Evaluación del rendimiento de redes neuronales en dispositivos edge mediante el framework NC-SDK

Performance benchmark of neural networks on edge devices using the NC-SDK framework

Trabajo de Fin de Máster
Curso 2023/2024

Autor

David Morán Montero

Directores

Francisco Daniel Igual Peña
Luis Piñuel Moreno

Máster en Internet de las Cosas
Facultad de Informática
Universidad Complutense de Madrid

Evaluación del rendimiento de redes neuronales en dispositivos edge mediante el framework NC-SDK

Performance benchmark of neural networks on edge devices using the NC-SDK framework

Trabajo de Fin de Máster en Internet de las Cosas
Dpto. de Arquitectura de Computadores y Automática

Autor

David Morán Montero

Directores

Francisco Daniel Igual Peña
Luis Piñuel Moreno

Convocatoria: *Septiembre 2024*
Calificación: *8,5*

Máster en Internet de las Cosas
Facultad de Informática
Universidad Complutense de Madrid

4 de septiembre de 2024

Resumen

Evaluación del rendimiento de redes neuronales en dispositivos edge mediante el framework NC-SDK

En el mundo de Internet de las Cosas (IoT), la computación en el borde se ha vuelto más popular en los últimos años. Este trabajo describe el framework Neural Compute Software Development Kit (NC-SDK), que contiene una serie de herramientas de desarrollo para poder adaptar, compilar y ejecutar inferencias de distintos modelos de aprendizaje profundo en ciertos dispositivos específicos.

Este proyecto muestra el flujo de trabajo con el paquete NC-SDK, haciendo hincapié en los procesos de adaptación de modelos de aprendizaje profundo desde distintas implementaciones, compilación de redes neuronales para su ejecución en la plataforma de despliegue, y ejecución de distintos modelos usando la GPU de PowerVR. Finalmente, se realiza una evaluación del rendimiento de distintos modelos de aprendizaje profundo sobre clasificación de imágenes, detección y segmentación de objectos, y procesamiento del lenguaje. Los resultados obtenidos serán comparados y analizados con el uso de CPU por otros frameworks.

El código personalizado utilizado para la creación de este proyecto se encuentra en Google Drive [1].

Palabras clave

NC-SDK — PowerVR — Redes neuronales — Rendimiento — Inferencia

Abstract

Performance benchmark of neural networks on edge devices using the NC-SDK framework

In the world of Internet of Things (IoT), edge computing has increased its popularity in recent years. This study describes the Neural Compute Software Development Kit (NC-SDK) package, which contains a set of development tools in order to be able to adapt, compile and run inferences from different deep learning models on certain specific devices.

This project demonstrates the workflow with the NC-SDK package, highlighting the processes of adapting deep learning models from various implementations, compiling neural networks for execution on the deployment platform, and executing distinct models using the PowerVR GPU. Finally, it is conducted a performance benchmark on different ML categories, such as image classification, object detection and segmentation, and language processing. The obtained results will be compared and analyzed with CPU usage by other frameworks.

The custom-code utilized for the creation of this project can be found on Google Drive [1].

Keywords

NC-SDK — PowerVR — Neural networks — Performance — Inference

Contents

1 Introduction **1**

1.1 Motivation . . . 2

1.2 Objectives . . . 2

1.3 Document structure . . . 3

2 State of the art **5**

2.1 The NC-SDK development tools . . . 5

2.1.1 NNVM and the IMGIR format . . . 6

2.1.2 IMGDNN runtime library . . . 7

2.1.3 Relay model language . . . 9

2.1.4 TVM optimization framework . . . 10

2.1.5 Data preprocessing . . . 14

2.2 Other edge computing packages . . . 15

2.2.1 TensorRT . . . 16

2.2.2 OpenVino . . . 16

2.2.3 Core ML . . . 17

2.3 Inference scenarios on edge computing . . . 17

2.4 Hardware for edge computing . . . 18

2.4.1 PowerVR hardware architecture . . . 18

3 NC-SDK methodology and workflow **21**

3.1 Network adaptation . . . 22

3.2 Transformation and profiling 24

3.3 PowerVR compilation 25

3.4 Input preprocessing 27

4 Benchmark architecture and results 29

4.1 Experimental setup 29

4.2 Benchmark design 30

4.2.1 Categories and models 30

4.2.2 Runtime environments 33

4.3 Evaluation and analysis 35

5 Conclusion and future work 39

5.1 Further customization of neural networks 40

5.2 Exploration of other ML areas 40

5.3 Comparation of edge computing packages 40

Bibliography 46

A Setting up NC-SDK development environment with Docker 47

Figures

2.1 Workflow of optimization of a neural network for a specific target. 6

2.2 Workflow of optimization of a neural network using TVM. 11

2.3 TVM runtime coordination with device drivers for binary execution. 12

2.4 Comparison between NCHW and NHWC image layouts [2]. 15

2.5 Workflow of optimization of a neural network using TensorRT [3]. 16

2.6 Workflow of optimization of a neural network using CoreML [4]. 17

2.7 Tile Based Deferred Rendering (TBDR) architecture used in PowerVR devices [5]. 19

3.1 Main workflow of the NC-SDK development tools for adapting, transforming and compiling neural networks to inference on PowerVR hardware. 22

3.2 Network adaptation process from a PyTorch model to Relay IR format. . . 23

3.3 Network transformation process of the Relay IR network from the network adaptation script to a specific target device. 24

3.4 Relay IR network compilation for a specific target device. 26

4.1 Main components of the Beaglebone AI-64 [6]. 30

4.2 A sample annotated image from the COCO dataset, showing the difference between image classification, object detection, and object segmentation [7]. 32

4.3 Network inference on the deployment platform using both TVM and IMGDNN runtime libraries. 34

4.4 Inference times of image classification models on the deployment platform. . 35

4.5 Inference times of object detection model on the deployment platform. . . . 36

4.6 Inference times of object segmentation models on the deployment platform. 36
4.7 Inference times of language processing model on the deployment platform. . 37

Tables

2.1 Main comparison between edge computing packages for optimizing deep learning models for AI inference. 15

4.1 Table with all inference times of the performance benchmark, indicating the model and its ML task associated. 37

Chapter 1

Introduction

Over the last decade, the landscape of artificial intelligence (AI) computation has started to transform from largely centralized data centers to real-time edge devices. Due to the continuously growing number of smart devices, cloud-based data centers have encountered different shortcomings [8], such as an increase in energy consumption or potential vulnerabilities with a high risk of privacy exposure. With the continuous development of Internet of Things (IoT) deployments, there is an increasing interest in running standard AI models on edge devices for different activities [9], such as image classification or language processing, among others.

Edge computing, an architecture designed to compute data as close to its source as possible, offers several advantages [10]. Among them, latency is significantly reduced by processing data locally on the edge, enabling real-time decision-making for applications like autonomous driving, industrial automation, or smart healthcare. Additionally, network access to cloud devices decreases, allowing bandwidth-intensive tasks like video streaming to achieve computational capabilities similar to end devices.

However, there are still some challenges that need to be addressed. Due to its resource-constrained nature compared to traditional cloud infrastructures, deploying and running complex neural networks on edge devices requires innovative forms of optimization, such as model compression, quantization, and the use of hardware accelerators (for example, NNA) [11]. A robust software and hardware infrastructure is essential for running deep learning networks efficiently on edge devices.

It must be taken into account that inference has become a critical workload, where AI models can serve as many as 200 trillion queries and perform over six billion translations a day [12]. In the course of three years, language processing networks have increased to 175 billion model parameters and 45 TB of pretrained data size [13]. Consequently, significant effort in optimization is required to infer diverse neural networks on edge devices by using specific frameworks, making standard and complex models effective across a wide range of applications and architectures.

Nowadays, there is a high diversity of AI models, machine learning (ML) frameworks, tools, libraries, and platforms with hardware-specific features that allow to obtain the best possible results on edge devices, achieving a balance between performance and accuracy. With such a number of possible combinations, there are benchmarks that prescribe a set of rules and best practices to create a standard evaluation criteria. In this context, the most famous example is the MLPerf Inference Benchmark Suite [14], which manages each task to be well-defined, ensuring the reproducibility and accessibility of the benchmarks.

1.1 Motivation

Currently, there exist several libraries and frameworks developed by different organizations to improve model accuracy and optimize model performance. These tools are also crucial for maximizing efficiency and ensuring that networks run effectively across various hardware devices.

Building on the importance of these optimization efforts, the main motivation of this study is to conduct an inference performance benchmark of deep learning models using the Neural Compute Software Development Kit (NC-SDK) from Imagination Technologies [15]. This toolkit permits heterogeneous compilation of neural networks across PowerVR hardware accelerators [16].

Another key point of motivation is to learn, explore, and infer well-known neural network models on edge devices using different common techniques over a wide variety of ML tasks, like image classification, object detection and segmentation, and language processing. With the help of the NC-SDK development tools, it will be possible to execute common neural network implementations on edge devices with hardware constraints and limitations.

1.2 Objectives

The main goal of this work is to perform a benchmark of various deep learning models using the NC-SDK development tools. By evaluating the inference performance of these neural networks, the results of this benchmark will provide a review of how optimized networks with the NC-SDK package interact with different software and hardware configurations on edge devices.

The approach of this study involves the use of the NC-SDK package to prepare and optimize standard ML models for deployment, followed by a performance evaluation of graph-compiled networks across distinct hardware components. This benchmark will assess the efficiency of specific hardware devices handling various optimized neural networks in resource-constrained edge computing environments. In general terms, this work makes the following key contributions:

- Manipulate and compile standard well-known networks on a development platform by using the NC-SDK development tools.
- Execute the processed models on a deployment platform across various runtime environments and hardware components.
- Conduct a comprehensive benchmark of the optimized models, analyzing their performance in terms of inference speed and providing valuable insights of the NC-SDK package on specific hardware devices.
- Compare the obtained execution results with other edge computing frameworks and hardware configurations.

1.3 Document structure

This study divides the information into the following chapters:

- Chapter 2 explains the tools of the NC-SDK package and compares it with other edge computing frameworks made by other organizations. It also highlights the structure of edge computing hardware accelerators, including PowerVR devices.
- Chapter 3 describes the main workflow of the NC-SDK development tools, from adapting neural networks from common implementations to compiling them for inference on PowerVR GPU devices.
- Chapter 4 presents the evaluation and analysis of the performance benchmark on inference speeds over various well-known deep learning models compiled to run on the Beaglebone AI-64 [17].
- Finally, chapter 5 summarizes the overall conclusions of this study and highlights future research on the edge computing topic.

Chapter 2

State of the art

In recent years, there has been significant progress in the field of IoT. Edge computing has become increasingly important due to the growing demand for AI inference on edge devices. The current availability of various frameworks, libraries, and platforms has simplified the task of running different types of neural networks in edge environments, making AI more accessible and efficient.

This chapter will review the main characteristics of the NC-SDK development tools. Additionally, this package will be compared with other edge computing frameworks that specialize in optimizing and deploying neural networks on edge devices. Finally, we will examine common scenarios involved in designing a performance benchmark, as considered by the MLPerf Inference Benchmark Suite.

2.1 The NC-SDK development tools

The Neural Compute Software Development Kit [15] is a set of model transformation tools, specific compiler frameworks, and runtime libraries developed by Imagination Technologies and responsible for preparing, optimizing, and deploying deep learning models from supported formats on PowerVR hardware. The NC-SDK development tools are capable of executing high-level neural networks, ensuring optimal performance and minimal inference times on specific hardware. The key features of these utilities are the following:

- **Transformation of models to Intermediate Representation (IR)**: These tools allow the compilation of neural networks from supported formats to two distinct types of IR (IMGIR and Relay), enabling simple transformations for every kind of target device [18].
- **Optimization of IR networks to specific hardware**: By using several techniques, it is possible to compile the IR and optimize the transformed model for its execution on PowerVR hardware (GPU and NNA) or other compatible hardware.

