Faculty of Engineering, Architecture and Information Technology
Technische Universität München

TUM

SIEMENS

Design Space Exploration for Efficient Inference of Machine Learning Models on Heterogeneous Hardware Devices

Ines Ben Hmida

Industrial Project Master's Thesis in Information Technology and Electrical Engineering

Chairman:
Prof. Dr.-Ing. habil. Daniel Müller-Gritschneder, Technical University of Munich

Supervisor
Alexander Hoffman M.E, M.Sc, Technical University of Munich

Second Supervisor
Dr. Andreas Mucha, Siemens AG

I confirm that this Master's thesis is my work with my supervisors' valuable contribution, and all sources and materials that I used are documented.
Ines Ben Hmida

Abstract

In recent years, machine learning methods have proven their efficiency and applicability in a broad range of fields and environments. However, the complexity and high computational density they require makes their implementation challenging, particularly in constrained environments like embedded systems. Therefore, on-going research has aimed at accelerating the execution of machine learning methods. Some solutions presented in research involve the use of design space exploration (DSE).

In this thesis, a method to accelerate the inference of machine learning models by applying state-of-the-art DSE techniques is presented.

By representing a machine learning model as an application, and the hardware it is deployed on as the architecture, optimally distributing its execution to accelerate it becomes a system-design level synthesis problem. This problem is modeled according to the Y-chart methodology, which allows the comparison of design points quantitatively according to performance data. Evaluating all design points is challenging when dealing with such complex algorithms. Hence, the design space is explored with an efficient algorithm selected from the existent methods used in DSE. The problem is then represented as an optimization for system-level DSE. The results of the optimization reside in optimal distributions of machine learning workloads to the target hardware architecture. Using SAT-Decoding with an evolutionary algorithm, each proposed solution is guaranteed to be valid and highly optimal.

An experimental evaluation was performed as a proof of concept of the theory. The evaluation of each design point during the DSE is carried out with a function computing an estimation of the overall execution time. This estimation is deduced from cost models created by performing benchmarks of operations encapsulated in machine learning models on the hardware.

The design space is represented and explored using existing frameworks. The resulting solutions generated by the DSE present optimal mappings, which could accelerate the inference if implemented.

Thus, it was shown that the application of design space exploration, specifically the Y-chart methodology, at a system-design level, to optimize the distribution of machine learning workloads across heterogeneous hardware systems leads to the acceleration of machine learning models' execution.

Contents

Abstract iii

Contents v

List of Figures ix

List of Tables xiii

Acronyms xv

1 Introduction 1
1.1 Motivation and Problem Statement . . . 1
1.1.1 Motivation . . . 1
1.1.2 Problem Statement . . . 1
1.2 Thesis Outline . . . 2
1.3 Contribution . . . 2

2 State of the Art 3

3 Preliminaries 5
3.1 Machine Learning . . . 5
3.1.1 Generalized Machine Learning . . . 5
3.1.2 Inference . . . 6
3.1.3 Deep Learning . . . 7
3.1.4 Machine Learning Models' Creation . . . 9
3.1.4.1 Introduction . . . 9
3.1.4.2 TensorFlow Models Creation as an example . . . 9
3.1.4.2.1 Introduction to TensorFlow's Programming Model . . . 10
3.1.4.2.2 Models Creation in TensorFlow . . . 10
3.1.4.2.3 Models Creation in TensorFlow Lite . . . 11
3.1.5 Artificial Intelligence Hardware Accelerators . . . 11
3.2 Design Space Exploration . . . 12
3.2.1 General Presentation . . . 12
3.2.2 Formal Definition . . . 12
3.2.3 The Y-chart Methodology to design programmable embedded systems . . . 13
3.2.4 Methods in DSE . . . 16
3.2.5 System-Level-Synthesis for DSE . . . 16
3.2.6 General Presentation of the Dataflow Graph Model of Computation . . . 17

4 Design Space Exploration for Hardware Accelerated Inference Distribution 19
4.1 Overview of DSE as a Solution to Distributed Inference . . . 19
4.1.1 Overview of the Problem . . . 19

4.1.2 Distribution of ML Models' Inference . . . 21
4.1.3 Application Specific Similar Example: The TensorFlow Placement Algorithm 21
4.2 Modeling of the Solution: Representation of the Problem According to the Y-chart Methodology . . . 23
4.2.1 Presentation of the Design Space . . . 23
4.2.2 Specification Formulation . . . 26
4.3 Optimization . . . 26
4.3.1 Optimization Problem Formulation . . . 27
4.3.2 The Fitness Function . . . 28
4.3.3 Choice of the DSE Algorithm . . . 28

5 Experimental Setup 33
5.1 Hardware setup . . . 33
5.1.1 Used AI Hardware Accelerators . . . 33
5.1.1.1 GPU . . . 33
5.1.1.2 NCS . . . 34
5.1.1.3 Coral . . . 35
5.2 Machine Learning Benchmarking . . . 37
5.2.1 Introduction to the Benchmarking Environment . . . 37
5.2.2 Benchmarking Tool Description and Setup . . . 37
5.2.3 Benchmarking Individual Operation Models . . . 40
5.3 DSE . . . 41
5.3.1 Used DSE Frameworks . . . 41
5.3.1.1 OpenDSE . . . 41
5.3.1.2 Opt4j Stack . . . 41
5.3.2 Implementation of the DSE . . . 41
5.3.2.1 Establishment of the design space according to the Y-chart methodology with OpenDSE . . . 41
5.3.2.2 Exploration with OpenDSE and Opt4j stack . . . 43
5.3.2.2.1 Presentation of the workflow of the Exploration with the used frameworks . . . 43
5.3.2.2.2 Encoding of the Specification by OpenDSE and creation of the genotypes . . . 45
5.3.2.2.3 Decoding . . . 45
5.3.2.2.4 Evaluating the Specification . . . 46
5.4 Assumptions . . . 47

6 Results 51
6.1 Benchmarking Results . . . 51
6.1.1 Results of Benchmarking Individual Operation Models . . . 51
6.1.2 Results Analysis and Establishment of cost models from the previous Results 57
6.2 Results of the Design Space Exploration . . . 60
6.2.1 Application of Design Space Exploration Methods to a Small TensorFlow Model . . . 60
6.2.1.1 Presentation of the Context . . . 60
6.2.1.2 Presentation of the Results . . . 62

7 Evaluation of the Design Space Exploration Outcome for Distributed ML Inference 65
7.1 Evaluation of the Results for Design Space Exploration Application on a Small TensorFlow Model . . . 65
7.2 Conclusions of the Evaluation and Discussion . . . 68

8 Conclusion 71

9 Future Work 73
9.1 Future Work in the Experimentation . . . 73
9.1.1 Applying the Design Space Exploration to a larger TensorFlow Machine Learning Model- Mobilenet as an example . . . 73
9.1.1.1 Context . . . 73
9.1.1.2 Primary Test Results Obtained with a Mobilenet TensorFlow MModel . . . 73
9.1.2 Experimental Setup in a more Constrained Environment . . . 73
9.1.3 Implementing the Hardware not supported by TensorFlow . . . 74
9.2 Future Work in the Strategy . . . 75

Bibliography 77

List of Figures

3.1 Representation of the important phases in machine learning models' workflow: Inference versus Training adapted from [1]. 6
3.2 A Venn diagram adapted from [2] showing the positioning of deep learning as a subgroup of representation learning, which itself is a subgroup of machine learning, which is included in AI technologies. 8
3.3 The Y-chart, with dashed lines indicating the areas that impact the performance of programmable architectures [3]. 14
3.4 Diagram adapted from [4] of a smooth mapping from an application to an architecture that requires the compatibility of the model of architecture with the model of computation, and the use of compatible data types. 15

4.1 Multi-Device Execution handled by TensorFlow's placement algorithm: node placement and cross-device communication (example diagram adapted from [5]). . . . 23
4.2 The Y-chart representation of the problem adapted from [3], with the set of applications consisting of a neural network. 24
4.3 Representation of a machine learning model as an application graph with the dataflow graph chosen as the model of computation. 25
4.4 Representation of the steps of an evolutionary algorithm (EA) used for DSE taking as input a specification of a system and giving as output a set of optimized implementations (adapted figure from [6]). 29
4.5 Bloc diagram summarizing the SAT-decoding approach (Figure adapted from [6]). 31
4.6 Representation of the chromosome space, the solution space, and the objective space used in the optimization combining the EA with SAT-decoding (Figure adapted from [7]). 31

5.1 A Diagram from [8] exposing the usual workflow of development using the Intel Movidius Neural Compute Stick. 35
5.2 Summary of the workflow of a model's creation for the Coral Edge TPU USB accelerator adapted from [9]. 36
5.3 Partition of the Flatbuffer TensorFlow Lite model's operations by the Edge TPU Compiler into feasible operations to run on the Coral accelerator and unfeasible operations to run on the CPU, adapted from the Coral Edge TPU official documentation [9]. 36
5.4 Benchmark tool Block Diagram. 38
5.5 Summary of the tasks that must be fulfilled to run the TensorFlow Lite benchmark on the Coral Edge TPU. 39
5.6 Operation models visualized from the created protocol buffer files with the online tool `Netron` [10]. 40
5.7 Visualization of the specification of the TensorFlow model mobilenet with OpenDSE viewer. 43
5.8 Summary of the exploration's workflow with OpenDSE. 44

5.9 Overview of the simplified architecture of the used classes in this thesis DSE with OpenDSE, adapted from [7]. 48

6.1 Average inference time in μs for a set of different types of operations for each available hardware with a number of runs=50, different input sizes, and different input types. 52
6.2 Average inference time on the NCS in μs versus the input SHAVEs of the NCS with 50 runs for each test, different operation types, different input sizes, and different input types. 52
6.3 Average inference time on the NCS in μs versus the input types to the operation models run on the NCS with 50 runs for each test, different operation types, different input sizes, and different input SHAVEs. 53
6.4 Average inference time on the host (CPU+GPU) in μs versus the input types to the operation models run on the host with 50 runs for each test, different operation types, and different input sizes. 54
6.5 Bar graphs showing the average inference time in μs in function of the input size of the operation model Conv2d (2D convolution of TensorFlow) on the following hardware: CPU and GPU, NCS, and Coral Edge TPU. 56
6.6 Average inference time in μs in the function of the input size of the operation model Conv2d (2D convolution of TensorFlow) and the operation model relu (the rectified linear unit from TensorFlow) on the following hardware: CPU and GPU, NCS (number of SHAVEs=4 and 12), and Coral Edge TPU. 58
6.7 Visualization of the generated example model from the Protocol Buffer file with `Netron` [10]. 60
6.8 Application graph representing the TensorFlow model example (visualized with the OpenDSE Viewer). 61
6.9 Bar graph representing the cost of mapping (`cost_of_mapping`) versus the crossover rate (`CrossoverRate`) over 10000 seperate runs of the DSE with a fixed number of generations equal to 100, a fixed size of population equal to 50, a fixed number of parents equal to 25, and a fixed number of offsprings per generation equal to 25. 62
6.10 Bar graph representing the Cost of Mapping versus population size over 10000 seperate runs of the DSE with a fixed number of generations equal to 100, a fixed crossover rate equal to 0.95, a fixed number of parents equal to 25, and a fixed number of offsprings per generation equal to 25. 63

7.1 Corresponding Architecture. 67
7.2 Mapping for 3 different implementations generated by the optimization which corresponding (cost of mapping, number of SHAVEs) tuples are respectively: (582.2, 12), (634.1, 6), and (643.0,9). 67
7.3 Mapping for cost of mapping= 676.7. 67
7.4 Optimal Implementations obtained by the DSE with OpenDSE summary for the TensorFlow model example. 67
7.5 Representation of the Pareto front of the implementations generated by the optimization that includes the NCS in the architecture according to the cost of mapping and the number of SHAVEs of the NCS as objectives. 69

9.1 Pareto chart representing the results of running the optimization on the mobilenet model over 69 runs. 74

List of Tables

3.1 TensorFlow operation types example adapted from [5] 10

6.1 Correlation between measurement data and regression cost models for a set of operations on the Host compute(CPU and GPU) and on the Coral Edge TPU. . 59
6.2 Correlation between measurement data and regression cost models for a set of operations running on the NCS with a fixed number of SHAVEs equal to 6. . . . 59

7.1 Table representing the frequencies of the resulting cost of mapping over 10000 separate runs with the following settings: number of generations=100, size of population=50, number of parents=25, number of offsprings per generation=25, and crossover rate=0.95. 66

Acronyms

AI	Artificial Intelligence.
Coral	Coral Edge TPU stick from Google.
CSV	Comma-Separated values.
DL	Deep Learning.
DSE	Design Space Exploration.
EA	Evolutionary Algorithm.
GUI	Graphical User Interface.
HA	hardware accelerator.
ML	Machine Learning.
NCS	Neural Compute Stick from intel.
SAT	Boolean Satisfiability.
SHAVEs	Streaming Hybrid Architecture Vector Engines.
tf	TensorFlow.
XML	Extensible Markup Language.

1 Introduction

1.1 Motivation and Problem Statement

1.1.1 Motivation

Artificial intelligence is extensively used in various fields of science and engineering, such as speech recognition, image classification, and data analysis. Thanks to its continuous progress and numerous breakthroughs, it is replacing multiple traditional methods. Specifically, Deep learning as a subset of AI has become widely used for a broad range of applications, outperforming conventional machine learning algorithms.

Within the Department of Chemical and Physical Sensing within the Corporate Technology sector in Siemens, there is an interest in utilizing deep neural networks in processing data for on-field sensors, with the aims of staying up-to-date and ensuring the best performance of sensor data analysis. Integrating AI models into the data analysis pipeline requires fast and efficient hardware to meet their high computational density. Hence, came the idea of using hardware accelerators which are designed to execute in a distributed and efficient manner alongside the existing hardware. In other words, the focus is turned to optimizing the deployment performance of the data analysis pipeline of sensor data. The latter would include deep neural network models, considering that the volume of collected data of the targeted sensors is huge and that the processing environment is constrained. The goal is then to be able to make use of heterogeneous hardware devices without utilizing cloud computing solutions, which are both expensive and limited.

The problem raised here is distributing the neural networks' execution across different hardware and ensuring that the implementation meets the optimal performance requirements demanded by the user.

1.1.2 Problem Statement

The deployment of deep learning models requires high computational resources due to their increasing complexity, size, and the number of layers (refer to Subsection 3.1.2). Therefore, distributing their execution on different hardware, including hardware accelerators, should be scrutinized, particularly when deployed on the field in constrained environments like embedded systems. There comes the need for a deliberate strategy of their implementation. The latter necessitates decisions to be taken regarding the hardware to be used and its configuration, and where to place the parts of the machine learning model with respect to the bounds of performance set by the user and the constraints introduced by the hardware.

When considering the low-level representation of a deep neural network consisting of elementary operations with or without built-in weights set during the training phase (refer to Subsection 3.1.1), it can be modeled as a dataflow graph, in which nodes represent the operations. In this

way, the inference distribution problem is equivalent to a system-level-synthesis one that includes the allocation of the hardware, the binding of the nodes to the allocated hardware, the routing of the data to the busses, and the scheduling, which is out of the scope of this thesis.

This system-level-synthesis problem is complex not only because of the large number of possible resulting implementations but also due to the numerous constraints introduced by the hardware and the high complexity of the considered application. This is why a design space exploration (DSE) of the various implementations needs to be conducted. Precisely, an automated DSE approach, according to the Y-chart methodology (refer to subsection 3.2.3), is an adequate strategy for this problem. This approach has been applied to similar problems and has led to satisfactory results. However, challenges also arise when using this methodology, namely obtaining the performance numbers and the choice of the algorithm used during the DSE as well as the execution cost of the DSE. Nevertheless, it is known to produce a reliable and satisfactory solution in a reasonable time lapse.

To conclude, the distribution of the inference (refer to Subsection 3.1.2) of machine learning models on heterogeneous hardware devices is assimilated to an optimization problem, which can be solved with a DSE according to the Y-chart methodology. This raises several challenges related to the hardware specifications, especially the hardware accelerators, the complexity of the considered class of applications, and the constraints of the overall system-model.

1.2 Thesis Outline

This thesis presents an approach to distributing inference of machine learning models on heterogeneous hardware devices, including artificial intelligence hardware accelerators. The approach relies on representing this problem according to the Y-chart methodology as a system-level design problem. The solution to the latter is addressed with a DSE applying existent methods. An experimental evaluation is then performed as a proof of concept of the presented method.

1.3 Contribution

In this thesis, a method that applies existing DSE techniques according to the Y-chart methodology to machine learning workloads is proposed. This method aims to optimally distribute inference of machine learning models on heterogeneous hardware devices, including artificial intelligence hardware accelerators. The DSE relies on an existing hybrid optimization algorithm that ensures the exploration of feasible solutions and which one of its characteristics is a fitness function. A practical approach, as to how this function was established, is outlined in this thesis. Indeed, by combining different benchmark algorithms, a benchmark tool was used to establish cost models for mapping nodes of machine learning models onto specific hardware devices. These cost models are used to rate the fitness of each implementation during DSE.

2 State of the Art

Design space exploration has been integrated into the design engineering workflow of complex embedded systems for years. However, with the tremendous growth and new technologies in the embedded software and hardware market, the system-level design of programmable embedded systems has been increasing in complexity, making the DSE process more challenging. Different approaches exist in between which the Y-chart methodology ensuring a scheme for DSE was considered as the go-to strategy in this thesis. This approach is commonly used for system-level synthesis problems, as shown in [6], [7], and in the OpenDSE framework [11]. This framework relies on the Y-chart approach and relies on the Opt4J [7] stack for metaheuristic optimizations designated to be applied to programmable embedded systems' design. In this thesis, this framework is applied to a reemerging class of algorithms, namely machine learning and deep learning models, which could be considered as an up-to-date application to the framework and these techniques.

The previously mentioned concepts are applied to distribute the execution of neural networks. This class of algorithms progressed in different domains such as image recognition, speech recognition, object detection, genomics, and others [12].

Different existent frameworks ease the implementation and deployment of these machine learning models. In particular, TensorFlow is the most popular and used interface, whose first open-source release was in 2015 presented by the Google Research group in [5]. In [5], TensorFlow was presented with a dataflow based programming model designated to run on a single machine or in a distributed manner. Particularly, its placement algorithm relying on greedy heuristics, in addition to the run-time placement decisions during the execution, was introduced. The difference in this thesis is that this placement algorithm is extended theoretically to hardware accelerators, which are not supported by TensorFlow, and the strategy of the placement is slightly different since it is based on a metaheuristic exploration according to the Y-chart approach.

Due to the complexity and high computational density that machine learning algorithms require, research has been on-going about how to accelerate their overall workflow, namely their training and inference. The focus was turned to improve the overall performance of the training phase to make more experimentation and lower its cost. However, nowadays, more interest is towards speeding up inference to bring it to edge devices. Several studies were conducted to speed up the inference execution of machine learning models. However, there is still a significant lack of research in applying DSE methods to this matter. In the existing recent research, a DSE framework for convolutional neural networks implemented on edge devices can be identified in [13]. The latter presents a framework to improve resource management and thus accelerate inference when using convolutional neural networks, and with an approach based on DSE, the execution time could be reduced up to 3.6 percent and energy consumption up to 7.7 percent. One key difference between their approach and the one in this thesis is that they look at the neural networks at a higher level. Specifically, they try to map layers to hardware contrariwise to the approach used here where every operation is considered separately to be mapped. Another key difference might be the methods used in DSE, but they do not expose the methods they are

using to conduct the DSE.

Another related research in the topic can be found in [14], which presents a software-defined DSE for Efficient DNN accelerator architecture where a framework is presented to make a DSE of the accelerator design space and optimize the architecture configurations using a genetic algorithm. The principal dissimilarities between this work and the work in this thesis are that their optimization targets an early-stage design of hardware accelerators, whereas what is targeted here is a system-level-synthesis at a later stage 4. Another difference is the genetic algorithm that they use, which is slightly different from the hybrid evolutionary algorithm that is used in this thesis, and that ensures that the exploration stays in the feasible part of the design space (Chapter 4).

While accelerating the inference of machine learning models has been the focus of on-going research in recent years, intending to bring their deployment to the edge. Different approaches are being followed, like design space exploration methods, which seem to be an adequate choice, as can be found in the recent on-going and mentioned research papers. These methods rely on different algorithms that have been proving their efficiency in multiple domains. Within this thesis, a similar approach is followed because DSE methods are used to accelerate inference. However, there is a lack in research papers on the use of the Y-chart methodology to represent the problem of accelerating the inference phase of a machine learning model by distributing it on heterogeneous hardware devices, including hardware accelerators. This thesis aims to present a precise method relying on the Y-chart methodology and using state-of-the-art efficient DSE algorithms, namely a hybrid optimization algorithm based on a genetic algorithm with SAT-decoding. The strategy is then applied as a proof of concept to accelerate inference of machine learning models represented as dataflow graphs on heterogeneous hardware devices, including recent AI hardware accelerators.

3 Preliminaries

In this Chapter, the main technologies and techniques used in the thesis are presented.

3.1 Machine Learning

3.1.1 Generalized Machine Learning

Machine learning can be defined as the set of methods that rely on applied statistics using hardware resources to perform predictions or decisions without explicit former programming [15]. ML workloads can learn by extracting features from real-world data, which resulting difficulty is the way of defining and the nature of the patterns that are to be extracted from the data. One solution to this can be avoided by representation learning, which is a set of techniques that enables the automation of features extraction by the machine having raw data as input [12].

A basic example of a representation learning algorithm is an auto-encoder. The encoder is trained to produce a new representation of the input data with various properties without losing information so that the decoder converts it back into the original format [2]. Nevertheless, when extracting abstract high-level features from raw data is difficult, representation learning is limited, and the solution would be to introduce simpler interconnected representations like it is the case in deep learning algorithms (refer to Subsection 3.1.3).

Machine learning approaches can be divided into three main types:

- Supervised learning: a predictive approach in which the main idea is to establish relationships and dependencies between labeled outputs and inputs in order to make the correct predictions.
- Unsupervised learning: a descriptive approach which core concept is to search for patterns from given unlabelled data.
- Reinforcement learning: reward or penalty based approach which principle scheme is to learn from collected interactions with the environment iteratively by giving rewards or punishments during the process of learning.

These groups of machine learning approaches and others (semi-supervised learning) usually adopt the following workflow: first, a phase of learning or training, and then a phase of inference. During the training phase, the algorithm is trained to learn from collected data (whether labeled or unlabeled). In other terms, training is equivalent to determining the best values for all the weights and bias of the considered machine learning model that minimize the training error. A good example of this is the logistic regression whose coefficients are determined during a training phase. The performance can be computed in this example with the mean squared error (MSE) of the model on the training set, this way, finding the best coefficients is equivalent to an optimization problem, whose goal is to minimize the value of MSE. After training, the machine

learning model is expected to have good performance even on new previously unseen data, and this capability is denoted as generalization. Therefore, in addition to minimizing the training error, the machine learning model's performance is also tested on a set of data that is separate from the training set with the goal of minimizing the resulting generalization error. Hence, a good machine learning model ensures a small training error and a small gap between the training and generalization error [2]. In other terms, it is a model that avoids underfitting and overfitting. One way of controlling that is tuning the hyperparameters of the machine learning model, which should be separately set and readjusted (in most cases). Thus, another step that usually comes before the generalization step during the training is the validation step. During this step, the model is tested on a validation set different from the training set in order to adapt its hyperparameters. All these steps can be considered as the training phase in which the output is a pre-trained model ready to infer.

The next phase for a machine learning model is executing inference, which calculates the output of the previously trained model for a random input (in or outside the training data set). Shortly, inference is the step where what is learned during the training is tested.

3.1.2 Inference

As mentioned in the previous Subsection 3.1.1 and as can be seen in Figure 3.1, inference is the step that comes after training and during which the machine learning model is deployed. More specifically, it consists of making a prediction of an output given an unknown input with the pre-trained machine learning model.

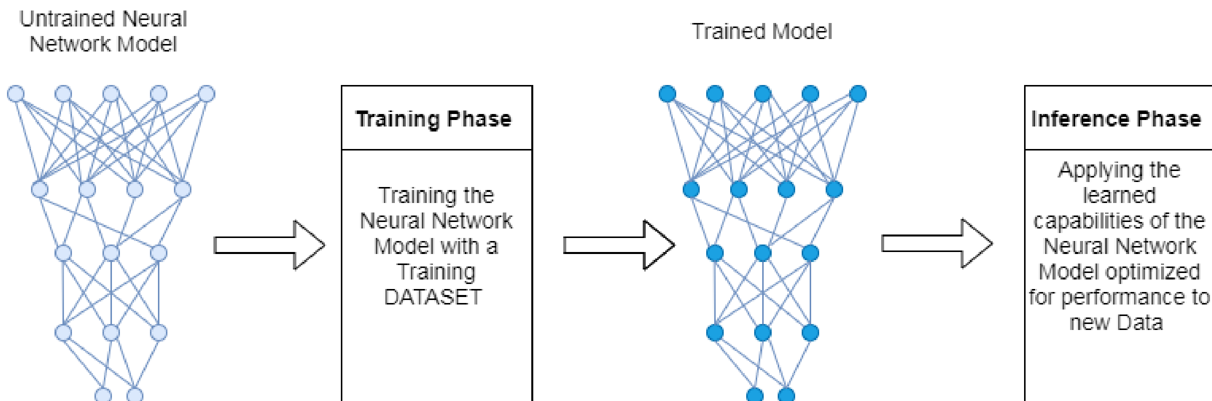

Figure 3.1: Representation of the important phases in machine learning models' workflow: Inference versus Training adapted from [1].

