach that supports multiple deep learning applications running concurrently with other dataflow applications in a model-based design methodology on a heterogeneous hardware platform, including NPUs.

5.3 System Model

5.3.1 Motivational Example

Figure 5.1 displays a motivational example where DL applications and dataflow applications are running together. The *Image processing* application is represented by a task graph that consists of eight tasks that process images. After processing the image, the *Det* application gets the output data from the *Image processing* application and performs object detection. The object detection network consists of many layers for inference in addition to pre-/post-processing tasks. The example has four sets of such combinations of a dataflow application and a deep learning (DL) application as shown in Figure 5.1. In

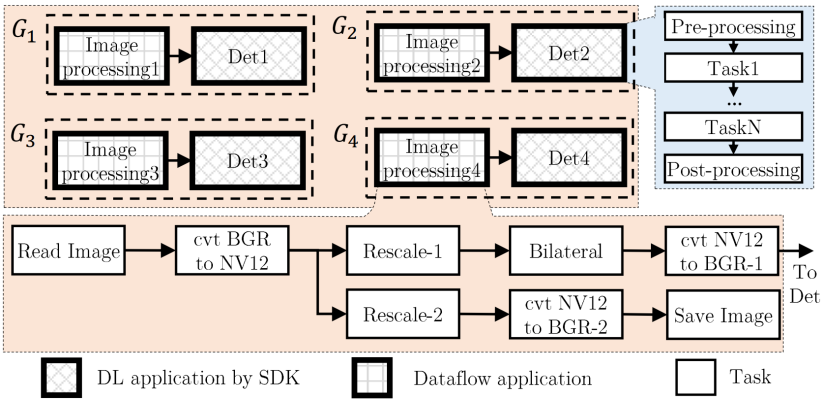


Figure 5.1: A motivational example

this research, we target the case where both deep learning applications and the dataflow applications run simultaneously, like the motivational example.

5.3.2 Notation and Problem Definition

Architecture specification: The target hardware platform consists of heterogeneous processing elements (PEs), $\mathcal{PE}$. In this work, three processor types are used in the Xavier board: CPU, GPU, and DLA. And a set of PEs for each processor type is represented as PE_{cpu}, PE_{gpu}, and PE_{dla}, respectively.

Application specification: Two types of applications are distinguished. DL applications are specified by a DL SDK, TensorRT in this work, while other applications are specified by a dataflow graph. We denote a given set of applications in each type as $\mathcal{D}$ and $\mathcal{A}$, respectively. As shown in Figure 5.1, an edge between two applications denotes the data dependency between them. The connected applications form an application group, denoted by G_i. Group G_i is defined by a tuple $\langle A^i, D^i, E^i, p^i, pr^i \rangle$. $A^i_j \in \mathcal{A}$ and $D^i_k \in \mathcal{D}$ are applications that belong to group G_i. The last three elements indicate edges between applications, the invocation period, and the priority, respectively. There may exist multiple groups, and the set of groups is denoted as $\mathcal{G}$. A dataflow application A^i_j is characterized by a tuple $\langle V^i_j, E^i_j \rangle$, where V^i_j and E^i_j represent the set of tasks and the set of edges be-

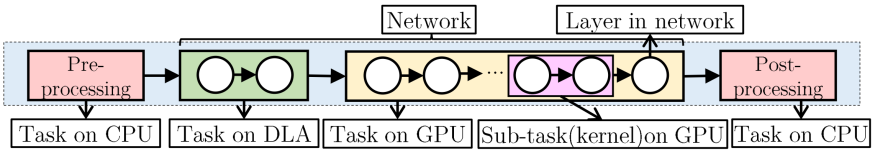


Figure 5.2: Task/Sub-task definition on the deep learning application specified by SDK

tween tasks, respectively. For the dataflow application, A^i_j, task $\tau^{i,j}_m$ is naturally defined by the model. For instance, in Figure 5.1, the *Image processing* application consists of eight tasks inside.

For a DL application D^i_k, however, it is characterized by a tuple $\langle V^i_k, E^i_k \rangle$ only after pipelining of the application is made, where V^i_k is a set of tasks and E^i_k is a set of edges between tasks. How to pipeline a DL application is explained in the next section. After pipelining decision is made, the DL application consists of multiple pipeline stages, each of which is mapped to a PE. Figure 5.2 shows an example that has four pipeline stages colored differently. Each pipeline stage is defined as a task for DL application, $\tau^{i,k}_n \in V^i_k$. For the task mapped to the GPU colored yellow, the SDK forms a set of kernels after optimization. The purple box in the yellow GPU-mapped task represents a kernel that merges two layers after layer fusion. Since a kernel is a unit of profiling, we define each kernel as a sub-task in the GPU.

Since per-kernel profiling is not possible on a DLA, however, the entire set of layers becomes a single task on a DLA. A DL application has a pre-processing task that feeds input data to the inference body and a post-processing task that processes the output data. Those tasks should be mapped to the CPU core. Note that a pipelined DL application has a chain structure of tasks, which is also assumed in [10] since there is a dependency between pipeline stages and the execution order of kernels is set by the SDK [19]. In summary, a DL application consists of chain-structured tasks ($\tau^{i,k}_n \in V^i_k$).

Scheduling specification: We denote the worst-case execution time (WCET) of task $\tau^{i,x}_y$ as $C(\tau^{i,x}_y)$. The invocation period of task $\tau^{i,x}_y$ is denoted as $P(\tau^{i,x}_y)$, which is the

same as p^i, the period of graph G_i. We assume that the graphs run periodically with the implicit deadline assumption that the period becomes the relative deadline. Thus, the deadline of group G_i is its period, p^i. The average latency of a group G_i is represented as $L(G_i)$, and the execution time on each processing element $pe \in \mathcal{PE}$ is denoted by $ET(pe, G_i)$. Similarly, the latency of a DL application D_k^i is denoted by $L(D_k^i)$. Meanwhile, $R(G_i)$ indicates the worst-case response time (WCRT) of a group G_i, and it is necessary to check whether the WCRT violates the deadline. The calculation of the WCRT is explained in Section 5.4.2.3.

Even though a group is assigned a priority, pr^i, it is applicable only for the tasks mapped on the CPU. Since there is no way to set the priority of a task on a DLA, tasks mapped on a DLA are executed in the FIFO order. Even though there are two priority levels in GPU, they are usually not used. Similar to DLAs, the GPU executes the mapped tasks in the FIFO order.

We denote the set of mapped tasks on processing element $pe \in \mathcal{PE}$ as $Map(pe)$. Based on the mapped tasks on pe, we calculate the utilization of pe, $U(pe)$, as $\sum_{\tau_y^{i,x} \in Map(pe)} (C(\tau_y^{i,x}))/p^i$. In the proposed methodology, we choose Pareto-optimal mapping candidates, explained in the next section. The set of mapping candidates of D_k^i is represented by $Cand(D_k^i)$ where $D_k^i \in \mathcal{D}$, and the x-th candidate is denoted by $Cand(D_k^i, x)$.

Table 5.1 summarized the notations in this work.