- **Execution of processed models by runtime libraries**: Final transformed and compiled networks are deployed on the target device, where inferences are executed by specialized runtime libraries, such as TVM or IMGDNN.

2.1.1 NNVM and the IMGIR format

NNVM [19] is an open source deep learning compiler focused on deploying neural network workloads on a variety of platforms and hardware backends. NNVM provides a way to import different networks from a wide variety of frontend frameworks. Then, the network is transformed and optimized at graph level before being compiled for a specific target by the TVM compiler stack [20] (see figure 2.1).

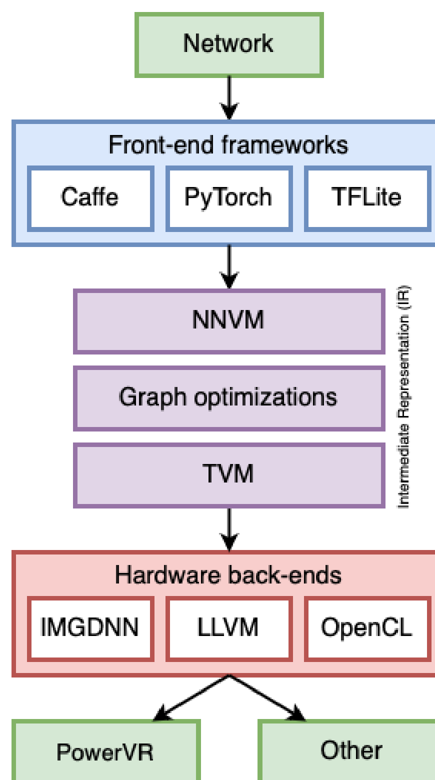

Figure 2.1: Workflow of optimization of a neural network for a specific target.

In summary, with the help of the TVM compiler framework, the NNVM compiler makes the following key contributions:

- Represent and optimize the common frontend frameworks in high-level graph IR, such as IMGIR.
- Transform the computation graph to minimize memory utilization, optimize data layout, and use computation patterns for different hardware backends.
- Present a compilation workflow from frontend deep learning frameworks to PowerVR hardware (or other compatible hardware).

IMGIR is an IR file format used by the IMGDNN runtime library. With this format, IMGIR wraps the compiled NNVM model description and the weight data parameters that can be read by IMGDNN for its execution on target devices. This format is presented by default with two different files:

- **Network (`.imgir`)**: Contains the compiled NNVM network description as text.
- **Parameters (`.imgir.params`)**: Contains the weight data parameters that are defined in the network file.

2.1.2 IMGDNN runtime library

The IMGDNN runtime library is an AI-oriented API that provides several functions to create network layers for deploying and executing deep learning models on PowerVR hardware. Its API also provides low-level hardware-specific optimizations of common neural networking functions to increase the model's performance.

IMGDNN uses OpenCL [21], enabling the creation of more complex neural networks by making use of distinct OpenCL kernels and constructs. Additionally, its driver makes use of the NNVM compiler as a way to internally represent networks through the low-level TVM compiler. Therefore, the implementation of this library allows to import different kinds of models in NNVM-compiled IR, like IMGIR.

Network execution with IMGDNN

This section explains how to execute optimized IMGIR models using the IMGDNN API in C++. Firstly, a basic setup of the target PowerVR hardware is required. Listing 2.1 demonstrates the functions for identifying PowerVR devices available in the system and creates the necessary context for these devices.

```
1 unsigned int num_devices;
2 imgdnn_device devices[10];
3 imgdnnGetDevices(IMGDNN_DEVICE_TYPE_ALL, 10, devices, &num_devices);
4
5 imgdnn_err_code err;
6 imgdnn_context context = imgdnnCreateContext(num_devices, devices, 0, &err);
```

Listing 2.1: Basic setup of supported PowerVR devices present in the target hardware.

Next, it is needed to define the architecture of the network with the `imgdnn_network` object. IMGDNN offers the opportunity to import an already processed IMGIR model to define the network, as shown in listing 2.2. Additionally, it is possible to create networks from scratch, allowing developers to create and design custom-made architectures for specific requirements through input and output tensor descriptors.

```
7  imgdnn_network network =
8      imgdnnCreateNetworkFromIR(model, model_size, params, params_size, &err);
```

Listing 2.2: Creation of the network architecture from a processed IMGIR model.

Once the network has been created, the next step is to instantiate an object from the `imgdnn_network_object` class, which will allow the network to be run. When creating this network object, it is essential to specify which tensors serve as the input and which serve as the output, as these will be used during execution to provide input data and retrieve output results.

```
9  imgdnn_tensor* input_tensor =
10     imgdnnNetworkFindInputs(network, NULL, &err);
11 imgdnn_tensor* output_tensor =
12     imgdnnNetworkFindDefaultOutputs(network, NULL, &err);
13
14 imgdnn_network_object network_object = imgdnnCreateNetworkObject(device[0],
15     context, network, 1, input_tensor, 1, output_tensor, 0, NULL, &err);
```

Listing 2.3: Creation of the network object, specifying input and output tensors.

Having created the network object, listing 2.4 shows how to get the inputs and output objects (which are different from the input and output tensors).

```
16 unsigned int num_inputs;
17 imgdnn_input input;
18 imgdnnNetworkObjectGetInputs(network_object, 1, &input, &num_inputs);
19
20 unsigned int num_outputs;
21 imgdnn_output output;
22 imgdnnNetworkObjectGetOutputs(network_object, 1, &output, &num_outputs);
```

Listing 2.4: Retrieval of input and output objects from the network object.

For any of the instantiated input and output objects, it is needed to allocate their memory for the network execution. Listing 2.5 indicates the process of memory allocation, in which the input memory is populated with data from an input file.

```
23 auto input_descriptor = imgdnnGetInputDescriptor(input, &err);
24 size_t in_size = imgdnnGetDescriptorSize(&input_descriptor, &err);
25 auto in_memory = imgdnnImportMemory(context, &input_data, in_size, 0, &err);
26
27 auto output_descriptor = imgdnnGetOutputDescriptor(output, &err);
28 size_t out_size = imgdnnGetDescriptorSize(&output_descriptor, &err);
29 auto out_memory = imgdnnAllocateMemory(context, out_size, &err);
```

Listing 2.5: Memory allocation for input and output objects.

Moreover, it is necessary to create a binding object, which is provided during execution and instructs IMGDNN on which memory objects to use for each input and output. Listing 2.6 demonstrates the functions needed to create the binding object and map the corresponding input and output memory objects to it.

```
30 imgdnn_binding binding = imgdnnCreateBinding(&err);
31 imgdnnBindingAddInput(binding, input, in_memory);
32 imgdnnBindingAddOutput(binding, output, out_memory);
```

Listing 2.6: Creation of the binding object for input and output memory management.

Finally, run the network with the input data using the previously created network object and memory binding. The `imgdnnNetworkObjectExecute` function executes inferences on the processed IMGIR model (`network_object`) using the specified input data (`binding`).

```
33 imgdnnNetworkObjectExecute(network_object, binding, true, 0, NULL, NULL);
```

Listing 2.7: Network execution and inference of the IMGIR model.

2.1.3 Relay model language

Another IR used in the NC-SDK development tools workflow is Relay [22]. Relay is a high-level IR, statically typed, and functional language:

- **High-level IR**: Relay is intended as the top layer of the TVM compiler stack, where the input network is parsed and converted into IR.
- **Statically typed**: Every variable, constant, operation, and function of this language has a shape and a data type.
- **Functional language**: Relay uses functions to represent the network. Each function is a sequence of operations, which are either mathematical operations (`conv2d`, `softmax`, etc.) or operations that control the flow of execution. The `main` function is the entry point of the execution, which can call other functions that might be compiled for a specific target device.

In addition, Relay is an expression-based language. In simple terms, every entity in Relay is represented as an expression. The main expressions of this language are detailed below [23]:

- **Variables**: A variable or an identifier is a placeholder with a unique name, shape, and data type. It does not hold any data at the time of compilation. All variables are of type `relay.Var`.

- **Constants**: Like variables, they have a name, shape, and data type. Additionally, each constant has data associated with it, which is stored as metadata. All constants are of type `relay.Constant`.

- **Operators**: An operator is a primitive operation, such as add, subtract, or a convolution. Every operator has a name, inputs, attributes, outputs, and an implementation that define how to perform the actual calculations of the operator.

- **Calls**: Every operator in Relay is a function, and it is *called* like any other function. All calls are of type `relay.Call`.

- **Tuples**: Relay tuples (`relay.Tuple`) store multiple items. A tuple contains one or more `relay.Var`, `relay.Constant` or `relay.Call` items.

- **Functions and modules**: Functions are represented as `relay.Function` nodes and a Relay module is a collection of one or more functions.

- **Types**: Out of many supported types, the most common are tensor and tuple types. The `relay.TensorType` is the type assigned to tensors (variables, constants, and functions) with a known shape and data type. The `relay.TupleType` contains one or more types, but used to represent types for `relay.Tuple` nodes.

2.1.4 TVM optimization framework

The Tensor Virtual Machine (TVM) [20] is a framework for ML systems dedicated to compile, optimize, and execute networks on a variety of hardware devices. As shown in figure 2.1, TVM compiles IR into binaries for its execution by distinct hardware backends on the target platform. However, it is also possible to run neural networks on target devices with the TVM runtime library.

Figure 2.2 illustrates the general workflow of the transformation of deep learning models into binaries for their execution on specific hardware devices. This process involves several stages, each focusing on different aspects of the model compilation and optimization to ensure compatibility and efficiency on the target hardware. In this context, three types of operations have been identified to adapt, transform, and compile different kinds of neural networks [24]:

- **TVM frontend**: It allows to compile models written in several frontend frameworks into Relay IR by using the NNVM compiler.

- **TVM backend**: TVM takes Relay IR as input, performs high-level optimizations, and generates binary files suitable for execution on specific target devices using the TVM Runtime. In our study, we emphasize the compilation and execution of neural networks on PowerVR hardware.

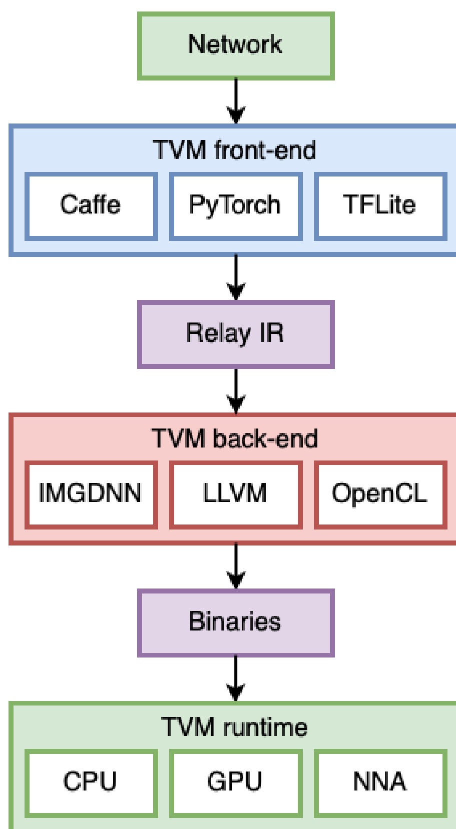

Figure 2.2: Workflow of optimization of a neural network using TVM.

- **TVM runtime**: It is a lightweight library that runs on the target host device and allows to load and execute binaries generated by the TVM backend across different hardware devices. It offers two runtime methods:
  - **Relay VM**, which provides a dynamic execution environment. Neural networks with control flow and dynamic tensors must be executed using this runtime only.
  - **Graph Runtime**, which offers a fast execution experience but only for a limited subset of programs such as static and sequential graphs.