This step has differences and similarities with training. Indeed, they both start with forward propagation calculations. During training, the results of forward propagation compared with the training set are used to determine an error rate, and a backward propagation takes place to propagate the error back through the model and adjust the weights in order to boost the efficiency of the model on accomplishing what it is trying to learn [16]. Therefore, the distribution of the inference computations can be easier than the training one since it does not require the backward propagation process to update the weights. Thus, the inference focus is on delivering the output result. For instance, when representing the considered machine learning model as

a dataflow graph (more details on the dataflow model of computation in Subsection 3.2.6), the distribution of inference is enabled. Dataflow graph nodes can be executed in parallel, and their execution depends only on their input data. Precisely, as presented in the general abstract model of dataflow graphs in [17], each node's execution generates an output token carrying the result, and similarly, the input data of the node is carried by a token. When a node has dependencies from more than one node, its input data is a composite of these nodes' output tokens. For inference, this means that the parallel distributed computations' results can be assembled to give the final result, which is the goal of inference.

Another difference is that during inference, parts that are not activated are removed, and layers are fused as long as making the prediction is not affected, and the loss in accuracy negligible. That does not mean that inference is computationally cheap. It could be less computationally intense than training, but it still requires high memory and compute costs with the rising improvements in the quality and complexity of machine learning models in general and deep neural networks in particular. For example, the Recurrent Neural Network ResNeXt-101-32x4d model in [18] that comprises 43M parameters and performs 8B multiply-add operations during inference and a group of convolutions.

In addition to that, the expectations of the user also have an impact on the needed computational resources to infer. For example, when inference is deployed in embedded systems with real-time constraints, the focus resides in meeting the deadlines and assuring low delays by choosing the adequate hardware and software combination. Contrariwise, in other applications, memory footprint or accuracy are the interesting metrics and impact the chosen hardware.

The inference performance metric targeted in this thesis is the time of execution, as shown in the next chapters.

3.1.3 Deep Learning

Deep learning is a subdivision of machine learning, which itself is a subdivision of artificial intelligence as shown in Figure 3.2.

The beginning of this class of algorithms goes back to the 1940s under different names, has gone through different phases, and highs and lows in popularity [2]. It was known that some of these algorithms work well, but one of the issues was their computational cost limiting experimentation with the available hardware back then, and also the difficulty of gathering data for the training phase, which is easier nowadays. Moreover, it encountered a lack of research funding except for some research groups like the Canadian Institute for Advanced Research (CIFAR) that contributed to maintaining the research in neural networks despite the downturn from the mid-1990s until 2007 of these technologies' reputation. These recent years, the use of DL has been thriving more and more in several applications. It has proven itself by efficiently progressing in areas such as image recognition, speech recognition, visual object detection, genomics, and others [12].

As its name indicates, two aspects are essential to define deep learning: the notion of learning and the notion of depth of a neural network in this type of algorithms. Deep learning algorithms deal with complex raw data by making more straightforward and interconnected representations. For instance, an image recognition deep learning model could represent the perception of an im-

age of a dog as a combination of simple notions like corners and contours from which edges are determined [2]. Hence, the representation learning approach in the case of deep learning consists of acquiring the best adequate simple and interconnected representations that enable building complex concepts.

In addition to that characteristic of deep neural networks, the notion of depth is also essential. A deep neural network comprises multiple processing layers. The depth can be both calculated as the depth of the computational graph or the depth of the probabilistic modeling graph because what is important about the notion of depth in deep learning is considering models with bigger and more complex architectures than traditional machine learning models [2].

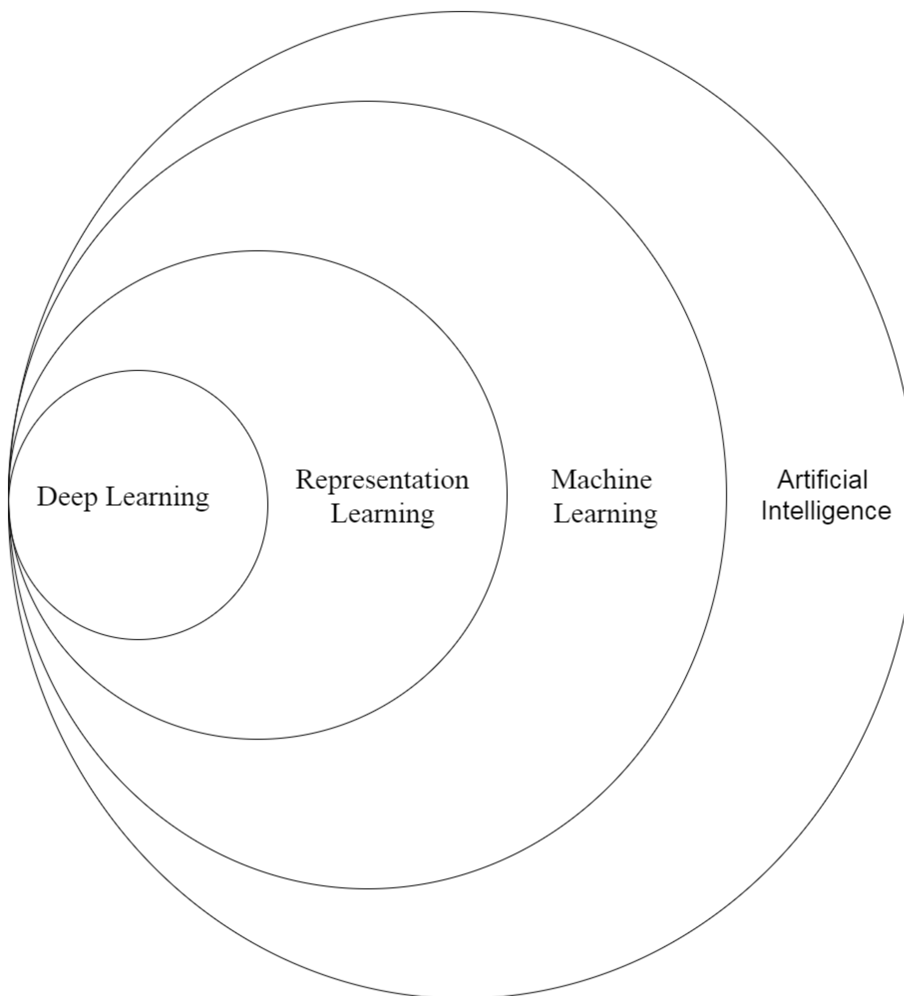

Figure 3.2: A Venn diagram adapted from [2] showing the positioning of deep learning as a subgroup of representation learning, which itself is a subgroup of machine learning, which is included in AI technologies.

Given a deep neural network model composed of multiple processing units or neurons that are connected and that generate activation values from their input, input neurons get activated by the direct input of the environment and start a series of activations for other neurons. The representation learning and depth characteristics here mean that the model will look for the ad-

equate weights so that the neural network achieves its task through many computational stages [19].

There are different deep learning architectures, the most used and famous ones are first deep convolutional neural networks (CNNs) in applications like video, image and speech processing, and second recurrent neural networks in sequential data applications. The example considered for testing later is the Mobilenet neural network, whose structure is based on the CNNs architecture. CNNs handle input data, in which topology is grid-structured, for instance, a grayscale image represented by a 2D array of the pixel intensities. Layers of CNNs usually comprise three stages. The first one is a convolutional layer that generates a group of linear activations, which are the input of the next layer known as the Detector layer, which consists of a non-linear activation function like the rectified linear activation function (ReLU). The third one is a pooling layer, whose role is to fuse comparable features. Thanks to their structure that is less computationally demanding than fully connected neural networks and that makes their training phase faster and easier, CNNs already succeeded when back-propagation networks were supposed to be a flop [2]. Nowadays, tests and experimentation with these algorithms and specifically with deep neural networks, have been made easier and faster, especially with the available hardware resources, for example, AI hardware accelerators.

3.1.4 Machine Learning Models' Creation

3.1.4.1 Introduction

Adopting machine learning methods implicates building machine learning models. Different types of models are used and researched. In particular, parametric models that have a finite set of parameters, for example, linear models, or non-parametric models, whose number of parameters can be infinite because it changes during the learning phase, for example, decision trees or K-nearest neighbor algorithms, which set of parameters is proportional to the size of the training data set. As mentioned in Subsection 3.1.1, these models are based upon statistics where assumptions are made, and priors are computed according to distributions. A good example that illustrates this is a linear regression model which takes as input a vector x$\in \mathbb{R}^n$, whose weights determined during the training phase are $\omega \epsilon \mathbb{R}^n$, and whose output y is a scalar resulting of matrix multiplication as follows: $\mathrm{y} = \omega^T \mathrm{x}$. Therefore, the user needs to represent the model by the latter matrix multiplication of the matrix of the learned weights and the input and also adapt the running of the computations on the hardware, whether a high-level programming language or low level is used.

In general, the computation model of the machine learning system needs to be adapted to the hardware where the inference is run (refer to 3.2.3). The user can choose the representation, but different frameworks exist and present interfaces to make building machine learning models easier than doing it from scratch. An example of these frameworks, one of the most used ones in machine learning is TensorFlow.

3.1.4.2 TensorFlow Models Creation as an example

A good example of how machine learning models are created is their conception with the TensorFlow framework. In the next paragraphs, the programming model of TensorFlow is introduced to

have an overview of the creation process, and then the creation of a TensorFlow and a TensorFlow lite model are presented in detail.

3.1.4.2.1 Introduction to TensorFlow's Programming Model

TensorFlow is a powerful open-source framework to represent and deploy machine learning algorithms. It started as a Google Brain project, namely DistBelief investigating the benefits of very-large-scale deep learning for research and Google products [5]. In late 2015, it was made public, and the first TensorFlow supported steady version was provided in 2017, and is currently the most popular machine learning interface.

Internally, TensorFlow comprises two core interacting parts: a library for expressing neural networks as dataflow-like graphs and the runtime for executing them on heterogeneous hardware devices. Computations are represented in TensorFlow as directed dataflow graphs. Every dataflow graph is constituted from a set of vertices that are the nodes of the graph and edges that are the data flowing between the nodes. As can be seen in Table 3.1, these nodes which are instances of TensorFlow operations (`tf.Operation`) are either abstract units of computations from different types, placeholders (`tf.Placeholder`) that are like empty nodes defining shape, data type, and usually used to feed inputs to the graph, or variables (`tf.Variable`) which are a different type of TensorFlow operations that are changeable parameters in the graph usually used for weights and biases in a neural network model. These units involve tensors, which are the generalization of the notion of matrices and vectors to a higher dimension, and they are represented as n-dimensional arrays of base datatype [20]. To build and then interact with a TensorFlow model, a `Session` needs to be created and run. The Session interface starts with an instantiated empty graph extended with the operations that are given by the program, which permits constructing the computation graph. The graph is executed with the `Run` method from the interface `Session` [5].

Table 3.1: TensorFlow operation types example adapted from [5]

Type	**Examples**
Element-wise mathematical operations	Add, Sub, Mul, Div, Exp, Log, Equal...
Array operations	Concat, Slice, Shape, Constant...
Matrix operations	MatMul, MatrixInverse, MatrixDeterminant...
Neural-net building blocks	SoftMax, ReLU, Convolution2D, MaxPool...
Stateful operations	Variable, Assign...
Control flow operations	Merge, Switch, NextIteration...

Consequently, creating a machine learning model in TensorFlow involves creating a dataflow graph with nodes as operations and vertices as tensors. This model is stored in a data structure to solve the specific machine learning problem it was trained for.

3.1.4.2.2 Models Creation in TensorFlow

The graph needs to be saved to a certain data structure to be trained, retrained, or execute inference, and one of the most used ones in TensorFlow (up to 1.15) is Protocol Buffers. Protocol

Buffer is a specific way presented by Google to efficiently and flexibly serialize structured data, like dataflow compute graphs in [21]. The user then creates a TensorFlow model or loads a pre-trained model from the supported TensorFlow models. In conclusion, a machine learning model created or loaded from pre-trained models in TensorFlow is stored as a protocol buffer encoding its logic and its learning (In TensorFlow 2 Saved model). Running inference then consists of loading this Protocol Buffer in a TensorFlow session, giving it an input, and getting the output. In TensorFlow Lite, the model creation is slightly different.

3.1.4.2.3 Models Creation in TensorFlow Lite

Creating a model in TensorFlow Lite requires passing through the same procedure described in the previous paragraph in this Section 3.1.4.2.2 and, in other words, starting by creating a TensorFlow model and training it. Then, transforming the latter with the TensorFlow Lite Converter to a TensorFlow Lite format, which consists of a reduced file size and an optimized model stored in a FlatBuffer. A Flatbuffer is another form of serialization developed by Google, which is somehow similar to Protocol Buffers but more lightweight and adapted to constrained environment devices coming with the advantage of not needing to de-serialize before accessing the stored data and consuming less memory than Protocol Buffers. Once the conversion is completed, the model is ready to be deployed using the TensorFlow Lite interpreter that takes as input the converted TensorFlow FlatBuffer model, runs its operations on the input, and gives the output, which is the result of the inference.

In conclusion, the example of TensorFlow models' creation gives an overview of the steps of a machine learning creation. Indeed, the machine learning model's design needs to include logic, knowledge, training, and inference on the targeted hardware. The user can choose to build the model from scratch, but it is easier to use the existing features in frameworks like TensorFlow, which offers many advantages such as optimizations and others.

3.1.5 Artificial Intelligence Hardware Accelerators

Artificial intelligence hardware accelerators (AI HA) are a class of hardware which purpose is to accelerate Artificial intelligence tasks, which are known to have special requirements of high computational power and high bandwidth memory during their different phases of development and deployment (refer to 3.1.1 and 3.1.2). For example, deep neural networks require high computation power for their convolutional layers or fully connected layers.

The first trials of AI hardware acceleration started as early as the 90s with Analog Neural Networks implementations, for instance, the ANNA chip that was targeting fully connected and recurrent neural networks [22], and also with Field Programmable Gate Arrays (FPGAs) like the one presented in [23]. Besides that, hardware acceleration of AI algorithms relied on the advancements of new central processing units (CPU) structures and multi-core heterogeneous powerful processors. Starting from the 2000s, graphical processing units, which were intended to handle image and video-related tasks, gained popularity for handling AI due to the similarity of the mathematical basis of the previously mentioned tasks, for example involving matrices multiplications and convolutions. Moreover, the interest was meanwhile and more recently also turned to develop application-specific integrated circuits (ASICs) as hardware accelerators, for example, the DianNao university dataflow research chips that target the acceleration of Machine Learning algorithms processing with different designs and targets [24].

To conclude, AI hardware accelerators rely on various technologies and architectures. This group of hardware includes optimized multi-core general-purpose processors (CPUs), graphical processing units (GPUs), Field Programmable Gate Arrays (FPGAs), application specific integrated circuits (ASICs), and others. Furthermore, the best architectures and types of AI hardware accelerators to choose from depend on multiple factors like the type of the targeted application, the type of used AI class, and the user's targeted goals that differ between accelerating tasks in data-centers and on edge devices. Therefore, AI systems' designers turn to DSE to find the best hardware acceleration they need for their considered application.

3.2 Design Space Exploration

3.2.1 General Presentation

Design space exploration (DSE) denotes, systematically investigating design alternatives before implementation to make the best design decisions. This activity is used in many domains, such as rapid prototyping, system integration, optimization, and others [25]. It is used at different system design steps (refer to Subsection 3.2.5), and designers decide the levels of abstraction that are considered when using it according to the system's requirements and conditions.

An advanced complex system can have millions or, in some cases having infinite design possibilities. For this reason, an exhaustive exploration of all the alternatives is usually either not possible or expensive in time and computations. Hence, the need for methods and approaches that ensure an automated or semi-automated DSE [25]. Numerous methodologies and frameworks were developed to facilitate automated DSE. The three following fundamental concepts are centric to DSE automation [25]:

- The first important concept is the representation scheme of the design space that should be general enough to permit the automation of the analysis.
- The second important concept is the capability of finding possible solutions that satisfy the precise conditions and constraints.
- The third important concept is the exploration methods that should ensure a smart and guided exploration of the design space.

To conclude, design space exploration is used in several domains helping designers, especially at early-design stages. It is mainly based on how the design space is represented, analyzed, and explored, and thus on the methodology and the tools that designers choose when they employ it.

3.2.2 Formal Definition

The formulation of design space exploration is complex and depends on the considered problem. However, the basic concepts can be summarized with the formulation presented below, following [26].

Given a design space X, for each design point of X, m decision variables representing the various degrees of freedom (e.g., hardware parameters) need to be set by the DSE. The DSE explores these design points, and evaluate them to find the optimal feasible ones. The assessment is

performed with a so-called fitness function. The latter evaluates a design point from the solution space X by giving it a value in the objective space Y. For each objective, a fitness function is defined by:

$$f_i : X \rightarrow \mathbb{R}^1.$$

A combination of the multiple fitness functions for each objective constitutes the final fitness function that evaluates each solution in the solution space X according to the n objectives in the objective space Y, and that can be written as follows:

$$f : X \rightarrow R^n \text{ where n=dim(Y).}$$

Hence the DSE can formally be expressed as an optimization problem aiming to minimize the fitness function according to the targeted objectives:

$$min\{y = f(x) = (f_1(x), ..., f_n(x))\}$$

$$\text{where } x = (x_1, ..., x_m) \in X,\ y = (y_1, ..., y_n) \in Y.$$

The results of the optimization are used to select the optimal design points. In the case of a single objective, the solution with the best fitness value is selected. In the case of multiple objectives, the optimal solutions need to be identified. The latter is obtained by applying the Pareto dominance relation, which consists of the following:

$$x_1 << x_2, x_1, x_2 \in X \Leftrightarrow \forall i \in \{1, ..., n\} : f_i(x_1) \leq f_i(x_2) \wedge \exists j \in \{1, ..., n\} : f_j(x_1) < f_j(x_2).$$

In other terms, a solution $x_1 \in X$ dominates a solution $x_2 \in X$ if and only if the objectives' values of x_1 are better or equal (in the definition less or equal) than the ones of x_2, and at least one objective value strictly better (strictly less in the definition). The defined Pareto dominance relation results in a partial ordering set of X of non-dominated solutions representing the Pareto front X'. This subset X' is defined as having no solution in X that dominates its solutions, as follows:

$$\{x' \in X' |\ \nexists x_i \in X : x_i << x\}.$$

The selection of the optimal solution from the Pareto front is a decision taken by the designer according to the best tradeoff that fits his problem.

To conclude, the DSE process's formulation is usually an optimization that aims to find the Pareto front with the optimal design points according to the targeted objectives. The search strategy and methods to cover the design space are multiple. One of the methodologies that have been considered for the design of programmable embedded systems and used in this thesis is the Y-chart approach that is defined next in Subsection 3.2.3.

3.2.3 The Y-chart Methodology to design programmable embedded systems

With the tremendous growth and new technologies in the embedded software and hardware market, the design of programmable embedded systems has been increasing in complexity, making the exploration process more challenging. Different strategies exist and have been followed in order to represent the system definition and requirements. These strategies have been through methodologies completely isolating HW and SW design with a system gap to describing and synthesizing with hardware description languages until HW/SW Codesign. The first strategies

leave less freedom in the architecture and narrow the design space because of the details' level when defining the specifications. Furthermore, HW/SW Codesign has proven its efficiency in implementing a single algorithm into hardware but has difficulty handling a set of applications efficiently.

Therefore, to represent the problem in this thesis, it was interesting to follow the Y-chart methodology that goes a step forward towards more abstraction in the designing of programmable embedded systems detailed in [3] (The Y-chart concept was presented first in [27], but the general and complete methodology for using it to design programmable embedded systems is illustrated in [3]).

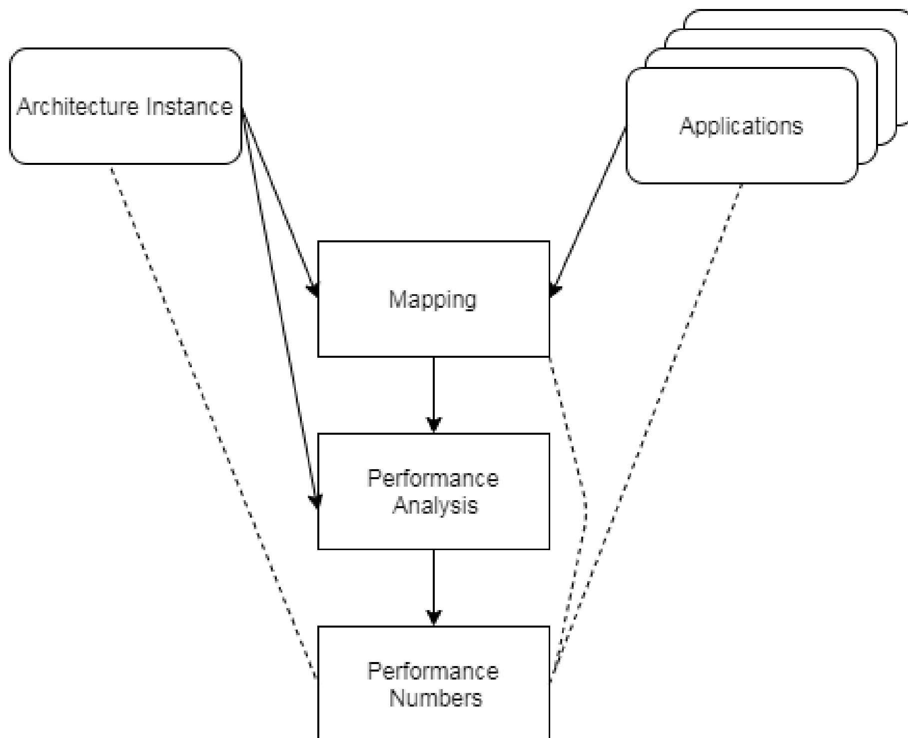

Figure 3.3: The Y-chart, with dashed lines indicating the areas that impact the performance of programmable architectures [3].

The Y-chart methodology aims at providing designers with quantitative data deducted from investigating the performance of architectures for a given set of applications [3]. The overview of this approach is depicted in Figure 3.3, identifying three essential matters: architecture, mapping, and applications. First, designers define architecture instances (Architecture Instance Box) beginning with the architecture template and go through the step of performance analysis of the architecture (Performance Analysis Box) to build a performance model that is assessed for the Mapped applications (Mapping Box and set of Applications Boxes). The results of this assessment (Performance Numbers Box) are used to make improvements that generate other architecture instances, which form the new starting points of the same previous steps until achieving the targeted suitable synergy between architecture and applications [3].

The mapping step represented by the Mapping Box in Figure 3.3 is important in the Y-chart methodology and can be a complex problem. To describe this problem, the following relevant concepts need to be defined as in [3] and [4]:

- A model of computation, as defined in [4] and originally inspired from [28], is "the formal representation of the operational semantics of networks of functional blocks describing computations". In other words, it is the formalism followed by the computations when established and executed. The operation semantics control how the functional blocks of the application are connected to execute the computations. Examples of models of computations are *Dataflow Models*, *Process Models*, *Finite State Models*, and *Imperative Models*.
- A model of Architecture is introduced and defined in analogy with the model of computation in [4], as "a formal representation of the operational semantics of networks of functional blocks describing architectures". The functional blocks depict the behavior of the architecture's elements, and the operational semantics handle how they communicate with each other.
- Data types concern the ones specified while defining the application and the architecture and consist of how the data organization and properties. Example of data types are *integers*, *floats*, *reals*, *integer matrices*, and others.

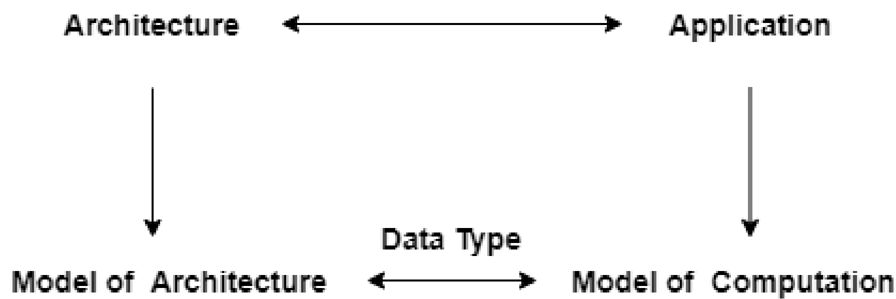

Figure 3.4: Diagram adapted from [4] of a smooth mapping from an application to an architecture that requires the compatibility of the model of architecture with the model of computation, and the use of compatible data types.

Following the previous definitions, mapping an application to a programmable architecture instance requires checking the compatibility between the application's model of computation and the architecture's model, and the consistency between their respective data types. When the previous conditions are filled, and as can be seen in 3.4, a smooth mapping is done (also called natural fit [4]). In the case of incompatibility, for example, of data types, additional modifications need to be added to the application when mapping it to an architecture instance.

To Formally define the Y-chart approach, let `P` be the set of all parameters of the architecture template `AT` and `I` the resulting architecture instances set `I={Io, I1,..., In}`. Exploring the design space of the architecture template `AT` is equivalent to setting up values for all the parameters `Pj` to generate a set of architecture instances and analyze their performance using the Y-chart approach. Nevertheless, the design space of an architecture template is usually large. Thus, stepwise refinement of the design space is proposed in [3] by using a stack of Y-charts.

The space exploration can be conducted as an exhaustive search from where the Pareto front of the optimal solutions is deducted. However, this strategy of exploring is not practical and, in some cases, not possible because of the high cost of the evaluation due to the huge number of design points to test and also due to the high complexity of system-design synthesis that gains more complexity with design constraints. In order to efficiently reach the optimal implementations that meet the expectations, the DSE must be partially automated using algorithms that need to be carefully scrutinized and selected from the existing state-of-the-art algorithms.