5.4 Proposed Methodology

Figure 5.3 (a) shows the traditional embedded software design flow based on the dataflow model, which is explained in Section 2.5. To support a deep learning applications in the model-based design, we propose the extensions which is described in Figure 5.3 (b). In step 1, a set of Pareto-optimal mapping candidates for each DL application is obtained independently and applied to the extended model-based design framework as additional input information. In step 2, we try to find an optimal mapping of tasks by a

Table 5.1: Notations used in a system model

Notation	Description
pe and $\mathcal{PE}$	A PE and a set of PEs ($pe \in \mathcal{PE}$)
PE_{proc}	A set of PEs for processor type $proc$ ($G_i \in \mathcal{G}$)
G_i and $\mathcal{G}$	A group and a set of groups ($G_i \in \mathcal{G}$)
A^i_j	A dataflow application in G_i
D^i_k	A DL application in G_i
V^i_x and E^i_x	A set of tasks and a set of edges in A^i_x or D^i_x
$\tau^{i,x}_y$	A task in A^i_x or D^i_x
$C(\tau^{i,x}_y)$	The WCET of $\tau^{i,x}_y$
p^i and pr^i	A period and priority of G_i
$L(D^i_k)$	The average latency of D^i_k
$ET(pe, G_i)$	The execution time on pe when running G_i
$R(G_i)$	The WCRT of G_i
$Map(pe)$	A set of mapped tasks on pe
$U(pe)$	The utilization of pe
$Cand(D^i_k)$	The mapping candidates of D^i_k

meta-heuristic with a given set of objectives. For each DL application, we select a mapping candidate while we decide on the mapping of dataflow tasks simultaneously in this step. The last step is to generate the target code for each processing element, which will not be discussed in this work.

5.4.1 Step 1: Finding the Pareto-optimal Mapping Solutions of Each Deep Learning Application

The problem addressed in this step is summarized as follows.

- **Input:** All deep learning applications in $\mathcal{D}$ are provided.
- **Objective:** We define multiple objectives to minimize the latency of each deep learning application $D^i_k \in \mathcal{D}$ and the execution time on GPU and DLA, which can be described in Equation 5.1.

$$\begin{aligned} &Minimize : (L(D^i_k), ET(D^i_k, pe)) \\ &where\, D^i_k \in \mathcal{D},\, pe \in PE_{gpu}\, and\, PE_{dla} \end{aligned} \tag{5.1}$$

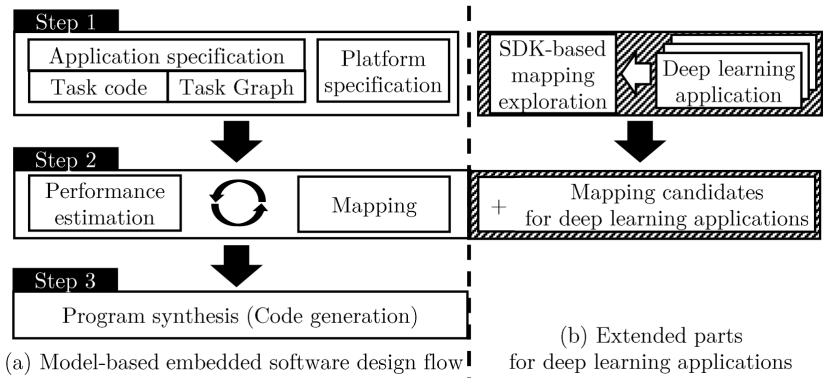


Figure 5.3: Overall flow of the model-based embedded software design and the proposed extension

- **Problem:** For each deep learning application $D_k^i \in \mathcal{D}$, find all Pareto-optimal mapping candidates. i.e. mapping of D_k^i: $D_k^i \rightarrow Cand(D_k^i)$.

- **Output:** The set of Pareto-optimal mappings of each deep learning application, $Cand(D_k^i)$. A mapping candidate indicates how a DL application is partitioned into pipeline stages and to which PEs the pipeline stages are mapped.

To solve this problem, the genetic algorithm is used in our implementation of the proposed methodology, as there are publicly available GA optimizers that are well-maintained. Other meta-heuristics that support multiple objectives can also be used.

Similar to the work in the previous chapter, we organize the mapping options as shown in Table 5.2 when partitioning the object detection application, *Det*, in the motivational example. For example, option *C* indicates the layers are partitioned into three stages. The pre-/post-processing tasks are mapped to the CPU while the inference body

Table 5.2: Mapping options for pipelining of a DL application

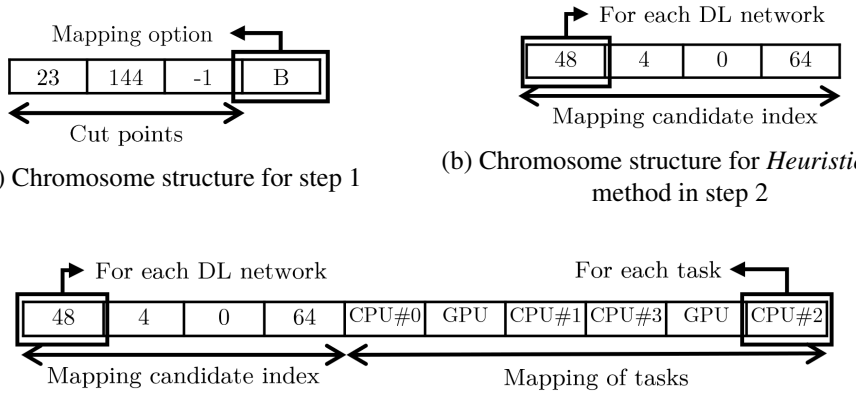
Options	# of stages	Composition of PEs
A	2	DLA0 - GPU
B	3	GPU - DLA0 - GPU
C	3	DLA0 - DLA1 - GPU
D	4	GPU - DLA0 - DLA1 - GPU
E	1	GPU

is partitioned into three pipeline stages that are mapped to two DLAs and GPU, respectively. Note that DLA0 and DLA1 are interchangeable without difference in terms of performance. The last pipeline stage is mapped onto the GPU in all mapping options since the last stage contains some layers that cannot be executed on a DLA. And no stage is mapped to the CPU. Those restrictions imposed by TensorRT need to be considered in the organization of the mapping options. It means that a set of mapping options may vary depending on the DL applications and the hardware platform.

After mapping options are identified, mapping candidates of the DL application are represented with the chromosome structure described in Figure 5.4 (a). The length of the chromosome is decided by the maximum number of pipeline stages. Since the maximum number of stages is four in Table 5.2, we set three genes for pipelining cut-points and one gene for the mapping option. In the example of Figure 5.4 (a), layers #0 $\sim$ #23 and layers #24 $\sim$ #144 are mapped to the GPU and DLA0, respectively, since the mapping option is option *B*. And the rest of the layers are also mapped to GPU. If there is no corresponding cut-point, the gene has a value of -1.

We define multiple objectives for GA fitness evaluation as described in Equation (5.1): end-to-end latency, the execution time on GPU, and the execution time on DLA and find Pareto-optimal solutions as mapping candidates for each deep learning application. We use the measured values as fitness values while running the network on a real board for each mapping candidate. Analytical performance estimation is not feasible because the layer-wise execution time could not be obtained for DLAs and the effect of optimization techniques of TensorRT on the performance cannot be estimated.

Suppose that a mapping is selected for a DL application, *Det*. Then the DL application can be transformed into a chain-structured task graph as shown in Figure 5.2. The dependency between the *Image processing* application and *Det* remains by grafting task *Pre-processing* to task *cvtBGRtoNV12-1*. Such graph transformation is necessary to determine the mapping dataflow tasks and check the schedulability in the next step.



(a) Chromosome structure for step 1

(b) Chromosome structure for *Heuristic+GA* method in step 2

(c) Chromosome structure for *Entire-GA* method in step 2

Figure 5.4: Chromosome structures: (a) for step1, (b) for *Heuristic+GA* method in step2, and (c) for *Entire-GA* method in step2.

5.4.2 Step 2: Mapping Exploration

In this step, we determine the mapping of each DL application among Pareto-optimal mapping candidates and the mapping of dataflow tasks onto PEs. The problem is summarized as follows.

