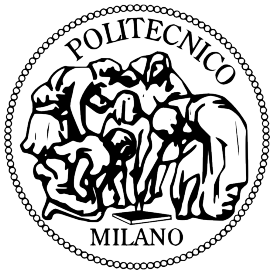
POLITECNICO DI MILANO
Corso di Laurea Magistrale in Ingegneria dell'Informazione
Corso di laurea in Ingegneria Informatica



A DESIGN FLOW BASED ON HALIDE AND BAMBU-HLS TO DEPLOY DEEP LEARNING MODELS ON FPGA

Relatore: Prof. Fabrizio Ferrandi
Correlatore: Serena Curzel

Tesi di Laurea di:
Ghitti Marco, matricola 893986

Anno Accademico 2019-2020

Abstract

Recent advances in the Artificial Intelligence field and the increasing computational power of HW accelerators have renewed the interest in Machine Learning models. In particular, the field is interested in Deep Learning models, powerful biologically inspired models that model how the model process sensory information to solve problems where defining an explicit algorithm is not a viable solution. Deep Learning models achieve outstanding results in applications where high dimensional data processing is needed and have the potential to solve yet unsolved problems such as medical diagnosis and autonomous driving.

Since Deep Learning models require intensive computational power researchers started using HW accelerated solutions to overcome the shortage of computational power and use more powerful models with more parameters. The first targets that have been used to accelerate Deep Learning applications are GPGPUs and many software frameworks are built to leverage the SIMD parallelism provided by feed-forward configurations.

Due to interest in sparse configurations and recent advances in High-Level Synthesis tools the research started considering FPGAs as HW accelerators. FPGAs are interesting solutions to deploy Deep models; their power efficiency and reconfigurability are interesting characteristics but the design process is slow, error-prone, and require knowledge about low-level hardware description languages. The use of High-Level Synthesis Tools eases the use of FPGAs while allowing the programmer to prototype different models before deciding which one best fits the target application.

This thesis proposes a design flow for the implementation of Deep Learning model on FPGAs; the design flow relies on the ONNX IR to provide a common entry point for the most common Deep Learning frameworks. The ONNX model is then passed to the Halide compilation infrastructure that decides how the computation is going to be performed on the target FPGA and produce a software description. The software description is then passed to the PandA-Bambu framework that translates the input C code into hardware descriptive code, ready to be deployed.

Abstract in italiano

Recenti miglioramenti nel campo dell'Intelligenza artificiale e l'aumento della capacità computazionale dei metodi di accelerazione HW hanno rinnovato l'interesse nell'utilizzo di modelli basati sull'apprendimento macchina (Machine Learning). In particolare l'intelligenza aritificiale è interessata alle "reti neurali profonde" (Deep Learning), modelli ispirati da come il cervello animale processa le informazioni sensoriali per risolvere problemi dove definire un algoritmo esplicito non è una soluzione praticabile. I modelli basati su reti neurali profonde ottengono risultati eccezionali in applicazioni che richiedono analisi di dati complessi e hanno il potenziale per risolvere problemi come la guida autonoma e l'analisi di immagini mediche.

Dato che le reti neurali profonde richiedono crescenti performance la ricerca ha deciso di utilizzare soluzioni HW per superare la carenza di potenza di calcolo delle CPU e poter'così utilizzare modelli più complessi e con più parametri. I primi acceleratori hardware che sono stati utilizzati per accelerare le reti neurali profonde sono le GPGPU e molti framework software sono stati costruiti con l'intenzione di utilizzare le potenzialità HW delle GPU.

A causa dell'interesse in reti neurali sparse e i recenti miglioramenti dei metodi di sintesi ad alto livello (HLS) la ricerca ha cominciato a considerare le FPGA come acceleratori HW. Le FPGA sono una soluzione interessante per eseguire i modelli Deep learning; l'efficienza energetica e la riconfigurabilità HW sono caratteristiche interessanti ma il processo di design è lento, è facile commettere errori e richiede conoscenza dell'hardware. L'utilizzo di metodi di sintesi ad alto livello permette a non esperti HW di utilizzare le FPGA permettendo inoltre di testare vari prototipi prima di decidere quale modello utilizzare.

Questa tesi propone un processo per implementare reti neurali profonde su FPGA; il design utilizza ONNX come punto di ingresso comune per diversi framework. Il modello ONNX viene poi passato ad Halide per produrre una descrizione software ottimizzata da utilizzare con Bambu-HLS. Il risultato finale è la descrizione hardware pronta per essere implementata su FPGA.

Ringraziamenti

Alle fine del mio percorso di studi voglior ringrazie la mia famiglia.

In particolare voglio ringrazie i miei genitori che mi hanno sempre sostenuto durante tutti i miei anni di studi.

Contents

Chapter 1

Introduction

Due to recent advances in AI and the creation of credible datasets the Deep Learning field has put an end to the AI Winter by achieving outstanding results in problems that were considered as an exclusive domain of human intelligence just a few years before. Artificial Neural Networks are powerful models that model what we know about how the brain process information coming from the outside world. Composed by an interconnected network of artificial neurons, ANNs model the firing mechanism of real neurons; the axon transmits accumulated charges through synapses and once the charge is above a certain threshold the neuron fires.

Deep Learning is the evolution of classic Artificial Neural Networks. Since the brain uses different structures to perform different tasks a DL model organizes the network as a sequence of interconnected layers; each layer implements different mechanisms useful for specific contexts. One example is the use of convolution layers in Convolutional Neural Networks. In the animal brain, the visual cortex organizes neurons in a hierarchical structure; neurons closer to the optic nerve are activated by simple features and neurons at higher levels are activated by more complex features and situations. CNNs are inspired by such mechanism; the model is organized as a sequence of convolutional layers that extract features of increasing complexity.