3.2.4 Methods in DSE

[29], [6], and [30] give an overview of state-of-the-art algorithms proposed and used for DSE automation and multi-objective optimizations. The scheme of these algorithms could be divided into two steps:

- Establishment of feasible solutions.
- Assessment or estimation of the objectives of the solutions towards the selection of the best parameters.

There is an extensive spectrum of algorithms to choose from, which are classified according to [29] into four broad classes:

- Heuristics and pseudo-random optimization methods try to reduce the design space and rely on full- search or pseudo-random methods to explore the regions of interest. However, the simplicity of the strategies in these algorithms makes them sub-optimal in terms of cost of time and resources.
- Evolutionary Algorithms employ random transformations on a set of initial individuals repeatedly until reaching the set of optimal solutions. The advantages are limited setup effort and less effort with knowing the search space or the optimization parameters. In practice, the set of optimal solutions can be reached when the algorithm runs for a sufficient number of times. Nevertheless, theoretically, it is not sure that the algorithm converges to results or find the optimal solutions.
- Statistical approaches without domain knowledge derive a metamodel from the design space and use it to determine the next design points to evaluate. In other words, using statistics to guide the DSE.
- Statistical approaches with domain knowledge exploit predetermined rules of the design space to reach the set of the most optimal solutions.

The algorithms to chose when carrying on design space exploration is a difficult decision and depends on many factors in between which the type of application that is considered. One of the most know applications is electronic system design that usually requires solving a system-level synthesis problem.

3.2.5 System-Level-Synthesis for DSE

System-level synthesis in this thesis's context resides in finding the optimal mapping of a task-level specification on a hardware architecture [31]. More specifically, the problem is equivalent to realizing the following steps:

- Resource allocation.
- Binding of process tasks to the allocated resources.
- Routing communication tasks to a tree of allocated resources.
- Scheduling of tasks[1].
- Exploring the design space to derive a set of optimal implementations meeting expected restrictions on costs and performance.

It will be explained later in Section 4.2 how the problem in the thesis can be formulated as a system-level-synthesis problem according to the Y-chart methodology presented in the previous Subsection 3.2.3. Establishing the system-model, particularly the mapping, following this approach requires the definition of a model of computation for the application, to further verify its compatibility with the model of architecture of the resources. In the next Subsection, the dataflow model of computation is introduced because it can be used to model machine learning systems, as it is the case in the TensorFlow framework.

3.2.6 General Presentation of the Dataflow Graph Model of Computation

A dataflow program is a directed graph whose nodes represent operations and edges designate the data dependencies between the operations. All mathematical or logical expressions can be interpreted as an acyclic dataflow graph. As explained in [17], in this model of computation, data values are carried on tokens, which are consumed when the node executes or as it is expressed in [17] "fires". Hence, when the node fires, the data token carrying the node's input data is removed, and the computed result is stored to a token on the output of the node. This way, the output of the node only depends on the input values. Therefore, dataflow graphs enable parallel execution of nodes unless there is a conflicting data dependency that links them and determinacy, which means the order of the parallel nodes' execution does not affect the result [17]. The formalism also includes building conditional, and loop program graphs with the control operators "switch" and "merge". In general, nodes can be representing computations at a different level of detail, going from elementary instructions to functions.

This abstract definition of the dataflow model is based on carrying the data on tokens flowing along the edges connecting the different nodes. There are different ways then to adapt this abstract dataflow model to the problem needed to represent. In particular, the representation of the data structures that handle the tokens in conventional programming languages needs to be defined and managed.

[1] Scheduling is out of the scope of this thesis.

4 Design Space Exploration for Hardware Accelerated Inference Distribution

The continuous advances in artificial intelligence in general and in machine learning, in particular, made them more and more used in several domains. Examples of their employment are image recognition, object detection, speech recognition, natural language processing, and others. These applications can both take place in very different environments that impose different performance capabilities like data centers or embedded systems. When working in constrained environments like on edge devices, several requirements need to be full-filled. Especially when working with real-time embedded systems, their deadlines must be met. That is why accelerating the deployment of neural networks is crucial. Certainly, executing a machine learning model is generally less computationally dense than training it (refer to Subsection 3.1.2 in the Preliminaries for the more details). Nevertheless, it still necessitates high computational capabilities and memory bandwidth that might lack when working in a constrained environment. Because of this, a good inference distribution management through the available hardware devices is of high importance.

Different approaches have been used and researched to accelerate inference distribution. Some solutions involve conducting a DSE preceding the execution. Indeed, considering the tremendous success of DSE methods in several domains, it is demonstrated in this thesis that DSE techniques and strategies are considered as a suitable solution to accelerating inference by distributing it.

This chapter presents DSE as an adequate solution to accelerate the inference of machine learning models across different hardware devices, including hardware accelerators. First, the idea of applying DSE methods to distribute the inference is defended in Section 4.1 by giving an overview of the problem in Subsection 4.1.1, explaining how the execution of machine learning can be distributed across different hardware in Subsection 4.1.2, and showing a similar example implemented in the TensorFlow framework in Subsection 4.1.3. Then, in Section 4.2, the working design space is represented according to the defined problem. Finally, in Section 4.3, the exploration process is established as an optimization problem, and the choices of the used methods and techniques during this process are explained.

4.1 Overview of DSE as a Solution to Distributed Inference

4.1.1 Overview of the Problem

Design space exploration (DSE) approaches have been proven efficient for designing complex embedded systems. Specifically, in the case of system-level synthesis [6], it allows exploring the design space of a system-model and searches for the optimal solutions. These solutions are equivalent to allocating hardware devices, binding the tasks constituting the system application to the allocated hardware, routing the communication tasks to the busses, and finally scheduling (more details in Subsection 3.2.5).

Distributing the inference of a machine learning model can be considered from a certain perspective[1] equivalent to the mentioned process. Indeed, distributing the execution of a neural network comes down to distributing the computations on multiple hardware then assembling the results to give the prediction with respect to the scheduling (out of the scope of this thesis). Therefore, this problem can be reduced to a system-level design space exploration where the machine learning model represents the set of applications that would be mapped to the hardware.

Applying DSE methods to accelerate inference requires going through several important steps and decisions. On the one hand, the problem has to be formulated as a system-level synthesis one. Therefore, in this thesis, this problem was faced by following the Y-chart methodology. This is a commonly used methodology for system-level synthesis, and it allows a flexible and efficient approach in modeling systems for DSE (the Y-chart methodology is introduced in the preliminaries in Subsection 3.2.3). Following the Y-chart approach at system-level requires defining a system-model, which can be reduced into a formal graph-based model with an application graph representing the set of tasks, an architecture graph representing the available resources from hardware and busses, and the set of mapping edges between them (formally defined in the Subsection 4.2.2). The adopted approach allows the designer to choose the level of abstraction freely when defining the system-model. It was especially important to decide about the level of definition of the machine learning model. For example, choosing between considering the neural network layers (e.g. convolutional layer) or the elementary computations (e.g. addition, matrix multiplication). In this thesis, the nodes of the application graph representing the neural networks represent elementary operations. The mappings of these operations to hardware is explored as detailed in Subsection 4.1.2 and following the example in Subsection 4.1.3. It is also relevant to mention that the chosen model of computation when representing the application needs to be compatible with the model of architecture in order to ensure a smooth mapping (The conditions required to have a smooth mapping are defined in Subsection 3.2.3). The chosen model of computation was the dataflow paradigm (more details were given in Subsection 3.2.6). The latter enables both parallelism and determinacy, which are advantageous properties for distributing the inference.

On the other hand, considering that the goal of the DSE is to find highly optimal implementations with the shortest inference time without searching the whole design space, each examined solution by the DSE needs to be assessed. Hence, the need for a function that quantitatively evaluates every implementation that is looked at by the exploration algorithm. From the example of the TensorFlow placement algorithm where cost models are built for the elementary operations. Then, according to these cost models, the operations are placed on the hardware (more details in Subsection 4.1.3). The idea of establishing cost models that constitute a function taking as input an implementation and assessing its performance came up. The goal of the DSE process is then finding the implementations with the best assessment. Thus, the exploration process needed to be formulated as an optimization problem, as presented in Section 4.3, whose aim is to minimize or maximize the mentioned assessment function.

Finally, it was essential to decide on an efficient approach during the exploration. In fact, the design space is typically large for complex systems, and system-level synthesis belongs to the class of NP-Complete problems [26]. The chosen algorithm and the reasons for opting for it are

[1] At a system-level design perspective, when representing a machine learning model as an application graph.

presented in Subsection 4.3.3.

To conclude, facing the matter of accelerating inference with conducting a DSE as a solution, the suggested strategy is to apply the Y-chart methodology for representing the problem at system-level. An examined choice of the model of computation and a prior decision of the abstraction level to represent the application is required. Furthermore, an efficient search algorithm has to be selected. Performing the DSE then comes down to an optimization problem based on prior defined cost models to efficiently distribute the inference of machine learning models. This strategy would not have been conceivable if there was no possibility of distributing the inference of ML models, presented in the next Subsection 4.1.2.

4.1.2 Distribution of ML Models' Inference

Given a pre-trained machine learning model, running inference consists of predicting an output to an unknown input. Depending on the model of computation used to translate the ML algorithm, this task could be distributed. Thus, when creating a neural network, and then training it by a chosen dataset, the outcome is a ready pre-trained model. The latter consists of a set of mathematical operations with the learned weights during training built into the operations. These operations need to be run by adequate processing units to compute the results (refer to machine learning models' creation in the preliminaries in 3.1.4). Therefore, they are scheduled and bound to the available hardware according to influencing factors like the dependencies and the type of the operation. Hence, depending on the model of computation of the ML model, distributing the inference goes back to realizing a system-level synthesis in order to find the best distribution to implement.

In this thesis, the chosen model of computation is the dataflow graph with elementary operations as nodes (e.g. arithmetic or logic operations). Representing the ML algorithm with a dataflow graph by its elementary operations and considering mapping the operations separately instead of mapping layers, for example, could represent a suitable approach to distribute inference efficiently. For instance, examining the mobilenet convolutional neural network on an operation level instead of looking at its layers.

Ultimately, efficient distribution of the inference necessitates good management of the available resources and taking into consideration all the factors that come with the model definition, for example, the data types and size. This can be achieved by conducting a system-level DSE with the goal of finding the best implementations. For instance, the placement algorithm of the framework TensorFlow represents an example of applying methods generally used in DSE and specifically heuristics and cost models to efficiently map operations to devices.

4.1.3 Application Specific Similar Example: The TensorFlow Placement Algorithm

TensorFlow is the machine learning interface that was employed in this thesis (refer to the tests in 5). Here its placement algorithm is given as an application-specific example of applying DSE methods to place the operations of the machine learning models. These models are represented in TensorFlow by means of dataflow graphs, which vertices are the computations and edges are the data flowing between the nodes (refer to basic concepts of TensorFlow in the preliminaries chapter in the example of machine learning models creation in Paragraph 3.1.4.2.2). In order to run inference with TensorFlow, the machine learning graph is executed by running an interface

named `Session`. TensorFlow relies on this `Session` interface to connect major elements. Specifically, in a TensorFlow system, there is a client that connects to a master, which itself sends commands to worker processes that handle the mapping of the graph nodes to the available devices [5]. In the simple case of a single worker process with a single device, the graph is executed with respect to the dependencies between nodes. By way of explanation, the dependencies of each node are followed and updated during execution, and the nodes with 0 dependencies are placed in a ready queue that is executed in a random order by the device object based on the corresponding kernels [5]. Otherwise, as in this thesis case, when a multi-device execution is considered, the placement algorithm gets involved in managing the additional difficulties that come along with using several types of devices. The main difficulties consist of the choice of the target hardware for each considered node when facing a multi-device architecture, and handling the transfer of data at the frontiers of the hardware.

On the one hand, the node placement is controlled by TensorFlow's placement algorithm, in which one of the inputs is a cost model assessing for each node the number of bytes per input and output tensors and its computation time. The cost model is determined as reported by [5] statically based on heuristics depending on the operation types or computed from previous runs of the graph. The first task the placement algorithm begins with is a simulation of the graph execution from the source nodes until the end. It checks for each reached node the group of possible devices that would have the kernel[2] implementing the node's operation. When there is more than one feasible device, the targeted device is the one where the node's execution is the soonest to end. The latter device is determined with a greedy heuristic based on the estimated or measured execution time of the node from a cost model and the additional eventual communication costs to transfer data to the node. The algorithm also respects the eventual device placement constraints given by the TensorFlow clients. For instance, when working with python, it is possible to have a device context within which a set of chosen operations run by using "with tf.device" and indicating either CPU or GPU with the index. This TensorFlow feature is not possible yet with other devices than CPUs, GPUs, or TPUs.

On the other hand, the placement algorithm needs to deal with the second complication residing in the management of cross-device communication. Indeed, after completing the previous step of placing the nodes of the compute graph of the machine learning model, the latter is divided into a set of subgraphs. Each one of these subgraphs is designated for running on an allocated device [5]. The cross-device edges are then replaced by a "Send" and "Receive" nodes coordinating to ensure data transfer across devices. These nodes eliminate unnecessary additional data transfer, as can be seen in the example in Figure 4.1, where the compute graph is divided into three subgraphs mapped to "Device 2" and "Device 1". For example, the added nodes between the node "a" which is mapped to "Device 2", and node "b" and "c" which are mapped to "Device 1", ensure the transfer of data only once instead of two times. The "Send" and "Receive" nodes facilitate the scheduling because they handle the necessary synchronization between workers and devices (more details in the TensorFlow paper [5]).

[2] A kernel in this context (and as defined in the TensorFlow paper [5]) is the implementation of an operation designated to run on a particular type of device. Indeed, every operation in TensorFlow represents an abstract computation with a name and other attributes which implementation consists of defining its multiple kernels that depend on the input/output types and the architectures. Hence, when a kernel implementing a particular TensorFlow operation to run on a particular device does not exist, then the device is not feasible for that particular operation.

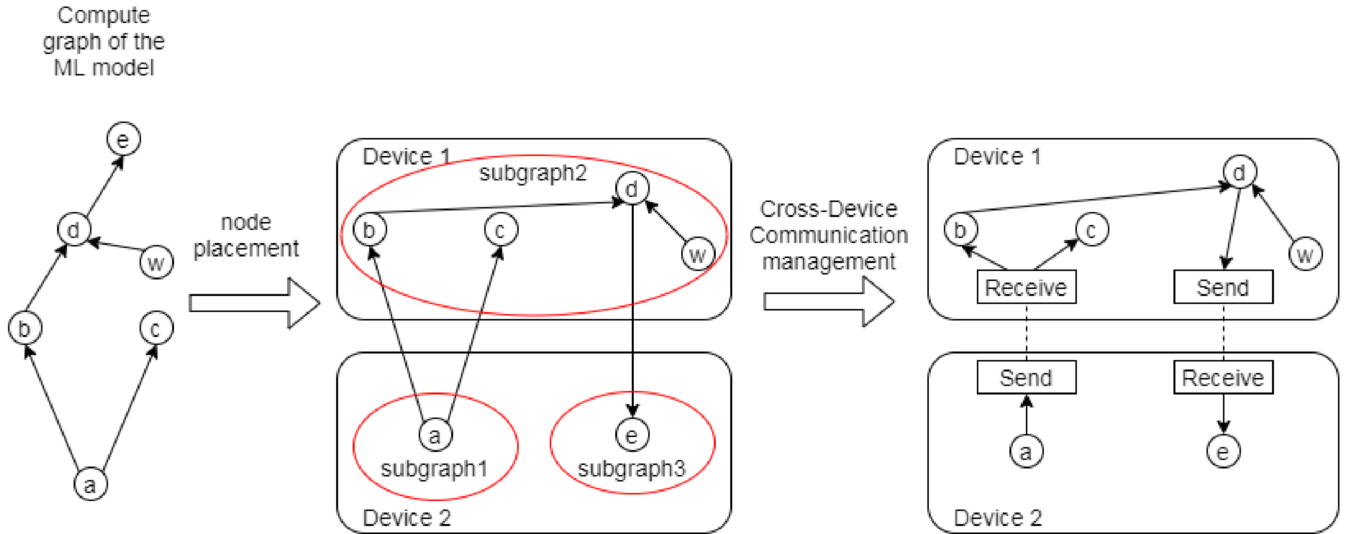

Figure 4.1: Multi-Device Execution handled by TensorFlow's placement algorithm: node placement and cross-device communication (example diagram adapted from [5]).

The placement algorithm is still a subject of continuous growth as cited in [5], and as it can be seen in the improvement suggestions comments within the TensorFlow source code in Github.

In conclusion, the TensorFlow placement algorithm is based on greedy heuristics, which ensure the best efficiency, specifically the shortest execution time by mapping the nodes to the supported TensorFlow devices with the corresponding operations' kernels. And when other devices that are not supported are used, the clients need to find the best solutions for efficiently using their hardware resources. From here came the idea of benchmarking operations to build cost models that are used in a design space exploration process that helps to place them on the right device. The similarity would be using DSE methods and cost models, but with a different perspective and approach.

In the next sections, the design space of generic ML models is represented according to the Y-chart methodology and taking into consideration representing the machine learning model at the elementary operations level in Section 4.2. The approach of conducting the exploration as an optimization problem searching for the best implementations is explained in Section 4.3.

4.2 Modeling of the Solution: Representation of the Problem According to the Y-chart Methodology

As mentioned in Section 4.1, the Y-chart methodology 3.2.3 is followed in this thesis and make it possible to define the design space.

4.2.1 Presentation of the Design Space

Modeling the Design space, in this case, comes down to specifying the set of applications, the set of hardware resources, and the parameters that need to be tuned to obtain the best implementations, as can be seen in Figure 4.2.

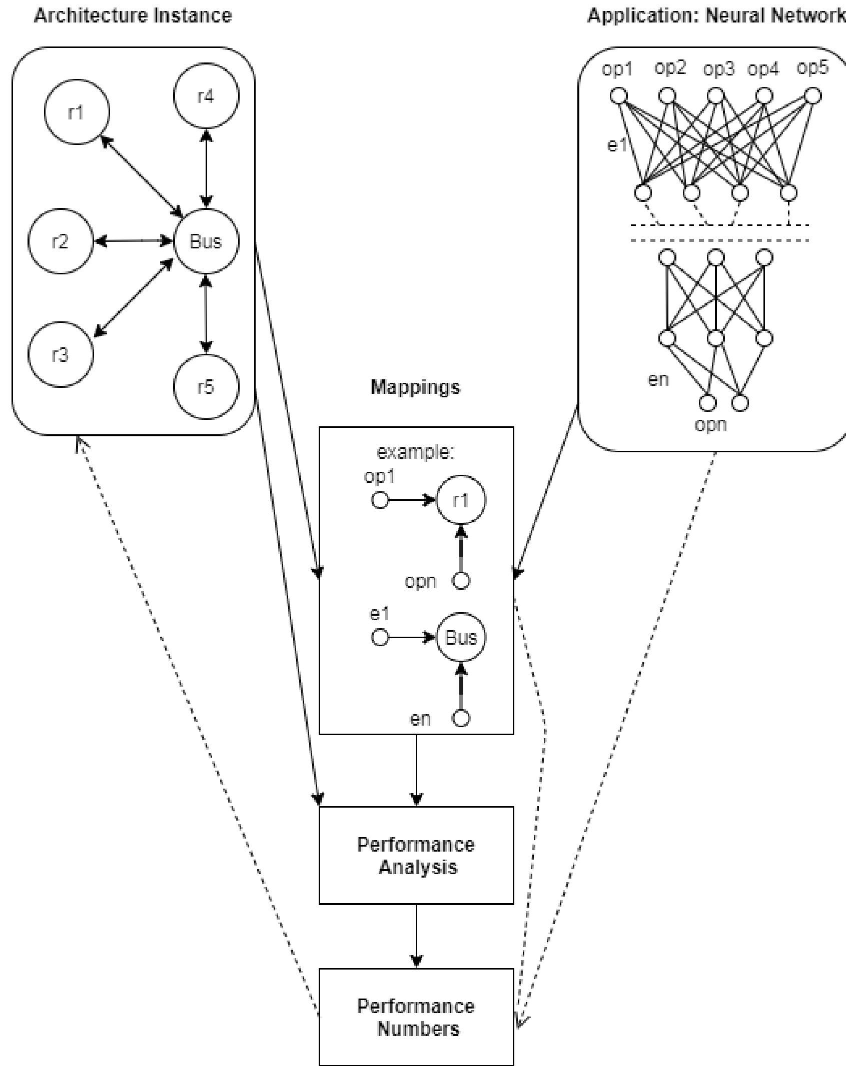

Figure 4.2: The Y-chart representation of the problem adapted from [3], with the set of applications consisting of a neural network.

Following the Y-chart methodology, the system model named specification comprising all the feasible implementations needed to be set. The specification of the problem is comprising three important components, as shown in Figure 4.2:

- The set of Applications of the Y-chart that describes the functional blocks and their interactions, and which is defined according to a model of computation carefully examined to ensure a smooth mapping (the definition of smooth mapping is explained in Subsection 3.2.3). The considered machine learning system, which in this case is the application, needs to be compatible with the model of computation or adapted to it. For instance, representing the logic of a decision tree is different from representing the logic behind an artificial neural network. In the context of this thesis, the considered model of computation to represent the machine learning systems was dataflow graphs. This model of computation permits the representation of a wide range of machine learning models as it is the case

in the TensorFlow framework that relies on dataflow-like[3] model of computation. Correspondingly, and as can be seen in Figure 4.3, the application graph is constituted of a set of tasks as nodes and dependencies as vertices between them. The tasks represent the operations, which are the nodes of the machine learning model compute graph, and the dependencies are the data (tensors in the case of TensorFlow) that flow between them and which are the edges of the graph. The edges between the nodes are communication tasks that are assigned to busses when going from one resource to another. Hence, the creation of the graph task of the specification is generic enough to be able to represent a wide range of machine learning models.

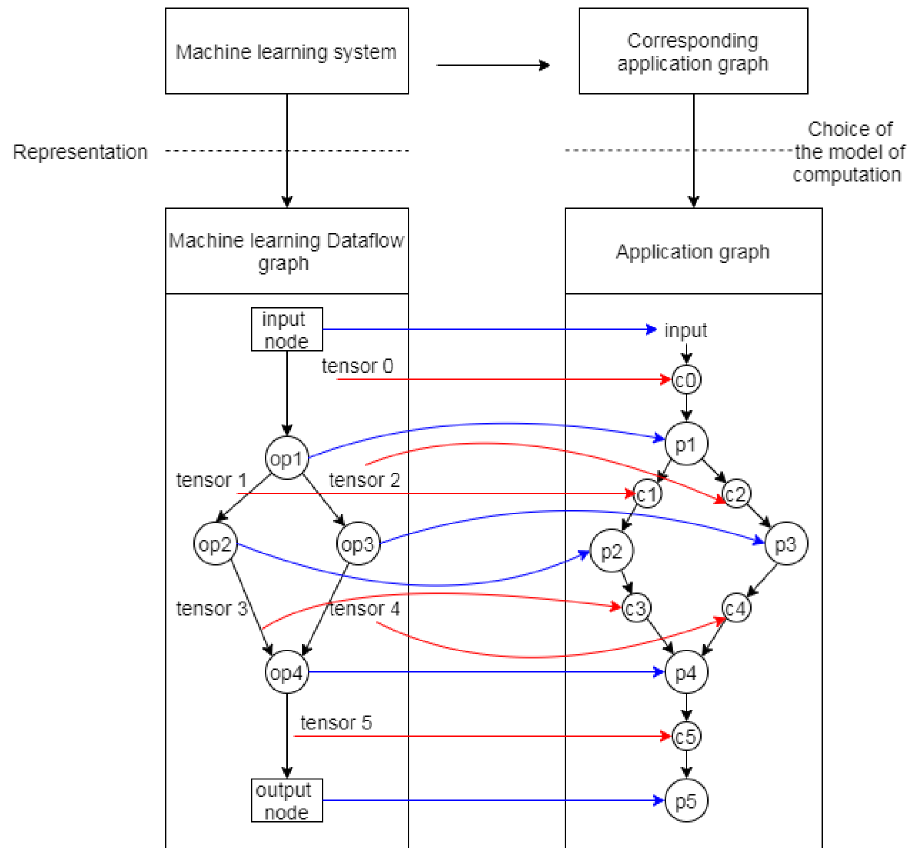

Figure 4.3: Representation of a machine learning model as an application graph with the dataflow graph chosen as the model of computation.

- The Architecture instance that comprises the hardware resources and the links between them. This architecture can be represented as an architecture graph in which nodes are the resources (processing resources and busses between them), and vertices are the links between the resources. This architecture instance is updated according to the performance analysis numbers.

[3]The model of computation in TensorFlow [5] is dataflow with extensions to permit to the nodes maintaining and updating persistent state, and for other branch and loop control structures in the graph similar to the timely dataflow graph model of computation presented in Naiad [32].

- The Mappings that define which task can be bound to which allocated hardware. This is important, especially that there could be operations that are not feasible on some hardware. The conducted design space exploration will search for the best mappings according to the performance analysis numbers (more details about the mapping process were presented in 3.2.3).

To conclude, a system model consisting of a specification comprising the application graph, the architecture graph, the mappings, and including all the constraints and conditions must be defined according to the Y-chart approach. This specification is shortly formally defined in the next Subsection 4.2.2.

4.2.2 Specification Formulation

The previously defined specification in 4.2.1 can be formally posed like the system representation in [6]. It comes down to defining a system model ζ where:

- The application graph is a bipartie directed graph $G_T(T, E_T)$ with $T = P \cup C$, where P is the set of the machine learning model's operations (the tasks) and C represent the set of communication tasks handling the data transfer between them (the tokens, in the case of TensorFlow the Tensors).