- **Input:** All dataflow applications in $\mathcal{A}$ and the set of Pareto-optimal mappings candidates for each deep learning application, $Cand(D_k^i)$ where $D_k^i \in \mathcal{D}$.
- **Constraint:** The WCRT of group G_i should be less than or equal to its deadline, period. i.e. $R(G_i) \leq p^i \; where \; \forall G_i \in \mathcal{G}$.
- **Objective:** We aim to minimize the maximum utilization to balance the utilization of PEs, $Max_{pe \in \mathcal{PE}}(Util(pe))$.

$$Minimize : Max_{pe \in \mathcal{PE}}(U(pe)) \tag{5.2}$$

- **Problem:** Find the mapping of tasks $\tau_m^{i,j}$ to PEs, or $\tau_m^{i,j} \rightarrow \mathcal{PE}$, in each dataflow application A_j^i and select the mapping candidate of each deep learning applications.

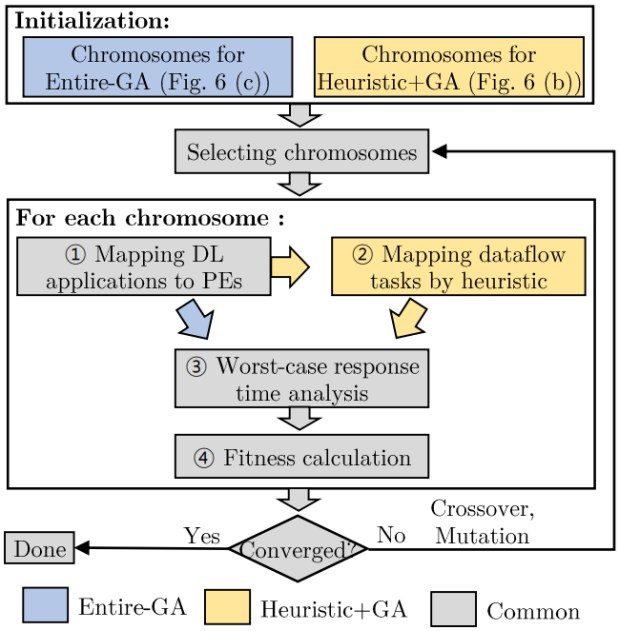


Figure 5.5: Procedure of the proposed mapping exploration technique

For the selected mapping candidate, we decide the mapping to PEs of each selected mapping candidate: $Cand(D_k^i, x) \rightarrow \mathcal{PE}$.

- **Output:** The mappings of dataflow applications and deep learning applications.

To solve this problem, we propose to use a GA algorithm. Since the execution time of GA increases as the problem size grows, two methods are devised. In the first method, denoted *Heuristic+GA*, we use a heuristic to determine the mapping of dataflow applications, while the GA decides the mapping of DL applications. A solution candidate is represented by a chromosome whose structure is depicted in Figure 5.4 (b). Each gene indicates the *index* of mapping candidates for each DL application. For example, 4 indicates the 4-th mapping candidate belonging to the second application. It contains as many genes as the number of DL applications. In the second method, denoted *Entire-GA*, we use the GA to select the mapping of dataflow applications simultaneously with the selection of the mapping of DL applications. The corresponding chromosome structure is shown in Figure 5.4 (c). The chromosome is divided into two parts. The left part indi-

cates the index of mapping candidates for each DL application, same as the first method. The right part indicates the mapping of dataflow tasks. For example, in the figure, the gene of *GPU* means that the corresponding tasks are mapped to the GPU while others are mapped to the specific core of the CPU.

Figure 5.5 shows the procedure of the proposed mapping exploration technique that consists of four main steps in the evolving process. In the initialization phase, we generate initial chromosomes randomly. Before entering into the evolving process, we first check if the combination of mapping candidates is executable on the target platform. As explained in Section 5.3, the number of DLA-mapped tasks is limited by the SDK[1]. If it violates the constraint, then the chromosome is discarded by setting the fitness value to the maximum value.

In the first step of the iteration, we perform mapping of DL applications to PEs. In this step, we determine which DLA and which CPU core is used for each DL application. After the mapping of DL application is determined, we use a heuristic to determine the mapping of dataflow applications in the *Heuristic+GA* method. This step is skipped in the *Entire-GA* method since the chromosome includes the mapping information of dataflow applications.

After all mappings are decided, we check the schedulability of applications through the worst-case response time analysis and compute the fitness value to select the dominant species in the evolutionary process. With the selected dominant solutions, we perform GA operations such as crossover and mutation, to define the next generation chromosomes. Such an evolutionary process is repeated until no better solution, or chromosome, is found during a given number of iterations.

Since the *Entire-GA* method explores a wider design space than the *Heuristic+GA* method, it is likely to find a better solution, taking much longer time. We improve the convergence speed of the *Entire-GA* method by using the mapping solutions found by

[1]In the version we used, the number of DLA-mapped tasks is limited to four.

the *Heuristic+GA* method as initial chromosomes. Using good initial chromosomes is useful for better exploration [80]. If no mapping is found by the *Heuristic+GA* method due to tight real-time constraint, we loosen the constraint to find solutions to obtain initial chromosomes for the *Entire-GA* method.

5.4.2.1 Mapping DL Applications to PEs

The mapping option in Table 5.2 indicates the processor type, not a specific processing element. For instance, we may use any DLA between two DLAs in the system when mapping option A or B is taken. Similarly, the pre-/post-processing tasks of a DL application can be mapped to any core in a multi-core CPU. To evaluate the fitness value of a chromosome, we need to determine which PE to use for each DL application. Since our objective is to balance the processor utilization, we use a simple scheme as follows: First, we sort the mapped tasks on each processor in the decreasing order of profiled execution time. Next, we perform a greedy mapping of tasks to PEs in a round-robin fashion starting from the longest task. For example, if there are three tasks mapped onto DLA, they are mapped to DLA0, DLA1, and DLA0 in the order of the execution length if there are two DLAs.

5.4.2.2 Mapping Dataflow Tasks by a Simple Heuristic

Algorithm 7 displays the pseudo-code of the proposed mapping heuristic for dataflow tasks in the *Heuristic+GA* scheme. This heuristic is called for each chromosome that indicates a candidate mapping combination of all DL applications. In other words, we determine the mapping of dataflow tasks after the mapping of pipeline stages of all DL applications onto processing elements is completed. Since mapping is performed for each chromosome during the evolution process, we use a simple greedy heuristic, sacrificing performance for faster execution speed. We first compute the PE utilization based on the mapping result of DL applications (line 2). Next, we sort the dataflow tasks in

Algorithm 7 Pseudo code of mapping heuristic for dataflow tasks

```
1:  procedure MAPMODELTASKS(chromosome)
2:      utils = calculateCurrentUtil()
3:      for each task in ordered task list do
4:          minValue = MaxInt
5:          for each mappable processor proc do
6:              minPEUtil, core = getMinUtil(proc, utils)
7:              value = getUtilIfMappedTo(task, proc, core)
8:              if value < minValue then
9:                  minValue, minProc, minCore = value, proc, core
10:             end if
11:         end for
12:         mappingInfo[task] = (minProc, minCore)
13:         updateUtil(minValue, minProc, minCore)
14:     end for
15: end procedure
```

the decreasing order of the maximum WCET value among the mappable processors and determine the mapping in the sorted order (line 3). We find the PE with the minimum utilization for each mappable processor (lines 5-6). And we compute the maximum utilization among all PEs in the selected processor when the current task is mapped to the corresponding PE (*getUtilIfMappedTo* function in line 7). To minimize the maximum utilization, we select the PE with the minimum value and map the task to the PE (lines 8-10). Afterwards, the PE utilization is updated according to the mapping (line 11).

The time complexity of Algorithm 7 is $O(|V_j| \cdot |\mathcal{PE}|)$ where $|V_j|$ indicates the number of tasks for all dataflow applications and $|\mathcal{PE}|$ is the number of PEs. This is because it checks the utilization of all PEs to find the PE with the lowest utilization for each task to map. The space complexity is $O(|V_j| + |\mathcal{PE}|)$ since the space to store the mapping information of tasks and PE utilizations depends on the number of tasks and PEs.