The state of the art Deep Learning models can achieve outstanding results in image, word, and speech processing applications but the number of neurons used by DL models is far smaller than the number of neurons present in a human brain. In an animal brain, the number of neurons is in the order of 10^{11} with 10^4 synapses per neuron while the most complicated Deep Learning models use at best millions of parameters(10^6 - 10^9).

Since hardware solutions can deliver performances of orders of magnitude higher than programmable architectures, research efforts have been directed toward developing HW accelerated solutions to train and deploy bigger and more powerful models. One possible solution to accelerate Deep Learning application is the use of Field Programmable Gate Arrays composed by simple reconfigurable blocks. The HW reconfigurability of FPGAs makes them a suitable target to implement sparse models with extremely tailored floating point precision.

Another advantage of FPGA-based solution is the power efficiency; FPGAs maximize performance per watt of energy and the reduction of floating point precision allows to decrease the size of memory buffer thus reducing the amount of energy used to perform memory transfers. The disadvantages of FPGA-based solutions are usually limited to the long design time and High-Level Synthesis solutions have been developed to overcome this problem.

This thesis proposes a design flow to ease the deployment of Deep Learning model on FPGA targets. By exploiting the Halide compilation infrastructure and the PandA-Bambu HLS framework the design flow starts from the ONNX intermediate representation of a Deep Learning model and produces the RTL Description necessary for the deployment. By using the Halide infrastructure the computation can be optimized by finding the right schedule for the target FPGA and specific application. The schedule can be designed to find the right trade-off between memory locality, parallelism, and storage granularity; this allows the programmer to find the schedule that satisfies application-specific constraints such as maximum latency, minimum throughput, and maximum power consumption.

The thesis is organized as follows: Chapter 2 provides an introduction to the field of Artificial Intelligence, Artificial Neural Networks, and Deep Learning. Chapter 3 reviews state of the art frameworks related to the work of the thesis. Chapter 4 describe the proposed design flow and the Bambu back-end implemented as part of the Halide infrastructure.

Chapter 2

Definitions

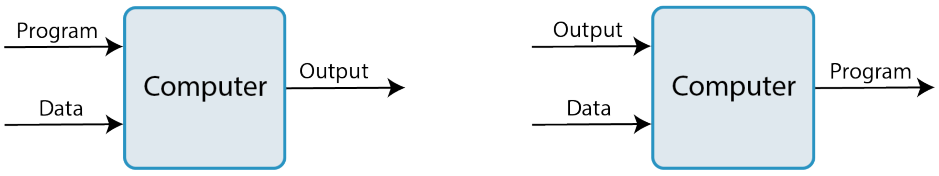
This chapter introduces some preliminary concepts. Section 2.1 introduces the early ideas in Artificial Intelligence and how they started advancements that led to the state of the art methods. Section 2.2 introduces the Artificial Neural Network (ANN) model and how it is used in the Machine Learning context; we introduce the concept of Deep Learning and how Convolutional Neural Networks can be used to perform pattern recognition on images and spatially organized features. We also introduce the problem of deploying Deep Learning models in resource-constrained applications. Sections 2.3 introduce the ONNX IR and why we need an intermediate representation for Deep Learning frameworks.

2.1 Artificial Intelligence and Machine Learning

The Artificial Intelligence field is born in the 1950s with the goal of creating thought-capable artificial beings. With the creation of the computational model of the Turing machine, the Computer Science field started studying how machines can act humanly and solve problems that were previously considered as an exclusive domain of human intelligence. Early efforts to create agents capable of showing intelligent behavior were directed toward creating decision-making algorithms, mostly focused on solving games with graph representations. Even if the first results were able to solve problems that were previously considered unsolvable, the field soon realized that the approach was not well suited to solve different and more general problems.

Even if the early approaches were too general and failed to realize how complex the problem of creating an Artificial General Intelligence is they sparked the emerging field of Machine Learning. The Computer Science field realized that the graph-based approach lacks the flexibility that is needed to create agents that show intelligent behavior; humans do not interact with their environment by optimizing a utility function and performing search procedures in a graph of possible states. Intelligent agents should be able to learn directly from the environment how to perform the computation needed to solve a specific problem.

The Machine Learning field changed the AI perspective by changing the paradigm of how the computation is performed. The program is not seen as given input of the system and the process is split into training and inference phases. The training phase uses a set of sample data to learn how to perform a task without being explicitly programmed to solve it. The learned model must then be able to solve the same problem while being able to generalize on previously unseen samples.



(a) Traditional programming *(b) Machine Learning*

Figure 2.1: Comparison between the Traditional Programming approach and the Machine Learning approach

Figure (a) show how the computation is usually performed in a traditional program execution; the computer produces the output by taking the program and the data as input. As a comparison figure (b) shows how the Machine Learning approach is fundamentally different. The program is not an input of the com-

puter; the program is the output of the learning phase and is meant to be used to generalize on unseen data.

By splitting the process into training and inference phases the algorithm learn a representation of the task to be used later as previous knowledge of the problem. Moreover, since the ML approach does not specify the kind of model that needs to be used it can adapt by using the model that better fits a given task.

One of the most interesting models generated by the ML field is the Artificial Neural Network that is going to be presented in the next section.