- The architecture is similarly represented by a directed graph $G_R(R, E_R)$ where R represents the Resources and $E_R \subseteq$ R $\times$ R.

- The set of mapping edges E_M where each m=(p,r) ϵ E_M is a mapping of a task p ϵ P into a resource r ϵ R.

To conclude this section, the design space can be set by building a specification with multiple mapping options for the tasks. The parameters of this specification depend on the provided machine learning model parameters, the hardware resources, and the constraints that come along with that. It will be demonstrated in the next section that exploring the design space of the defined specification is equivalent to an optimization problem where the fitness of the implementations according to a set of objectives is the target value to optimize (minimize or maximize depending on the problem).

4.3 Optimization

The design space exploration problem comes down to an optimization problem whose goal is to find the best implementations by finding the allocation, the routing, and the binding (scheduling is out of the scope of this thesis) corresponding to the specification and the expectations of the designer. In other words, when searching for the ideal implementations, the aim is at finding the solutions presenting the best performance at the objectives defined by the designer. The evaluation is accomplished with a function called fitness function, which resides in an objective function that evaluates every solution quantitatively. The optimization ought to maximize or minimize it. The approach to defining this fitness function in the thesis was similar to the scheme of TensorFlow's placement algorithm, in which the operations are mapped to resources according to pre-defined cost models. It will be shown later how these cost models are established.

In this section, a formulation of the optimization problem is given in Subsection 4.3.1, followed by an introduction to the fitness function in Subsection 4.3.2, and finally the explanation behind

the choice of the exploration algorithm which was carefully selected is presented in Subsection 4.3.3.

4.3.1 Optimization Problem Formulation

The formulation of the optimization problem relies on the specification one. In fact, the optimization is expected to generate highly optimal implementations. This is achieved through allocating the adequate resources from the architecture, binding the operation nodes to allocated resources, and routing the communication tasks to the busses, and usually in system-level synthesis scheduling, which was out of the scope of this thesis [6]. Thus formally, as posed in [6], an implementation is feasible if the following constraints are verified:

- An allocation directed graph $G_\alpha(\alpha, E_\alpha)$ is created as a sub-graph of the architecture graph G_R. This graph represents the set of resources R into which the tasks are mapped. E_α are the bus resources from E_R between the resources in α respecting the condition translated in equation 4.1:
$$e = (r, \tilde{r}) \in E_\alpha \iff r, \tilde{r} \in \alpha \tag{4.1}$$
- The bindings $E_\beta \subseteq E_M$ are set to depict the mapping of operation tasks to the allocated resources, and adhere to multiple conditions. As depicted in Equation 4.2, in an application of a feasible implementation each process task $p \in P$ is mapped to exactly one resource, and according to Equation 4.3, it can be mapped into only one allocated resource in the previously defined allocation graph:
$$\forall p \in P : card(\{m | m = (p, r) \in E_\beta\}) = 1 \tag{4.2}$$
$$\forall m = (p, r) \in E_\beta : r \in \alpha \tag{4.3}$$
- The union of routing trees of communication tasks $\cup G_\gamma$ such as $G_\gamma = (R_{\gamma,c}, E_{\gamma,c})$ represent a routing tree for all $c \in C$. Each directed tree $G_\gamma = (R_{\gamma,c}, E_{\gamma,c})$ is defined as a subgraph of the allocation one G_α according to Equation 4.4. Moreover, the root of $G_{\gamma,c}$ must be equal to the resource on which the previous sender process task is mapped to, following Equation 4.6:
$$R_{\gamma,c} \subseteq \alpha and E_{\gamma,c} \subseteq E_\alpha \tag{4.4}$$
$$\forall (p, c) \in E_T, m = (p, r) \in E_\beta : r \in R_{\gamma,c} \tag{4.5}$$
$$\forall (c, p) \in E_T, m = (p, r) \in E_\beta : card(\{e | e = (\tilde{r}, r) \in E_{\gamma,c}\}) = 0 \tag{4.6}$$

In addition to the previously defined linear constraints, the problem is bound by the system specification constraints. For example, the hardware resources configurations and the compatibility of some tasks to the specific target hardware. In general, these constraints can also be represented as linear equations or inequalities.

Allocation, binding, and routing are design steps solving combinatorial problems of mapping nodes to resources. Hence, the DSE comes down to a constrained combinatorial problem, where

the goal is to minimize the fitness function with respect to the problem's constraints. Hence, the problem can be formulated according to the following definition in [6] (and with respect to the general formulation of DSE in Subsection 3.2.2):

$$minimize f(x) \text{ with } a_i \leq bi, \forall i \in \{1, .., q\} \text{ with bi} \in \mathbb{Z}$$

In conclusion, every implementation ω synthesized by the DSE is in this thesis the result of determining the best resource allocation G_α, the adequate binding E_β, and the routings γ with respect to a set of linear constraints, according to a fitness function (refer to next Subsection 4.3.2) that guides the DSE by giving the overall cost of mapping.

4.3.2 The Fitness Function

As defined in the introduction of this Section 4.3, the fitness function evaluates every possible implementation by summarizing its performance regarding the targeted qualities quantitatively. For example, when designing embedded systems, there is a huge interest in finding the tradeoffs between cost and power. The fitness function gives performance numbers that help the designer make his decision about the tradeoffs. Fitness functions are usually used when employing genetic algorithms and in DSE in general, like it is the case in this thesis to direct it towards the optimal design implementations.

To establish a fitness function to guide the genetic algorithm used in the exploration process (refer to the next Subsection 4.3.3), a scheme needs to be set according to the expectations. Since in this thesis, the plan is to efficiently map elementary operations to hardware and the data between them to the busses to accelerate a neural network inference, the approach that was followed is to make estimations of the inference time of operations on the targeted hardware and build upon those cost models that are used by the fitness function to rate every solution. This approach is similar to what is used by the placement algorithm of TensorFlow, as explained in Subsection 4.1.3. The cost models can be built in different ways, for instance, benchmarks like what was done in this thesis.

To conclude, the fitness function is crucial for the exploration simulations to guide the algorithm that is presented in the next Subsection 4.3.3.

4.3.3 Choice of the DSE Algorithm

An exhaustive search and computing of the overall design space is not an adequate solution to the optimization problem because of the ample design space and the complexities coming from the constraints. Therefore, the strategy was to use a metaheuristic approach to explore the design space. Namely, evolutionary algorithms (EAs), which are commonly used in automated DSE. As can be seen in Figure 4.4, the EA performs modifications on a starting set of implementations derived from the specification and varies them with operators like mutation or crossover. A mutation resides in randomly varying a part of the genetic encoding of the considered implementation to obtain a new one. A crossover consists of crossing over by swapping chunks of the genetic representation of two implementations to produce new ones called offsprings. Hence, both operators generate a new set of offsprings that are evaluated with the fitness function. The fittest individuals are added to an archive A, whereas the ones that are Pareto-dominated by the implementations existing in the archive are dropped. This scheme is repeated iteratively until reaching the final number of runs, usually defined by the EA's parameters that have to be tuned

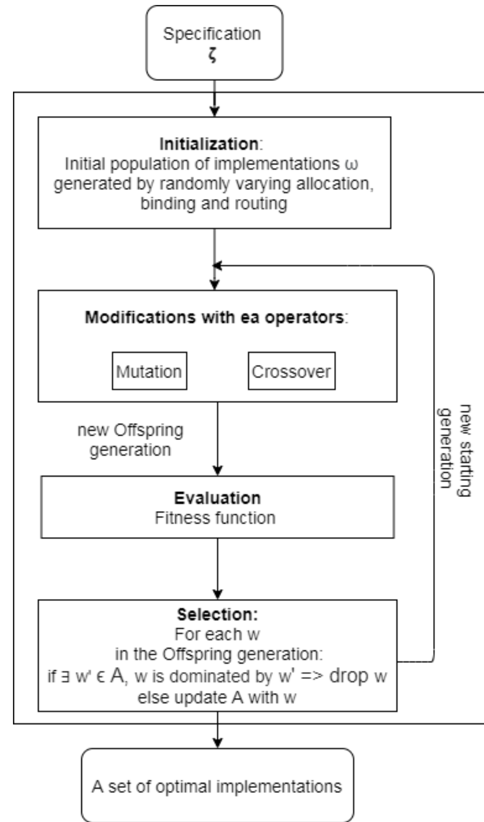

Figure 4.4: Representation of the steps of an evolutionary algorithm (EA) used for DSE taking as input a specification of a system and giving as output a set of optimized implementations (adapted figure from [6]).

by the user.

The issue here is that the constraints (for example, related to tasks not feasible on all hardware) make a part of the search space unfeasible. Furthermore, randomly varying the variables of the specification directly in the solution space of implementations, which is the main concept of the EAs algorithms, does not guarantee always obtaining feasible solutions (considering that the variations are random, for example, mutations or crossovers). Several solutions exist to solve this problem, such as repair heuristics where the infeasible reached implementation is repaired, but since system-level synthesis problems are NP-Complete, the repair heuristic would also be NP-Complete [6]. Another possible approach is using penalty functions, which central concept is about assigning penalties depending on the violation of the constraints. This approach was avoided because, in the literature, it is said to have a slow convergence, and it would also add extra modeling costs.

The solution to this problem, chosen in this thesis, is to use a hybrid optimization strategy combining an evolutionary algorithm and a backtracking-based Pseudo-Boolean solver that works only with the feasible region of the design space during DSE [6]. This hybrid optimization technique is called SAT-Decoding, and its main idea is relying on using a metaheuristic method that changes the branching strategy of the backtracking-based solver.

As explained in [6], this technique is based on Boolean Satisfiability solvers whose goal is to find a variable assignment satisfying all the constraints in a Propositional Satisfiability Problem[4]. In this thesis, the SAT-solver's task is to find a feasible solution satisfying the defined specification and its constraints. Most of modern complete SAT Solvers rely on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm, which is a complete backtracking search algorithm aiming at determining if a propositional logic formula is satisfied or not [33]. This algorithm starts from an initial set of unassigned variables, and then in a loop, a backtracking [5] procedure makes assignments and resolves the eventual conflicts. The algorithm uses a branching strategy guided by two vectors $\sigma \in \{0,1\}^n$ and $\rho \in \mathbb{R}^n$. When running this algorithm, every selection of an unassigned variable x_i by the algorithm is performed according to its corresponding ρ_i and assigned the value of σ_i. Once the assignment for x_i is done, the algorithm checks if there is a conflict that consists of a constraint being not satisfiable. In case there is one, the backtracking takes place, and the previous assignment is reversed. The previous steps are repeated in a loop until all the variables $x_i \in x = (x_0, .., x_n)$ are assigned, and in case the latter is achieved, and no conflict arises, the obtained assignment is translating a feasible solution to the considered problem.

The first step in using this technique is converting the overall problem into SAT (except the fitness function). In this thesis, it is done by establishing a Pseudo-Boolean encoding Ψ_ζ of the system-level synthesis problem[6] and the considered specification. As in [6], the following Boolean variables are laid out to encode the Pseudo Boolean constraints (previously formulated in Subsection 4.3.1):

- r as a Boolean variable for each resource $r \in R$ to specify if r is allocated or not, equal to 1 if $r \in \alpha$, and to 0 if $r \notin \alpha$.
- m as a Boolean variable for each mapping edge $m \in E_m$, equal to 1 if $m \in E_\beta$ and to 0 if $m \notin E_\beta$.
- c_r as a Boolean variable corresponding to each communication task $c \in C$ and resource $r \in R$, equal to 1 if c is routed over r and 0 if not.
- $c_{r,\tau}$ as a Boolean variable relative to each communication task $c \in C$ and resource $r \in R$ to indicate at which step the communication is routed over the resource.

Once the Boolean variables are set, the specification with its different elements and constraints are encoded. As an example to how it is done, the constraints related to the binding in Equation 4.2 and 4.3, presented in Subsection 4.3.1, are encoded as follows:

$$\forall p \in P : \sum_{m=(p,r) \in E_M} m = 1 \tag{4.7}$$

$$\forall m = (p,r) \in E_M : r - m \geq 0 \tag{4.8}$$

[4] A Propositional Satisfiability Problem (also called Boolean Satisfiability Problem) is the problem of identifying if an interpretation exists to satisfy a given Boolean formula[11].

[5] Backtracking in this context means reversion of the variable assignments done before.

[6] except scheduling, which is out of the scope of this thesis.

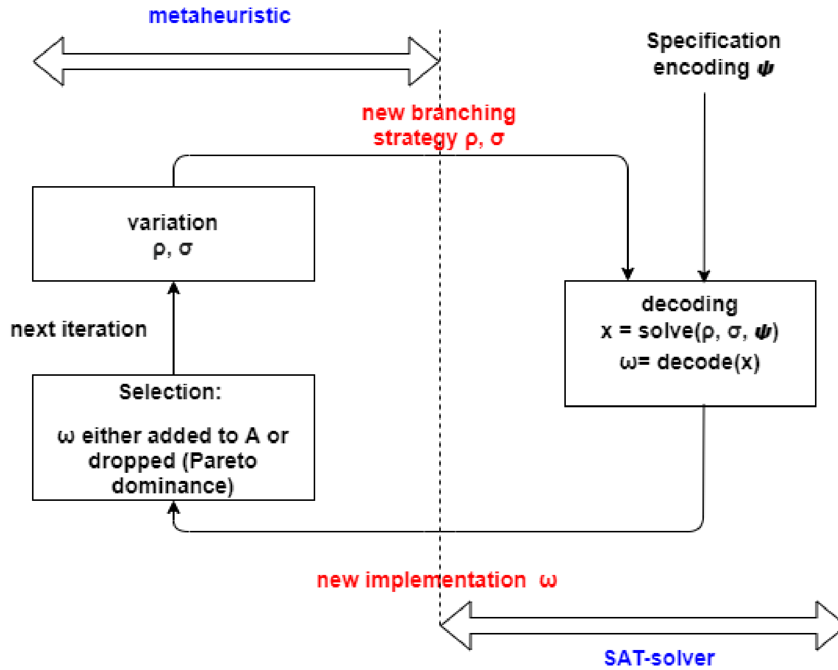

Figure 4.5: Bloc diagram summarizing the SAT-decoding approach (Figure adapted from [6]).

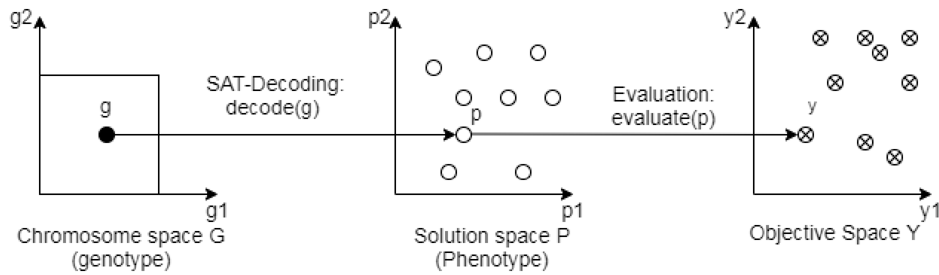

Figure 4.6: Representation of the chromosome space, the solution space, and the objective space used in the optimization combining the EA with SAT-decoding (Figure adapted from [7]).

The previous encoding process is similarly performed for all the elements of the problem to obtain Ψ_ζ. As can be seen in Figure 4.5, the metaheuristic algorithm, here the EA, interferes in the branching strategy of the SAT-solver. It varies the vectors ρ and σ of the SAT-solver that results in a variable assignment x in the chromosome space. This way, every genotype generated by the EA in the chromosome space is satisfying the constraints of the problem. An SAT-decoder decodes the obtained genotype into a phenotype, which corresponds to a feasible implementation in the solution space P. The next step is evaluating with the fitness function the phenotype and either adding it to the archive A of the fittest implementations or dropping it.
Hence, as can be seen in Figure 4.6, when using this method, there is a clear distinction between the bounded chromosome space where the search takes place, the solution space corresponding to only feasible solutions, and the objective space where the implementations are evaluated.

Therefore, SAT-decoding was the go-to strategy in the exploration process.

To shortly conclude this chapter, a scheme and the various used algorithms in this thesis are presented as a solution to accelerating inference of machine learning models by efficiently distributing the inference based on the conduct of a design space exploration according to the Y-chart methodology.

5 Experimental Setup

The goal of the implemented experimental evaluation is to present a proof of concept of conducting a design space exploration for machine learning models on different resources, including hardware accelerators, in order to accelerate inference. The implementation is following what was theoretically presented in Chapter 4, and can be divided into two major phases: a machine learning benchmarking phase and a DSE phase. The first phase consists of running benchmarks on the available hardware. The results are analyzed to build cost models that constitute the fitness function adopted by the DSE. The second phase consists of modeling the problem according to the Y-chart approach utilizing existing frameworks and performing the DSE.

In this chapter, the experimental setup is introduced. First, in Section 5.1, the HW setup is presented with a focus on the used hardware accelerators. Then, in Section 5.2, the benchmarking process is described. Finally, the tools used in the DSE and method are shown in Section 5.3.

5.1 Hardware setup

One of the goals of this thesis is to make use of the capacities of hardware accelerators. That is why the used hardware includes state-of-the-art technologies in artificial intelligence hardware acceleration and is presented in the next Subsection 5.1.1.

5.1.1 Used AI Hardware Accelerators

5.1.1.1 GPU

Graphic processing units are electronic circuits that were created to boost the performance of video graphics previously, and starting from the end of the 2000s, they were proven to be good AI hardware accelerators. Defacto, working alongside CPUs, they are efficient thanks to their parallel computing capabilities. The GPU used in this thesis is the Quadro M4000 professional graphics card with an NVIDIA Maxwell GPU architecture. It is designated for demanding computing applications and large complex datasets with its 8GB GDDR5 GPU memory and the rest of its features, as stated in its datasheet [34].

More powerful GPUs are provided by the same vendor on the market, but this one was used because it was available on the host computer where the tests were carried out. GPUs working alongside CPUs have made the deployment of deep learning models easier. Nevertheless, they have some disadvantages. For instance, they are not suitable for edge devices because of their high power consumption and heat generation, where embedded systems use self-cooling methods. Therefore further solutions can be considered, as the AI hardware accelerators USB sticks presented in the next subsections.

5.1.1.2 NCS

NCS denotes the Neural Compute Stick released by Intel as a highly efficient and low power AI inference accelerator at an affordable price. It incorporates the Intel Movidius Myriad 2 Vision Processing Unit, which features 12 SHAVEs (Streaming Hybrid Acceleration Vector Engines) that are re-programmable processing cores adapted for delivering maximum performance with complex visions and image workloads. One of the most interesting features of this device is that it allows inferencing complex and large machine learning models on edge devices without going through cloud solutions. Hence, it is suitable for real-time applications and inference in areas without connectivity.

Intel made available two software interfaces to get started using their hardware: the "NCSDK2" and the "openvinotoolkit". Nowadays, they advise using the second one because they do not support the "NCSDK" anymore, and they stated that in their Github end of August 2019. Nevertheless, because the tests with the hardware accelerator during this thesis started before the announcement, the "NCSDK2" was used. Another point that encouraged to keep on using it is to see the impact of varying the number of used SHAVEs during the tests, which is only possible with the "NCSDK2".

This SW Development Kit comprises three command-line tools that ensure profiling, tuning, and compiling neural network models on host systems. The "mvNCCompile" command-line tool permits compiling a TensorFlow or Caffe machine learning model to an internal representation comprehensible by the Intel Movidius Neural Compute SDK. In the case of TensorFlow, it takes as input a meta graph of the TensorFlow model (a type of file to which a machine learning model built with the framework TensorFlow and which extension is ".meta"), the maximum number of SHAVEs that the user wants to allocate for running inference which can be set to a value varying between 1 to 12 SHAVEs, the input node, the output node, and other optional parameters. It is relevant to note that by giving the input and output node of the TensorFlow graph as inputs to the tool, it is possible to give to the NCS only a subgraph of the considered graph to process. It is also relevant to mention that the specified number of SHAVEs corresponds to the maximum number of allocated ones for inference of the input neural network. However, the device runtime code decides the number of used SHAVEs where there is no inference performance deterioration. It is consequently not possible to control the internal mapping of tasks to the SHAVEs, and only indicating the maximum number of SHAVEs that are allowed to be used (maybe when Intel releases the library with an open-source license, this possibility could be further examined). The command-line tool gives as output an Intel Movidius graph file format that can be read and executed by the NCS. The second tool in the SDK is "mvNCProfile", which executes the same steps as the "mvNCCompile" tool and runs the NCS compatible output graph on a USB-connected NCS and logs profiling results from the run to texts and HTML profile reports. The "mvNCProfile" was used in the benchmarking tool. Last, the command-line tool "mvNCCheck", which was only shortly tested during this thesis and which can be used to check the validity of a specific neural network model on the neural compute device. The latter test of accuracy is done by running one inference of the model on the NCS and one on the host computer and comparing the loss. Indeed, it is essential to make sure that the `fp16` representation on the NCS does not significantly deteriorate the deep learning model's accuracy. The mvNCCheck is limited to image recognition models, but the idea behind it is to check the possible loss of accuracy during inference because of the 16-bit quantization, although there could be an improvement that resides in

execution time for example.

The usual workflow when using the NCS can be seen in Figure 5.1. The user starts by training the model on different hardware (for example, a server or a laptop). The trained model is then fed to a development host and the NCS to undergo profiling, tuning, and compiling steps using the tools of the previously introduced software development kit. The next activity is prototyping with the user interface made available by Intel (not used in this thesis).

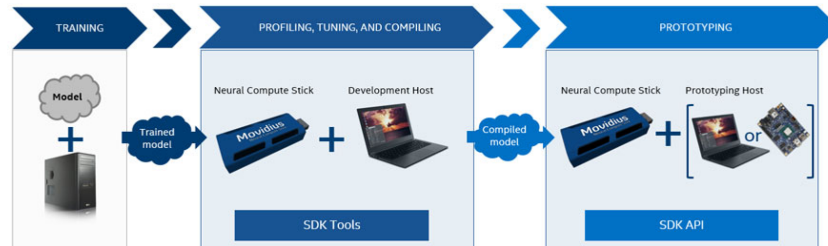

Figure 5.1: A Diagram from [8] exposing the usual workflow of development using the Intel Movidius Neural Compute Stick.

In summary, the NCS is an artificial intelligence hardware accelerator from Movidius Intel that comes with the mentioned characteristics and tools that are used in this thesis in the benchmark suite, and which aim is only inferencing (not training) on edge devices.

5.1.1.3 Coral

The Coral USB accelerator is a USB gadget provided by Google that comprises an onboard electronic circuit as a co-processor to accelerate Machine learning [35]. This latter system-on-ship is the Edge TPU ASIC that stands for the Tensor Processing Unit, which was built by Google with a domain-specific architecture [9]. More specifically, since it targets neural networks training and inference, which depend on vast amounts of additions and multiplications between inputs and the model's parameters (e.g., Matrix multiplications), these units have been conceived as Matrix processors for machine learning tasks. The advantage versus general-purpose processors is the possibility of executing the previous operations faster and cheaper like most of ASICs versus GPPs despite the design that comes at an expensive cost. This USB hardware accelerator has proven itself to perform very well ML inference with low power consumption. As an example given by Google, it executes the Mobilenet v2 at 400 FPS with low power [35].

The vendor imposes some limitations. It has system requirements like being connected to a Linux host computer with Debian 6.0 and higher operating systems, x86-64 or ARM32/64 with ARMv8 instruction set (Recently in January 2020, it was released that the Coral Edge TPU could be used with other operating systems like windows). Moreover, the EdgeTPU library is not open source, and only a compiler is provided for compiling the ML model to a representation comprehensible by the hardware [35]. These models have to be TensorFlow Lite fully 8-bit quantized models (with other conditions) to be compiled.

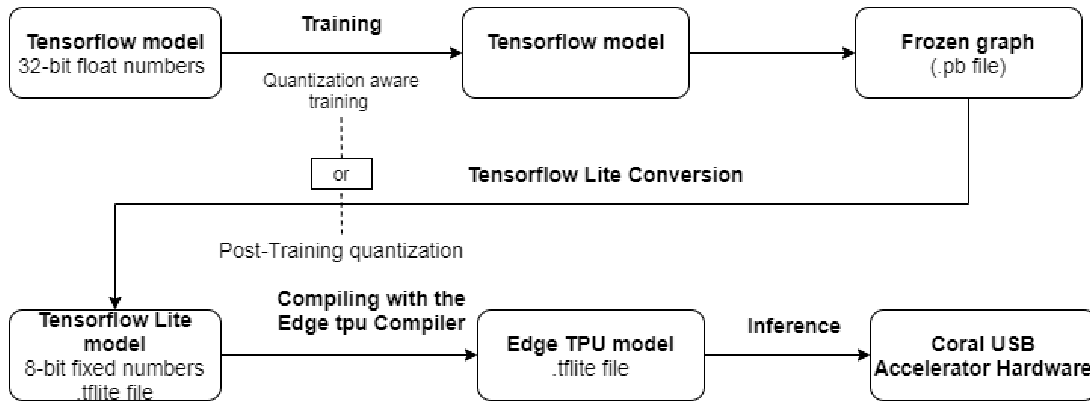

Figure 5.2: Summary of the workflow of a model's creation for the Coral Edge TPU USB accelerator adapted from [9].

The basic workflow for developing a model on the Coral can be seen in Figure 5.2. The user starts with training a TensorFlow pre-trained model or with a TensorFlow ML model that he constructed. It then transforms the model into a frozen TensorFlow protocol buffer graph to convert it to a TensorFlow Lite model. Since one of the requirements of deployment on the Coral is the 8-bit quantization, the model can be quantized during the training or after the training. After that, as can be seen in Figure 5.3, the TensorFlow lite 8-bit quantized obtained model is passed to the Edge TPU compiler that partitions the model only once to unsupported operations and supported operations. The Edge TPU's feasible operations are transformed into an Edge TPU Custom operation that runs on the Coral. The conditions, consisting of limited feasible operations on the Coral, constrain the DSE process. Indeed, the exploration algorithm has to avoid unfeasible regions of the design space automatically (more details in Subsection 6.2).