5.4.2.3 Worst-case Response Time Analysis

For each group, the worst-case response time (WCRT) analysis is conducted and the schedulability is checked by comparing the deadline constraint and the estimated WCRT. We use a compositional performance analysis (CPA) to estimate the WCRT for each group [21, 22]. In the CPA, the WCRT analysis is performed for each PE separately and

the dependency between tasks mapped to different PEs is modeled as an event stream that is specified by a tuple (period, jitter, the minimum distance between two events). Starting from the PE that the source task in a group is mapped to, WCRT analysis is performed one PE at a time up to the PE where the last task in the group is mapped to, following the task dependency. In case there exist multiple dependency paths in a group, we choose the maximum WCRT of all paths as the WCRT of the group.

Since the scheduling policy of processors is not identical, we apply a different WCRT analysis method for each processor type. For CPU that uses a fixed-priority preemptive scheduling [73] scheme, we use a well-known response time analysis formulated as follows:

$$r^{m+1} = C(\tau_y^{i,x}) + \sum_{\tau_h \in hp(\tau_y^{i,x})} \lceil \frac{r^m + J_{\tau_h}}{P(\tau_h)} \rceil \cdot C(\tau_h) \qquad \text{where } r^0 = C_{\tau_y^{i,x}} \tag{5.3}$$

Equation (5.3) estimates the WCRT of a task $\tau_y^{i,x}$ which is mapped to the CPU. Task set $hp(\tau_y^{i,x})$ is a set of higher or equal priority tasks that are mapped to the same PE with task $\tau_y^{i,x}$. J_{τ_h} is the jitter of task τ_h. The estimated WCRT of CPU task $\tau_y^{i,x}$ becomes the converged value of r^m.

In the GPU, the mapped tasks are executed in a FIFO order without preemption [69]. The WCRT of task $\tau_y^{i,x}$ mapped to the GPU can be computed by the following equation.

$$r = C(\tau_y^{i,x}) + \sum_{\tau_e \in ep(\tau_y^{i,x})} B_{\tau_e}^{\tau_y^{i,x}} \tag{5.4}$$

Task set $ep(\tau_y^{i,x})$ indicates a set of tasks that are mapped to the same PE with $\tau_y^{i,x}$. $B_{\tau_e}^{\tau_y^{i,x}}$ represents the maximum interference from task $\tau_e \in ep(\tau_y^{i,x})$ to the target task $\tau_y^{i,x}$.

If τ_e is a dataflow task, the number of interference is bounded by $\lceil \frac{P(\tau_y^{i,x})}{P(\tau_e)} \rceil + 1$. If it is a pipeline stage of another DL application, we need to consider the number of sub-tasks in the task. Suppose task $\tau_y^{i,x}$ and τ_e are both pipeline stages that have five and two sub-tasks, respectively. Sub-tasks in $\tau_y^{i,x}$ can be interfered at most five times since sub-tasks are scheduled in the FIFO order. On the other hand, task τ_e can interfere with task $\tau_y^{i,x}$ at most $\lceil \frac{P(\tau_y^{i,x})}{P(\tau_e)} \rceil + 1$ times. If the task τ_e can interfere with at most twice, four sub-tasks ($min(5, 2 \cdot 2)$) in τ_e may interfere with task $\tau_y^{i,x}$.

Equation (5.4) is also applied to DLA since the same non-preemptive scheduling policy is used as the GPU. We check whether the estimated WCRT violates the given deadline constraint. If the estimated WCRT is bigger than the deadline, the mapping does not satisfy the schedulability constraint.

5.4.2.4 Fitness Calculation

As the objective function of GA in the second step, we aim to minimize the maximum PE utilization in order to balance the utilization of PEs, which is also taken in the work of [63]. With this objective in mind, we define two types of fitness values as described in Equation (5.5). One is the maximum utilization of each processing element ($Max_{pe \in \mathcal{PE}}(U(pe))$) and the other is the maximum WCRT value of each group ($Max_{G_i \in \mathcal{G}}(R(G_i))$. Since the implicit deadline constraint should be satisfied for each group, by setting the latter fitness value, dominant solutions at each generation are more likely to satisfy the deadline constraint. If the estimated WCRT does not satisfy the deadline constraint, the solution is not feasible. Based on the fitness values of each candidate solution, we select the solution with the minimum value of the maximum utilization.

$$Minimize : (\, Max_{pe \in \mathcal{PE}}(U(pe)), Max_{G_i \in \mathcal{G}}(R(G_i))\,) \tag{5.5}$$

Table 5.3: Benchmark networks and the number of mapping candidates obtained by step 1

Subgraph	Network	# of layers	# of selected candidates
*Det*1	Yolov4 [47]	269	51
*Det*2	Yolov2tiny [45]	24	14
*Det*3	Yolvo3tiny [46]	35	12
*Det*4	Yolov4csp [48]	290	65

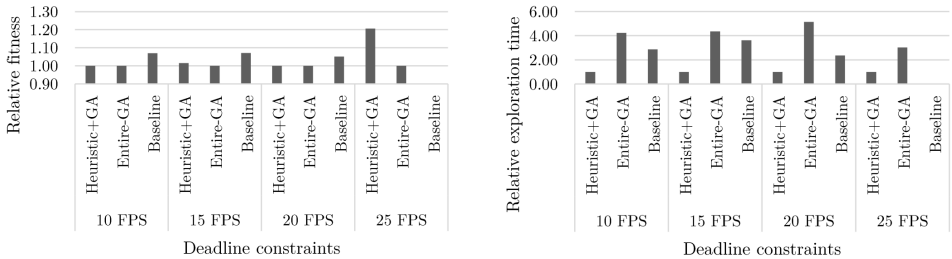
Table 5.4: Mappable processors of tasks in dataflow applications

Tasks	Processor	Tasks	Processor
ReadImage	CPU	cvt_NV12_BGR-1	CPU,GPU
cvt_BGR_NV12	CPU,GPU	Rescale-2	GPU
Rescale-1	GPU	cvt_NV12_BGR-2	CPU,GPU
Bilateral	CPU,GPU	SaveImage	CPU

5.5 Experiments

5.5.1 Comparison with a Previous Work

As a preliminary experiment, we evaluate the approach taken by previous works, which is to translate a DL application to a dataflow model. The work of [18] provides an example in which a Resnet152 network [81] is specified by an extended SDF model, called SDF/L. We implement the same network using TensorRT and compare both implementations on the same target board. The pre-/post-processing parts are mapped to the CPU, and other parts are mapped to GPU in both methods. It is observed that the previous approach could achieve only 18 FPS (frame per second) performance while the version of TensorRT achieves 81 FPS performance. It confirms that transforming a DL application to a dataflow may suffer from significant performance loss if the same degree of optimizations is not applied as TensorRT. In addition, model conversion takes a huge amount of effort.



(a) Fitness

(b) Exploration time

Figure 5.6: Comparison of three methods for the motivational example: Heuristic+GA, Entire-GA, and Baseline

5.5.2 Set-up

Since there is no previous work that tackles the same mapping problem, we devise a GA-based scheme by merging two recent previous works, [79] and the work of the previous chapter, and take it as a baseline technique to compare, denoted by *Baseline* in the experimental results. The former work is used to map multiple DL applications on the target board first, and the latter is used to map dataflow tasks after DL application mapping is completed. In contrast, two proposed methods, *Heuristic+GA* and *Entire-GA* perform the mapping of DL applications and dataflow applications simultaneously.