2.2 Artificial Neural Networks

Artificial Neural Networks are powerful Machine Learning models inspired by how animal brains work. ANNs are data driven models composed of artificial neurons and synapses that perform nonlinear transformations of input data. This section presents how ANNs work and how they evolved to solve more complex problems by using DL and CNN layers.

2.2.1 Vanilla neural networks

An Artificial Neural network is composed of multiple interconnected neurons. Each neuron is based on the perceptron model; a model invented in the 1950s to perform binary classification on input features.

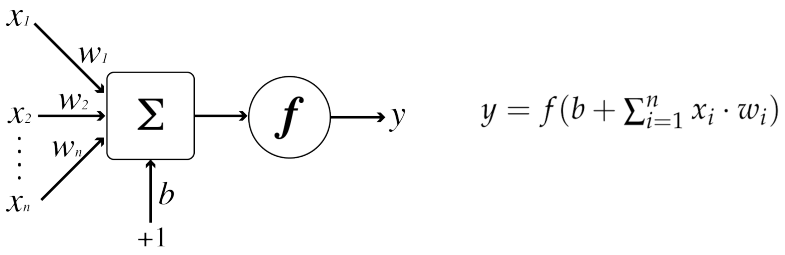
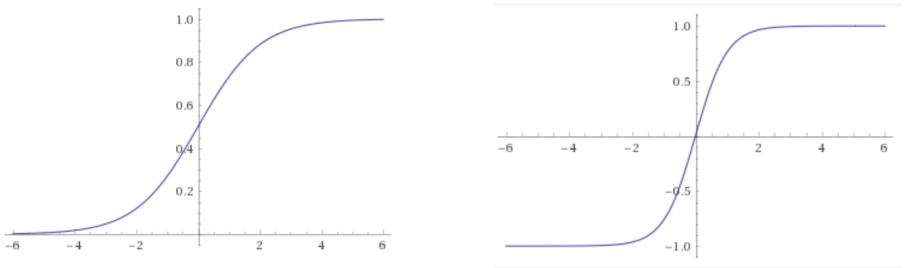


Figure 2.2: Perceptron model

To learn nonlinear representations of the input data the perceptron model uses a nonlinear activation function before propagating the output value to the output connections. Two simple and common activation functions are the sigmoid and tanh functions.



(a) $y = \frac{1}{(1+e^{-x})}$ *(b)* $y = \tanh(x)$

Figure 2.3: Sigmoid (a) and Tanh (b) functions

The perceptron model can be composed creating a network of interconnected neurons, usually referred to as Multi-Layer Perceptron.

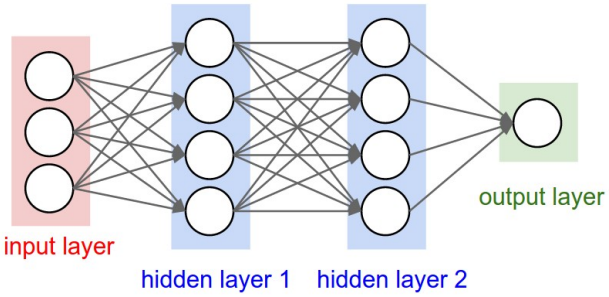


Figure 2.4: Example of Multi-layer Perceptron network. [3]

The MLP model uses a topology named Feed Forward configuration. A FF neural network is an MLP composed of a sequence of fully connected layers; a FF network always has one input and one output layer with a set of hidden layers in between.

Since the ANN model is computationally demanding and its use has been held back by the limited computational resources available, the use of the Feed Forward configuration is usually preferred to exploit SIMD instructions. By seeing the network as a sequence of transformations over the input features, the algorithm can easily parallelize the training and inference phases on specialized HW by using SIMD instructions. This allows the use of bigger and more complex networks leading to better performing models.

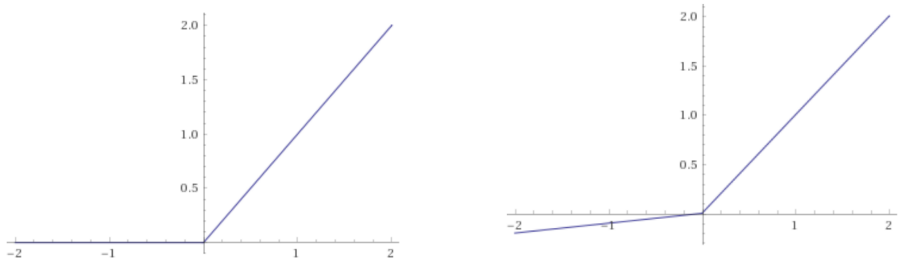
2.2.2 ANN Training

As previously stated a Machine Learning model need to be trained on a set of training data before being used with new and previously unseen samples. The most common method to train a Neural Network is through supervised learning using the back-propagation algorithm.

The back-propagation algorithm is a gradient-based optimization method. Since the Neural Network is a differentiable model, given a differentiable function the back-propagation algorithm can calculate the network gradient (Also known as backward-pass) and iteratively optimize the network weights toward values corresponding to lower loss values.

The back-propagation algorithm is able to learn a set of weights that approximate the desired output, even in the case of Neural Networks with multiple hidden layers. The weights belonging to hidden layers can not be directly calculated; the back-propagation algorithm needs to use the chain rule to calculate the gradient of hidden layers. This leads to the problem of the vanishing gradient. Since the gradient vanishes while propagating through the layers this limit how deep a Neural Network can be. The gradient shrinks and layers distant to the output get trained more slowly than layers close to the output.