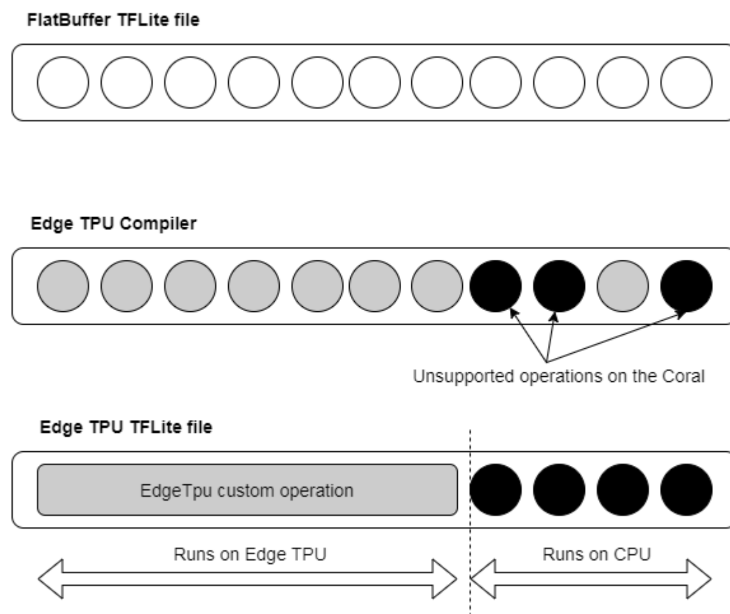

Figure 5.3: Partition of the Flatbuffer TensorFlow Lite model's operations by the Edge TPU Compiler into feasible operations to run on the Coral accelerator and unfeasible operations to run on the CPU, adapted from the Coral Edge TPU official documentation [9].

Finally, the Coral Edge TPU presented in this Subsection is an AI hardware accelerator presenting several benefits and constraints and supporting both training and inference.

The previously presented hardware accelerators were used in the thesis despite their limitations, the constraints that they introduce, and the SW specifications requirements.

5.2 Machine Learning Benchmarking

5.2.1 Introduction to the Benchmarking Environment

There are distinct AI benchmarking suites. Building an adequate benchmarking tool that meets the expectations, as detailed in the next subsections, was required. The addressed benchmarking tool had to take into consideration the available HW Resources presented in the previous Section 5.1 and the requirements of the problem. On the one hand, using TensorFlow as the framework for building and executing machine learning models. Hence, the decision to rely on the TensorFlow Model Benchmark tool, whose usage is detailed later (refer to the next Subsection 5.2.2). This tool only supports CPU or GPU hardware. Thus, it was also necessary to integrate with it the Coral USB Edge TPU stick. Contrariwise, for benchmarking the execution on the NCS USB stick, the mvNCProfile command-line tool was used to first compile the TensorFlow model to a readable Intel Movidius NCS graph, and then profile the graph on it. On the other hand, the tool should systematically realize the benchmark (by the automation of several tasks). Finally, the benchmark results are later analyzed and interpreted to build the cost models needed during the DSE.

Shortly, the benchmarking environment relies on the available hardware introduced in Section 5.1 and a software tool presented with more details in the next Subsection 5.2.2.

5.2.2 Benchmarking Tool Description and Setup

As can be seen in Figure 5.4, the benchmark tool includes three modules aiming at benchmarking the input on the CPU and GPU, the NCS, and the Coral.

The benchmark on the CPU and GPU of the host computer is based on the TensorFlow benchmark. The latter is a C++ binary that takes as input a TensorFlow compute graph and some information about it like the input layer, the output layer, the input shape, and others (as indicated in the TensorFlow Github [36]). It loads the graph first into a new Session to make a single initialization run through the graph. Then, it runs the graph multiple times according to the default number of runs or a number of runs given by the user. It saves timing information about the inner inference (the average execution time of each node) and the outer inference(the average execution time of the whole model).

The inputs to the graph during the benchmark are constructed as dummy randomly generated inputs from the information of the input shape and input type to the model that is given by the user. A part of the outputs is saved to protocol buffers upon request or can be fetched from the command line. It was more interesting (more information) to parse the output logged to the command line.

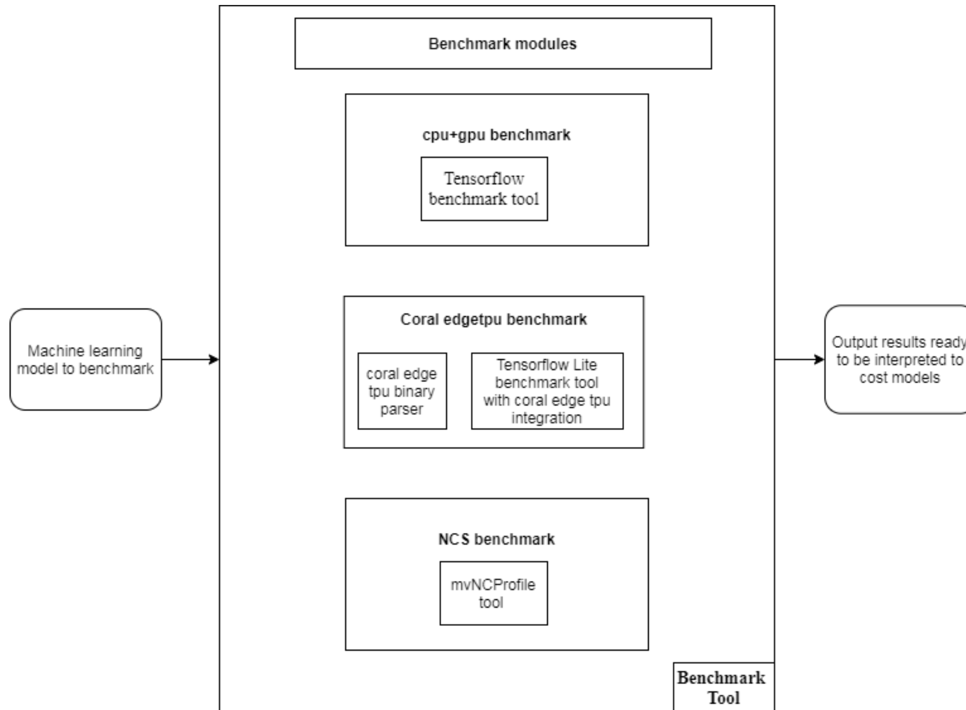

Figure 5.4: Benchmark tool Block Diagram.

The TensorFlow benchmark has to be built from the source code of TensorFlow (currently in the TensorFlow version r1.14) and is not included in the built-in versions of TensorFlow that can be installed or used through different IDEs. As a direct consequence, it was necessary to build TensorFlow from source and then build the benchmark binary. The first step was to ensure all the required setups for building a TensorFlow package from source. As the used host computer in this thesis is working with a Red hat Enterprise operating system (namely Red Hat Enterprise Linux Server release 7.7, Maipo) and TensorFlow source build requires the Ubuntu Linux Distribution, there was an obligation for setting up an environment independent from the current operating system. One good and plausible solution that was used in this thesis is Docker containers [37]. The first idea was to run a Docker container for a specific hardware benchmark, but, considering that the initialization of such containers comes already with an overhead, it was more practical to run all the benchmarks in one Docker container. Specifically, the choice was first to run a TensorFlow GPU Docker container from the official TensorFlow Docker images in the Docker hub repository within the privileged mode to have access to the USB hardware accelerators connected to the host to prepare a container that would have all the requirements to run the benchmark and which is pushed as a Docker image for re-usability. Several setups needed to be done, most importantly:

- Downloading and setting up the NCSDK2 for running the profiling on the NCS with the mvNCProfile tool.
- Adapting the TensorFlow lite benchmark code to run the benchmark on the Coral Edge TPU. This step is summarized in the block diagram of Figure 5.5 and consists of two main tasks. On the one hand, for having the same input for the benchmark tool, it was necessary to include compiling the input deep learning model that is given by the user as a TensorFlow model (a protocol buffer with the extension .pb) to a TensorFlow

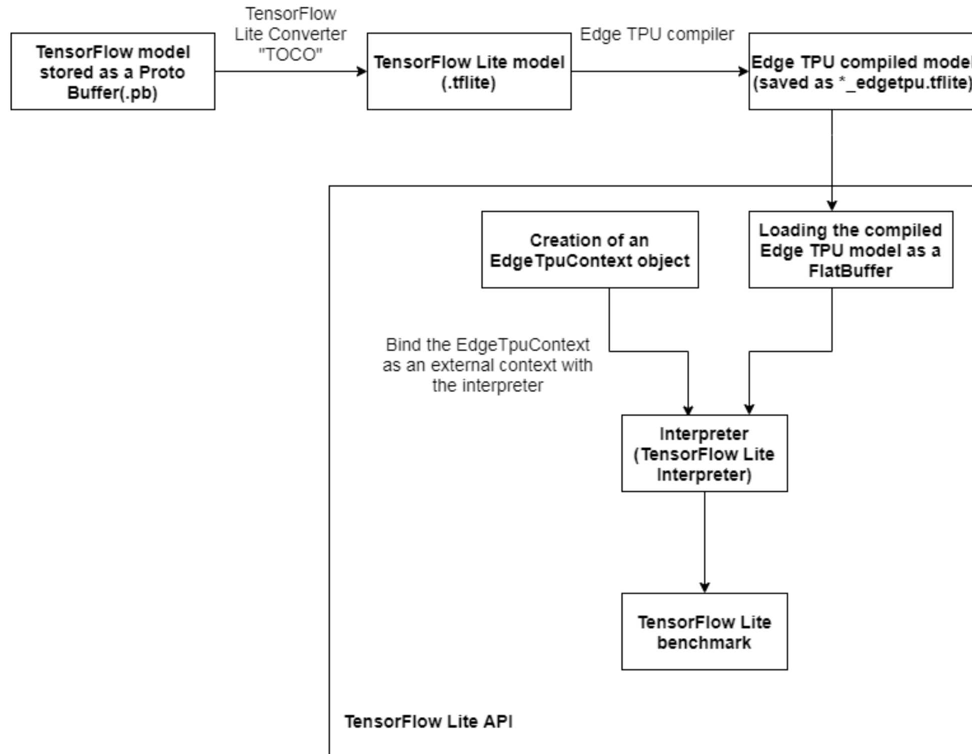

Figure 5.5: Summary of the tasks that must be fulfilled to run the TensorFlow Lite benchmark on the Coral Edge TPU.

Lite model (.tflite) which was performed with the TensorFlow Lite Converter command-line tool “TOCO”. The TensorFlow Lite model is then compiled into a file that would be compatible with the Edge TPU with the command line compiler Edge TPU. On the other hand, running the TensorFlow Lite benchmark is equivalent to running inference of a TensorFlow Lite model on a HW device with dummy input data. Hence, it is necessary to go through an interpreter, which is designed to work with a static graph ordering and a less-dynamic memory allocator to minimize the execution time, the initialization, and the overall load [20]. Different configurations needed to be done but, most importantly, loading the compiled Edge TPU model as a FlatBufferModel, creating an EdgeTpuContext, and then building an Edge TPU interpreter similar to the one needed by the TensorFlow Lite APIs to inference. Consequently, the compiled Edge TPU model's inference time is given as if it was run as a TF Lite model.

- Building TensorFlow from source with all the components with the Bazel SW (Free Tool developed by Google to automate building and testing software. In the case of TensorFlow, everything is built with Bazel).

Finally, the Docker container with the TensorFlow built-in package and specifically benchmark tools that fit the requirements are saved as a Docker image in the local Docker repository. Thus, running the benchmark requires that the user provides the models to benchmark and some parameters. The latter includes the number of runs of the benchmark because every time the benchmark was needed to be run a sufficient number of runs in order to have an average inference time on this range. The benchmark output results are analyzed in Chapter 7 to be used

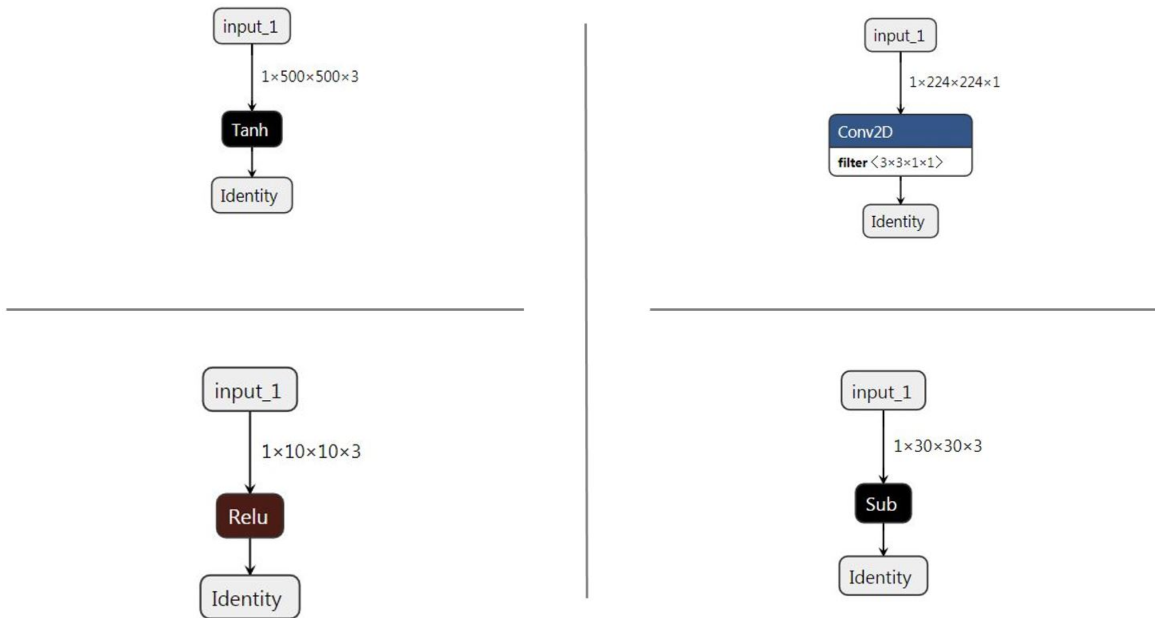

Figure 5.6: Operation models visualized from the created protocol buffer files with the online tool `Netron` [10].

by the DSE. Following the TensorFlow placement algorithm in Subsection 4.1.3, the goal of the benchmarking is to estimate the cost of mapping for ML operations on the available hardware. To obtain that, the benchmarking tests were conducted on TensorFlow "operation models", which are explained in the next subsection.

5.2.3 Benchmarking Individual Operation Models

TensorFlow models are represented as dataflow graphs whose edges are tensors (`tf.Tensor`), and nodes are TensorFlow operations (`tf.Operation`). In order to build a cost model for individual operations that are used by the DSE, TensorFlow single operations models were created. The creation was partially automated with a script. Some examples of the "one-operation" models that were created in this context can be visualized in Figure 5.6.

Then, the benchmark was performed 50 times for each encapsulated operation model, and the average value of the inference time through the runs for each hardware was saved. During each of these 50 runs, the following parameters have been changed: the input shape and consequently, the input size of the model, the input type, and the number of the used SHAVEs in the case of the NCS.

The results of the benchmarking are presented and analyzed in Section 6.1.

5.3 DSE

5.3.1 Used DSE Frameworks

5.3.1.1 OpenDSE

OpenDSE is a framework of design space exploration for embedded systems whose core is written in Java. This framework was chosen because it follows the Y-chart methodology and is based on a system model appropriate for system-level DSE, which fits the desired approach, as presented in Chapter 4. The considered problem presented in 4 consists of a DSE at system-level for machine learning systems inference following the Y-chart methodology.

Another reason for choosing OpenDSE was its modularity and scalability because it is built with Google Guice, a dependency injection framework. Hence, more freedom during the optimization in the choice of the optimization algorithm and the definition of the fitness function. Finally, the OpenDSE framework optimization part is based on the opt4j metaheuristic optimization framework, which fits the optimization strategy targeted in this thesis.

5.3.1.2 Opt4j Stack

The Opt4j stack used in this thesis through OpenDSE constitutes the metaheuristic base of the framework. It presents different DSE algorithms from which it was chosen to use an evolutionary algorithm. Moreover, it implements SAT-Decoding, distinguishing the genotype space, the phenotype space, and the objective space where the solution is evaluated with the fitness function.

5.3.2 Implementation of the DSE

The realization of the DSE with the chosen frameworks according to the designated strategy presented in Chapter 4 can be divided into two phases. First, the creation of the design space representing the problem is presented in 5.3.2.1. Second, the exploration process is laid out in 5.3.2.2.

5.3.2.1 Establishment of the design space according to the Y-chart methodology with OpenDSE

As demonstrated in Section 4.2, a specification including all the components to represent the design space needs to be defined. The latter is established in this thesis as a java class named `FullSpec` from where it is possible to instantiate a specification by just providing the TensorFlow model summarized text file generated from the Protocol Buffer frozen file as can be seen in Listing 5.1.

```
FullSpecDef fullspecdef = new FullSpecDef(MLmodel);
```

Listing 5.1: Code line where a specification is instantiated given the machine learning model to consider.

Following the OpenDSE programming model and the requirements set in this thesis, every defined specification includes:

- An application (class `Application` in OpenDSE), which consists of a set of tasks (class `Task`) and dependencies (class `Dependency`) between them as defined in Listing 5.2. The

tasks include processes and communication tasks. As in the experimentation, the considered application was any TensorFlow machine learning model, the tasks representing processes then are the computations, the communication tasks are the tensors, and the edges are the dependencies between them. This could be generalized to dataflow graphs representing machine learning models as well.

```
Application<Task, Dependency> application = new Application<Task, Dependency>();
```

Listing 5.2: Code line where an application is instantiated in OpenDSE.

- An architecture (class `Architecture` in OpenDSE), which, in this thesis, consists of the CPU and GPU, the NCS, the Coral Edge TPU accelerator, and the USB busses between them. Hence, its definition, as can be seen in Listing 5.3 with OpenDSE. The resulting instantiated graph, as can be seen in the example in Figure 5.7, is the architecture graph in which vertexes are the available resources and edges are the links between them. Specifically, two different types of resources can be seen, the ones representing the processing hardware and the USB busses defined as `bus12` and `bus23` between them.

```
Architecture<Resource, Link> architecture = new Architecture<Resource, Link>();
Resource r1 = new Resource("NCS");
Resource r2 = new Resource("Desktop(CPU + GPU)");
Resource r3 = new Resource("coral_edgetpu");
Resource bus12 = new Resource("bus12");
Resource bus23 = new Resource("bus23");
architecture.addVertex(r1);
architecture.addVertex(bus12);
architecture.addVertex(bus23);
architecture.addVertex(r2);
architecture.addVertex(r3);
Link l1 = new Link("l1");
Link l2 = new Link("l2");
Link l3 = new Link("l3");
Link l4 = new Link("l4");
architecture.addEdge(l1, r1, bus12);
architecture.addEdge(l2, bus12, r2);
architecture.addEdge(l3,r2,bus23);
architecture.addEdge(l4, bus23, r3);
```

Listing 5.3: Code defining the architecture in OpenDSE. First, the resources are instantiated from the available hardware and busses between them and added to the architecture as vertexes. Then the different links between the hardware are created to represent the edges between the vertexes of the architecture.

- Mappings (class `Mappings` in OpenDSE) that delimits how tasks are mapped to resources. In OpenDSE, they are first instantiated, as can be seen in 5.4. Then, the possible mappings are added one by one to them, knowing that there is more than one possible mapping to each task, and the exploration selects the optimal ones.

```
Mappings<Task, Resource> mappings = new Mappings<Task, Resource>();
```

Listing 5.4: Code line where mappings are instantiated in OpenDSE.

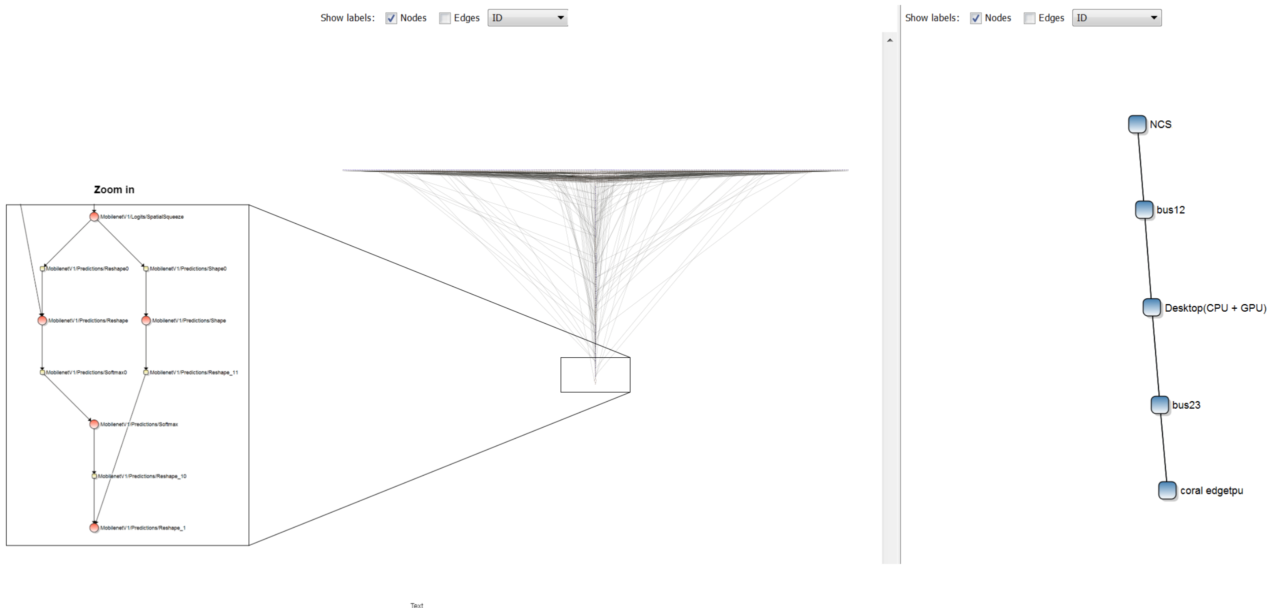

Figure 5.7: Visualization of the specification of the TensorFlow model mobilenet with OpenDSE viewer.

The resulting specification is created in OpenDSE as a class, as shown in Listing 5.5.

```
Specification specification = new Specification(application, architecture, mappings);
```

Listing 5.5: Code line where the Specification is instantiated in OpenDSE.

Once created, the latter can be saved to an XML file that stores all its details and can be viewed with the OpenDSE user interface, for example, the output specification of the mobilenet v1 TensorFlow supported model in Figure 5.7. The architecture instance of the specification can be seen in the right of the figure. It consists of the CPU and GPU modeled as one resource, the NCS accelerator, and the Coral Edge TPU. It also includes the communication connections between them modeled as busses. The application graph of mobilenet, which has a large number of nodes, is modeled as a large application graph, whose details can be seen only by zooming in, as shown in Figure 5.7.

The next step after creating the specification representing the design space is the exploration process.

5.3.2.2 Exploration with OpenDSE and Opt4j stack

5.3.2.2.1 Presentation of the workflow of the Exploration with the used frameworks

The DSE implemented with OpenDSE and the underlying metaheuristic stack Opt4j is based on the SAT-Decoding approach enabling one of the supported metaheuristic algorithms to search through solutions' encodings (genotypes) restricted by a set of SAT constraints in the chromosome space. Since the chosen exploration approach in this thesis consists of using an evolutionary algorithm for the search coupled with an SAT-solver, it was possible to implement the targeted approach (more details about the theory behind approach were given in 4.3).

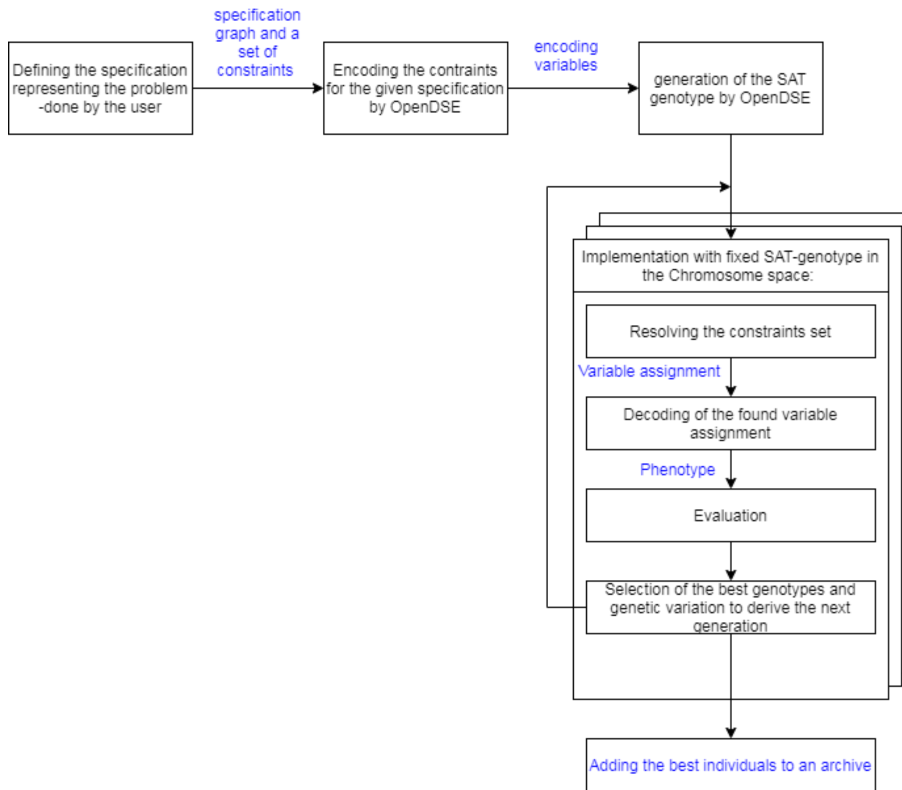

Figure 5.8: Summary of the exploration's workflow with OpenDSE.