We implement the GA meta-heuristic with the DEAP library [75]. The GA runs on a host computer consisting of AMD Ryzen 9 3950X Processor. Experiments are made with the aforementioned motivational example in Figure 5.1 and randomly generated graphs. We conduct each experiment three times and get the average value to measure the mapping exploration time and the best fitness value. The target system is the Xavier board with Jetpack 4.6 and TensorRT 8.0.1. We reduce the available number of CPU cores to four to observe the effect of resource contention between tasks on the CPU.

5.5.3 Experimental Results: Motivational Example

We profile the tasks with TensorRT IProfiler and POSIX time library. We set the WCET of the task as the value obtained by adding six times the standard deviation to the

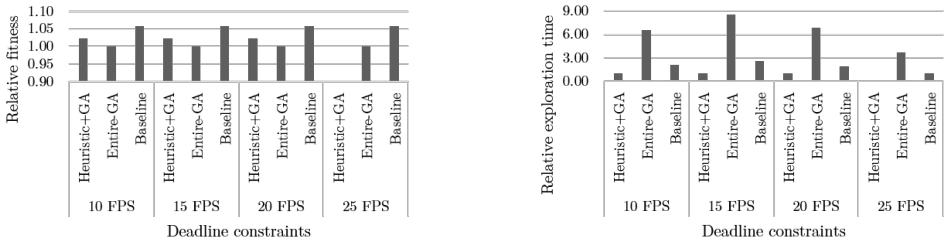
mean of the profiled execution times. We deploy four different DL networks as described in Table 5.3, each of which corresponds to a *Det* in Figure 5.1. For example, *Det4*, which is connected to *Image processing4*, is a *Yolov4csp* network in Figure 5.1. Also, we set the priorities of application groups in the following order: G_1(Image processing1 + Det1), G_4, G_2, G_3. The mappable processors of each task are described in Table 5.4.

Finding mapping candidates: The number of Pareto-optimal mappings selected from step 1 for each DL application is shown in Table 5.3. In this step, we run the GA with 65 chromosomes for 1000 iterations with uniform crossover, one-point mutation, and Lexicase selection [76]. Multiplying those numbers for all DL applications defines the design space size for selecting the mapping candidates in step 2.

Design space exploration: We vary the deadline constraint of all groups from 10 FPS to 25 FPS, assuming implicit deadlines[2]. In this experiment, we run the GA at most 100 iterations with 2048 chromosomes by making the GA terminate if there is no better solution found in 10 iterations.

Figure 5.6 shows the relative fitness (maximum utilization) and exploration time over the minimum value in each bundle. The lower value is better in the figure. For all deadline constraints, the *Entire-GA* method shows the best results as shown in Figure 5.6 (a). Up to a deadline constraint of 20 FPS, the *Heuristic+GA* and *Entire-GA* methods give a similar result. But with the deadline constraint of 25 FPS, *Entire-GA* gives better results by a large margin. This is because the size of the design space for *Heuristic+GA* is too narrow to find good solutions. The *Baseline* method could not find a feasible solution at a tight deadline constraint of 25 FPS. Also, at lower deadline constraints, it shows 5% $\sim$ 7% worse fitness than *Entire-GA*. As for exploration time, the *Heuristic+GA* method took the least amount of time to perform, as depicted in Figure 5.6 (b) since it explores a smaller design space than the other methods. The Entire-GA method takes the longest time as expected.

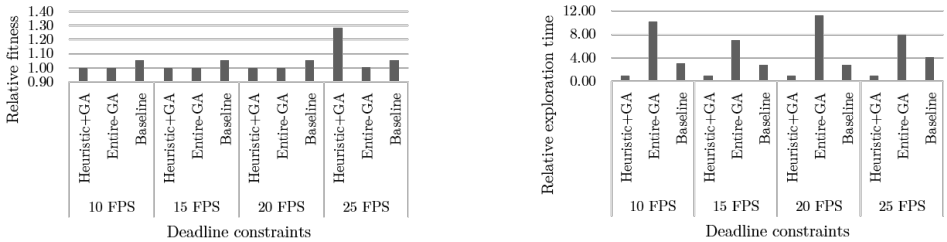
[2]Implicit deadline means the period is equal to the relative deadline. For example, 10 FPS means a deadline of 100 milliseconds.



(a) Fitness

(b) Exploration time

Figure 5.7: Comparison of three methods with four randomly generated dataflow applications.



(a) Fitness

(b) Exploration time

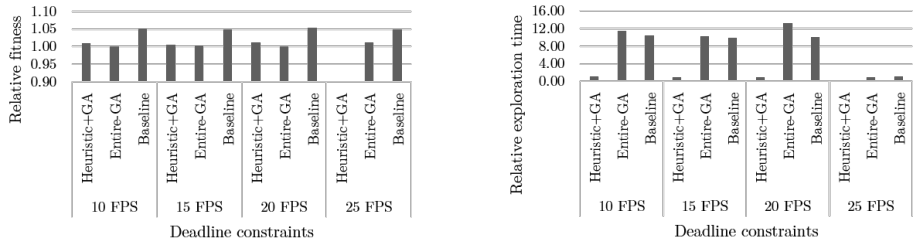
Figure 5.8: Comparison of three methods with eight randomly generated dataflow applications.

Code generation: To check the viability of the proposed methodology, we synthesize the target codes with a model-based design framework [18] and run them on the target hardware platform. The sample of the motivational example is available on a GitHub[3].

5.5.4 Experimental Results: Randomly Generated Dataflow Graphs

In this experiment, we randomly generate dataflow graphs by the SDF3 [82] while using the same four DL applications as the motivational example. We assume that all

[3] `https://github.com/cap-lab/HOPES/tree/master/HOPES_UI/schematics/Test_Examples/ExternalTask/ImgProcessing.files`



(a) Fitness

(b) Exploration time

Figure 5.9: Comparison of three methods with sixteen randomly generated dataflow applications.

applications are independent. Each dataflow graph consists of 10 nodes. We vary the number of dataflow graphs and the average WCET of the dataflow tasks. The average WCET of dataflow tasks is inversely proportional to the number of graphs to make the total workload of dataflow applications remain similar. It means that the average WCET of dataflow tasks is four times larger with 4 dataflow graphs than that with 16 dataflow graphs. In the case that the number of graphs is 16, the WCET of a dataflow task is randomly set in the range of $10 \sim 100us$ on the GPU, and $100 \sim 500us$ on the CPU. And we set a half of dataflow tasks can be run on GPU only. We make the average WCET of CPU-mapped tasks be about five times of that of GPU-mapped tasks.

Figures 5.7-5.9 show the results of the experiment when the number of graphs is 4, 8, and 16, respectively. As expected, the *Entire-GA* method shows the best fitness. The fitness gap between the *Entire-GA* method and the *Heuristic+GA* method tends to increase as the deadline constraint is tightened. In some cases, the *Heuristic+GA* method could not find a solution even when the *Baseline* method found a solution. But in most cases when the *Heuristic+GA* method finds a solution, it finds a better solution than the *Baseline* method. Hence these two methods are not dominating each other in terms of fitness. When no solution could be found by the *Heuristic+GA* method when the deadline constraint is 25 FPS, we loosened the constraint to 10 FPS and found the solution that is used as the initial chromosome in the *Entire-GA* method.

As for the exploration time, the *Heuristic+GA* method takes the least time by more than three times than the *Entire-GA* method. This experiment clearly demonstrates the trade-off between two methods in terms of fitness and exploration time. And it also shows that the proposed two methods are significantly better than the previous state-of-the-art method which is the *Baseline* method.

Chapter 6

Conclusion and Future work

In this dissertation, we propose the system-level optimization methodology for a single deep learning application as well as multiple deep learning applications. In addition, we extend the model-based methodology to support deep learning applications.