The problem is solved by using activation functions that do not shrink the gradient at each layer propagation; a possible example is the use of the ReLU activation function.



(a) $y = \max(0, x)$

(b) $y = \begin{cases} x & x \geq 0 \\ ax & otherwise \end{cases}$

Figure 2.5: Relu (a) and Leaky ReLU (b) functions.

The ReLU function is introduced to avoid the problem of the vanishing gradient while being able to introduce nonlinearities. Since the ReLU function has gradient equal to 0 when $x \leq 0$ most of the connections become deactivated during the training phase. To solve this problem multiple activation functions have been proposed; one example is the Leaky ReLU function that assigns a small gradient $a \leq 1$ when x is negative.

2.2.3 Deep learning and Convolutional Neural Networks

The Artificial Neural Networks are powerful models to process high dimensional features but there are situations where the MLP model would require too many parameters to find a good approximation. To make an example in spatially organized data, such as images and sequence of words, the number of weights that an MLP would require to find a good model would be intractable.

The term Deep Learning is often used to refer to the state of the art NN models that use different types of layers to tailor the model to the type of task that the network is supposed to learn. DL models can use layers specifically designed to analyze sequences of data by using LSTMs and Attention layers. Other kinds of networks can use convolutional layers to extract features among spatially organized data such as 2D convolutions for images and 3D convolutions for videos.

In the field of Computer Vision, a convolution operation is a well-known operation used to extract interesting features in an image.

The equation to compute a convolution on an image I and a kernel h is

$$G(r,c) = (I \circledast h)(r,c) = \sum_{u=-L}^{L} \sum_{v=-L}^{L} I(r+u, c+v) \cdot h(-u,-v) \tag{2.1}$$

For example when performing the edge detection the Canny edge detection algorithm use convolution operations to smooth the input image and detect the gradient wrt the x and y coordinates of the image.

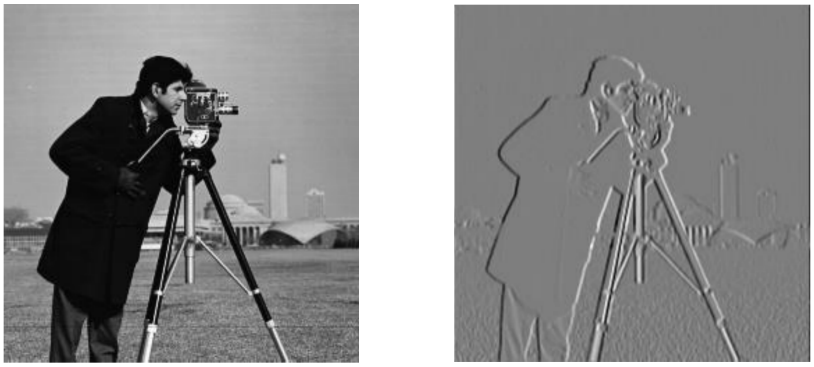
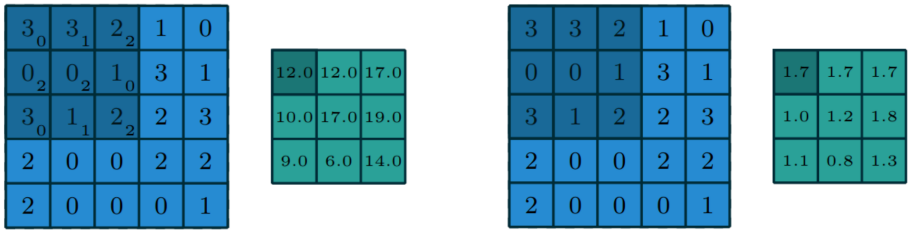


Figure 2.6: Horizontal derivative with Sobel operator

In the DL field, a Convolutional Neural Network is a NN that uses convolutional layers, usually to analyze images. In a CNN the network is organized as a sequence of convolutional layers usually followed by few fully connected layers. The sequence of convolutional layers learn the filters to be extracted directly

from the input data; the layers extract features of increasing complexity as the layers progress from the input to the output layer. Between convolutional layers is common practice to use pooling layers. This is done to decrease the data dimensionality and be able to use more kernels to extract high-level features.



(a) 2D Convolution *(b) AVG Pool*

Figure 2.7: Examples of 2D convolution (a) and AVG Pool operation (b). [4]

One interesting difference between how convolutions are typically handled and how a CNN handle convolution is how they are performed on images with multiple channels. The convolution operation is performed as usual but in the end, all convolved channels are added pixelwise. The number of channels of the output tensor is equal to the number of kernels considered by the convolution operation.

The CNN topology is inspired by how the animal visual cortex work; multiple neurons are connected in a hierarchical way to recognize features of increasing complexity. Neurons on the low level of the hierarchy recognize small features such as simple lines and corners; neurons at higher levels recognize more abstract features such as more complex shapes and objects.

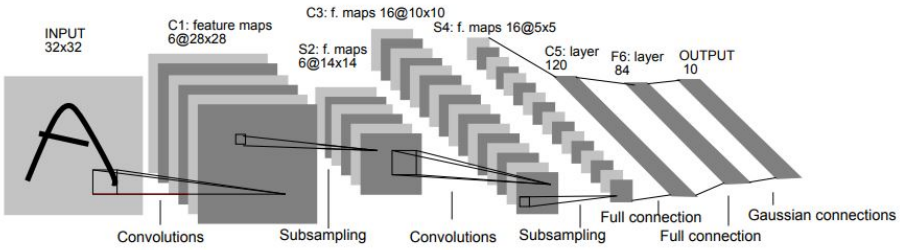


Figure 2.8: LeNet-5 architecture. Hand written digit recognition in 28x28 gray scale images. [8]

2.2.4 DL accelerators

When a DL model must be deployed in a real system the forward pass can not always be done by simply using a CPU to perform the computation. The system is

likely to have latency, throughput, and power constraints that a simple CPU implementation is not able to deliver. CPUs have limited throughput performance and can exploit a limited amount of coarse and fine-grained parallelism.