Figure 5.8 summarizes the overall workflow when using them to implement the selected strategy. First, the user defines the specification and its constraints, which represent the design space of the problem, as explained previously in Subsection 5.3.2.1. The problem is then converted to SAT by the `Encoding` module of OpenDSE and its different dependencies in paragraph 5.3.2.2.2, which encodes the specification and its constraints into a Pseudo-Boolean encoding. The resulting encoded variables are then used in combination with the EA to generate the SAT genotypes [1]. The next step concerns every implementation with a selected SAT genotype in the chromosome space. This step starts with resolving the constraints with the strategy defined in the genotype into a variable assignment. The latter is decoded into a phenotype, i.e., a feasible implementation. This implementation is then evaluated by the fitness function, Pareto compared to the archived implementations, and either stored to the archive with the fittest individuals or eliminated. The archive is also used to derive the next generation, and then the same process is repeated for each implementation until, in the case of using an EA, the number of generations that needs to be studied is reached.

To summarize, the steps can be reduced to defining the problem, followed by its encoding into SAT constraints and variables. The EA varies the SAT solver's branching strategy so that it generates genotypes in the chromosome space. Then, a number of the genotypes in the chromosome space are randomly selected to be decoded into phenotypes constituting an implementation.

[1] The genotype as defined in Subsection 4.3.3 is the genetic representation of a feasible implementation in the chromosome space (in the case of this thesis, thanks to the SAT-constraints). Specifically, it is a variable assignment of the Pseudo-Boolean variable x, which is encoding an implementation.

The latter is evaluated and either selected within the archive of the fittest individuals or discarded. Since the chromosome space was defined already limited by the problem's constraints and conditions, the resulting individuals in the solution space are only feasible implementations.

In the next paragraphs, a closer look at how these steps are implemented under OpenDSE is presented.

5.3.2.2.2 Encoding of the Specification by OpenDSE and creation of the genotypes

The `DesignSpaceExplorationCreator` class is responsible for the Pseudo-Boolean encoding of the specification.

Two types of constraints are encoded. The ones that are already built-in OpenDSE when defining a specification, for example, a rule that for each process task in the application graph, only one mapping edge has to be activated in the implementation. These types of rules are related to how OpenDSE was implemented, and especially to the constraints that come with a system-level-synthesis problem. Indeed, for an implementation to be feasible, several conditions must be verified by the allocation, binding, routing, and scheduling (usual requirements when conducting a system-level-synthesis, as shown in Subsection 4.3.3, and presented in [6]). The latter described constraints are encoded into a genotype by the `SATCreatorDecoder`, associated with `SATConstraints`, which itself is associated with the underlying `Encoding` module. The other constraints are user-specific and need to be introduced where OpenDSE allows some degrees of freedom, where constraints can be added as parameters (class `Parameters`), Attributes (class `Attribute`), or constraints (class `SpecificationConstraints`).

The constraints of the system-model in this thesis concern:

- the system-level-synthesis problem, which is already built-in with OpenDSE.
- the mapping of operations to only hardware that supports them is implemented by the user. It was important to precise the operations that are feasible on the Coral Edge TPU and which are listed in the official website documenting this AI HA [35]. For the NCS, some of the operations tested during the benchmark appeared to be not compatible with the hardware and hence excluded.
- The number of SHAVEs of the NCS, which is a number between 1 to 12, was implemented as a parameter whose value is selected during the exploration (Parameters.select(), as can be seen in Listing 5.6).

```
Resource r1 = new Resource("NCS");
r1.setAttribute("num_of_shaves",
Parameters.select(12, 12, 1, 2, 3, 4, 5, 6 ,7, 8, 9 , 10, 11));
```

Listing 5.6: Code showing how the number of SHAVEs of the NCS was implemented in the specification.

The resulting encoding is used to generate SAT Genotypes forming the chromosome space.

5.3.2.2.3 Decoding

From the latter resulting chromosome space, the evolutionary algorithm selects SAT genotypes that undergo operators like mutations and crossovers to generate offsprings, and the tasks

of the EA algorithm are performed as explained in Subsection 4.3.3. Each considered genotype is decoded with the injected class `DesignSpaceExplorationDecoder` that has several dependencies in between which the associated module `SATCreatorDecoder` that implements the `AbstractSATDecoder` interface from Opt4j. This module translates the strategy of encoding and decoding the SAT genotypes that are encoding implementations.

Each decoded phenotype which resides in a feasible implementation needs to be evaluated during the exploration process.

5.3.2.2.4 Evaluating the Specification

It was needed to inject an external evaluator to the `DesignSpaceExplorationEvaluator` module in OpenDSE. This evaluator is defining a fitness function that computes the overall cost of mapping of the considered implementation. It looks at every mapping of operation to a resource, assigns to it a cost of mapping, and every routing to which a cost of communication is assigned.

A coding part of the `evaluate()` method of the `ExternalEvaluator` can be seen in Listing 5.7. The latter shows how the overall cost of mapping (`cost_of_mapping`) is given by adding each operation's cost to it (`MappingCost(m)` in the example). `MappingCost` is a method that outputs the cost of mapping of one task to one hardware based upon the benchmarking results. The second loop in 5.7 shows that the costs of communication are also added where defined.

```java
Architecture<Resource, Link> architecture = impl.getArchitecture();
Mappings<Task, Resource> mappings = impl.getMappings();
Routings<Task,Resource, Link> routings = impl.getRoutings();
Set<Element> elements = new HashSet<Element>();
elements.addAll(architecture.getVertices());
elements.addAll(architecture.getEdges());
elements.addAll(mappings.getAll());
Application<Task, Dependency> app = impl.getApplication();
double cost_of_mapping = 0.0;
for (Mapping<Task, Resource> m: mappings) {
    Task current_task = m.getSource();
    if (current_task.isDefined("input_shape")){
    cost_of_mapping = cost_of_mapping + MappingCost(m);
    }
}
for (Architecture<Resource, Link> r : routings.getRoutings()) {
    Iterator<Link> routing_it = r.getEdges().iterator();
    while (routing_it.hasNext()){
        Link link_n = routing_it.next();
        cost_of_mapping = cost_of_mapping +
        ((Double) link_n.getAttribute("cost")).doubleValue();

    }
}
```

Listing 5.7: Code showing a part of the method evaluate that can be found in the ExternalEvaluator injected to the DesignSpaceExploration problem and which role is to give the overall cost of mapping of each implementation.

To conclude this Subsection 5.3.2, the design space exploration was implemented with OpenDSE. An overall simplified architecture of the used classes is presented in Figure 5.9, which shows that the DSE depends on 3 classes making the distinction between genotype in the chromo-

some space, phenotype in the solution space, and evaluator in the objective space. Mainly, a `DesignSpaceExplorationCreator` module creates genotypes from the encoding of the user-defined specification. Then the `DesignSpaceExplorationDecoder` module using an SAT-Decoder decodes the genotypes into feasible implementations. Next, the `DesignSpaceExplorationEvaluator` computes the fitness of these implementations. Since the fitness function depends on the benchmarking results, an external evaluator was implemented (and injected as a dependency with Google Guice into the framework Evaluator). The fitness function needed to be set with the corresponding objective.

Several assumptions were needed in order to facilitate the experimental setup and are presented in the next section.

5.4 Assumptions

Assumptions were made. Indeed, the cost models that are established later in Section 6.1 from the benchmark depend on the type of the considered operation, the target hardware, the input type, the input size, and in the case of the NCS, the number of SHAVEs. Considering the large number of operation types in the TensorFlow framework that need to be encapsulated individually in TensorFlow models to be tested with the benchmarking tool, the cost models were built only for a set of operations as a proof of concept. Therefore, in the fitness function, the operations that do not have a cost model were considered to have a cost of mapping equal to 0. As the fitness function gives the overall cost of mapping of each obtained solution by summing the cost of mapping of the operations and the communication costs, the operations that do not have a cost model do not affect the cost given to the mapping of the considered solution. Thus, they are mapped to the closest hardware to avoid additional communication costs. Hence, it was considered that this assumption does not have a significant impact on the overall optimization of the inference machine learning model.

The second important assumption that was made is related to the communication cost. This needed to be taken into consideration in the computing of the cost of mapping. The problem was that while performing the benchmark tests, the cost of communication was not examined. It is though included in the execution time when sending the model from the host computer to the hardware accelerators. This communication time is logically depending on different factors, like the type of USB connection that defines the throughput and the input size of the data that is being sent to it. Nevertheless, here, the decision was to assume that the communication time for each of the two USB busses is fixed to the minimum communication time that was deducted from the cost models. In other words, considering all the benchmarks that were made for each of the USB HAs (the Coral Edge TPU and the NCS) the minimum bias of all the cost models was considered as the communication cost and subtracted from all the linear cost models (it is shown in Subsection 6.1.2 that the cost models are modeled as linear regressions). Assuming that the evaluator checks the routings, and the fitness function adds up the communication costs to the value of the objective when routed communication to busses.

Hence, the fitness function takes into account these set assumptions. It then gives as output an estimation of the cost of mapping of the implementation whose fitness is tested.

In conclusion, the experimentation included two complementary parts. First, a benchmarking phase whose results are analyzed to build cost models. The latter is used to construct the fitness

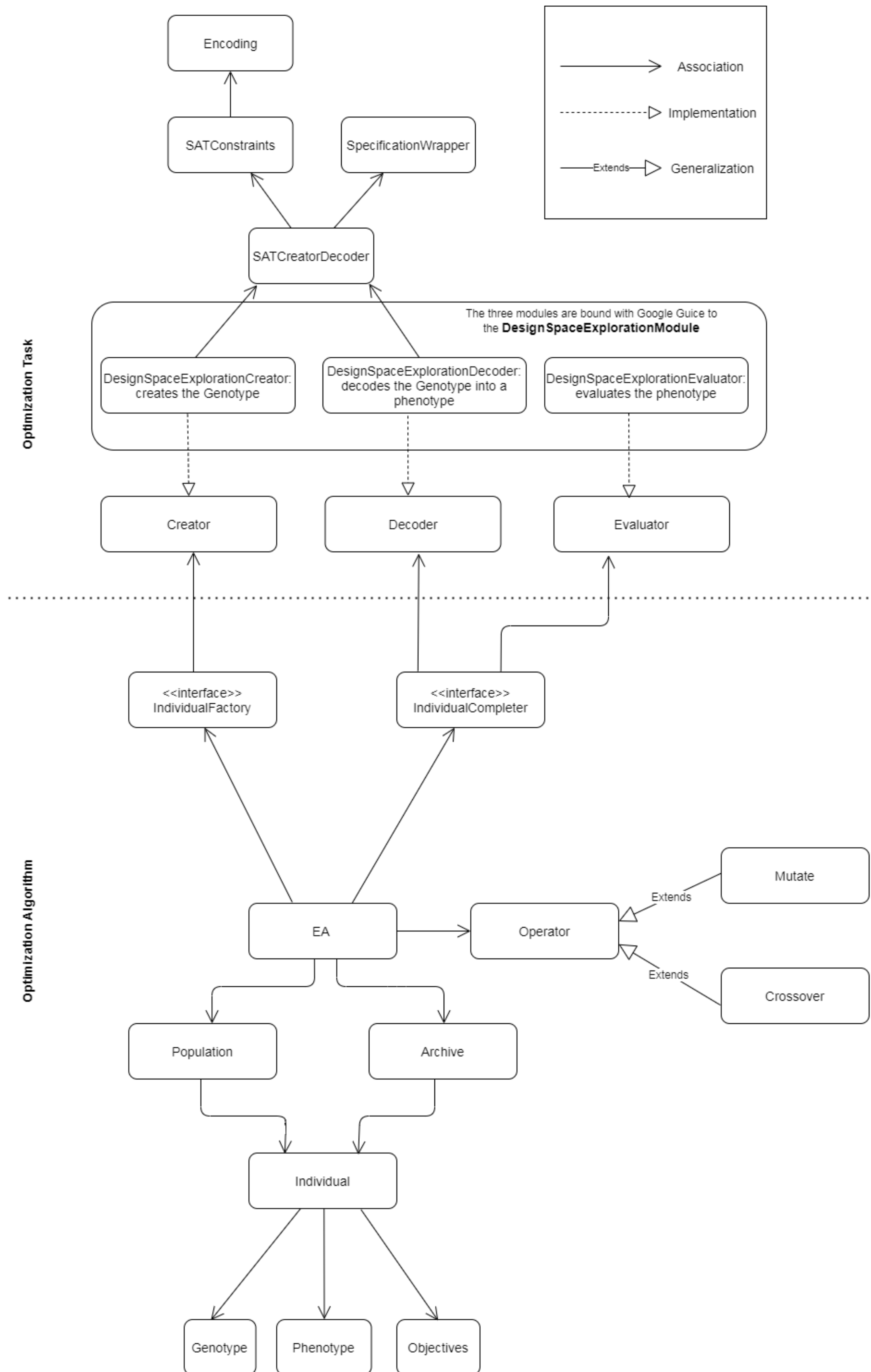

Figure 5.9: Overview of the simplified architecture of the used classes in this thesis DSE with OpenDSE, adapted from [7].

function that is used for the DSE during the second phase. The obtained results are presented in the next Chapter 6.

6 Results

In this Chapter, the results gathered from the performed tests are presented. First, the results of the benchmarking of TensorFlow operation models are exposed and analyzed to lead to cost models for each operation in Section 6.1. Then, the results of the implemented DSE tested on an example model are shown in Section 6.2.

6.1 Benchmarking Results

6.1.1 Results of Benchmarking Individual Operation Models

The results of benchmarking TensorFlow models encapsulating one operation with the benchmark tool following the setup described in Subsection 5.2.3 are exposed in this subsection.

As shown in Figure 6.1, for the displayed set of TensorFlow operations, the host computer achieved the shortest overall average inference time in total, followed by the NCS, and then followed by the Coral. It can be explained by the high computational power of the host computer used for the tests during this thesis with the described powerful GPU (refer to Subsection 5.1.1.1). It is also to be noted that the recorded average inference time on the hardware accelerators includes the additional communication costs of transferring the model from the host to them.

Looking at the overall average inference time, the changed parameters during the benchmarks do not reflect all the inference timing trends. It was obvious from the collected data that the measured inference time was related to the type of the encapsulated operation within the TensorFlow model, the input SHAVEs in the case of the NCS, and the input size of the model (and the input type in some cases). This was the main motivation to analyze the impact of those different configurations separately.

First, the type of operation has an impact on the inference time, as shown in Figure 6.1. The latter was expected because some operations' execution is more complex than others and because each TensorFlow operation has a different kernel implementation on the targeted hardware that strongly depends on the circuitry and the CPU frequency, for example, when it is a CPU hardware. It is known, for instance, that convolutions in deep neural networks are computationally dense and subject to interest when accelerating deep neural networks. It can be observed in the data in Figure 6.1 that the `conv2d` operation model (2D convolution in TensorFlow) has a significant inference time on the three hardware comparing to the addition operation.

Concerning the input SHAVEs of the NCS, assigning more SHAVEs for the computations decreases the inference time. An increase in the number of SHAVEs shows large improvements initially with rapidly diminishing returns when greatly increasing the used amount.

According to Figure 6.2, representing the overall average inference time (average on different configurations) for the different number of SHAVEs values, the inference time using one SHAVE of the NCS is around 2366.7 us which is 1.8 times larger than using 2 SHAVEs and 2.3 times

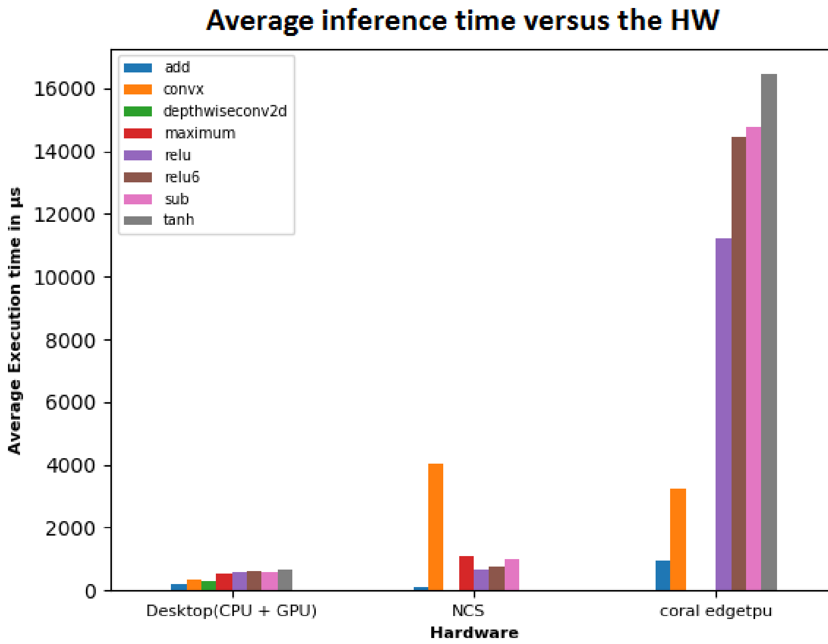

Figure 6.1: Average inference time in μs for a set of different types of operations for each available hardware with a number of runs=50, different input sizes, and different input types.

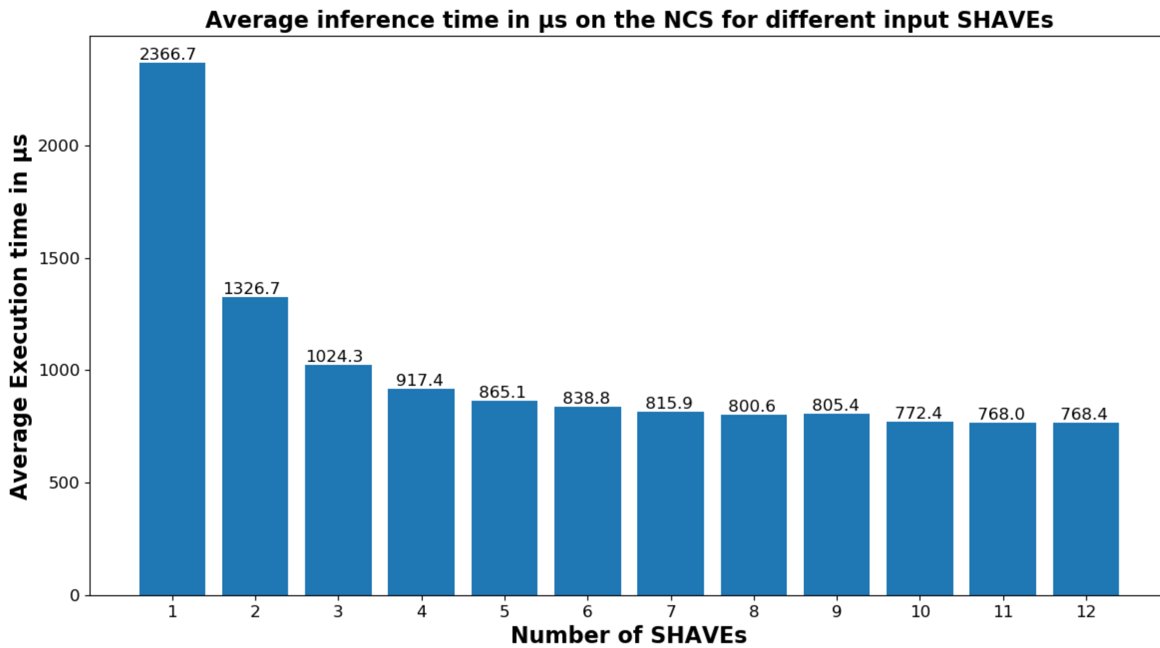

Figure 6.2: Average inference time on the NCS in μs versus the input SHAVEs of the NCS with 50 runs for each test, different operation types, different input sizes, and different input types.

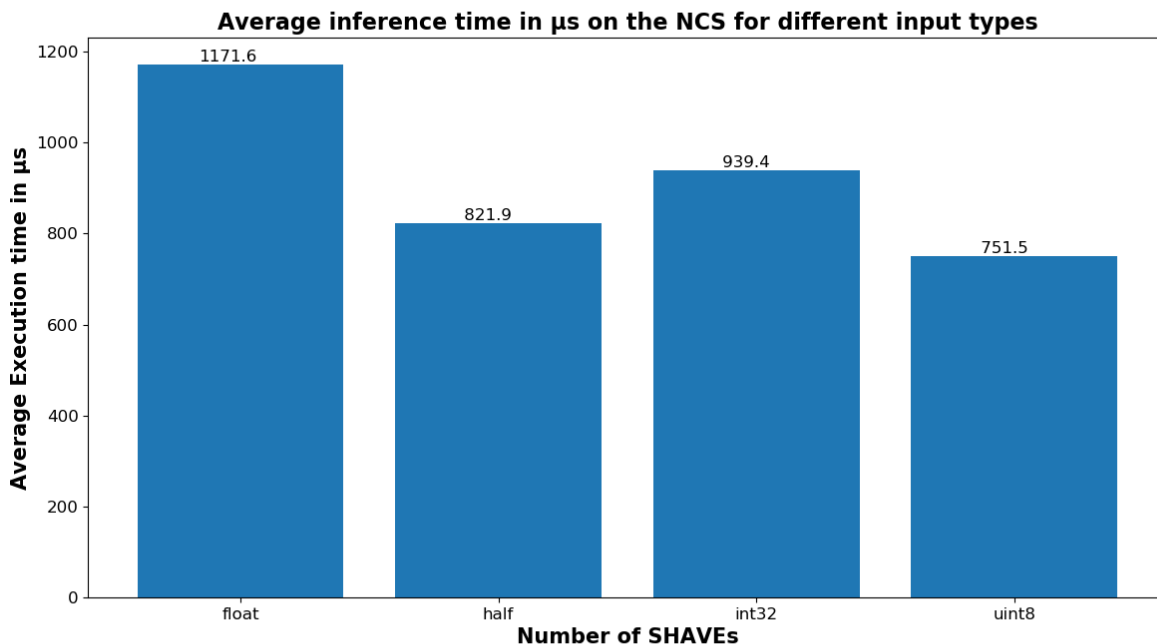

Figure 6.3: Average inference time on the NCS in μs versus the input types to the operation models run on the NCS with 50 runs for each test, different operation types, different input sizes, and different input SHAVEs.

larger than using 3 SHAVEs. Starting from 5 SHAVEs, the inference time average decrease is negligible compared to the previous decrease and is maximum around 30 μs (with an increase of 0.4 μs for 12 SHAVEs). This can be explained by the primary conception of the SHAVE core, which relies on deriving the maximum data-level parallelism. In other words, the decrease of the inference time is due to the high parallelism of execution thanks to the SHAVEs, and the decrease reaches its maximum when the parallelism cannot go further. Another explanation to the hardware reaching a stagnation in inference decrease is that the device runtime code controls the number of used SHAVEs, which is bounded by the number of SHAVEs that the user indicates. This feature allows the user to control the power consumption of the NCS, which can be beneficial when the USB stick is used on edge devices. It is also relevant to mention that since the communication time to transfer the data from the host to the accelerator through USB has not been separated from the timing measurements while doing the benchmarks, the observed stagnation can also be considered as a communication overhead from the host to the NCS.

Another important parameter that was changed during the benchmarks is the input type to the model. More precisely, when generating the models to benchmark on the different hardware, the input type was fixed for all the operations going from the input tensor to the model that is expected by the input placeholder node to the rest of the operations. It has to be defined when building TensorFlow models because it also gives information about the precision in which the computations are run, how the data's representation is expected (`int32`, `uint8`, `float32` and half-precision), and is expected to have an impact on the inference. During the tests, the input

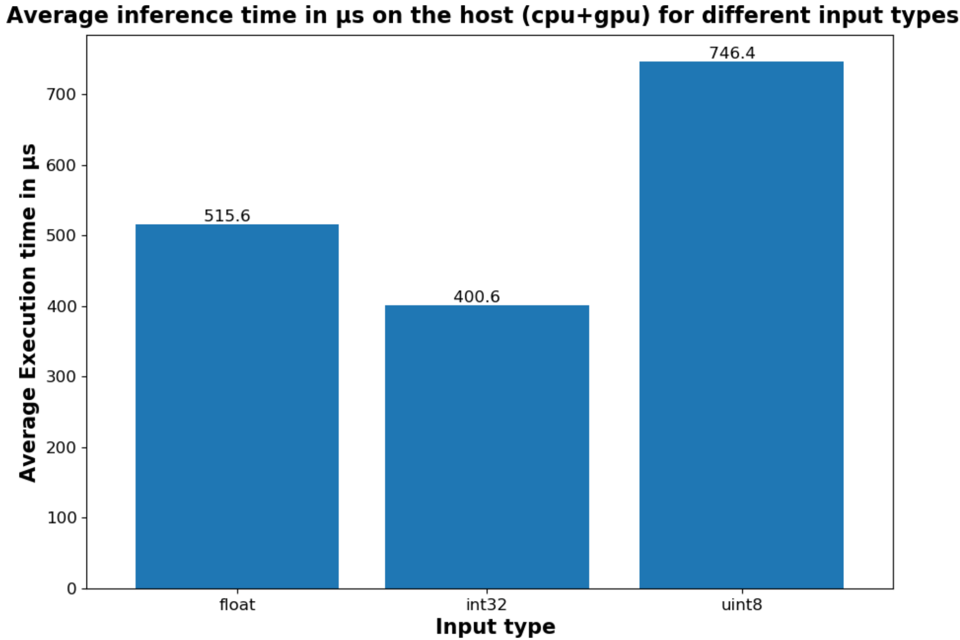

Figure 6.4: Average inference time on the host (CPU+GPU) in μs versus the input types to the operation models run on the host with 50 runs for each test, different operation types, and different input sizes.

type had to be fixed for the Coral because of the library limitations[1]. Hence, the models that were inferenced on the Coral Edge TPU had to be quantized to `uint8` from the beginning (as explained in Subsection 5.1.1.3). Unlike that, the NCS uses `fp16` precision, and as shown in Figure 6.3, there are significant differences due to changing the input type. The overall inference time when the considered type is `uint8` is the shortest, followed by the half-precision representation, followed by `int32`, and the longest one is scored when the input type is `float32`. Nevertheless, according to the datasheet of the NCS [38], all the operations are run in `fp16` (half- precision), which means that the observed difference in inference time comes from the time that the internal optimizer takes to convert the type to half[2]. It can also be due to the fact that less data precision to compute leads to less inference time. When representing the information with a precision of 8 bits, it comprises less information than the one with 32 bits. The latter can be seen in Figure 6.4, where the average inference time on the host (CPU and GPU) for a `float32` is higher than for an input type of an `int32` representation. The inference on the host for an input type of `uint8` is higher in average can be explained by the lack of accuracy when using only an 8-bit precision. Consequently, when building the cost models later in Section 6.1.2 input types were considered separately.