  This runtime handles the coordination of different network segments executing on various devices and interacting with its drivers to allocate memory and execute binaries (see figure 2.3).

Network execution with TVM

This section explains how to execute transformed and compiled Relay IR networks using the TVM runtime API in C++. Firstly, we will explore network execution using the Relay VM method. After that, we will shift to the Graph Runtime, providing a comparison between these two execution strategies on the PowerVR hardware.

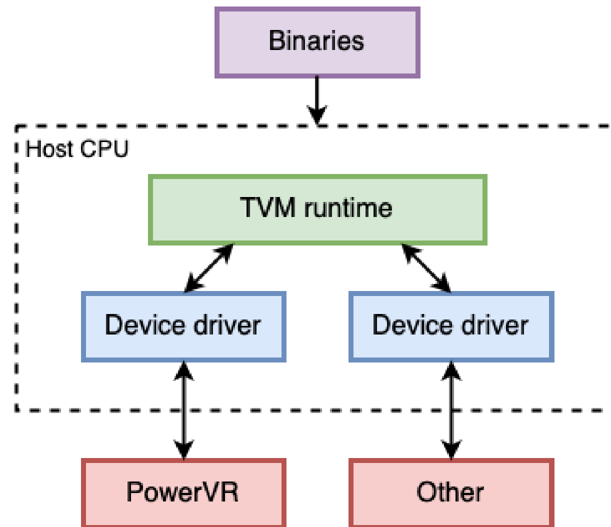

Figure 2.3: TVM runtime coordination with device drivers for binary execution.

When utilizing the TVM runtime with the Relay VM strategy, the initial step is to load the packed functions from the dynamic library (`.so`) that contains the executable binaries. The TVM runtime accomplishes this by using the `LoadFromFile` function to retrieve the necessary data into a `tvm::runtime::Module` object.

```
1 using namespace tvm;
2 using namespace tvm::runtime;
3 Module mod_syslib = Module::LoadFromFile("deploy.so");
```

Listing 2.8: Load Relay VM packed functions from the dynamic library.

Next, an instance of the virtual machine must be created to execute the Relay VM bytecode (`.ro`). As shown in listing 2.9, this is achieved by calling the packed function `runtime._VirtualMachine`, passing in the target executable generated from the Relay VM bytecode along with the loaded module containing the packed functions from the dynamic library.

```
4 Module target_exec = vm::Executable::Load(str_vm, mod_syslib);
5 const PackedFunc *create = Registry::Get("runtime._VirtualMachine");
6 Module vm_instance = (*create)(target_exec);
```

Listing 2.9: Create the instance of the virtual machine through the target executable from the Relay VM bytecode and the loaded packed functions module.

The next step is to initialize the virtual machine with a set of device contexts. A device context includes a device type and a device identifier (set to zero unless there is a need to differentiate between multiple devices on the same hardware). Listing 2.10 demonstrates the initialization of the virtual machine instance (`vm_instance`) using the `init` packed function.

```
7  PackedFunc init = vm_instance.GetFunction("init");
8  init((int)kDLCPU,        (int) 0, (int) 1,
9       (int)kDLPowerVRNNA, (int) 0, (int) 1,
10      (int)kDLPowerVRGPU, (int) 0, (int) 1,
11      (int)kDLOpenCL,     (int) 0, (int) 1);
```

Listing 2.10: Initialization of the virtual machine instance with a set of device contexts.

Before executing the network, it is necessary to establish the input data values on the virtual machine instance. This involves creating an `NDArray` object to hold the data and then passing it to the `set_input` packed function, which manages the index of the input nodes and its corresponding data.

```
12 PackedFunc set_input = vm_instance.GetFunction("set_input");
13 NDArray narray = NDArray::Empty(shape, dtype, ctx);
14 set_input("main", narray);
```

Listing 2.11: Add input data to the virtual machine instance for its execution.

Finally, execute the network by calling the main function using the `invoke` method, as shown in listing 2.12.

```
15 PackedFunc invoke = vm_instance.GetFunction("invoke");
16 invoke("main");
```

Listing 2.12: Relay VM network execution using the `invoke` packed function.

On the other hand, it is shown the network execution process of a compiled Relay IR network with the Graph Runtime strategy. Similar to the Relay VM approach, the packed functions are loaded from the dynamic library (`.so`) in the same manner through the `LoadFromFile` function.

Like the Relay VM strategy, to execute the computational graph (`.json`), it is required to create an instance of the runtime. This instance is established using the packed function `tvm.graph_runtime.create`, as demonstrated in listing 2.13. Additionally, the PowerVR device must be specified during the creation of the runtime instance.

```
1  using namespace tvm;
2  using namespace tvm::runtime;
3  Module mod_syslib = Module::LoadFromFile("deploy.so");
4
5  const PackedFunc *create = Registry::Get("tvm.graph_runtime.create");
6  Module runtime = (*create)(str_graph, mod_syslib, (int) kDLPowerVR, 0);
```

Listing 2.13: Creation of the graph runtime through the computation graph and the loaded packed functions module.

The next step is to load the network weight data parameters into the graph runtime instance. To achieve this, the parameters file (`.params`) is read as a `TVMByteArray` object and subsequently loaded into the runtime.

```
7  TVMByteArray params_arr;
8  params_arr.data = params_data.c_str();
9  params_arr.size = params_data.length();
10
11 PackedFunc load_params = runtime.GetFunction("load_params");
12 load_params(params_arr);
```

Listing 2.14: Load network weight data parameters into the graph runtime instance.

In listing 2.15, the input data for the network execution is managed using the `NDArray` object. This object provides the shape and data type to the `DLTensor` class, which populates the input tensor with the necessary data through the `set_input` packed function.

```
13 PackedFunc get_input = runtime.GetFunction("get_input");
14 PackedFunc set_input = runtime.GetFunction("set_input");
15
16 NDArray narray = get_input(0);
17 const DLTensor *tensor = in_array.operator->();
18 set_input("input", tensor);
```

Listing 2.15: Add input data to the graph runtime instance for its execution.

Lastly, the network is executed using the `run` packed function from the graph runtime instance, as illustrated in listing 2.16.

```
19 PackedFunc run = runtime.GetFunction("run");
20 run();
```

Listing 2.16: Graph Runtime network execution using the `run` packed function.

2.1.5 Data preprocessing

Data or input preprocessing is the process of converting an input into the format required by a specific model. The NC-SDK development tools include scripts to perform this data conversion. Generally, typical image datasets available do not present the data in the format required, usually specified by a number of parameters:

- **Image dimensions**: Contains the height, width, and a number of channels.
- **Image layout**: For a batch of images, the order in which each image value is stored is critical to being compatible with a specific network. The common layouts are NCHW and NHWC (its differences can be seen in figure 2.4).

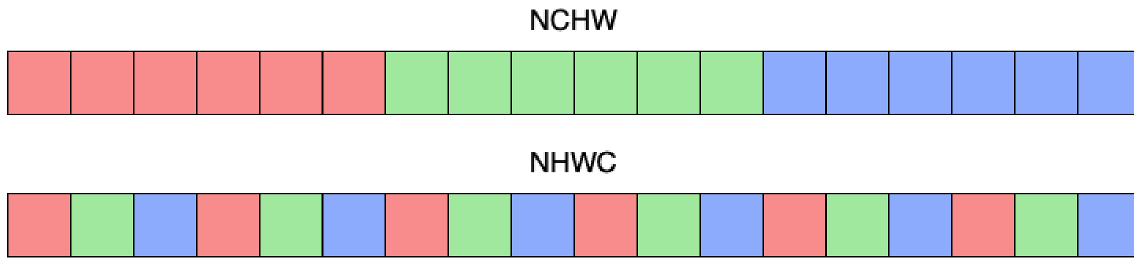

Figure 2.4: Comparison between NCHW and NHWC image layouts [2].

In the case of NCHW layout, for each image in the batch (N), pixels in the same channel (C) are stored close together. On the contrary, for the NHWC layout, pixels that correspond to close spatial positions of multiple channels are stored together. Apart from the format modification on an image, other usual preprocessing operations include value normalization and resolution modification.

2.2 Other edge computing packages

The NC-SDK package includes tools to optimize neural network models for AI inference on resource-constrained devices with PowerVR hardware (or other compatible hardware). However, there are other alternatives created by different organizations focused on their own hardware, such as TensorRT [25], OpenVINO [26], and Core ML [4].

This section will review the main features of these packages, which are highlighted in table 2.1. The NC-SDK package provides two distinct workflows for transforming and compiling neural networks for PowerVR devices. In contrast, other edge computing frameworks offer a single primary workflow for accelerating inference on edge devices.

Package	Compiler	Runtime	IR format	Hardware support
NC-SDK	TVM	TVM	Relay IR	PowerVR devices
	NNVM	IMGDNN	IMGIR	
TensorRT	TensorRT	CUDA	PLAN	NVIDIA devices
OpenVINO	NNCF	OpenVINO	OpenVINO IR	Intel devices
CoreML	Xcode	BNNS	Core ML model	Apple devices

Table 2.1: Main comparison between edge computing packages for optimizing deep learning models for AI inference.

For instance, TensorRT utilizes its own API to transform and compile networks that will be executed using CUDA [27] on edge devices. Similarly, OpenVINO employs its own API for executing optimized models and integrates NNCF [28] for model compression and quantization. On the other hand, Core ML relies on Xcode for compiling deep learning models and uses BNNS [29] to execute them on Apple devices.

2.2.1 TensorRT

TensorRT [25] is an ecosystem of APIs developed by NVIDIA for high-performance neural network inference. This package includes development tools for model optimization and an inference runtime that deliver low latency for production applications using NVIDIA hardware.

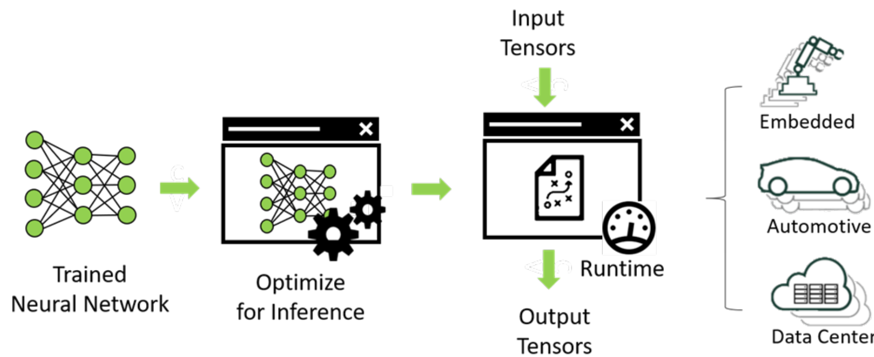

Figure 2.5: Workflow of optimization of a neural network using TensorRT [3].

TensorRT is built on top of CUDA [27], providing several model optimizations using techniques such as quantization, tensor fusion, or kernel tuning. TensorRT uses PLAN as an IR file format for transforming models from well-known ML frameworks for its optimization and execution on NVIDIA hardware (similar as the NC-SDK does for PowerVR devices). Figure 2.5 shows the workflow of optimization and inference of a neural network using this package.

2.2.2 OpenVino

OpenVINO [26] is an open-source toolkit developed by Intel to optimize and deploy deep learning models on Intel devices. This package can convert neural networks from a series of supported ML frameworks and deploy them across all Intel hardware. It offers model optimization as an optional offline step of improving the final model performance and reduces the model size by applying special optimization methods, such as post-training quantization, training-time optimization, or weight compression [30].

The workflow of OpenVINO is very similar to NC-SDK and TensorRT. The model compression and optimization is done by the NNCF [28], a suite of algorithms for optimizing inference of neural networks with a minimal accuracy drop. Then, its internal compiler transforms neural networks to OpenVINO IR file format, which can be executed by the OpenVINO inference runtime on Intel hardware. This runtime offers several APIs that abstract the low-level programming with each target device.