Since the training phase uses batches of data to calculate the gradient for the next weight update GPGPUs are the most common choice. The high degree of parallelism that can be exploited from different weights and the different samples of the batch allows a GPU implementation to exploit all the available parallelism. In the case of multiple GPUs available there are methods allow to use all the available computational power; multiple GPUs can be used to train bigger models with more parameters and use bigger batches to better approximate the true gradient.

Even if GPUs work great for parallelism when training a model they aren't always a good option when deploying a model. GPUs are power-hungry and since have a dedicated memory to perform computation they require time-consuming memory transfers every time a kernel is executed. This might be a problem, especially in latency constrained applications where the system is required to process a stream of data where each sample depends on the decisions that the model has made at previous steps.

Multiple solutions are available to solve the problem of delivering high throughput and low latency while not exploiting the parallelism created by processing batches of data. The obvious and best possible solution to solve the problem would be to use specialized ASICs to perform the computation. This would deliver the best possible results in terms of final system performance but have its drawbacks. The long design time of specialized HW solutions is impractical in situations where fast prototyping is a desirable feature of the design process. Moreover, ASIC solutions are not adaptable to new models; an ASIC would be fixed not able to implement new and yet undiscovered layers.

A different and more adaptable solution while preserving the low latency and high throughput performances on streams of data is to use reconfigurable HW. A possibility is the use of FPGAs as deployment targets. FPGAs are configurable devices that incorporate logic and memory blocks; the configuration can be designed to implement the specific function that needs to be computed by the specific application. The use of FPGAs as targets allows the deployment of low-level HW while being able to deploy different models by just reconfiguring the HW configuration [1].

Another advantage of FPGAs over other solutions is the power efficiency attainable; as long as the implementation manages properly the memory, the use of FPGAs allows to maximize performance per watt. Moreover, the reconfigurability allows to implement operations whit a floating point precision with an

arbitrary number of bits; the HW solution is not restricted to use a number of bits that is a power of 2. This leads to a more tailored implementation improving area utilization and buffer sizes.

The disadvantages of FPGA implementations are usually limited to a maximum buffer size and the long design time required to create a working and efficient implementation. The long design time can be reduced by using High-Level Synthesis tools that take as input a high-level specification of the computation (as an example the tool can take as input a filter written in C language) and output the bitstream ready to be deployed.

The result provided by an HLS tool does not have the same performance as a manually designed solution but the results are comparable. This also allows fast prototyping while enabling non HW experts to use specialized HW to deploy their trained models in real applications.

2.3 ONNX

On the software side of the Deep Learning field, multiple frameworks emerged to ease the process of creating and training new models. Since the DL field is relatively new there is no established intermediate representation to optimize the abstract computational graph before producing the actual implementation; different frameworks use different representations designed to work only with one framework.

The Open Neural Network eXchange (ONNX [11]) has been created by Microsoft and Facebook to allow the portability of computational graphs between different frameworks. The goal is to create an intermediate representation that can be used by different frameworks and be used to implement optimizations common to all frameworks.

The ONNX representation can also be used as a common input representation for compilation stacks. Since ONNX supports the translation from computational graphs of main DL frameworks to an ONNX representation, the compilation stack can avoid the burden of implementing a different input procedure for each input DL framework and just rely on the ONNX standard. ONNX is also likely to be maintained and if new and more recent frameworks are going to be developed the stack can rely on ONNX to provide a translation from the new computational graph to the already specified representation.

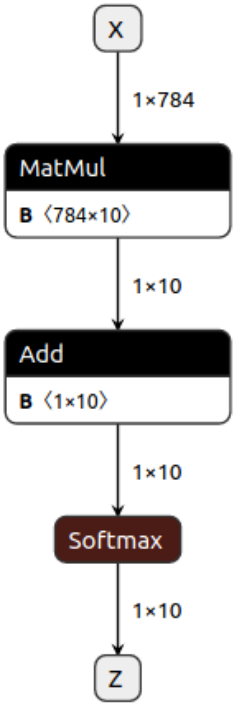


Figure 2.9: Example of ONNX model.

2.4 Conclusions

This chapter has provided the concepts necessary for the thesis. We described the main characteristics of Artificial Intelligence and what lead to the Machine Learning field. The chapter also described Deep Learning models and how they are implemented in the context of image processing applications. The next chapter is dedicated to the exploration of tools related to the contribution of the thesis.

Chapter 3

State of the art summary

This chapter explores previously existing implementations of development stacks used to produce stream computing applications and deep learning accelerators. This chapter also explores tools that have been used in the proposed design; their understanding is necessary to explain how they have been used to reach the final software representation.

3.1 Chapter structure

Section 3.2 introduce StreamIt, a research compilation infrastructure to deploy optimized streaming applications from a high level definition. Section 3.3 describe VitisAI, a proprietary end-to-end development stack to deploy DL models on Xilinx HW. Section 3.4 describes Halide, an open source language and compilation infrastructure to produce optimized code for image and tensor processing.