[1]The library that compiles the model is not an open-source library, and to compile the model on the Coral it is passed to a binary that either validates it or not.

[2]The user can check the loss in accuracy induced by the quantization with the `mvNCCheck`, however it was only shortly tested in this thesis. Indeed, the benchmarks are performed on operation models. The test of the accuracy is usually made for a complete neural network, for example, comparing the accuracy of inference of 1 image between two image recognition neural networks, which differ in quantization.

Finally, a look into the impact of the input size to the operation model on the overall inference time was taken. On the one hand, because larger input size is equivalent to a more significant number of elements on which the operations perform (This is especially true in the case of dense input tensors contrary to sparse tensors that were not particularly investigated in the context of this thesis). Moreover, the benchmark measurements include the timing cost of sending data from the CPU to the GPU in the case of the host benchmark, from the host to the Coral in the case of the Coral benchmark and the NCS in the case of the NCSDK2 benchmark. On the other hand, the current TensorFlow placement algorithm considers the input size as a parameter for its greedy heuristic [5] cost model for placing the operation. Hence, it was interesting to consider this parameter when trying to establish cost models. As can be seen in Figure 6.5, the average inference time increases with the input shape of the input, and hence of the input size. From here, it was interesting to have a closer look at the impact of changing the shape and thus the size of the input to the TensorFlow operation models while taking into account the previous influent parameters, namely the input SHAVES for the NCS, the input type, and the type of the concerned operation.

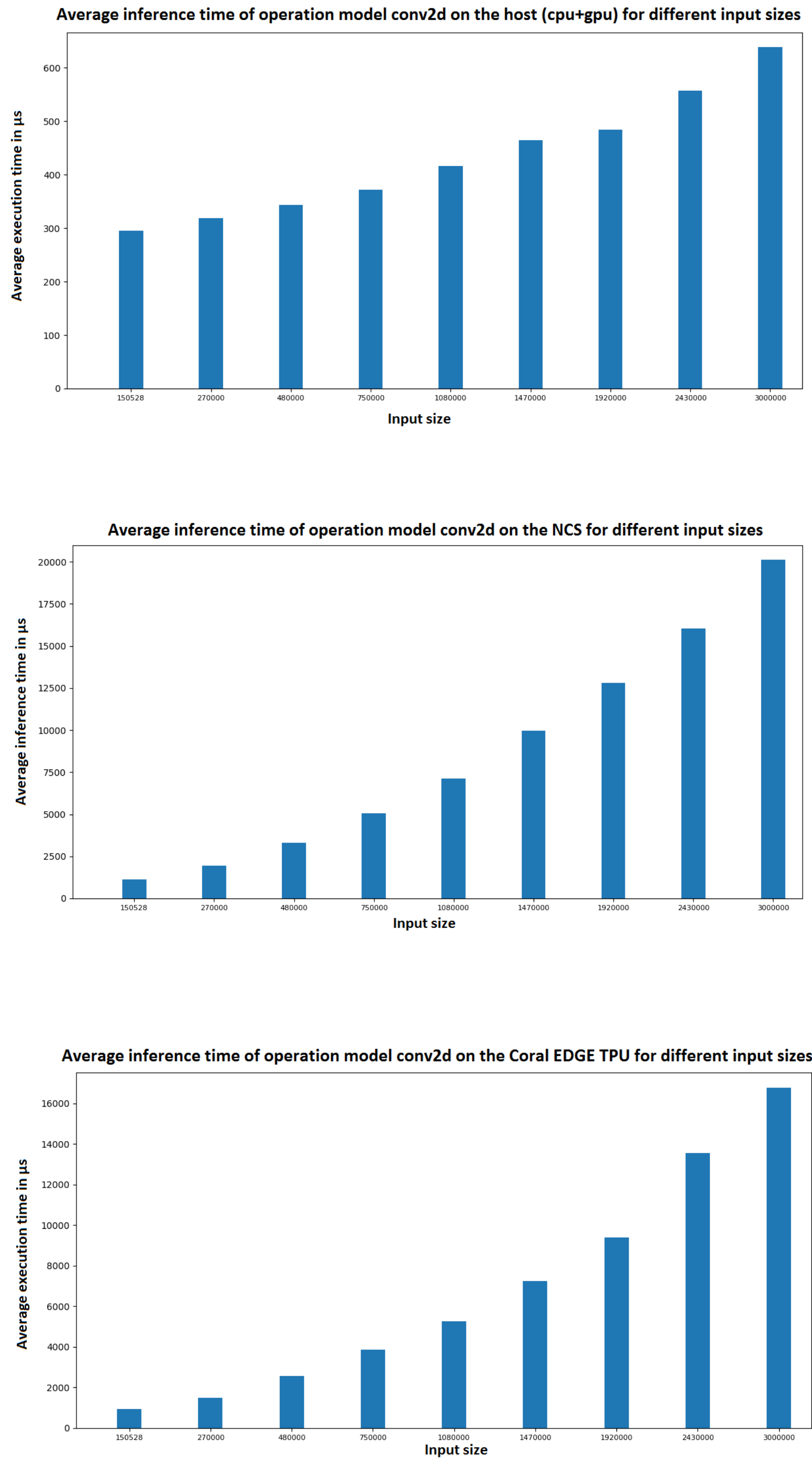

Figure 6.5: Bar graphs showing the average inference time in μs in function of the input size of the operation model Conv2d (2D convolution of TensorFlow) on the following hardware: CPU and GPU, NCS, and Coral Edge TPU.

6.1.2 Results Analysis and Establishment of cost models from the previous Results

The TensorFlow placement algorithm greedy heuristic assesses the execution time of the node according to some parameters like the type of the operation and the input shape [5]. Likewise, it was relevant to establish cost models that estimate the mapping cost of several TensorFlow operations on the available heterogeneous hardware, especially that TensorFlow does not support the used hardware accelerators. Hence, these cost models can be used later for the DSE to find the best HW/SW combination.

The first steps were the observations made while benchmarking individual operation models. Indeed, it was noticed that the execution time depends on several parameters, namely the input shape to the operation, the number of SHAVEs for the NCS, and the input type, as explained in the previous Subsection. For some operations, the results were possible to fit into regressions. For example, the plots in figures 6.6, the results obtained for mapping the operations `relu` (`tf.Operation` that is used in deep learning as an activation function) and `Conv2d` (TensorFlow `Operation` that computes a 2-D convolution) on the host using CPU and GPU, on the NCS and the Coral. Similarly, the cost models of the tested operations were assumed to be linear. In order to establish them, the "curve_fit" function from the package `Scipy` in Python was employed. This function uses non-linear least squares to fit a regression to the benchmark data. In other words, it takes as input the data collected from the benchmark for fixed parameters and finds the linear regression that fits the data best. Therefore, for every tested operation, it was establishing the execution time as a linear regression of the input size for fixed input type and number of SHAVEs for the NCS.

The results of the conducted curve fittings are that for each type of operation in the considered set of operations, for each available hardware that the operation could be mapped to, and for different number of SHAVEs for the NCS, the execution time is estimated according to a linear regression that depends on the input size (The size of the tensor is the number of elements of the tensor). The parameters of the linear regression are saved to a `CSV` file, which is used later in the optimization to fetch the cost models of each benchmarked operation.
The regression cost models to predict the trend of the inference time are built from benchmark test data that come with averaging over multiple runs and randomly generated inputs. Therefore the models are expected to have flaws, and establishing them is based on a practical approach that is sufficient but not optimal. Hence, building these cost models is based on a heuristic approach. The latter was sufficient in this thesis for considering these cost models to establish the fitness function in the optimization. Nevertheless, to verify that these cost models could be trusted later for the optimization, the correlation (Pearson correlation coefficient) between the curve fitting (the model) and the data (the results) was computed in each separate case. As shown in tables 6.1 and 6.2 in the example set of operations, the correlation value varies between 0.9 and 1. Thus, the established cost models are not optimal, but reliable[3].

In conclusion, using the benchmark tool tests was made on TensorFlow single operation models. The results were analyzed and transformed into cost models. These cost models are used next in the DSE to establish the fitness function.

[3] The correlation between the benchmark data and the regression established by the curve fitting was between 0.9 and 1 for all the cost models. The presented tables are shown as an example to support the reliability of the cost models.

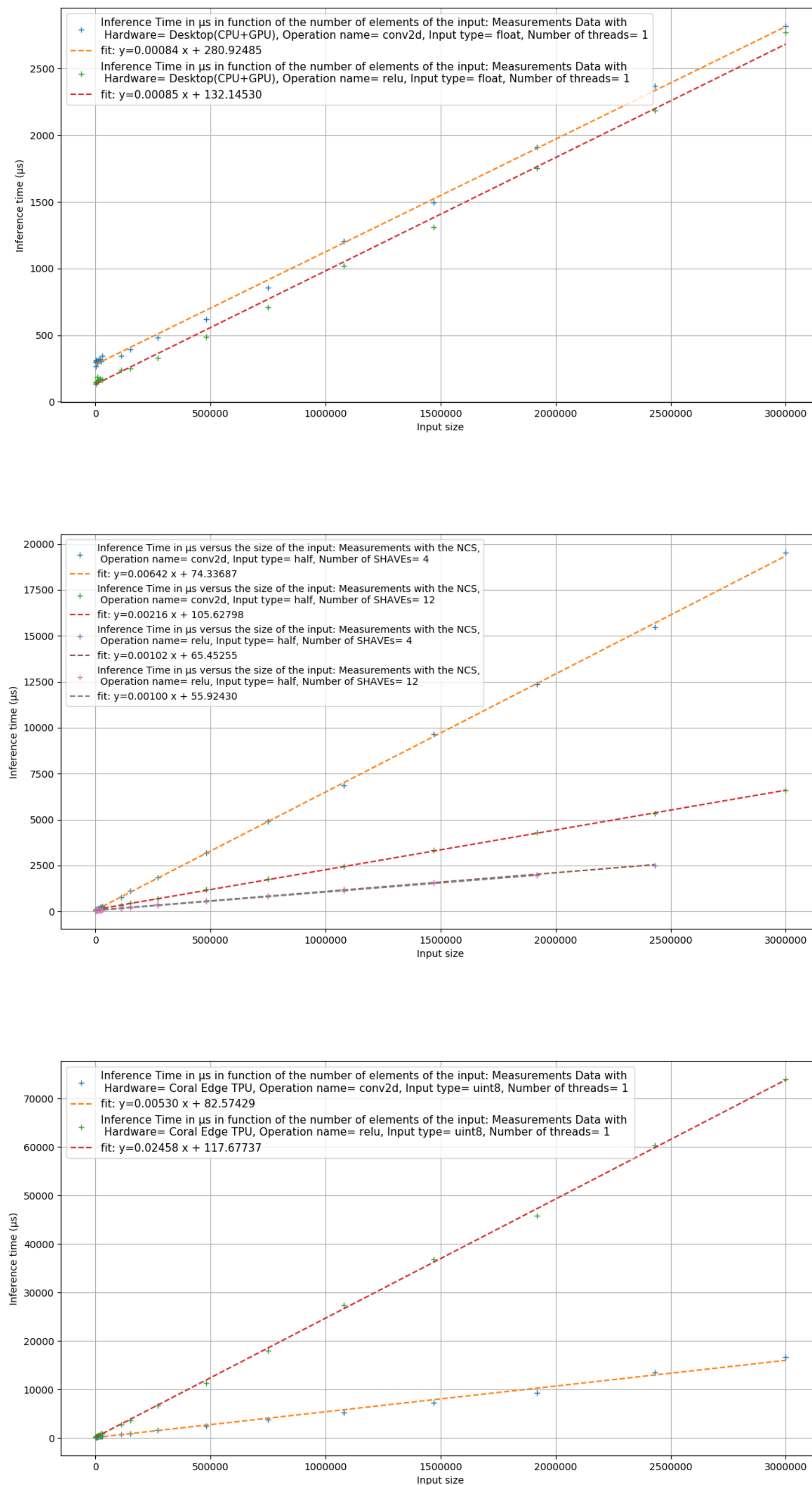

Figure 6.6: Average inference time in μs in the function of the input size of the operation model Conv2d (2D convolution of TensorFlow) and the operation model relu (the rectified linear unit from TensorFlow) on the following hardware: CPU and GPU, NCS (number of SHAVEs=4 and 12), and Coral Edge TPU.

Table 6.1: Correlation between measurement data and regression cost models for a set of operations on the Host compute(CPU and GPU) and on the Coral Edge TPU.

Operation Name	Type	Hardware	Correlation coefficient
add	float	CPU + GPU	0,98
conv2d	float	CPU + GPU	1,00
depthwiseconv2d	float	CPU + GPU	1,00
maximum	float	CPU + GPU	1,00
maximum	int32	CPU + GPU	0,99
relu	float	CPU + GPU	1,00
relu6	float	CPU + GPU	1,00
sub	int32	CPU + GPU	0,97
tanh	float	CPU + GPU	1,00
add	uint8	Coral Edge TPU	1,00
conv2d	uint8	Coral Edge TPU	1,00
relu	uint8	Coral Edge TPU	1,00

Table 6.2: Correlation between measurement data and regression cost models for a set of operations running on the NCS with a fixed number of SHAVEs equal to 6.

Operation name	Number of SHAVEs	Type	Hardware	Correlation Coefficient
add	6	float	NCS	0,99
conv2d	6	float	NCS	1,00
maximum	6	float	NCS	1,00
maximum	6	half	NCS	1,00
maximum	6	int32	NCS	1,00
relu	6	float	NCS	1,00
relu	6	half	NCS	1,00
relu	6	int32	NCS	1,00
relu6	6	float	NCS	1,00
relu6	6	half	NCS	1,00
relu6	6	int32	NCS	1,00
sub	6	float	NCS	1,00

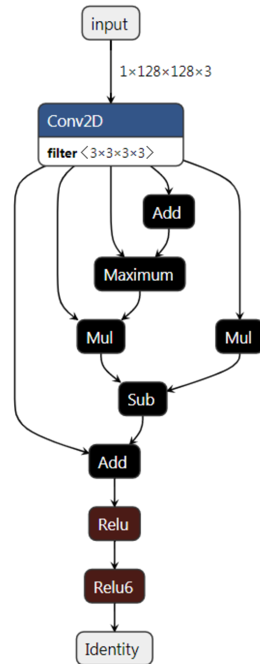

Figure 6.7: Visualization of the generated example model from the Protocol Buffer file with `Netron` [10].

6.2 Results of the Design Space Exploration

6.2.1 Application of Design Space Exploration Methods to a Small TensorFlow Model

6.2.1.1 Presentation of the Context

The optimization model is tested for a relatively small TensorFlow model constituted from 9 TensorFlow operations (`tf.Operation`), an input Placeholder, and an output Identity node 6.7. The first step was to pass the model to the python script that was written to dump the details of the model into a `CSV` file representing a summary of the created graph, which resulted in a set of 13 operations (2 nodes are added as a constant that is the input filter to the conv2d and reading its value). Therefore, it is a design space of at most 3^{13} possible implementations[4] if the fact that some operations are not always feasible on the three hardware and that there are additional parameters like the input SHAVEs (more details about the number of SHAVEs of the NCS were presented in Subsection 5.1.1) of the NCS that widens the design space is not considered. The goal is to find the best implementations with the shortest execution time and the adequate configuration of hardware resources (In this case, the number of SHAVEs for the NCS as a parameter).

[4]The maximum number of mapping possibilities is bounded by: $card(R)^{card(P)}$, where R is the set of the hardware resources (processing resources like CPU, TPU...) of the architecture, and P are the nodes of the architecture.

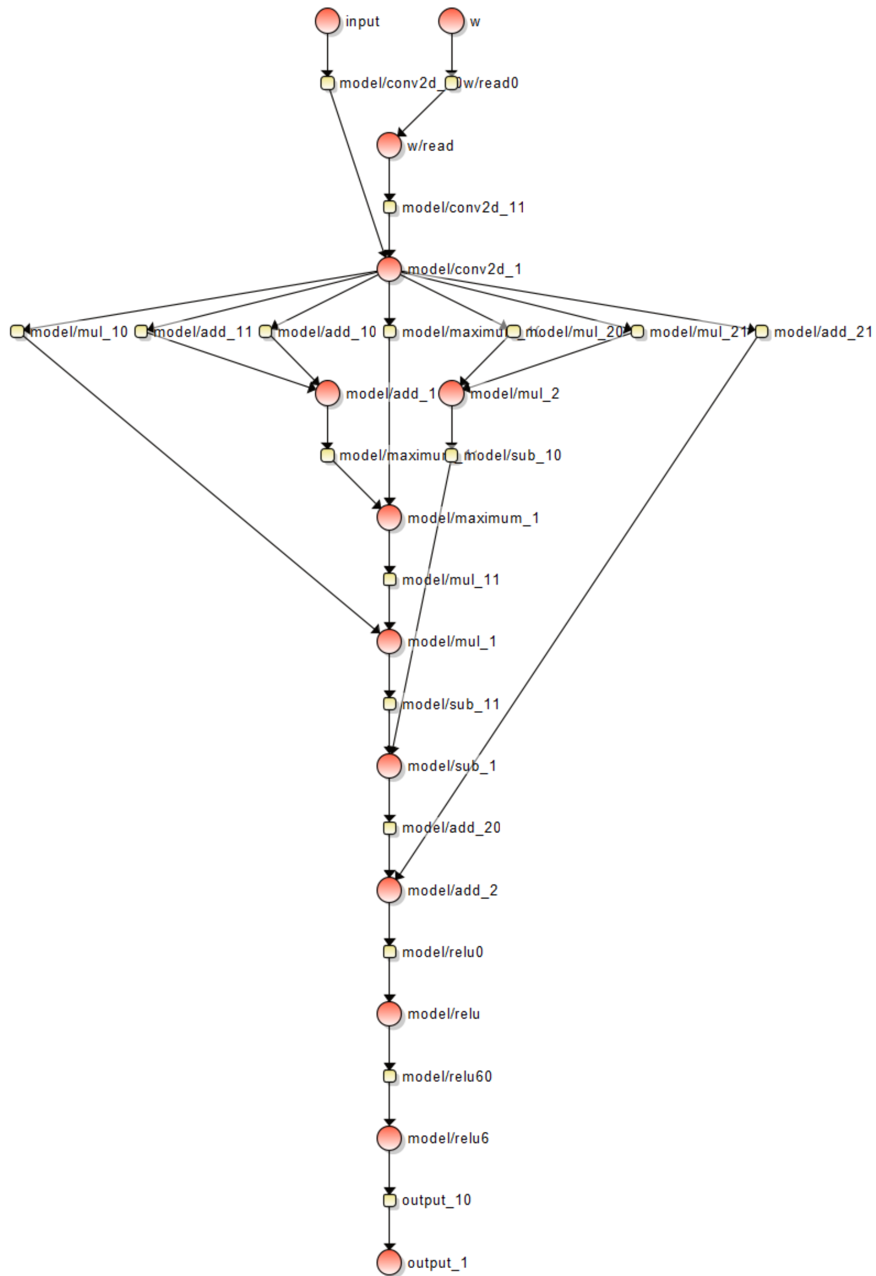

Figure 6.8: Application graph representing the TensorFlow model example (visualized with the OpenDSE Viewer).

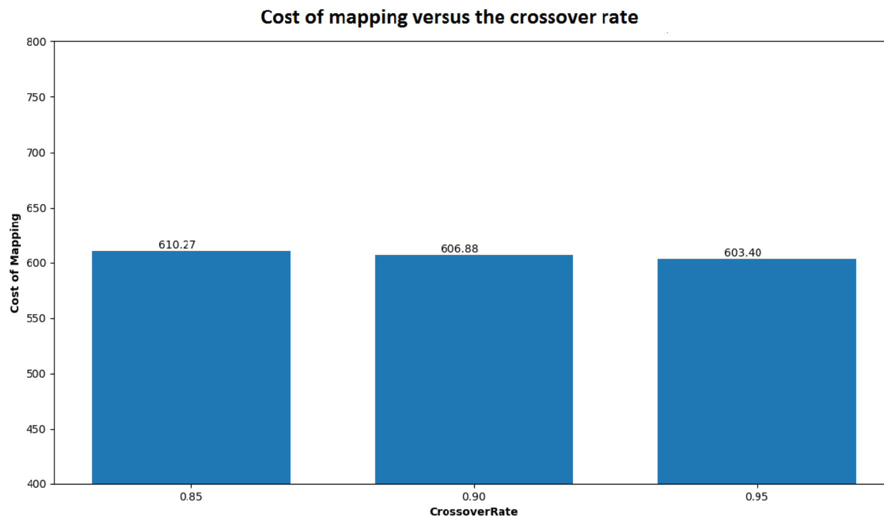

Figure 6.9: Bar graph representing the cost of mapping (`cost_of_mapping`) versus the crossover rate (`CrossoverRate`) over 10000 seperate runs of the DSE with a fixed number of generations equal to 100, a fixed size of population equal to 50, a fixed number of parents equal to 25, and a fixed number of offsprings per generation equal to 25.

The generated application graph can be seen with the OpenDSE viewer in Figure 6.8. A script runs the optimization for 10000 runs with varying parameters of the evolutionary algorithm with the aim of tuning them and converging towards the optimal implementation.

6.2.1.2 Presentation of the Results

For this example, a sensitivity analysis had to be conducted to find optimal EA parameters. These parameters consist of the number of generations, the size of the population α, the number of parents μ, and the number of offsprings per generation λ. First, all of them were fixed, and the crossover rate of the `Crossover` operator was varied. As can be seen in Figure 6.9, the cost of mapping is inversely proportional to the crossover rate. The value of 0.95 crossover rate permits the optimization to output the minimum, and thus the best average cost of mapping. That is why the crossover rate was fixed to 0.95. Similarly, tests were made by fixing all the parameters except α, and as can be noticed in Figure 6.10, the size of the population also have an impact on the result of the optimization. With this tested combination, the results show that choosing a smaller initial size of the population makes the optimization converge to a better solution. Concerning the number of parents μ and the number of offsprings per generation λ, the tests showed that after fixing the previous parameters, the results are the same when setting these two parameters to 25 or 50.

After setting the crossover rate, the size of the population, the number of parents, and the number of offsprings per generation respectively to 0.95, 50, 25, and 25, the number of generations was considered. It has a substantial impact on convergence. Indeed, it was noticed while

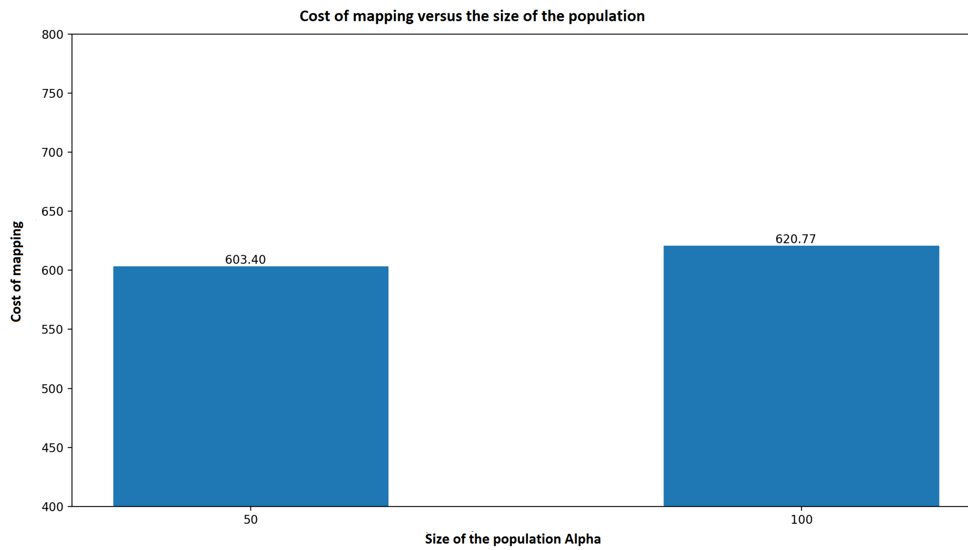

Figure 6.10: Bar graph representing the Cost of Mapping versus population size over 10000 seperate runs of the DSE with a fixed number of generations equal to 100, a fixed crossover rate equal to 0.95, a fixed number of parents equal to 25, and a fixed number of offsprings per generation equal to 25.

doing the tests that when setting the number of generations to 100, as shown in Table 7.1, the algorithm converges to a highly optimal implementation with a frequency of around 0.67. In other words, it does not always converge to the global minimum of the cost of mapping and is sometimes trapped in local minima, as shown in the results Table 7.1. A stable fixed solution is reached when the number of generations is set to 400 or more.