2.2.3 Core ML

Core ML [4] is an API developed by Apple to integrate ML models into its applications. This package optimizes on-device performance by leveraging CPU, GPU, and NNA while minimizing its memory footprint and power consumption. As well as other edge computing packages, Core ML uses several tools for creating and importing AI models into IR file format (`.mlmodel`) [31].

Figure 2.6: Workflow of optimization of a neural network using CoreML [4].

As shown in figure 2.6, Core ML models use Xcode compiler [32] to transform the Core ML model into a resource that's been optimized to run on-device. The optimized representation of the model is included in the bundle of the application. Then, it is executed using the BNNS library [29], a collection of functions that provides routines optimized for high performance and low energy consumption across all Apple platforms.

2.3 Inference scenarios on edge computing

After reviewing a series of edge computing packages for optimizing neural networks, it is also necessary to know how to infer different kinds of applications with simple performance metrics that satisfy all use cases. In order to enable representative testing of a wide variety of development tools, well-known ML benchmarks have defined four different scenarios when running inferences on deep learning models [14]:

- **Single-stream**: In this scenario, the model processes one input at a time by injecting a single query into the inference system, recording the completion time, and continuing to inject the next query. The performance metric of this scenario is the 90th percentile latency.
- **Multistream**: This scenario represents applications that can handle a stream of queries, but each query includes multiple inferences. The performance metric is the number of streams that the system supports while keeping the latency constraints.
- **Offline**: The offline scenario represents batch-processing applications where all data is immediately available and latency is unconstrained. For this scenario, it is sent a single query that the system is free to process in any order. The metric for this scenario is measured in samples per second.

- **Server**: This scenario, which is not applicable for edge computing, represents online applications where query arrival is random and the latency is important. The performance metric of this scenario indicates the queries per second achievable while meeting all constraints.

2.4 Hardware for edge computing

Edge computing applications need to use highly efficient systems to process large workloads. By using network optimization techniques, such as weight compression, parameter pruning, or quantization, it is possible to improve the energy efficiency by lowering the computing complexity and data volume.

However, there also exist specific hardware accelerator devices that can enhance the inference of deep learning models by optimizing the low-level operations done by ML algorithms [33]. Edge computing applications can be implemented through a Central Processing Unit (CPU), Graphics Processing Unit (GPU), Application-Specific Integrated Circuit (ASIC), and Field-Programmable Gate Array (FPGA) [34]:

- **CPU**: It's the general-purpose device of the system, which supports high-precision numerical representations that could improve prediction accuracy, at a cost of power consumption [35]. There exist several optimization techniques to improve the power management when executing deep learning tasks on this hardware.
- **GPU**: This device is originally designed for image processing and computer graphics acceleration. It offers a higher computing precision with a higher power consumption rate.
- **ASIC**: This type of hardware device uses several strategies to accelerate and increase the computation operations. Tensor Processing Unit (TPU) and Neural Processing Unit (NPU) are some of the devices included in this group, improving the execution latency but decreasing the model accuracy.
- **FPGA**: It offers a flexible and cost-effective hardware implementation, allowing a better balance of power consumption, response latency, and prediction accuracy.

2.4.1 PowerVR hardware architecture

PowerVR [16] is the graphics hardware family developed by Imagination Technologies. These devices optimize the power consumption of data transfers in the system memory. With the reduction in memory bandwidth use and the implementation of specific hardware optimizations, it enables the execution of applications at a higher performance than other graphics architectures [5]. PowerVR GPUs are one of the most popular GPU families used in commercial smartphones [36].

The architecture of the PowerVR devices is based on Tile Based Deferred Rendering (TBDR). The core design principle of this architecture is to keep the system memory bandwidth requirements of the graphics hardware to the bare minimum [5]. This architecture splits the per-tile rendering process into two main stages: Hidden Surface Removal (HSR) and deferred pixel shading. In this context, the geometry that is obscured by other geometry is detected and removed from the render. Therefore, certain geometries and textures will be fetched only if they have an impact on the final image [37].

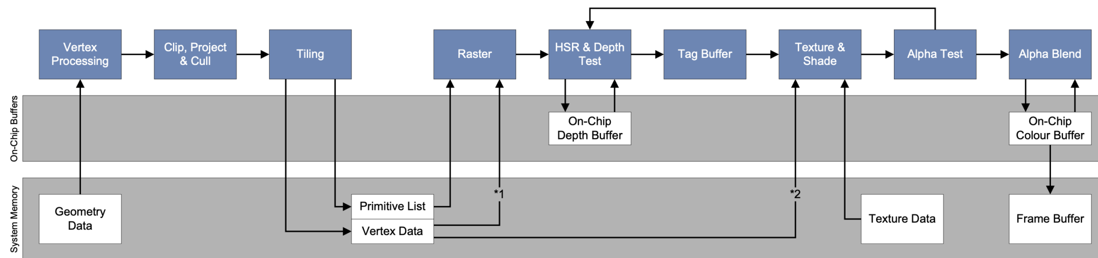

Figure 2.7: Tile Based Deferred Rendering (TBDR) architecture used in PowerVR devices [5].

Chapter 3

NC-SDK methodology and workflow

As explained in the previous chapter, the NC-SDK package includes a series of tools with the aim of converting and enhancing networks for running inferences on PowerVR devices. The process of transforming, optimizing, and compiling deep learning models is complex and involves multiple steps to achieve a successful deployment.

This chapter will review the main methodology of importing deep learning models from well-known ML frameworks, adapting each model to Relay IR format, transforming and optimizing the network for PowerVR hardware, and compiling the processed models for its execution on the target device.

As an example, the ResNet [38] model implemented in PyTorch [39] will be used to represent the NC-SDK workflow of adaptation, transformation, and compilation to execute on the deployment platform. For this purpose, this workflow defines the following main steps (see figure 3.1):

1. Convert pretrained deep learning models implemented with well-known ML frameworks to IR file format. In this process, the IR network is adapted by the NNVM compiler and transformed internally for the target device architecture.

2. Compile the Relay IR network files to generate executable binaries and dynamic libraries for PowerVR hardware by using the TVM compiler.

3. Transfer the compiled network binaries from the TVM compiler to the target device for deployment. This step also includes the transfer of IMGIR files to the deployment platform.

4. Execute the processed neural network with the TVM runtime library on the target hardware. This runtime environment communicates with the device drivers to execute the network on the specific PowerVR device. On the other hand, IMGIR files are executed using the IMGDNN runtime library.

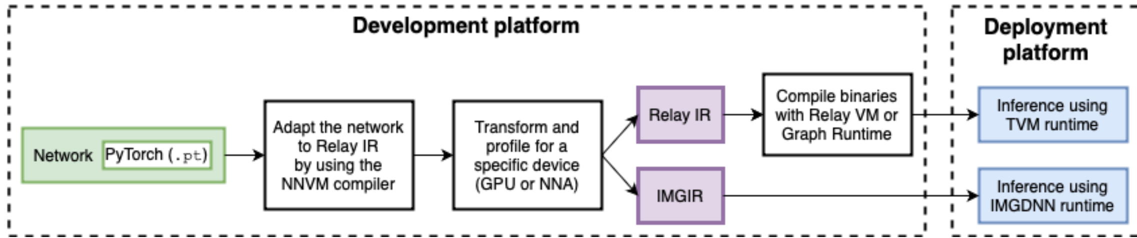

Figure 3.1: Main workflow of the NC-SDK development tools for adapting, transforming and compiling neural networks to inference on PowerVR hardware.

All steps related to modifying the neural network will be carried out on the development platform, while the execution and inference of each processed model will take place on the deployment platform:

- **Development platform**: It's a Docker container running the Ubuntu 20.04 x86_64 operating system, with a minimum of 4GB of RAM. The installation of the NC-SDK development tools is explained in appendix A.
- **Deployment platform**: This platform is the Beaglebone AI-64 board, which features a Texas Instruments Jacinto TDA4VM SoC with a dual-core 64-bit ARM Cortex-A72 microprocessor [17]. This board uses the PowerVR Rogue8XE GE8430 GPU [40] as a network acceleration device. The deployment package includes all the essential runtime binaries and libraries required for network inferences on this hardware.

Compiling a network for PowerVR devices using the NC-SDK development tools involves a four-step process: network adaptation, transformation and profiling, PowerVR compilation, and input preprocessing. Each of these steps is discussed in detail in the following sections.

3.1 Network adaptation

The adaptation stage converts deep learning models, originally implemented with well-known ML frameworks, into Relay IR format. This conversion ensures that the models are compatible with the TVM compiler, allowing for further optimization for the target hardware. Apart from generating Relay IR files, adapters also help understand the following aspects of the network [24]:

- **Unsupported layers**: Layers that are supported by the TVM backend but not by the IMGDNN runtime library are called unsupported. By default, the adapters generate an error if an unsupported layer is detected in the network.

- **Unimplemented layers**: Layers that are unsupported by both the TVM backend and the IMGDNN runtime library are called unimplemented. By default, the adapters generate an error if an unimplemented layer is detected in the network.
- **Custom layers**: It is possible to provide an implementation for an unimplemented layer or override an implementation of any existing layer by using custom layers.

Although the adapters convert neural networks into Relay IR, the network may still be in a format that is incompatible with the IMGDNN runtime library. Once the network is adapted into Relay IR, a series of graph transformations are applied to ensure that the Relay IR is suitable for compilation on PowerVR devices. These transformations, known as default transforms, may include the fusion of batch normalization layers for a particular PowerVR device.

Figure 3.2 illustrates the process of network adaptation from a standard ML framework to the Relay IR format. To convert from a PyTorch model (in this case, a ResNet implementation) to Relay IR, it is necessary to run the `pytorch_imgrelay.py` adapter script.

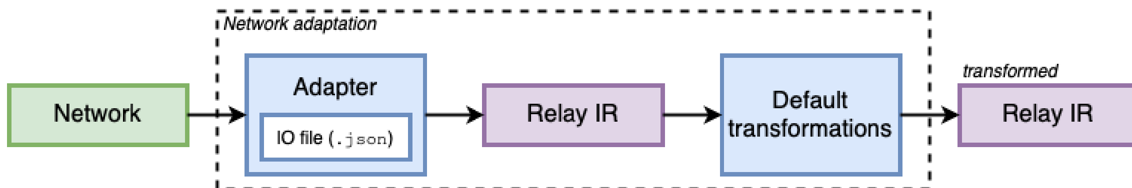

Figure 3.2: Network adaptation process from a PyTorch model to Relay IR format.

In this process, the adapter utilizes a network IO file that contains details about the model's inputs and outputs, facilitating the generation of the Relay IR. This file also allows the specification of shapes for unimplemented or custom layers.

```
1 [
2     {
3         "dtype": "float32",
4         "layout": "NCHW",
5         "name": "input",
6         "shape": [1, 3, 224, 224],
7         "type": "INPUT"
8     }
9 ]
```

Listing 3.1: ResNet IO file containing information about its input configuration.

In this scenario, listing 3.1 presents the ResNet IO file information, detailing the tensor types, image layout, shape, and data type. Before executing the network, it is essential to preprocess the input data. In section 3.4, we will prepare several input images to conform to the model's required layout, shape, and data type.

The following command executes the `pytorch_imgrelay.py` script: the `-i` argument specifies the PyTorch model file for the ResNet network, while the `-nf` argument denotes the ResNet IO file. The output of this script gives two files: the network in Relay IR format (`resnet.relayir`) and its associated weight data parameters (`resnet.relayir.params`).

```
1 pytorch_imgrelay.py -i resnet.pt -nf resnet_io.json
```

Listing 3.2: Command to adapt the PyTorch implementation of the ResNet model into Relay IR format.

3.2 Transformation and profiling

The transform and profile stage processes the Relay IR generated by the network adaptation scripts. During this stage, the Relay IR is annotated for microdevices, transformed according to the target hardware, and optionally profiled for the specific PowerVR device selected.