3.2 StreamIt

Writing a program in the context of a streaming application while satisfying a set of requirements in terms of throughput and latency requires careful management of memory transfer and HW resources. That problem can be solved by writing custom code that directly manages these aspects but the development process is error-prone and slow.

StreamIt is a programming language and research compilation infrastructure for streaming systems. It is an MIT project that started in the year 2000 with the goal of executing high-performance streaming applications while increasing the programmer productivity through stream-specific abstractions. The compilation infrastructure takes as input a program written with the StreamIt programming language, performs stream specific optimizations, and produces the optimized final program to be deployed on the target architecture.

3.2.1 Streaming Application Domain

The application domain of a streaming program is characterized by a set of assumptions [14] that can be leveraged by a development stack. By leveraging these characteristics a streaming programming language can increase performances while providing a set of abstractions that make easier for the programmer to write and maintain the source code of the application.

Large stream of data

The execution model of a general program is fundamentally different than the model of a streaming application: A common program is usually invoked with a finite input set that is kept until termination; instead, in the context of a streaming application the input is virtually infinite and is a sequence of data items where each element requires a finite amount of operations before being discarded. A stream program can run indefinitely, waiting for items to be processed from an external input source.

Independent stream filters

To exploit the parallelism of a streaming application the programmer must specify a set of interconnected filters. The whole set of interconnected filters is referred to as Stream Graph and contains information about communication among filters and how to exploit the intrinsic parallelism of the specific application.

Each filter defines a self-contained transformation on data items that can be executed independently from all others, while the connections among filters define the computation pattern, and thus the kind of parallelism that can be exploited.

Stable computation pattern

To be able to exploit the parallelism among filters and all other advantages of the streaming domain, the stream graph is assumed to be constant during steady state execution. The domain allows occasional modifications to the computation pattern but can lead to significant performance degradation if used improperly by changing its structure frequently.

Occasional out-of-stream communication

Even if in a streaming application most of the communication volume is dedicated to data items, there is still a need to send control messages on an irregular and infrequent basis. The domain allows for infrequent control messages from host to filter and between filters to react to specific situations. For example, the host might want to change a filter's parameter during runtime execution.

High performance expectations

The streaming application is going to be deployed in a real system that is going to have its own application-specific real-time constraints (Throughput, Latency, Power consumption, Memory, ...).

3.2.2 Streamit Program

The StreamIt programming language is a high-level specification of the application; it defines what the final application should do in terms of item transformations and communication without defining how the final code will perform these operations.

A StreamIt program is composed of 4 main blocks defining the model of computation:

Filter Basic unit of computation of a StreamIt program.

Contain two main functions, work and init; the work function defines the transformation performed on input data items whenever the actor is fired, the init function defines the initialization procedure of the filter needed to initialize the first invocation. A filter can be stateful or stateless and must always define at compile time the amount of push, pop, and peek operations that are performed at each invocation of the work function.

In a streaming application, each filter must represent a self-contained transformation on data items and the only way for a filter to communicate with other filters is through FIFO channels. StreamIt applies an additional constraint on the communication among filters, each filter has only one input and one output channels; this allows the compiler to further optimize the final code and extract more parallelism from the application.

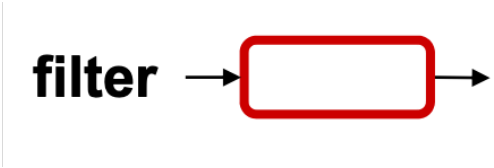


Figure 3.1: StreamIt Filter

Pipeline High-level abstraction of a software pipeline among filters.

Allow the programmer to specify a set of filters that are connected through a pipeline connection; this allows the compiler to exploit pipeline parallelism [5].

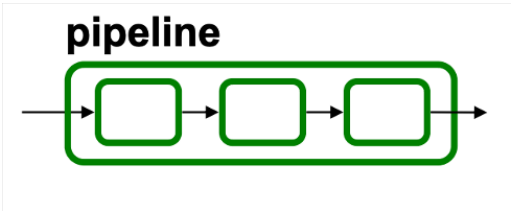


Figure 3.2: StreamIt Pipeline

SplitJoin Allow the specification of multiple parallel and independent computation paths that diverge from a common splitter and merge into a common joiner.

Can be used to extract both task level and data-level parallelism. If all items are sent to all filters and each filter performs a different transformation on the data item it extracts task-level parallelism; if the incoming data stream is load balanced among the different filters and all filters perform the same operation it extracts data-level parallelism. The behavior of the SplitJoin construct is thus defined by the behavior of the splitter and the joiner blocks.

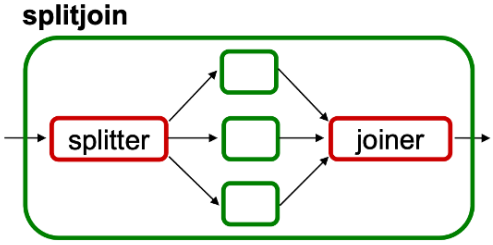


Figure 3.3: StreamIt SplitJoin

Feedback loop Allow the creation of loops in the computation.

Wrap a body filter in a feedback loop; the data items that enter the section are merged with the feedback connection through a joiner block while the items that exit the body filter are split between the feedback connection and the next filter according to a splitter block.

The programmer can also define a computation path along the feedback connection, this allows transformations on items that are sent along the feedback connection.