A deeper look into the obtained results is investigated in the next Chapter 7.

7 Evaluation of the Design Space Exploration Outcome for Distributed ML Inference

In this Chapter, the results previously presented are evaluated in Section 7.1. The reliability of these results and further conclusions are discussed in 7.2.

7.1 Evaluation of the Results for Design Space Exploration Application on a Small TensorFlow Model

As presented in Section 6.2, a sensitivity analysis was conducted in order to find the adequate evolutionary algorithm parameters to adopt for running the exploration. It was particularly noticed that the convergence of the EA depends on the chosen number of generations. As stated in the results, when the number of generations was set to 100, the output of the exploration algorithm is different over the runs. It converges to the known optimal solution with a frequency of 0.67 over 10000 runs of the optimization. It means that the rest of the solutions to which it converges are also optimal but not the best. If a look is taken at the most probable implementations given by the algorithm in the case of running it with the number of generations equal to 100, which are the following:

- The three following tuples represent three implementations generated with the optimization. Their mapping and architecture can be seen respectively in Subfigure 7.2 and 7.1, and the only difference between them is the setting of the number of SHAVEs of the NCS:
  - (Cost of Mapping: 582.2, Frequency: 0.675) with 12 SHAVEs.
  - (Cost of Mapping: 634.1, Frequency: 0.135) with 6 SHAVEs.
  - (Cost of Mapping: 643.0, Frequency: 0.054) with 9 SHAVEs.
- The following solution with (Cost of Mapping: 676.7, Frequency: 0.054), and which mapping can be seen in Subfigure 7.3 has a different mapping, and the number of SHAVEs of the NCS is set to 12.

The previously reached solutions by the optimization on different runs represent highly optimal feasible solutions, and the user can decide which implementation to select according to his expectations. The user could choose the solution with fewer SHAVES with a satisfying cost of mapping. For example, the solution, which shows 634.1 as the cost of mapping, could be sufficient to the user in terms of execution time, and he would then need to set the maximum number of SHAVEs of the NCS to only six, gain in terms of computation power for the NCS. In other words, the decision depends on the objectives of the design to optimize (refer to next Section 7.2).

Only the execution time cost was considered as an objective named the cost of mapping in the thesis. That is why more tests were conducted to be able to have an excellent convergence

Table 7.1: Table representing the frequencies of the resulting cost of mapping over 10000 separate runs with the following settings: number of generations=100, size of population=50, number of parents=25, number of offsprings per generation=25, and crossover rate=0.95.

Cost of Mapping	Frequency
582.2	0.675
618.8	0.017
634.1	0.135
643.0	0.054
652.3	0.023
653.7	0.013
661.5	0.001
662.0	0.002
663.4	0.006
664.9	0.005
667.9	0.004
668.5	0.002
670.7	0.002
674.2	0.002
675.5	0.001
676.7	0.054
713.2	0.001
721.3	0.002
730.5	0.001

to the optimal solution considering this objective, which is, in this example, the implementation with 582.2 Cost of mapping (7.2). The observations have shown that the number of generations for the chosen previous configuration of the EA parameters makes the optimization more likely to converge to the expected solution.

However, the problem of increasing the number of generations of the EA increases the time of the EA run. That is why an adequate number of generations would be sufficient. For instance, for a fixed number of generations equal to 400, the frequency of obtaining the previous solution over 10000 runs reached 0.97.

To conclude, despite the effort of tuning the EA parameters and the modelization of the Optimization problem and running it, the resulting optimal optimization is obtained with a high probability over a large number of runs. This implementation is presented as optimal for the following TensorFlow model and would accelerate its inference if it was possible to implement it. In the next subsection, the reliability of the final results obtained by the optimization is discussed.

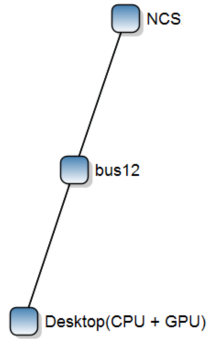

Figure 7.1: Corresponding Architecture.

Mapping	Task	Resource
inputDesktop(CPU + GPU)	input	Desktop(CPU + GPU)
model/add_1NCS	model/add_1	NCS
model/add_2NCS	model/add_2	NCS
model/conv2d_1NCS	model/conv2d_1	NCS
model/maximum_1NCS	model/maximum_1	NCS
model/mul_1Desktop(CPU ...	model/mul_1	Desktop(CPU + GPU)
model/mul_2Desktop(CPU ...	model/mul_2	Desktop(CPU + GPU)
model/reluNCS	model/relu	NCS
model/relu6NCS	model/relu6	NCS
model/sub_1Desktop(CPU ...	model/sub_1	Desktop(CPU + GPU)
output_1Desktop(CPU + GP...	output_1	Desktop(CPU + GPU)
w/readDesktop(CPU + GPU)	w/read	Desktop(CPU + GPU)
wDesktop(CPU + GPU)	w	Desktop(CPU + GPU)

Figure 7.2: Mapping for 3 different implementations generated by the optimization which corresponding (cost of mapping, number of SHAVEs) tuples are respectively: (582.2, 12), (634.1, 6), and (643.0,9).

Mapping	Task	Resource
inputDesktop(CPU + GPU)	input	Desktop(CPU + GPU)
model/add_1NCS	model/add_1	NCS
model/add_2NCS	model/add_2	NCS
model/conv2d_1NCS	model/conv2d_1	NCS
model/maximum_1NCS	model/maximum_1	NCS
model/mul_1Desktop(CPU ...	model/mul_1	Desktop(CPU + GPU)
model/mul_2Desktop(CPU ...	model/mul_2	Desktop(CPU + GPU)
model/reluNCS	model/relu	NCS
model/relu6NCS	model/relu6	NCS
model/sub_1NCS	model/sub_1	NCS
output_1Desktop(CPU + GP...	output_1	Desktop(CPU + GPU)
w/readDesktop(CPU + GPU)	w/read	Desktop(CPU + GPU)
wDesktop(CPU + GPU)	w	Desktop(CPU + GPU)

Figure 7.3: Mapping for cost of mapping= 676.7.

Figure 7.4: Optimal Implementations obtained by the DSE with OpenDSE summary for the TensorFlow model example.

7.2 Conclusions of the Evaluation and Discussion

The experimental setup aimed to present a proof of concept of the chosen strategy during this thesis. The DSE was conducted on the example TensorFlow model, which has 9 TensorFlow operations. The output of the optimization consists of implementations with a highly optimal cost of mapping. Several points around this outcome need to be discussed.

The goal of the optimization is to provide implementations with the lowest execution time, which corresponds in the experimental evaluation to what was called the cost of mapping. The value of this objective for each implementation was computed as the total sum of the cost of mapping for each operation. The latter was defined with cost models that rely on benchmarking data from tests performed on the available hardware. Hence, the evaluation of the implementation was directly dependent on estimations made from a practical method, and thus it was relying on a heuristic method. In other terms, the approach used for establishing the cost models is sufficient but not optimal. For example, these estimations depend upon the used hardware, and when using different hardware, which is not supported by the benchmark tool, the user would have to redo the benchmarks, maybe from scratch. Moreover, these cost models were computed with the function `curve_fit` from `scipy`. It is possible that better and more accurate cost models could fit the data, but that was not further investigated during the thesis.

The final output of the DSE in the tested example consists of highly optimal implementations according to the cost of mapping as an objective. The goal was to search for the implementation that permits the best distribution of inference to accelerate it, and thus, the focus was turned to minimizing the execution time. However, the optimization could be extended to a multi-objective optimization when other objectives are targeted in the design, for example, the power or the memory. This can be extended in the thesis by either adding parameters and variables when defining the design space or investigating the existing degrees of freedom. A good example is what was noticed previously concerning the maximum number of SHAVEs of the NCS. Indeed, minimizing the number of SHAVEs could benefit the implementation, for example, in reducing power consumption. Hence, this parameter could constitute a second objective to minimize in addition to the execution time.

In the particular case of the conducted tests, the solutions generated by the optimization include different NCS configurations. The choice of the optimal solution considering the two objectives is a trade-off that the designer needs to make. A look at the Pareto front in Figure 7.5 delimiting the solutions where the architecture includes the NCS can help to make this decision. This Pareto front represents the set of optimal outcomes of the exploration that the designer can choose from.

Another point to mention is that the DSE outcome gives the user only an idea of the optimal implementations. However, considering that the NCS and the Coral are not supported by TensorFlow. In order to apply the results of the DSE, the user has two options. Either implement the NCS and Coral in the source code of TensorFlow, rendering them visible by the placement algorithm that would treat them like the other hardware. For example, in the latest releases of TensorFlow, they presented the possibility of adding hardware as delegates. The second option would be to take care of dividing the application graph representing the machine learning model into subgraphs. Each subgraph would be mapped to dedicated hardware. The user would also

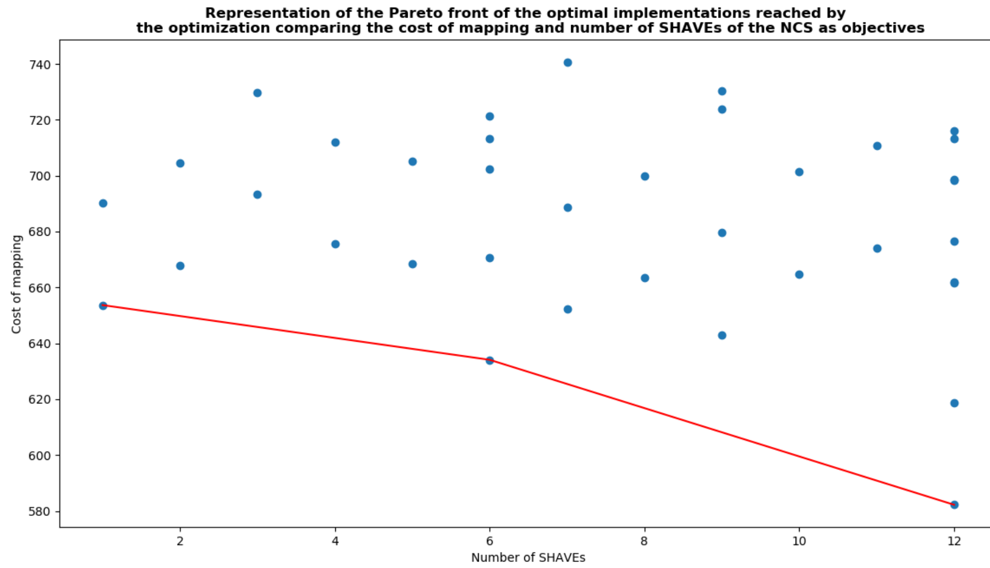

Figure 7.5: Representation of the Pareto front of the implementations generated by the optimization that includes the NCS in the architecture according to the cost of mapping and the number of SHAVEs of the NCS as objectives.

take care of assembling the result chunks that leads to the inference output (more details in the suggestions of future work in Subsection 9.1.3).

8 Conclusion

This thesis aimed to present a method to accelerate the inference of machine learning models by applying state-of-the-art DSE techniques. According to the Y-chart methodology, the problem was turned to an optimization one by representing it as a system-level design problem.

This optimization in the experimental evaluation is presented as a proof of concept of accelerating the inference of machine learning models. When a machine learning model is represented as a dataflow graph like in the TensorFlow framework, it could be modeled as an application in the OpenDSE framework, which is generic enough to fit different machine learning models. Next, the available hardware needs to be defined in the architecture. The various constraints related to the mappings and the objectives of the optimization are also specified. These objectives depend on the decision of what needs to be optimized in the system. Consequently, a fitness function that computes each implementation's quality according to the defined objectives is laid out. In this thesis, the fitness function relied on the cost models built from the benchmarking results. Several assumptions needed to be made in order to complement the establishment of the fitness function. These assumptions include: the TensorFlow operations that were not tested were considered to have no cost of mapping, and the cost of communication was assumed to be fixed.

The chosen design space exploration algorithm in this thesis was a hybrid metaheuristic algorithm relying on an evolutionary algorithm and an SAT-Decoder. These Algorithms are already implemented in the used OpenDSE framework. Nevertheless, the operators of the algorithms needed to be set, and the most common ones, the mutation, and crossover operators, were selected. It is relevant to mention that the EA algorithm also depends on parameters that control its convergence towards the optimal solutions. Indeed, in some cases, the EA tries to find the optimal solution but can get trapped in local minima, like in the presented example in Chapter 7. Therefore, the following parameters must be fine-tuned: the number of generations, the size of the population α, the number of parents μ, and the number of offsprings per generation λ.

The resulting implementations from the DSE present highly optimal bindings of the operations to allocated resources, routings, and allocation of resources (Scheduling is out of the scope of this thesis). These findings could be further followed to accelerate the inference. In the case of the experimental evaluation in this thesis, since the targeted hardware was not fully integrated into TensorFlow, the optimization gives an idea of what could be the best distribution of the considered TensorFlow machine learning model.

Several suggestions are presented in the next Chapter to extend this thesis.

9 Future Work

In this Chapter, several suggestions for future research on this topic are given to extend the thesis. First, concerning the experimentation and then concerning the overall followed strategy.

9.1 Future Work in the Experimentation

9.1.1 Applying the Design Space Exploration to a larger TensorFlow Machine Learning Model- Mobilenet as an example

9.1.1.1 Context

The DSE implementation was tested on an example TensorFlow model, which comprises only 13 operations. The latter constitutes a simple example. The idea of applying it to a larger TensorFlow model was thought of during the thesis, but not pursued for two reasons. First, the benchmarking tests were conducted on a limited set of operations that might not include all the operations from which a large machine learning model would be made. Second, the cost of computation of such tests in terms of running time. That is why to have a general idea of what would the DSE outputs just with the current configurations, some tests were conducted on the mobilenet TensorFlow model for a limited number of runs (limited to 69 runs). The results of these tests are presented in 9.1.1.2.

It would then be interesting to extend these tests to larger TensorFlow models and make a more significant number of runs of the DSE.

9.1.1.2 Primary Test Results Obtained with a Mobilenet TensorFlow MModel

As previously mentioned, the DSE implementation was applied to the mobilenet TensorFlow model for a limited number of runs (69 runs with the number of generations of the EA set to 100). The results on this limited number of runs show that the generated optimal implementation with a cost of mapping equal to around 16291,32 is reached only 29 times against the 69 runs. As can be seen in Figure 9.1, the optimization generated implementations with a cost of mapping close to the minimum with a percentage of 49 percent. The rest of the time, as can be seen with the Pareto line, the other generated solutions are trapped near other minima that could be satisfying but not highly optimal considering the cost of mapping and thus the execution time as an objective.

These tests on the mobilenet could be extended with more runs for the optimization.

9.1.2 Experimental Setup in a more Constrained Environment

The hardware that was used for the experimental setup relies on a powerful host computer with a powerful CPU and GPU and two AI hardware accelerators: the Coral and the NCS. It would be interesting to make the testing environment more constrained, for example, by replacing the host computer with a Raspberry Pi. It would only then be necessary to install Docker on it and resolve the different dependencies to run the same benchmarking tests and build the cost models

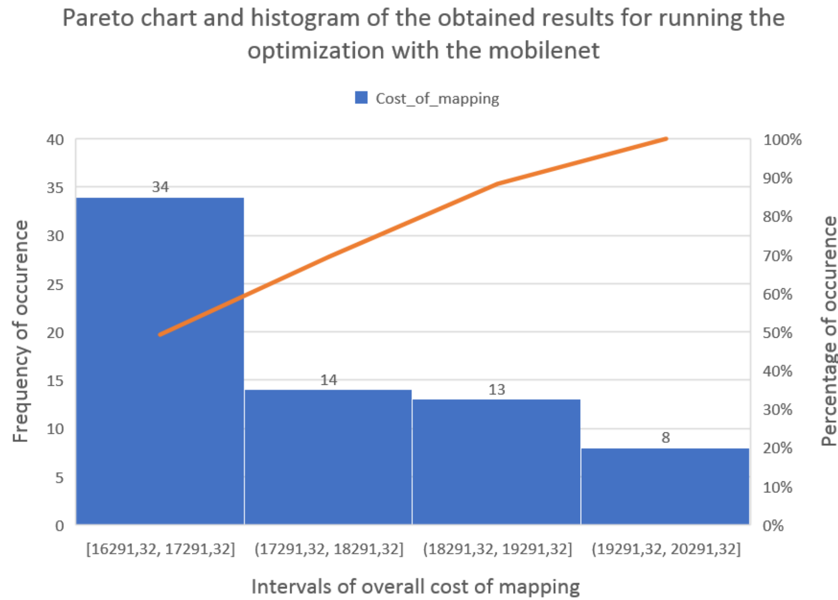

Figure 9.1: Pareto chart representing the results of running the optimization on the mobilenet model over 69 runs.

for the Raspberry Pi. Then, the architecture in the OpenDSE is changed by adding the Pi, and the same approach would be valid for conducting the DSE.

In this case, it can be expected that more operations will be mapped to the hardware accelerators because they are expected to be way more powerful than the Pi resources compared with the CPU of the host computer used for the tests.

9.1.3 Implementing the Hardware not supported by TensorFlow

The outcome of the DSE resides in highly the optimal implementations (or implementation) for distributing a TensorFlow machine learning model through the available hardware. It would be interesting in the future to test these resulted implementations and measure their cost of mapping to the one given by the DSE. The problem is that the TensorFlow framework does not support the hardware accelerators that were used. The solutions around this problem, which would be worth trying in future studies are either:

- Dividing the machine learning model into fragments encapsulated into small TensorFlow models. The latter would be assigned to hardware by the user, and the different transformations that are required by the AI HA (for example, the Coral Edge TPU compiler and the NCS compiler) need to be added by the user. Finally, the results of the different machine learning encapsulated subgraphs can be assembled to give the inference's final result. This solution would be only for verifying the DSE result and is not practical for future inference use. It also requires handling dividing the graph into subgraphs and taking care of the communication and the compilers of the AI HA, and this was out of the scope of this thesis.

- Integrating the hardware into the TensorFlow source code and its placement algorithm. Similar work to what was done to include the CPU or GPU by the Google team should be

done and followed. In other words, implementing the kernels of the supported operations of the hardware that is wanted to be added to the core (as well as cost models estimating their execution time). The placement algorithm will then take in responsibility the dividing of the machine learning graph into subgraphs and coordinate between the graph chunks to compute the final result. This solution would be adequate for future inference use.

9.2 Future Work in the Strategy

The DSE focus was to solve the system-level design problem. Nevertheless, the scheduling was excluded and left out of the scope of this thesis. It would be interesting to extend then the DSE with a real-time scheduling study. It could be implemented with the real-time scheduling module from OpenDSE that uses the Quadratic Programming approach to determine priorities for event-triggered systems such that all deadlines are satisfied.

Bibliography

[1] NVIDIA. What's the Difference Between Deep Learning Training and Inference? | The Official NVIDIA Blog. URL: https://blogs.nvidia.com/blog/2016/08/22/difference-deep-learning-training-inference-ai/.

[2] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[3] B. Kienhuis, E. F. Deprettere, P. van der Wolf, and K. Vissers. A methodology to design programmable embedded systems the y-chart approach. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2268:18–37, 2002. doi:10.1007/3-540-45874-3_2.

[4] B. Kienhuis and A. Jan. Design space exploration of stream-based dataflow architectures : methods and tools /. 01 1999.

[5] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. 2016. URL: http://arxiv.org/abs/1603.04467, arXiv:1603.04467.

[6] M. Glaß, J. Teich, M. Lukasiewycz, and F. Reimann. *Handbook of Hardware/Software Codesign*. 2017. doi:10.1007/978-94-017-7358-4.

[7] F. R. J. T. Martin Lukasiewycz, Michael Glaß. Opt4J – A Modular Framework for Metaheuristic Optimization. URL: http://opt4j.sourceforge.net/paper/opt4jpaper.pdf.

[8] Intel Corporation. NCSDK2. URL: https://movidius.github.io/ncsdk/index.html.

[9] Google. Coral edge TPU. URL: https://coral.ai/docs/edgetpu/models-intro/.

[10] L. Roeder. netron. URL: https://lutzroeder.github.io/netron/.

[11] F. Lukasiewycz, Martin Reimann, F. Smirnov, and F. Hoefte. OpenDSE. URL: https://mvnrepository.com/artifact/net.sf.opendse.

[12] Y. Lecun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015. doi:10.1038/nature14539.

[13] A. B. Foivos Tsimpourlas, Lazaros Papadopoulos and D. Soudris. A Design Space Exploration Framework for Convolutional Neural Networks Implemented on Edge Devices . 2018.

[14] S. C. N. K. J. W. Z. Ye Yu, Yingmin Li. Software-Defined Design Space Exploration for an Efficient DNN Accelerator Architecture. 2020. URL: https://arxiv.org/pdf/1903.07676.pdf.

[15] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 44(1-2):207–219, 2000. doi:10.1147/rd.441.0206.

[16] NVIDIA. NVIDIA Deep Learning Platform - Giant Leaps In Performance And Efficiency For Ai Services, From The Data Center To The Network's Edge. 2017. URL: https://developer.nvidia.com/deep-learning-software.

[17] D. E. C. Arvind. Dataflow architectures. pages 225–253, 1986.

[18] J. Park, M. Naumov, P. Basu, S. Deng, A. Kalaiah, D. Khudia, J. Law, P. Malani, A. Malevich, S. Nadathur, J. Pino, M. Schatz, A. Sidorov, V. Sivakumar, A. Tulloch, X. Wang, Y. Wu, H. Yuen, U. Diril, D. Dzhulgakov, K. Hazelwood, B. Jia, Y. Jia, L. Qiao, V. Rao, N. Rotem, S. Yoo, and M. Smelyanskiy. Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications. 2018. URL: http://arxiv.org/abs/1811.09886, arXiv:1811.09886.

[19] J. Schmidhuber. Deep Learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015. arXiv:1404.7828, doi:10.1016/j.neunet.2014.09.003.

[20] Google. Tensorflow Official Website. URL: https://www.tensorflow.org/.

[21] Google. Protocol Buffers website. URL: https://developers.google.com/protocol-buffers.

[22] E. Sackinger, B. E. Boser, J. Bromley, Y. LeCun, and L. D. Jackel. Application of the ANNA Neural Network Chip to High-Speed Character Recognition. *IEEE Transactions on Neural Networks*, 3(3):498–505, 1992. doi:10.1109/72.129422.

[23] M. Gschwind, V. Salapura, and O. Maischberger. Space Efficient Neural Net Implementation. (February), 1995.

[24] Y. Chen, T. Chen, X. Zhiwei, and O. Temam. DianNao Family: Energy-Efficient Hardware Accelerators for Machine Learning. *Communications of the ACM*, 57(5):109, 2014. URL: 10.1145/2594446{%}5Cnhttps://ejwl.idm.oclc.org/login?url=http://search.ebscohost.com/login.aspx?direct=true{&}db=bth{&}AN=95797996{&}site=ehost-live, doi:10.1145/2.

[25] Luqi and G. Jacoby. Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6662(March):228–238, 2011. URL: http://www.scopus.com/inward/record.url?eid=2-s2.0-79959990771{&}partnerID=tZOtx3y1, doi:10.1007/978-3-642-21292-5.

[26] A. D. Pimentel. Exploring Exploration: A Tutorial Introduction to Embedded Systems Design Space Exploration. *IEEE Design and Test*, 34(1):77–90, 2017. doi:10.1109/MDAT.2016.2626445.

[27] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf. Approach for quantitative analysis of application-specific dataflow architectures. *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors*, pages 338–349, 1997. doi:10.1109/asap.1997.606839.

[28] L. Ht, C. Hylands, E. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, Y. Zhao, and H. Zheng. Overview of the ptolemy project. 08 2003.

[29] B. G. Panerati J., Sciuto D. No Title. *Optimization Strategies in Design Space Exploration. In: Ha S., Teich J. (eds) Handbook of Hardware/Software Codesign. Springer, Dordrecht*, Optimizati, 2017.

[30] R. F. Glaß M., Teich J., Lukasiewycz M. No Title. *Hybrid Optimization Techniques for System-Level Design Space Exploration. In: Ha S., Teich J. (eds) Handbook of Hardware/-Software Codesign. Springer, Dordrecht*, Handbook o.

[31] T. Blickle, J. Teich, and L. Thiele. System-Level Synthesis Using Evolutionary Algorithms. *Design Automation for Embedded Systems*, 3(1):23–58, 1998. doi:10.1023/A:1008899229802.

[32] D. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. pages 439–455, 11 2013. doi:10.1145/2517349.2522738.

[33] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Journal of Symbolic Logic*, 32(1):118–118, 1967. doi:10.2307/2271269.

[34] Nvidia. Real interactive expression. page 4000, 2015. URL: http://www.nvidia.com/object/quadro.html.

[35] Coral. USB Accelerator datasheet | Coral. 0(September), 2019. URL: https://coral.withgoogle.com/docs/accelerator/datasheet/.

[36] Google. Tensorflow github. URL: https://github.com/tensorflow/tensorflow.

[37] docker. Docker Containers. URL: https://www.docker.com/resources/what-container.

[38] Intel Corporation. Intel Neural Compute Stick. page 2018, 2019. URL: https://software.intel.com/en-us/neural-compute-stick.