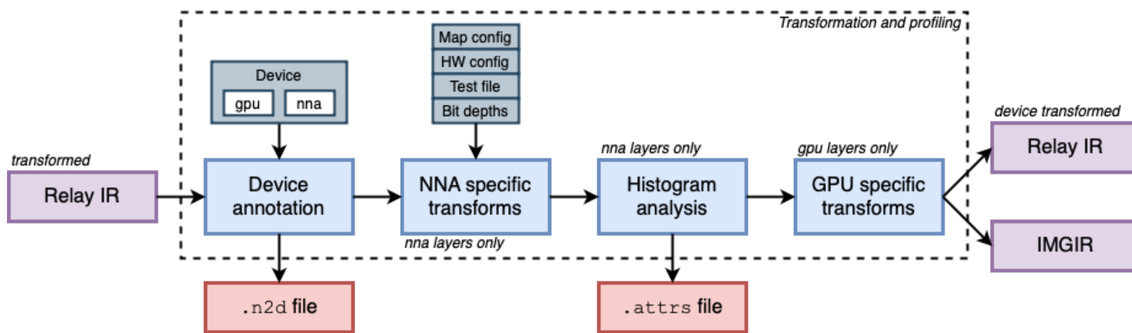

Figure 3.3: Network transformation process of the Relay IR network from the network adaptation script to a specific target device.

To compile a network for specific PowerVR devices, it is essential to identify the target device for each layer in order to generate the appropriate executable code. As illustrated in figure 3.3, one of the initial steps in the network transformation process is the device annotation task, which creates a device map assigning one of the following microdevices to each layer in the network: `nna`, `gpu`, `opencl`, or `cpu` [24]. The device annotation process generates a text file containing the mapping results, which the PowerVR compiler uses for graph partitioning and compilation. This file has the `.n2d` extension and is formatted as a JSON file.

Once the target device for each layer is determined in the device annotation process, device-specific transformations are applied to the entire graph. These transformations are crucial, not only for optimizing performance but also for ensuring compatibility between the Relay IR network and the network compiler. There are two types of network transformations:

- **Default transforms**: These transformations are run immediately after adapting the network and deal with general optimizations and incompatibilities between Relay IR and the PowerVR compiler.

- **Non-default transforms**: As the name suggests, these transforms are not called by default. To invoke non-default transformations, the `imgrelay_graph_transform.py` script can be used.

Another important step is profiling [24], which is mandatory for networks targeted to run on `nna` devices. Profiling is performed to determine the quantified exponents for each layer. As shown in figure 3.3, the script conducts a histogram analysis to obtain these exponents. The profiling options are specified using a JSON-formatted test file, which contains information about the sample data to be used during profiling. The profiling process generates an output file with `.attrs` extension, formatted as a JSON file. This file must be provided to the PowerVR compiler to compile the functions intended to run on `nna` devices.

Given the Relay IR files produced from the network adaptation process of the ResNet model, we will use the `imgrelay_transform_and_profile.py` tool to transform the Relay IR network for PowerVR GPU devices. The following arguments must be specified: `-i` for the input Relay IR file, `-o` for the output basename of the generated files, `-d` for the target device, and `-sn` to create the transformed IMGIR network as well.

```
1 imgrelay_transform_and_profile.py -i resnet.relayir\
2     -o resnet_gpu -d gpu -sn
```

Listing 3.3: Command of the ResNet Relay IR network transformation for PowerVR GPU.

The output of this script includes the device-transformed Relay IR network ready to compile for the target device (`resnet_gpu.relayir` and `resnet_gpu.relayir.params`), as well as the device mapping file (`resnet_gpu.n2d`) and the IMGIR network, which will be executed using the IMGDNN runtime on the deployment platform (`resnet_gpu.imgir` and `resnet_gpu.imgir.params`).

3.3 PowerVR compilation

The PowerVR compiler is in charge of compiling the device-transformed Relay IR network to generate executable code for the target PowerVR device. The main feature of this tool is the automatic graph partitioning. The PowerVR compiler utilizes the device annotation file to partition the IR graph into subgraphs for each microdevice available on the target hardware. Then, each function is compiled using the appropriate target compiler for its respective microdevice. The compilation strategies for each microdevice are outlined below [24], as illustrated in figure 3.4:

- `nna`: The compiler converts the Relay IR into IMGIR and invokes the `nna_compiler` tool to generate a binary file with MBS format. This MBS file is then packaged into a packed function that can be executed using the TVM Runtime.
- `gpu`: The compiler converts the Relay IR into IMGIR and bundles it into a packed function for runtime compilation and execution.
- `opencl`: The compiler generates OpenCL kernels from the Relay IR network, which are then packaged into a packed function for runtime compilation and execution.
- `cpu`: The TVM backend supports various LLVM IR-based target architectures, including Intel, ARM, and RISC-V.

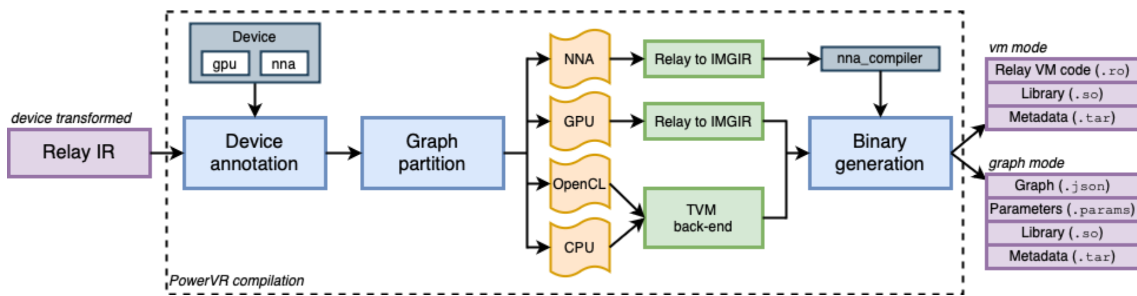

Figure 3.4: Relay IR network compilation for a specific target device.

As shown in figure 3.4, there are two compilation modes that are supported by the PowerVR compiler:

- **VM mode**: Relay VM uses instructions that enable execution of networks with control flow and dynamic tensors. The compiler generates three binary files: a file containing VM bytecode (`.ro`), a dynamic library (`.so`), and a metadata (`.tar`) file.
- **Graph mode**: The Graph Runtime is a fast execution experience, but only for static and sequential Relay IR networks. The compiler generates four files: a computation graph (`.json`), a dynamic library (`.so`) containing executable code, weight data parameters (`.params`), and a metadata (`.tar`) file.

The script used for compiling the device-transformed Relay IR network of the ResNet model is `powervr_build.py`. Listing 3.4 demonstrates the execution of this tool with the following arguments: `-i` to specify the input transformed Relay IR network, `-o` to define the basename for the output binaries, `-target` to indicate the target architecture for the executables, and `-d` to specify the PowerVR device for which the compilation is intended.

```
1 powervr_build.py -i resnet_gpu.relayir -o resnet_powervr.ro\
2     -target aarch64-linux-gnu -d gpu
```

Listing 3.4: Command of the compilation of the device-transformed ResNet Relay IR network for PowerVR GPU devices.

By default, the Relay IR network is compiled for Relay VM. The output files of the compiler are the Relay VM bytecode (`resnet_powervr.ro`), the dynamic library, which contains executable code (`resnet_powervr.so`), and the metadata (`resnet_powervr.tar`) file.

3.4 Input preprocessing

To perform inference on the compiled ResNet model on the deployment platform, the input data must be preprocessed to match the input tensor specifications described in listing 3.1. This can be achieved using the `preprocess.py` tool included in the NC-SDK package, which is specifically designed for data preprocessing.

This tool relies on a JSON-formatted file that outlines the preprocessing specifications. As shown in listing 3.5, these configurations include adjustments for input layout, data dimensions, and input scale normalization factors.

```
1 {
2     "input_offset": "128, 128, 128",
3     "raw_scale": 1.0,
4     "input_scale": 0.0078125,
5     "in_graph": false,
6     "input_layout": "NCHW",
7     "dimensions": "1, 3, 224, 224",
8     "ipp": "PIL"
9 }
```

Listing 3.5: ResNet preprocess file containing information about the input preprocessing.

The preprocessing script accepts the following arguments: `-id` to designate the input directory containing the raw data, `-o` to specify the output directory where preprocessed data will be saved, `-p` to indicate the configuration file, and `-t` to define the input data type for the model. The preprocessed data is saved in the output directory with a `.f32` extension if the specified data type is `float32`.

```
1 preprocess.py -id images -o images -p resnet_preprocess.json -t f32
```

Listing 3.6: Command to preprocess input data for the compiled ResNet network.

Chapter 4

Benchmark architecture and results

Before conducting a benchmark, it's essential to understand the architecture and carefully design the environment to ensure that the performance evaluation is both accurate and meaningful. For this task, the deployment platform described in chapter 3 will be utilized.

This chapter provides a review of the NC-SDK deployment package, a summary of the various models employed for the performance inference evaluation, and an in-depth analysis and assessment of the gathered results.

4.1 Experimental setup

This section explains in detail the hardware components of the Beaglebone AI-64, the deployment platform used to run inferences of optimized neural networks for this benchmark. This board features a Texas Instruments Jacinto TDA4VM SoC with a dual-core 64-bit ARM Cortex-A72 microprocessor with 4GB of RAM and a PowerVR Rogue8XE GE8430 GPU as a network acceleration device [17].

According to the documentation [6], the Beaglebone AI-64 is designed as a low-power, high-performance, and highly integrated device architecture, which improves the performance on processing power, graphics capability, and image processing. The Beaglebone AI-64 is capable of integrating vision hardware processing accelerators to facilitate extensive processing requirements in a low-power budget for ML applications.

With the 64-bit ARM Cortex-A72 microprocessor running at up to 2 GHz, it provides a general-purpose platform suitable for industrial automation, mobile robotics, building automation, and ML applications. It also can integrate hardware features that help applications achieve functional safety mechanisms [6]. Figure 4.1 illustrates the location of the main components of the Beaglebone AI-64:

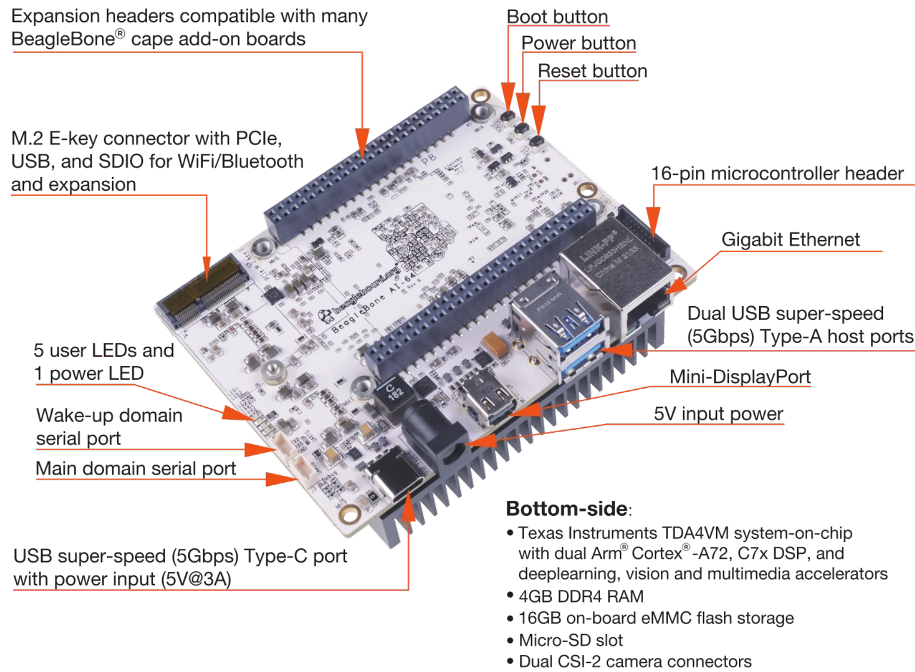

Figure 4.1: Main components of the Beaglebone AI-64 [6].