For a single deep learning application, we introduce a novel framework, called JEDI, for deep learning inference acceleration with TensorRT on NVIDIA Jetson embedded platforms. The proposed framework allows users to exploit various types of parallelism, including multi-threading, multiple streams, pipelining of the network, and partial network duplication. Also, various optimization parameters can be configured in JEDI to increase the throughput performance. In addition, we devise a novel method to efficiently explore the huge design space consisting of optimization parameters. We apply two heuristics to automatically explore the other optimization parameters. The JEDI framework is used to obtain the performance by running the network on the real hardware platform during the optimization process. The proposed framework and methodology are evaluated with real-life object detection networks on a real hardware platform, NVIDIA Jetson AGX Xavier. The proposed method achieves significant FPS performance improvement by 101% $\sim$ 680% and reduces the energy consumption up to 55% over the baseline that uses the GPU only.

As for multiple deep learning applications, we propose a scenario-based mapping

methodology to run multiple deep learning applications on heterogeneous processors, aiming to reduce energy consumption while satisfying the real-time constraints. The proposed technique considers several technical challenges imposed by the use of NPU and its associated SDK. The proposed method consists of three steps. In the first step, we obtain the Pareto-optimal mappings for each application. In the second step, we seek a sub-optimal mapping combination of applications, considering the scenarios and real-time constraints. In the last step, we fine-tune the frequencies of processors if the deadline constraint is loosened. The proposed method is confirmed by using real-life applications with different scenarios. We could satisfy up to 40% higher deadline constraints and reduce the energy consumption by 22% $\sim$ 31% compared to the state-of-the-art static mapping methods with real-life applications and different scenarios on a real platform.

Lastly, we propose a novel technique to support deep learning applications in a model-based embedded software design methodology leveraging the optimization capability of a deep learning SDK. We first find a set of Pareto-optimal mapping candidates for each deep learning application, independently of the model-based design flow. Adding the obtained mapping solution to the model-based design framework, we explore the mapping of dataflow applications and the mapping candidates of deep learning applications together with the meta-heuristic. The viability and efficacy of the technique are confirmed by experiments with a non-trivial real-life example and randomly generated graphs. We could reduce at least 5% of the maximum utilization over the previous state-of-the-art method that separates the mapping of deep learning applications and dataflow applications. More importantly, the proposed technique could find a solution when the previous method fails to find one.

While much of the research has been presented in this thesis, it is left to future work to apply the proposed framework and methodologies to new hardware platforms, SDKs, and applications that are continuously being developed. In the case of hardware platforms and SDKs, Intel's OpenVINO is one example. However, OpenVINO, which is an SDK

provided by Intel, does not support many layers like TensorRT on GPUs and ARM-based processors, which are widely used in embedded systems. Therefore, we have not been able to apply our proposed methodology using OpenVINO. In addition, as the trend of deep neural networks changes from CNNs to transformers, it is necessary to develop a methodology for transformers. However, commercially available NPUs do not currently support the operations mainly used by transformers. Therefore, it is difficult to apply the proposed methodology to transformers current.

However, we believe that the proposed methodology is still effective if platforms and SDKs are continually developed and new NPUs supporting transformer networks are released. In this process, the methodologies may need to be enhanced due to the additional constraints imposed by new processors, associated SDK, and applications.

Bibliography

[1] Zirui Xu, Fuxun Yu, Chenchen Liu, and Xiang Chen. Reform: Static and dynamic resource-aware dnn reconfiguration framework for mobile device. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[2] S Rallapalli, H Qiu, A Bency, S Karthikeyan, R Govindan, B Manjunath, and R Urgaonkar. Are very deep neural networks feasible on mobile devices. *IEEE Trans. Circ. Syst. Video Technol*, 2016.

[3] Loc N Huynh, Youngki Lee, and Rajesh Krishna Balan. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 82–95, 2017.

[4] Akhil Mathur, Nicholas D Lane, Sourav Bhattacharya, Aidan Boran, Claudio Forlivesi, and Fahim Kawsar. Deepeye: Resource efficient local execution of multiple deep vision models using wearable commodity hardware. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 68–81, 2017.

[5] Duseok Kang, Euiseok Kim, Inpyo Bae, Bernhard Egger, and Soonhoi Ha. C-good: C-code generation framework for optimized on-device deep learning. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2018.

[6] Siqi Wang, Gayathri Ananthanarayanan, Yifan Zeng, Neeraj Goel, Anuj Pathania, and Tulika Mitra. High-throughput cnn inference on embedded arm big. little multicore processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2254–2267, 2019.

[7] Linpeng Tang, Yida Wang, Theodore L Willke, and Kai Li. Scheduling computation graphs of deep learning models on manycore cpus. *arXiv preprint arXiv:1807.09667*, 2018.

[8] Husheng Zhou, Soroush Bateni, and Cong Liu. Sˆ 3dnn: Supervised streaming and scheduling for gpu-accelerated real-time dnn workloads. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 190–201. IEEE, 2018.

[9] Roger Pujol, Hamid Tabani, Leonidas Kosmidis, Enrico Mezzetti, Jaume Abella Ferrer, and Francisco J Cazorla. Generating and exploiting deep learning variants to increase heterogeneous resource utilization in the nvidia xavier. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 23, 2019.

[10] Yecheng Xiang and Hyoseung Kim. Pipelined data-parallel cpu/gpu scheduling for multi-dnn real-time inference. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 392–405. IEEE, 2019.

[11] Svetlana Minakova, Erqian Tang, and Todor Stefanov. Combining task- and data-level parallelism for high-throughput cnn inference on embedded cpus-gpus mpsocs. In Alex Orailoglu, Matthias Jung, and Marc Reichenbach, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 18–35, Cham, 2020. Springer International Publishing.

[12] Duseok Kang, Jinwoo Oh, Jongwoo Choi, Youngmin Yi, and Soonhoi Ha. Scheduling of deep learning applications onto heterogeneous processors in an embedded device. *IEEE Access*, 8:43980–43991, 2020.

[13] Woosung Kang et al. Lalarand: Flexible layer-by-layer cpu/gpu scheduling for real-time dnn tasks. In *Proceedings of the RTSS*, 2021.

[14] Soonhoi Ha and Hyunok Oh. Decidable dataflow models for signal processing: Synchronous dataflow and its extensions. *Handbook of Signal Processing Systems*, pages 1083–1109, 2013.

[15] Edward A Lee and David G Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9), 1987.

[16] Greet Bilsen et al. Cycle-static dataflow. *IEEE Transactions on signal processing*, 44(2):397–408, 1996.

[17] Hyesun Hong, Hyunok Oh, and Soonhoi Ha. Hierarchical dataflow modeling of iterative applications. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.

[18] Eunjin Jeong, Dowhan Jeong, and Soonhoi Ha. Dataflow model–based software synthesis framework for parallel and distributed embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 26(5):1–38, 2021.

[19] NVIDIA TensorRT. `https://developer.nvidia.com/tensorrt/`, 2023. [Online; accessed 02-June-2023].

[20] Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. A review on genetic algorithm: past, present, and future. *Multimedia Tools and Applications*, 80:8091–8126, 2021.

[21] JC Palencia Gutiérrez, JJ Gutiérrez García, and M González Harbour. On the schedulability analysis for distributed hard real-time systems. In *Proceedings Ninth Euromicro Workshop on Real Time Systems*, pages 136–143. IEEE, 1997.

[22] Marek Jersak. *Compositional performance analysis for complex embedded applications*. PhD thesis, Braunschweig, Techn. Univ., 2005.

[23] David Harel and Michal Politi. *Modeling reactive systems with statecharts: the STATEMATE approach*. McGraw-Hill, Inc., 1998.

[24] Rajesh Devaraj, Arnab Sarkar, and Santosh Biswas. Supervisory control approach and its symbolic computation for power-aware rt scheduling. *IEEE Transactions on Industrial Informatics*, 15(2):787–799, 2018.