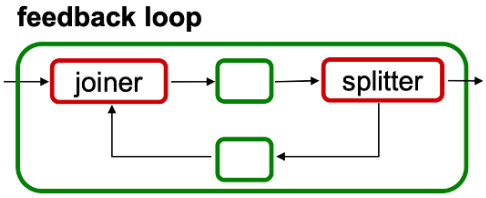


Figure 3.4: StreamIt Feedback Loop

3.2.3 Compilation Infrastructure

The StreamIt compilation infrastructure [6] is designed to start from the high-level abstraction of a StreamIt program and produce the final application to be deployed. The programmer describes the application in terms of which transformations need to be applied to each item and how different transformations are

connected; then, the StreamIt compiler automatically optimizes the StreamIt program for the streaming context and produce the final optimized code for the target back end.

Stream Graph Scheduling

One important information that is compiled from the StreamIt code is the Stream Graph of the application. The Stream graph is an internal representation of the streaming application that represents how different actors interact; the StreamIt compiler uses it to retain information about how different filters interact and the amount of parallelism that can be exploited among filters.

The goal of the Stream Graph Scheduling phase is to find a steady state schedule that allows the program to process the stream of input items while maintaining constant the number of live items on each communication channel.

Partitioning

Given the Stream Graph and the steady state schedule of the application, the compiler needs to adapt the computation to the granularity of the target architecture. The Stream Graph compiled from the StreamIt program does not take into consideration the number of processing elements. Mapping directly the original nodes to physical processors would lead to unoptimized results; in case of a too coarse-grained graph, the program would not use some processing elements, in case of a too fine-grained graph the final program would require unnecessary memory communication. The Streamit compiler uses a heuristic to decide the number of nodes to be mapped to physical elements, the graph is adapted by producing a new graph with a number of nodes that match the number of processing elements of the target architecture. To create the new stream graph, the compiler must implement Fusion and Fission operations [15]. Fusion merges two filters into one and Fission does the opposite operation. Fusion and Fission are not simple operations, the compiler must take into consideration the type of connections between filters to apply these operations efficiently.

To partition the stream graph in a set of balanced filters, the compiler must be able to estimate the amount of work performed by the work function of each filter. StreamIt estimates the work through static inspection. Since the number of iterations of each loop is known at compile-time, the compiler can estimate the total amount of work by unrolling each loop and considering the number and the type of instructions executed at each fire.

The partitioning phase can then be performed automatically with a simple greedy algorithm that split the most demanding filters and merges the least demanding ones.

Layout

Given the work balanced nodes from the partitioning phase, the StreamIt compiler needs to assign each node to a physical node on the target architecture while minimizing the communication overhead between nodes. To optimize the communication and synchronization overhead a back end dependent cost function must be defined and optimized. The Cost function should accurately measure the added communication and synchronization generated by mapping the work balanced Stream Graph to the communication model of the target.

Communication scheduler

When the layout phase has mapped filters to physical computation nodes the only remaining abstraction that needs to be mapped to the target architecture is the communication queues between filters. The communication scheduler maps the infinite FIFO abstraction to the limited resources of the target architecture while avoiding deadlocks and starvation.

Once the Communication Scheduler phase is finished the StreamIt compiler is ready to generate the final code for the target back-end.

3.3 VitisAI

VitisAI [17] is a development stack for AI inference on Xilinx's HW; it is a complete development stack to optimize and deploy a pre-trained model on FPGA without the need of knowing any implementation detail about the underlying HW.

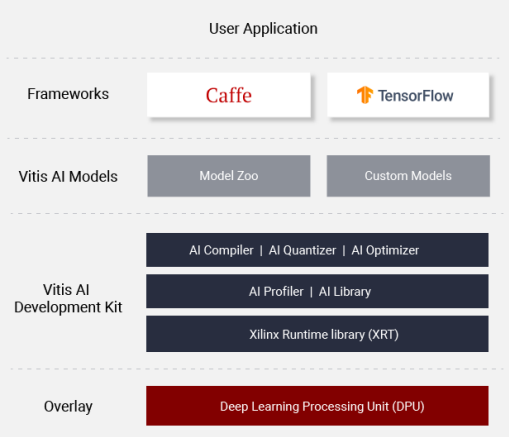


Figure 3.5: Vitis AI stack

The programmer can feed to the VitisAI stack an abstract model in ONNX format or defined with one of the most common frameworks. The stack automatically optimizes the model for the target architecture with few optimization tools and produce the final program to be deployed on the target Xilinx architecture.

3.3.1 Optimization tools

The stack has multiple free tools that can be used to perform optimizations on the input model; these optimization tools can also be used as preprocessing steps on an abstract model before feeding it to a different stack.

AIOptimizer Perform pruning on the input model, it automatically prunes the connections between artificial neurons that less affect the accuracy of the final model.

AIQuantizer Automatically convert the floating-point precision of the input model's weights to fixed-point representation. This is especially useful on FPGA

since it is possible to use a completely custom number of bits without being restricted by the HW design.

AICompiler The quantizer performs HW dependant optimizations on the quantized model, it maps the model to optimized DPU instructions while performing optimizations like node partitioning and instruction scheduling.

3.4 Halide

With the increasing size of deep learning models and the widening gap between processing power and memory bandwidth, it is increasingly difficult to train and deploy new and more computationally demanding DL models. A possible solution might be to write custom code to exploit the right trade-off among data locality, recomputation, and parallelism but the process is slow, prone to errors, and does not allow for fast prototyping of different scheduling solutions and the programmer has a constant focus on implementation details. Halide [12] is a language and compiler to write high performance tensor processing pipelines for multiple target platforms with the goal of solving the emerging problems of image processing pipelines and deep learning applications; the Halide domain-specific language allows to separately define the algorithm that needs to be executed, as functions of the input tensor, and the scheduling strategy to be applied for the specific algorithm on the target architecture. Decoupling the definition of a Halide program in two different components allows the exploration of a large number of scheduling strategies with far fewer lines of code and without affecting the correctness of the program.