4.2 Benchmark design

The main goal of this benchmark is to include a representative set of deep learning models from various ML categories, such as image classification, object detection and segmentation, and language processing. To achieve this, these networks have been adapted, transformed, and compiled using the NC-SDK development tools on the development platform.

For the performance evaluation, all compiled networks will be executed on the PowerVR GPU of the deployment platform. Additionally, a comparison will be made using the host CPU of the Beaglebone AI-64 to highlight the performance differences between the GPU and CPU.

This section explores the experimental setup used in the benchmark, the ML models executed to retrieve the results, and the different runtime environments present on the deployment platform. Continuing with the example of chapter 3, we will represent the execution workflow of the processed ResNet model.

4.2.1 Categories and models

In the field of ML, different tasks are categorized based on their objectives and the nature of the data they process. In this case, by contributing to the goal of improving intelligent systems, each category employs various techniques to handle specific challenges and be able to improve the model with each execution.

The diversity of deep learning models with the NC-SDK package can be analyzed to explore various ML areas and their unique challenges. Due to the high number of ML categories, this study will focus on evaluating and analyzing the execution results of distinct neural networks of various ML categories, which include computer vision and language processing.

The most common ML area is computer vision. This category focuses on the interpretation and understanding of visual data. Some examples of applications that use this kind of deep learning model are autonomous driving, security surveillance systems, and visual recognition in healthcare. In this case, the performance benchmark will analyze and evaluate various AI models across three key vision tasks: image classification, object detection, and object segmentation.

- **Image classification**: It is a branch of computer vision that deals with categorizing images using a set of predetermined tags (also known as labels or classes) on which an algorithm has been trained. In the top-left example of figure 4.2, given a list of labels, the ML algorithm infers the probability if any of the classes are inside the image. In this work, we will be analyzing different image classification networks implemented in PyTorch:
  - ResNet [38]: ML model that uses residual learning to overcome the vanishing gradient problem, allowing it to train very deep neural networks. In this case, we will analyze the ResNet50 model.
  - Inception [41]: A deep convolutional neural network that uses multiple filter sizes in each layer of the neural network to capture various visual features at the same time, allowing to process images at different scales and reduce the computational costs. For this study, we will examine the InceptionV3 model.
  - MobileNet [42]: Is a compact, efficient convolutional neural network optimized for mobile devices, designed to provide high performance with low latency and reduced computational requirements. In this benchmark, we will utilize the MobileNetV3 model.
- **Object detection**: This computer vision task aims to identify and locate objects within digital images by combining object localization and classification. Object localization determines the position of specific objects by drawing bounding boxes around them, while object classification assigns a category to each detected object. In the bottom-left image of figure 4.2, it is shown the object localization algorithm by identifying and marking the bounding box around detected objects. For this benchmark, we will work with the following model:
  - YOLO [43]: This object detection model detects and classifies multiple objects in an image by dividing an input image into a grid and using a single forward pass of a neural network to predict bounding boxes and class probabilities. In this case, the model to use in the benchmark will be YOLOv5.

- **Object segmentation**: This computer vision task identifies the precise boundaries of objects in an image. Object segmentation works by partitioning an image into regions where each region corresponds to a specific object. For this category, the algorithms used classify each pixel in the image into distinct categories. The pictures on the right side of figure 4.2 illustrate two types of object segmentation: semantic segmentation, which differentiates between different classes of objects, and instance segmentation, which creates distinct instances for each object, even if they belong to the same class. In this study, we will analyze the following object segmentation models:
  - DeepLab [44]: This semantic segmentation model is designed to classify each pixel in an image into predefined categories. This model employs dilated convolutional layers to capture multiscale context and preserve spatial resolution. This approach enables precise delineation of object boundaries and enhances the model's performance. In this context, we will use the DeepLabV3 model using a ResNet50 backbone.
  - FCN [45]: It is a convolutional network architecture that replaces connected layers with convolutional layers, allowing the network to take input images of arbitrary size and produce output maps of the corresponding size. For this benchmark, we will analyze a FCN model with a ResNet50 backbone.
  - U-Net [46]: This convolutional neural network architecture is designed with an encoder-decoder structure that allows to capture high-level context as well as small details. The use of skip connections to map features from the encoder to the corresponding layers in the decoder makes U-Net a distinctive object segmentation model from other similar networks. U-Net is particularly effective for medical image segmentation.

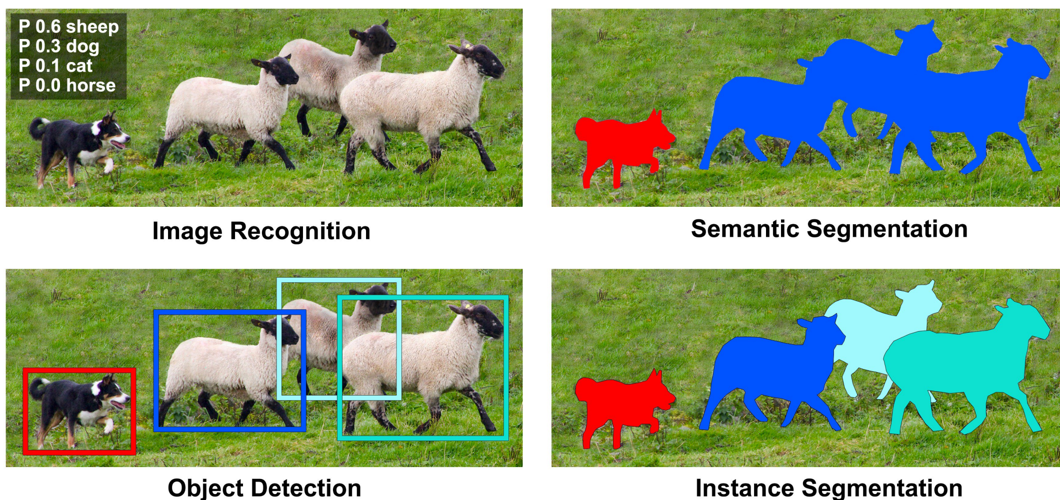

Figure 4.2: A sample annotated image from the COCO dataset, showing the difference between image classification, object detection, and object segmentation [7].

Another important area that we will be evaluating in this benchmark is language processing. This ML category enables computers to recognize, understand, and generate text by combining computational linguistics with statistical modeling. The most common applications of this category are customer assistance, grammar correction, and machine translation. For this purpose, this benchmark will be using BERT [47], a language model designed for natural language understanding tasks that take advantage of bidirectional context, considering both preceding and following words in a sentence, and fine-tunes on specific tasks like question answering or sentiment analysis.

4.2.2 Runtime environments

As described in chapter 3, the Beaglebone AI-64 deployment platform equipped with a PowerVR GPU will be used as the target hardware. The script for compiling the NC-SDK deployment package for the specified architecture is `prepare_deployment_package.sh`, which uses the `-c` argument to designate the target architecture.

```
1 bash prepare_deployment_package.sh\
2     -c ./deployment/source/aarch64/aarch64.cmake
```

Listing 4.1: Command to compile the deployment package for the `aarch64` architecture.

Listing 4.1 shows that the target architecture of the deployment platform is `aarch64`. However, the NC-SDK development tools can compile the deployment package into other architectures, such as `x86_64`. It is also possible to cross-compile for non-present architectures by adding the necessary assets to the deployment folder.

The compiled deployment package (`NC-SDK_Deployment_*.run`) is then copied to the deployment platform. Its execution will unzip all the deployment components to the `ncsdk` directory and automatically call the `deployment_install.sh` script to install libraries and prepare the deployment tools.

- **TVM runtime with `powervr_execute`**: It is an example application that uses the TVM runtime API to execute compiled Relay IR networks on PowerVR hardware, using both Relay VM and Graph Runtime binaries. Additionally, this application interacts with device drivers via the IMGDNN runtime API to configure input and output buffers and facilitate the execution of layers on the target device.
- **IMGDNN runtime with `nnvm_testbench`**: This application is designed to perform inference of IMGIR networks on PowerVR devices using the IMGDNN runtime library. By using the argument `--list-devices`, you can execute the application to list all available PowerVR devices on the deployment platform.
- **Examples**: The deployment package includes IMGDNN examples of various ML categories to execute PowerVR-compiled networks on the deployment platform. It also includes the `imgdnn_test` binary to test the PowerVR hardware.

Building on the example process of network transformation, optimization, and compilation of the ResNet model in chapter 3, this section will demonstrate how to perform inference on the deployment platform using both TVM and IMGDNN runtime libraries. Figure 4.3 illustrates the workflow for network inference on PowerVR GPU and host CPU devices used for this benchmark.

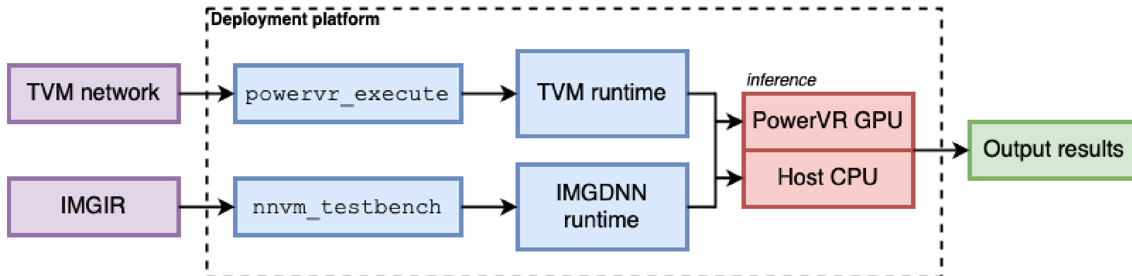

Figure 4.3: Network inference on the deployment platform using both TVM and IMGDNN runtime libraries.

The `powervr_execute` application is used to run TVM networks (either Relay VM or Graph Runtime binaries) on PowerVR hardware. Listing 4.2 details the arguments for this application, which include: `-c` to specify the Relay VM bytecode or Graph Runtime computational graph, `-i` to provide the input name and path to the input file, and optionally, `-n` to specify the number of inferences to execute. The output results of the ResNet network inference contain the adjusted probability for each class in `float32` format (it is needed an external script to visualize the output).

```
1 powervr_execute -c resnet_powervr.ro\
2     -i input:image_preprocessed.f32 -n 16
```

Listing 4.2: ResNet compiled network inference of a preprocessed image using Relay VM.

On the other hand, the `nnvm_testbench` application runs IMGIR networks on PowerVR devices using the IMGDNN runtime library. The required arguments for this binary are: `-n` to specify the IMGIR network file, `-p` for the network parameters file, and `-d` to provide the input name and path to the input file. Similar to the `powervr_execute` application, the output results include the probability for each class in `float32` format.

```
1 nnvm_testbench -n resnet_gpu.imgir -p resnet_gpu.imgir.params\
2     -d input:image_preprocessed.f32 -l 16
```

Listing 4.3: ResNet processed IMGIR network inference of a preprocessed image.

Additionally, the `torch_execute.py` script has been created to infer PyTorch network implementations using the host CPU on the deployment platform. This script is used in the benchmark to compare the performance of NC-SDK networks with those from standard frameworks (in this case, PyTorch).

4.3 Evaluation and analysis

For the evaluation and analysis of the performance benchmark of optimized neural networks using the NC-SDK development tools, the models outlined in section 4.2.1 will be executed and inferred under a single-stream scenario. In this case, the metric used in this scenario is the 90th percentile over 1024 queries.