[25] Jeronimo Castrillon et al. Maps: Mapping concurrent dataflow applications to heterogeneous mpsocs. *IEEE Transactions on Industrial Informatics*, 9(1):527–545, 2011.

[26] M. Pelcat et al. Preesm: A dataflow-based rapid prototyping framework for simplifying multicore dsp programming. In *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*, pages 36–40, Sept 2014.

[27] Joseph Buck et al. Ptolemy: A framework for simulating and prototyping heterogeneous systems. In *Readings in hardware/software co-design*, pages 527–543. 2001.

[28] Claudius Ptolemaeus. *System design, modeling, and simulation: using Ptolemy II*, volume 1. Ptolemy. org Berkeley, 2014.

[29] Rajesh Devaraj and Arnab Sarkar. Resource-optimal fault-tolerant scheduler design for task graphs using supervisory control. *IEEE Transactions on Industrial Informatics*, 17(11):7325–7337, 2020.

[30] Martín Abadi et al. Tensorflow: A system for large-scale machine learning. In *ODSI*, 2016.

[31] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *CoRR*, abs/1408.5093, 2014.

[32] Adam Paszke et al. Pytorch: An imperative style, high-performance deep learning library. In *NIPS*. 2019.

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

[34] Google TensorFlow Lite. `https://www.tensorflow.org/lite/`, 2023. [Online; accessed 02-June-2023].

[35] Gopalakrishna Hegde, Siddhartha, Nachiappan Ramasamy, and Nachiket Kapre. Caffepresso: An optimized library for deep learning on embedded accelerator-based platforms. In *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*, pages 1–10, 2016.

[36] 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.

[37] 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 the 24th ACM international conference on Multimedia*, pages 1201–1205, 2016.

[38] 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 *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 1–12, 2016.

[39] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V Le, and Jeff Dean. A hierarchical model for device placement. In *International Conference on Learning Representations*, 2018.

[40] Duseok Kang, DongHyun Kang, Jintaek Kang, Sungjoo Yoo, and Soonhoi Ha. Joint optimization of speed, accuracy, and energy for embedded image recognition systems. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 715–720, 2018.

[41] Micaela Verucchi, Gianluca Brilli, Davide Sapienza, Mattia Verasani, Marco Arena, Francesco Gatti, Alessandro Capotondi, Roberto Cavicchioli, Marko Bertogna, and Marco Solieri. A systematic assessment of embedded neural networks for object detection. In *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 937–944. IEEE, 2020.

[42] ONNX. `https://github.com/onnx/onnx`, 2023. [Online; accessed 02-June-2023].

[43] NVIDIA Ploygraphy. `https://docs.nvidia.com/deeplearning/tensorrt/polygraphy/docs/index.html`, 2023. [Online; accessed 02-June-2023].

[44] Martin Fowler. *Patterns of Enterprise Application Architecture: Pattern Enterpr Applica Arch*. Addison-Wesley, 2012.

[45] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

[46] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[47] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *CoRR*, abs/2004.10934, 2020.

[48] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. Scaled-yolov4: Scaling cross stage partial network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13029–13038, 2021.

[49] Chien-Yao Wang, Hong-Yuan Mark Liao, Yueh-Hua Wu, Ping-Yang Chen, Jun-Wei Hsieh, and I-Hau Yeh. Cspnet: A new backbone that can enhance learning capability of cnn. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*, pages 390–391, 2020.

[50] Densenet201+Yolo. `https://github.com/AlexeyAB/darknet/`, 2020. [Online; accessed 01-July-2021].

[51] CodaLab. `https://competitions.codalab.org/`, 2023. [Online; accessed 02-June-2023].

[52] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[53] Yanyu Li, Geng Yuan, Yang Wen, Ju Hu, Georgios Evangelidis, Sergey Tulyakov, Yanzhi Wang, and Jian Ren. Efficientformer: Vision transformers at mobilenet speed. *Advances in Neural Information Processing Systems*, 35:12934–12949, 2022.

[54] Michael R. Garey et al. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[55] Amit Kumar Singh, Muhammad Shafique, Akash Kumar, and Jörg Henkel. Mapping on multi/many-core systems: survey of current and emerging trends. In *Proceedings of the 50th Annual Design Automation Conference*, pages 1–10, 2013.

[56] Hyunok Oh and Soonhoi Ha. A static scheduling heuristic for heterogeneous processors. In *Euro-Par'96 Parallel Processing: Second International Euro-Par Conference Lyon, France, August 26–29, 1996 Proceedings, Volume II 2*, pages 573–577. Springer, 1996.

[57] 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.

[58] 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 the 53rd Annual Design Automation Conference*, pages 1–6, 2016.

[59] Mina Niknafs et al. Runtime resource management with workload prediction. In *Proceedings of the DAC*, DAC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[60] Robert Khasanov and Jeronimo Castrillon. Energy-efficient runtime resource management for adaptable multi-application mapping. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 909–914. IEEE, 2020.

[61] Bryan Donyanavard, Tiago Mück, Santanu Sarma, and Nikil Dutt. Sparta: Run-time task allocation for energy efficient heterogeneous many-cores. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 1–10, 2016.

[62] Stefan Valentin Gheorghita, Martin Palkovic, Juan Hamers, Arnout Vandecappelle, Stelios Mamagkakis, Twan Basten, Lieven Eeckhout, Henk Corporaal, Francky Catthoor, Frederik Vandeputte, et al. System-scenario-based design of dynamic embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14(1):1–45, 2009.

[63] Lars Schor, Iuliana Bacivarov, Devendra Rai, Hoeseok Yang, Shin-Haeng Kang, and Lothar Thiele. Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*, pages 71–80, 2012.

[64] Hanwoong Jung, Chanhee Lee, Shin-Haeng Kang, Sungchan Kim, Hyunok Oh, and Soonhoi Ha. Dynamic behavior specification and dynamic mapping for real-time embedded systems: Hopes approach. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):1–26, 2014.

[65] Wei Quan and Andy D Pimentel. A scenario-based run-time task mapping algorithm for mpsocs. In *Proceedings of the 50th Annual Design Automation Conference*, pages 1–6, 2013.

[66] Wei Quan et al. Scenario-based run-time adaptive mpsoc systems. *Journal of Systems Architecture*, 2016.

[67] Biyi Fang, Xiao Zeng, and Mi Zhang. Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 115–127, 2018.

[68] APOLLO: an open autonomous driving platform. `https://apollo.auto/`, 2018. [Online; accessed 15-Mar-2022].

[69] Tanya Amert et al. Gpu scheduling on the nvidia tx2: Hidden details revealed. In *Proceedings of the RTSS*, 2017.

[70] Ming Yang. Avoiding pitfalls when using nvidia gpus for real-time tasks in autonomous systems. In *Proceedings of the 30th Euromicro Conference on Real-Time Systems*, 2018.

[71] NVIDIA Jetson Developer Guide. `https://docs.nvidia.com/jetson/archives/l4t-archived/l4t-3231/`, 2023. [Online; accessed 02-June-2023].

[72] Erqian Tang, Svetlana Minakova, and Todor Stefanov. Energy-efficient and high-throughput cnn inference on embedded cpus-gpus mpsocs. In *Embedded Computer Systems: Architectures, Modeling, and Simulation: 21st International Conference, SAMOS 2021, Virtual Event, July 4–8, 2021, Proceedings*, pages 127–143. Springer, 2022.

[73] John P Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *[1990] Proceedings 11th Real-Time Systems Symposium*, pages 201–209. IEEE, 1990.

[74] Friedrich Eisenbrand and Thomas Rothvoß. Static-priority real-time scheduling: Response time computation is np-hard. In *2008 Real-Time Systems Symposium*, pages 397–406. IEEE, 2008.

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

[76] Thomas Helmuth, Lee Spector, and James Matheson. Solving uncompromising problems with lexicase selection. *IEEE Transactions on Evolutionary Computation*, 19(5):630–643, 2015.