For this purpose, the `powervr_execute` and the `nnvm_testbench` NC-SDK deployment example applications have been modified to facilitate the analysis of the results. In addition, a warm-up phase of 32 queries has been introduced to mitigate longer inference times associated with loading the network into the device memory. The benchmark includes four distinct network inference categories:

- TVM runtime execution on the PowerVR GPU device using the modified application `powervr_execute`.
- TVM runtime execution on the host CPU by using the application `powervr_execute` with the argument `-t`.
- IMGDNN inference on the PowerVR GPU device by using `nnvm_testbench` binary.
- PyTorch execution on the host CPU with the `torch_execute.py` script.

The first experiment shows the inference times of the image classification networks across the four inference categories. Figure 4.4 demonstrates that networks executed on the PowerVR GPU using the TVM runtime process inputs at least eight times faster than those executed on the host CPU. Furthermore, networks executed with PyTorch on the CPU are nearly 90% faster than those using the TVM runtime on the host CPU. Among the deep learning models tested, the MobileNetV3 network demonstrates significantly shorter inference times than the ResNet50 and InceptionV3 networks due to its optimization for mobile devices.

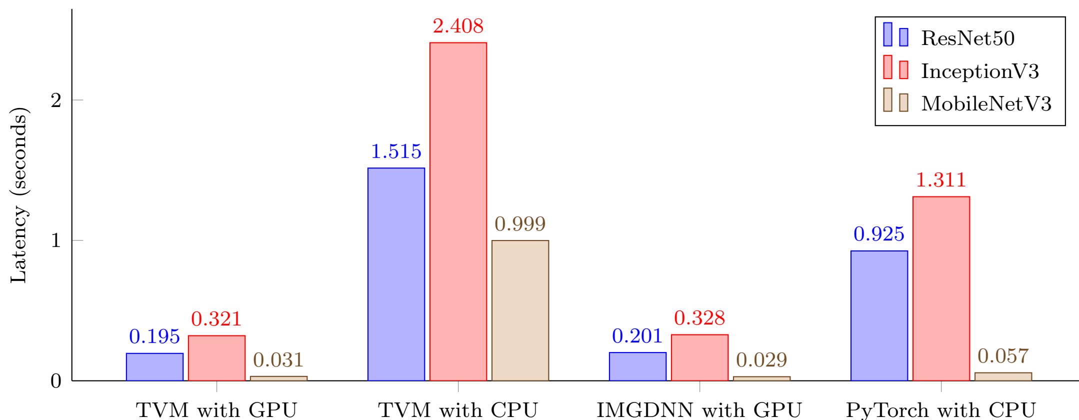

Figure 4.4: Inference times of image classification models on the deployment platform.

On the other hand, the second experiment presents in figure 4.5 the inference times for the YOLOv5 network, an object detection model. Similar to the results for image classification networks, this plot shows minimal differences in GPU performance when comparing the execution of the compiled Relay IR network with the TVM runtime library to the transformed IMGIR network using the IMGDNN runtime API. This consistency across different model types indicates that the performance that both runtime libraries deliver is very similar on PowerVR hardware.

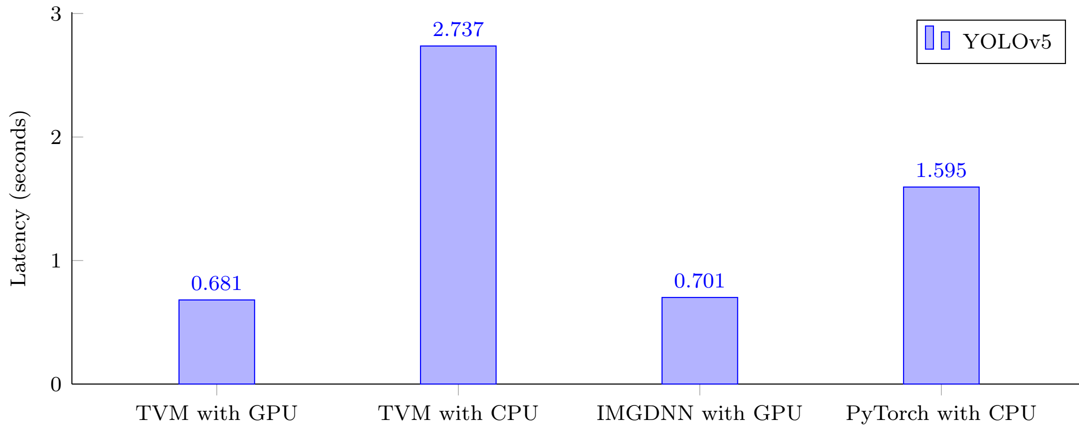

Figure 4.5: Inference times of object detection model on the deployment platform.

The third experiment shows the inference times of DeeplabV3, FCN, and U-Net object segmentation models. As seen in the experiment, this category of deep learning models experiences longer inference times than other categories due to the larger size of the model and their inputs. However, as illustrated in figure 4.6, the use of PowerVR hardware results in a reduction of up to 600% in inference times compared to running the same models on the host CPU when using the TVM runtime library. It is also comparable to the results on the host CPU of the Beaglebone AI-64, where the TVM runtime library execution performs worse than the execution of PyTorch implementations on the same device across all experiments.

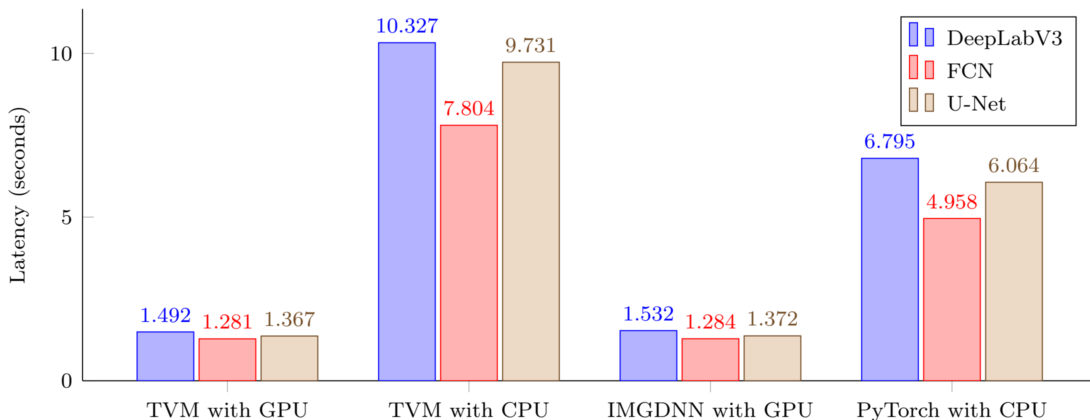

Figure 4.6: Inference times of object segmentation models on the deployment platform.

Finally, plot 4.7 represents the fourth experiment with the inference times for executing the natural language processing BERT network. As well as other evaluated neural networks, this model shows a substantial performance enhancement in inference speed when run on the PowerVR hardware. Inference times are also greater than other ML models due to the larger model size.

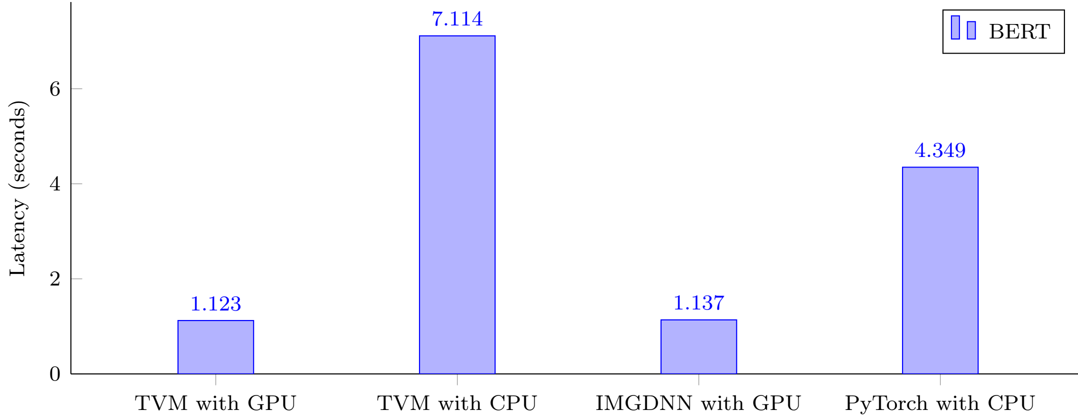

Figure 4.7: Inference times of language processing model on the deployment platform.

With all evaluated models across various inference categories, the results clearly highlight the significant influence that hardware-specific neural network optimizations have on inference performance. The inference results show that the use of PowerVR hardware can lead to substantial performance improvements across a wide array of deep learning models.

Table 4.1 provides a detailed summary of the measurements recorded during this benchmark, offering an in-depth overview of performance across different models and hardware configurations. It indicates the model used in the performance benchmark, the associated ML category, and the inference times retrieved from the results of the four inference categories.

Model	**Task**	**TVM (GPU)**	**TVM (CPU)**	**IMGDNN (GPU)**	**PyTorch (CPU)**
ResNet50	Image classification	0.195s	1.515s	0.201s	0.925s
InceptionV3	Image classification	0.321s	2.408s	0.328s	1.311s
MobileNetV3	Image classification	0.031s	0.999s	0.029s	0.057s
YOLOv5	Object detection	0.681s	2.737s	0.701s	1.595s
DeepLabV3	Object segmentation	1.492s	10.327s	1.532s	6.795s
FCN	Object segmentation	1.281s	7.804s	1.284s	4.958s
U-Net	Object segmentation	1.367s	9.731s	1.372s	6.064s
BERT	Language processing	1.123s	7.114s	1.137s	4.349s

Table 4.1: Table with all inference times of the performance benchmark, indicating the model and its ML task associated.

These findings emphasize the importance of choosing the appropriate edge computing framework and hardware configurations to optimize the efficiency and speed of neural network inferences. This benchmark demonstrates that using device-specific compiled models is essential for the deployment of applications in resource-constrained environments. By investing in network optimization tools, it is possible to improve various ML applications on edge devices while maintaining high performance.

Chapter 5

Conclusion and future work

The demand for edge computing applications has been increasing over the past few years, enabling data to be processed locally, reducing inference latency, and enhancing the speed of decision-making systems without the need to access a centralized cloud server. In this context, various edge computing frameworks have been developed to optimize the execution of deep learning models on edge devices, improving their performance in resource-constrained environments.

This study has focused on the use of the Neural Compute Software Development Kit (NC-SDK), a suite of development tools designed to facilitate the deployment of edge computing applications by adapting, transforming, and compiling common neural network implementations to run inferences on edge devices equipped with a PowerVR GPU. This workflow has been carried out with various ML categories, including image recognition, object detection and segmentation, and language processing.

NC-SDK also supports network cross-compilation, allowing deep learning models to be deployed on various hardware architectures. In the world of IoT, with a high number of heterogeneous devices (where devices can range from high-performance hardware to more constrained configurations), this advantage is very beneficial for edge computing applications. Additionally, the NC-SDK package is also capable of deploying neural networks in different scenarios. With the use of runtime environments like TVM and IMGDNN, it is possible to improve the network performance by executing device-compiled deep learning models on specific target devices. This workflow is much more efficient compared to using general-purpose hardware to infer ML models.

The evaluation and analysis of the performance benchmark results in chapter 4 shows the advantages of using PowerVR GPUs to accelerate and improve the inference times of neural network execution on edge devices. These results represent the importance of selecting the best suitable edge computing framework with the most efficient hardware configuration to maximize the network performance and inference speed in resource-constrained environments.

In summary, the NC-SDK package offers a robust framework for adapting, compiling, and deploying deep learning models on edge devices. As the demand for real-time decision-making systems across various applications grows, these neural network development tools become indispensable for enhancing the performance, speed, and responsiveness of ML models.

5.1 Further customization of neural networks

One of the possible topics to research in the future is the further customization of deep learning models using the NC-SDK development tools. With the integration of custom layers, it is possible to focus on expanding the capabilities of this edge computing framework in order to support different network architectures. The customization of neural networks can introduce new optimization techniques that can take advantage of various ML models.

Focusing on a single model type and evaluating its performance with various custom-made layers across different devices will provide deeper insights into the capabilities and limitations of the NC-SDK toolkit. Additionally, experimenting with other PowerVR hardware, such as the NNA, will help identify which layers are most effective for a specific device, improving the overall model performance.

5.2 Exploration of other ML areas

Another possible future work could be the exploration of the full capabilities of the NC-SDK package in other ML areas, such as audio processing. For this task, it would be needed to research how the toolkit handles different strategies, such as speech recognition, sound classification, or audio signal processing. This investigation could reveal the versatility of this edge computing framework across a wide range of applications.

The exploration of other ML areas could demonstrate the adaptability of the NC-SDK development tools in various contexts, improving the overall utility of this package by uncovering new optimization techniques and use cases for edge devices. On the other hand, by looking into the low-level network architecture, it is possible to find more interesting topics for newly developed custom networks.

5.3 Comparation of edge computing packages

An additional area of interest for future work involves benchmarking the inference performance of various neural networks using other edge computing frameworks, such as TensorRT, OpenVINO, or Core ML. This comparison could establish the NC-SDK package as a leading choice among other edge computing frameworks and show the performance increase optimizations of the PowerVR hardware.

Therefore, it is needed to ensure a consistent comparison across different software and hardware configurations. To achieve this, it is necessary to follow a set of standard rules and guidelines, such as those presented in the MLPerf Inference Benchmark Suite. These guidelines will help to create a robust benchmark in which the results reflect the exact performance of various edge computing frameworks.

Bibliography

[1] [Online]. Available: https://drive.google.com/drive/folders/1QimqavBN0O5Gl85BCGvkpkvIsSrYOgX1?usp=share_link (accessed on Sep. 5, 2024).

[2] Imagination, "Introduction to NC-SDK," Private documentation, 2022.

[3] Devstack, "Inference optimization using TensorRT." [Online]. Available: https://www.devstack.co.kr/inference-optimization-using-tensorrt/ (accessed on Aug. 17, 2024).

[4] Apple, "Core ML." [Online]. Available: https://developer.apple.com/documentation/coreml (accessed on Aug. 17, 2024).

[5] Imagination Technologies, "PowerVR Hardware Architecture Overview for Developers." [Online]. Available: http://powervr-graphics.github.io/WebGL_SDK/WebGL_SDK/Documentation/Architecture%20Guides/PowerVR%20Hardware.Architecture%20Overview%20for%20Developers.pdf (accessed on Sep. 5, 2024).

[6] Beagleboard, "Beaglebone AI-64," June 2022. [Online]. Available: https://git.beagleboard.org/beagleboard/beaglebone-ai-64 (accessed on Sep. 5, 2024).

[7] R. Tedrake, *Robotic Manipulation*. MIT, 2024. [Online]. Available: http://manipulation.mit.edu (accessed on Aug. 25, 2024).

[8] K. Cao, Y. Liu, G. Meng, and Q. Sun, "An overview on edge computing research," *IEEE access*, vol. 8, pp. 85 714–85 728, 2020.

[9] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[10] J. Chen and X. Ran, "Deep learning with edge computing: a review," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1655–1674, 2019.

[11] H. Hua, Y. Li, T. Wang, N. Dong, W. Li, and J. Cao, "Edge computing with artificial intelligence: a machine learning perspective," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.

[12] K. Lee, V. Rao, and W. C. Arnold, "Accelerating Facebook's infrastructure with application-specific hardware," *Facebook. Retrieved August*, vol. 20, p. 2020, 2019.

[13] T. Wu, S. He, J. Liu, S. Sun, K. Liu, Q.-L. Han, and Y. Tang, "A brief overview of ChatGPT: the history, status quo and potential future development," *IEEE/CAA Journal of Automatica Sinica*, vol. 10, no. 5, pp. 1122–1136, 2023.

[14] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou *et al.*, "MLPerf inference benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 446–459.

[15] Imagination Technologies, "Neural Compute SDK," February 2023. [Online]. Available: https://developer.imaginationtech.com/solutions/neural-compute-sdk/ (accessed on Aug. 11, 2024).

[16] Wikipedia, "PowerVR." [Online]. Available: https://en.wikipedia.org/wiki/PowerVR (accessed on Aug. 14, 2024).

[17] Beagleboard, "Beaglebone AI-64," June 2022. [Online]. Available: https://www.beagleboard.org/boards/beaglebone-ai-64 (accessed on Aug. 11, 2024).

[18] Anonymous, "Intermediate representations." [Online]. Available: https://cs.lmu.edu/~ray/notes/ir/ (accessed on Aug. 15, 2024).

[19] AWS AI team, "NNVM Compiler: Open Compiler for AI Frameworks," Paul G. Allen School of Computer Science & Engineering, University of Washington, Tech. Rep., October 2017. [Online]. Available: https://tvm.apache.org/2017/10/06/nnvm-compiler-announcement (accessed on Aug. 14, 2024).

[20] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: end-to-end optimization stack for deep learning," *arXiv preprint arXiv:1802.04799*, vol. 11, no. 2018, p. 20, 2018.

[21] A. Munshi, "The OpenCL specification," in *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, 2009, pp. 1–314.

[22] J. Roesch, S. Lyubomirsky, M. Kirisame, J. Pollock, L. Weber, Z. Jiang, T. Chen, T. Moreau, and Z. Tatlock, "Relay: a high-level IR for deep learning," *arXiv preprint arXiv:1904.08368*, 2019.

[23] Imagination, "NC-SDK Graph Transforms," Private documentation, February 2023.

[24] Imagination, "NC-SDK Compilation And Inference," Private documentation, February 2023.

[25] NVIDIA, "NVIDIA TensorRT." [Online]. Available: https://developer.nvidia.com/tensorrt (accessed on Aug. 17, 2024).

[26] Intel, "OpenVINO." [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/openvino-toolkit/overview.html (accessed on Aug. 17, 2024).

[27] NVIDIA, "CUDA Toolkit." [Online]. Available: https://developer.nvidia.com/cuda-toolkit (accessed on Aug. 17, 2024).

[28] Intel, "Neural Network Compression Framework (NNCF)." [Online]. Available: https://github.com/openvinotoolkit/nncf (accessed on Sep. 2, 2024).

[29] Apple, "BNNS." [Online]. Available: https://developer.apple.com/documentation/accelerate/bnns (accessed on Sep. 2, 2024).

[30] Intel, "OpenVINO model optimization guide." [Online]. Available: https://docs.openvino.ai/2024/openvino-workflow/model-optimization.html (accessed on Aug. 17, 2024).

[31] Apple, "Getting a Core ML model." [Online]. Available: https://developer.apple.com/documentation/coreml/getting-a-core-ml-model (accessed on Aug. 17, 2024).

[32] ——, "Xcode." [Online]. Available: https://developer.apple.com/xcode/ (accessed on Aug. 17, 2024).

[33] Wikipedia, "AI accelerator." [Online]. Available: https://en.wikipedia.org/wiki/AI_accelerator (accessed on Sep. 5, 2024).

[34] L. Martin Wisniewski, J.-M. Bec, G. Boguszewski, and A. Gamatié, "Hardware solutions for low-power smart edge computing," *Journal of Low Power Electronics and Applications*, vol. 12, no. 4, p. 61, 2022.

[35] M. Capra, R. Peloso, G. Masera, M. Ruo Roch, and M. Martina, "Edge computing: a survey on the hardware requirements in the Internet of Things world," *Future Internet*, vol. 11, no. 4, p. 100, 2019.

[36] S. De Dominicis, "A new era in mobile GPUs: the increasing relevance of energy efficiency, security and virtualization," in *2016 Mobile System Technologies Workshop (MST)*. IEEE, 2016, pp. 10–11.

[37] B. Jákó, "Hardware accelerated hybrid rendering on PowerVR GPUs," in *2016 IEEE 20th Jubilee International Conference on Intelligent Engineering Systems (INES)*. IEEE, 2016, pp. 257–262.

[38] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[39] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," in *NIPS-W*, 2017.

[40] Imagination Technologies, "GPU (Graphics Processing Unit)." [Online]. Available: https://www.imaginationtech.com/products/gpu/ (accessed on Aug. 11, 2024).

[41] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[42] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan *et al.*, "Searching for MobileNetV3," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019, pp. 1314–1324.

[43] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[44] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam, "Rethinking atrous convolution for semantic image segmentation," *arXiv preprint arXiv:1706.05587*, 2017.

[45] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3431–3440.

[46] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: convolutional networks for biomedical image segmentation," in *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18.* Springer, 2015, pp. 234–241.

[47] J. Devlin, "Bert: pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

Appendix A

Setting up NC-SDK development environment with Docker

This appendix explains the installation of the Neural Compute Software Development Kit (NC-SDK) development tools in a Docker container. The development platform is required to be an Ubuntu 20.04 with an x86-64 architecture (or a similar supported operating system).

In order to create a container that executes the NC-SDK development tools, it's essential to build a Dockerfile to create a software image of the development platform. This image inherits an Ubuntu 20.04 image with a `/bin/bash` shell. In addition, the argument `DEBIAN_FRONTEND` is set up to `noninteractive` to speed up the NC-SDK package installation process.

```
1 FROM amd64/ubuntu:20.04
2 ARG DEBIAN_FRONTEND=noninteractive
3 SHELL ["/bin/bash", "-c"]
```

Listing A.1: Preliminary settings of the Dockerfile to create the development platform.

All software prerequisites for the NC-SDK development environment are installed in the next layer of the Dockerfile. These prerequisites are essential for installing the NC-SDK development tools, as well as generating the deployment package for common target archictectures on the development platform.

```
5 RUN apt update\
6     && apt install -y vim wget libgl1 libxml2-dev libtinfo-dev\
7         libncurses5 libz-dev build-essential\
8     && apt install -y gcc make gcc-aarch64-linux-gnu\
9         binutils-aarch64-linux-gnu g++-aarch64-linux-gnu
```

Listing A.2: Installation of prerequisites of the NC-SDK package in the Dockerfile.

Finally, the NC-SDK development installer is copied and installed in the `ncsdk` directory. It's necessary to have the offline Conda libraries tarball, as it is required for the installation and execution of the NC-SDK development tools.

```
10 COPY *DevTools.run .
11 COPY ncsdk_conda_env_offline.tar.gz .
12 RUN chmod +x *DevTools.run && ./*DevTools.run --accept
```

Listing A.3: Installation of the NC-SDK package in the Dockerfile.

Listing A.4 shows the commands to build the Dockerfile and run a container from the built image in detached mode. It's mandatory to have in the same directory the offline Conda libraries tarball (`ncsdk_conda_env_offline.tar.gz` and the installer of the NC-SDK development package (`*DevTools.run`). This task can take up to twenty minutes (a working internet connection is required at this point).

```
1 docker build -t ncsdk-dev .
2 docker run --name ncsdk-dev -h ncsdk-dev -dit ncsdk-dev
```

Listing A.4: Execution the `ncsdk-dev` container image from the Dockerfile.

Once the container is up and running, we can access to its shell at any given moment with the command of listing A.5.

```
1 docker exec -it ncsdk-dev bash
```

Listing A.5: Command to open a shell inside the `ncsdk-dev` container.

To verify the installation, you can run the installer verification tests using the `-v` argument, as shown in listing A.6. After the tests are complete, the results should indicate that all tests have passed successfully, confirming that the NC-SDK development platform is ready to use.

```
1 /ncsdk/img_ndk_tools_install -v
```

Listing A.6: Verification script of the NC-SDK development tools.