[77] Edwin SH Hou, Nirwan Ansari, and Hong Ren. A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed systems*, 5(2):113–120, 1994.

[78] Amit Kumar Singh, Piotr Dziurzanski, Hashan Roshantha Mendis, and Leandro Soares Indrusiak. A survey and comparative study of hard and soft real-time dynamic resource allocation strategies for multi-/many-core systems. *ACM Computing Surveys (CSUR)*, 50(2):1–40, 2017.

[79] Dowhan Jeong et al. Parallel scheduling of multiple sdf graphs onto heterogeneous processors. *IEEE Access*, 2021.

[80] Borhan Kazimipour, Xiaodong Li, and A Kai Qin. A review of population initialization techniques for evolutionary algorithms. In *2014 IEEE congress on evolutionary computation (CEC)*, pages 2585–2592. IEEE, 2014.

[81] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[82] Sander Stuijk, Marc Geilen, and Twan Basten. Sdfˆ 3: Sdf for free. In *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*, pages 276–278. IEEE, 2006.

요 약

임베디드 시스템에서 딥 러닝 애플리케이션에 대한 증가하는 수요를 충족하기 위해 새로운 임베디드 디바이스에는 GPU와 뉴럴 프로세싱 유닛(NPU)이라고 하는 딥 러닝 하드웨어 가속기를 비롯한 여러 이기종 프로세서가 포함되는 경향이 나타나고 있다. 또한, 딥 러닝 애플리케이션의 빠르고 효율적인 개발을 위해 소프트웨어 개발 키트(SDK)가 제공된다. 딥 러닝 SDK에는 딥 러닝 어플리케이션의 짧은 지연 시간과 높은 처리량을 위한 옵티마이저가 포함되어 있다.

딥 러닝 SDK는 내부적으로 추론을 최적화하지만, SDK는 추론이 GPU 또는 NPU 중 하나의 처리 요소에서 수행하며 두 프로세서를 같이 사용하여 추론을 수행하지 않는다. 그러나 단일 처리 요소에서 추론을 실행하면 시스템을 완전히 활용하지 못 한다. 시스템이 이기종 프로세서로 구성되어 있기 때문에 효율적으로 실행하려면 이러한 프로세서를 동시에 사용해야 할 필요가 있다.

다시 말해서 딥 러닝 어플리케이션을 시스템 레벨에서 최적화하는 것이 필요하다. 이러한 맥락에서 우리는 크게 세가지 주제로 해당 문제를 접근하였다. 이 논문에서는 하나의 딥 러닝 어플리케이션의 최적화, 실시간 제약 조건 하에서 여러 딥 러닝 어플리케이션의 최적화, 모델 기반 설계 방법론에서 딥 러닝 어플리케이션 지원이라는 세 가지 주요 주제를 다룬다. 본 논문에서는 NPU를 비롯한 이기종 프로세서가 탑재된 NVIDIA Jetson 임베디드 플랫폼과 빠른 추론을 위한 대표적인 딥 러닝 SDK인 TensorRT를 대상으로 한다.

먼저, 딥 러닝 추론의 처리량을 높이기 위한 체계적인 최적화 기법과 방법론을 제안한다. 멀티 스레딩, 파이프라이닝, 버퍼 할당, 네트워크 복제 등 딥 러닝 애플리케이션을 위한 병렬화 기법을 소개한다. 또한 딥 러닝 애플리케이션을 가속화하기 위한 다양한 최적화 파라미터를 지원하는 프레임워크를 공개한다. 최적화 기법은 파라미터화되어 있어 프레임워크의 입력 파일에서 파라미터를 조정하는 것만으로 딥 러닝 애플리케이션에 적용할 수 있다. 서로 다른 프로세싱 요소에 레이어를 할당하고 다른 파라미터를 최적화하는 설계 공간은 방대하기 때문에 이기종 프로세서 간의 파이프라인 단계 균형을 맞추기 위한 휴리스틱과 파라미터 탐색 프로세스로 구성된 파라미터

최적화 방법론을 제안한다. 이는 TensorRT를 사용하는 딥 러닝 애플리케이션을 파티셔닝하고 NPU를 포함한 이기종 프로세서 시스템에서 처리량을 개선한 최초의 작업이다. 9개의 실제 벤치마크를 통해 GPU만을 사용한 추론에 비해 101% ∼ 680% 처리량 향상과 최대 55% 에너지 감소를 달성할 수 있었다.

두 번째로, 여러 기능을 제공하기 위해 여러 딥러닝 애플리케이션을 동시에 실행하는 것이 대중화되고 있다. 이 연구에서는 애플리케이션에 런타임에 따라 달라질 수 있는 실시간 제약 조건이 있다고 가정한다. 최근 다양한 하드웨어 플랫폼에서 여러 딥러닝 애플리케이션의 효율적인 매핑을 찾기 위한 광범위한 연구가 수행되었지만, 실제 임베디드 플랫폼에서 NPU와 해당 SDK에 의해 부과되는 제약 조건은 고려하지 않았다. 이 연구에서는 여러 이기종 프로세서가 있는 실제 임베디드 시스템에서 여러 딥 러닝 애플리케이션의 새로운 에너지 인지형 매핑 방법론을 제안한다. 모든 애플리케이션의 실시간 제약 조건을 만족하면서 에너지 소비를 최소화하는 것이 목표이다. 제안한 방식에서는 먼저 각 애플리케이션에 대한 파레토 최적 매핑 솔루션을 선택한다. 그런 다음 제약 조건을 만족하면서 애플리케이션의 동적 특성을 보여주는 시나리오를 고려하여 매핑 조합을 탐색한다. 또한 프로세서의 주파수를 조정하여 에너지 소비를 줄인다. 이는 NPU를 포함하는 실제 하드웨어 플랫폼에서 TensorRT 기반 멀티플 애플리케이션의 동시 실행을 한 최초의 작업이다. 실제 플랫폼에서 실제 애플리케이션과 다양한 시나리오를 사용하여 정적 매핑 방법에 비해 최대 40% 더 높은 마감 시간 제약을 만족하고 에너지 소비를 22% ∼ 31%까지 줄일 수 있었다.

마지막으로 딥 러닝 애플리케이션이 임베디드 시스템에 널리 보급됨에 따라 모델 기반 임베디드 소프트웨어 설계 방법론에서 딥 러닝 애플리케이션을 지원하는 방법은 어려운 문제가 되고 있다. 지금까지의 해결책은 각 딥 러닝 애플리케이션을 모델로 표현하는 것이다. 그러나 딥 러닝 애플리케이션에 최적화 기법을 적용하여 모델로 변환하고 좋은 성능을 얻기 위해서는 상당한 노력이 필요하다. 본 연구에서는 성능 최적화를 위해 딥 러닝 SDK를 활용하는 새로운 방법론을 제안한다. 제안하는 방법론에서는 먼저 하드웨어 플랫폼과 연동된 SDK를 이용하여 딥 러닝 애플리케이션의 파레토 최적 매핑 솔루션을 얻는다. 그런 다음 유전 알고리즘을 사용하여 데이터플로우 태스

크의 매핑과 딥러닝 애플리케이션의 매핑 솔루션 탐색을 동시에 수행한다. 동기 부여 예제와 무작위로 생성된 그래프를 사용한 실험 결과, 딥 러닝 애플리케이션과 데이터 플로우 기반 애플리케이션을 순차적으로 매핑하는 이전 작업과 비교했을 때 프로세싱 요소의 최대 사용률을 최소 5% 이상 줄일 수 있는 것도 확인하였다.

주요어 : 매핑 및 스케줄링, 설계 공간 탐색, 딥 러닝 응용, 소프트웨어 최적화, 이기종 프로세서 시스템

학번 : 2017-22440