3.4.1 Algorithm definition

The Halide algorithm describes what the programmer wants to compute without defining how the final result should be computed; this guarantees the correctness of the final program while experimenting with different scheduling strategies and leave to the compiler the responsibility of optimizing the final code. The Halide DSL represents the algorithm definition as an Abstract Syntax Tree of 3 main components: Variables, expressions, and functions.

Each function is defined over a set of variables and it's value is computed as an expression of other functions. As example we can use the 3x3 blur filter definition:

$$\mathsf{blur\_x}(x, y) = (\mathsf{input}(x-1, y) + \mathsf{input}(x, y) + \mathsf{input}(x+1, y))/3;$$

$$\mathsf{blur\_y}(x, y) = (\mathsf{blur\_x}(x, y-1) + \mathsf{blur\_x}(x, y) + \mathsf{blur\_x}(x, y+1))/3;$$

The pipeline is composed of 2 functions, blur_x and blur_y; both functions are defined over two different dimensions that define the x and y coordinates of the image. The value of each function is an expression composed as the mean of the 3 nearest pixels in both directions.

3.4.2 Schedule definition

Now that the algorithm is defined Halide need to know how to compute the results in terms of storage and compute granularity. Storage granularity and compute granularity refers to how big the buffers between stages need to be and how frequently consumers and producers are interleaved.

One possible naive solution might be to compute and store each stage of the pipeline one at a time and separately from others; this is the approach of the most common frameworks, after each stage the results are stored in main memory before proceeding with the next stage of the pipeline. This approach suffers from the drawback of having poor locality since every tensor must be computed and stored in its entirety and is unlikely to fit into a low-level memory. A second possible approach is to compute only the last stage of the pipeline without storing intermediate results; this maximizes locality by recomputing every value each time but requires an amount of work that grows exponentially with the number of stages in the pipeline. A third approach is to allocate large buffers while performing fine-grained computation, this allows for solving both problems by reusing all previously computed results while exploiting locality. The problem of this last approach is the introduction of dependencies into computed results thus reducing the amount of parallelism that can be exploited from different chunks of work. The programmer should find the right trade-off by using store_at and compute_at directives and by performing tiling, unrolling, and reordering operations on different pipeline stages.

Another important decision to be made by the schedule is how to use HW specific features to increase the pipeline performances (such as multiple cores, vectorized instructions, GPUs, and so on..). The schedule allows the programmer to combine in different ways different features for each variable and function;

this makes easier for the programmer to explore different acceleration strategies without touching the code that defines the algorithm.

Finding the right schedule to manage storage and compute granularity while exploiting the HW specific features is not a trivial task, especially when considering complex pipelines for complex image filters and DL applications. For that reason Halide has a built-in autoscheduler [2] to automatically find a schedule; the current version requires extensive optimization to find a schedule that is as good as one manually created by an expert programmer and can be used to find a baseline in a small amount of time.

To make an example we can take again the 3x3 blur filter:

$$\mathsf{blur\_y.tile}(x, y, xi, yi, 256, 32).\mathsf{vectorize}(xi, 8).\mathsf{parallel}(y);$$

$$\mathsf{blur\_x.compute\_at}(blur\_y, x).\mathsf{vectorize}(x, 8);$$

The schedule uses both parallelization and vectorization operations, tile the loop nest and compute blur_x for each unique value of the variable x of blur_y. The programmer only needs to specify how the computation is performed, Halide takes care of all implementation details and generates efficient code.

3.4.3 Halide Compiler

The Halide compiler [12] takes as input both the algorithm definition and the schedule and produce a cross-platform internal representation. The internal representation is then optimized by performing target-independent optimizations and compiled into the final code of the target back-end.

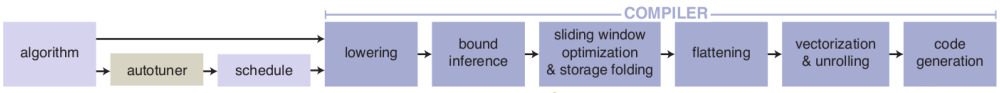


Figure 3.6: Halide stack [12]

The first step of the Halide compilation infrastructure is to lower the algorithm and schedule definition to a set of loop nests and buffer allocations. Each loop is labeled as serial, parallel, unrolled, or vectorized and loop bounds are left as symbolic expressions of the required region of the output function. The lowering process starts from the output function and recursively proceeds backward toward input functions. The process is complete once all functions have been lowered.

Once the Lowering process has been completed the Halide compiler stack has a complete representation of the Halide program as a loop nest operating on multidimensional tensor objects. The compiler knows the set of loops and the number of dimensions that are required to compute the final result but do not know the extent of such dimensions. The Bound Inference procedure is a two step process. The first step calculates the extends of each tensor; the compiler propagates backward the information about the size of the output tensor and calculates recursively the extents of intermediate bounds. The second step uses this information to calculate the extent of each loop.

By knowing the extent of each dimension, Halide can traverse the loop nest to seek for sliding window and storage folding optimizations. These two optimizations passes are used to leverage the storage granularity allowed in the current schedule and reuse data already computed in previous iterations.

The optimized sequence of loop nests with multi-dimensional store and load is then lowered to a single-dimensional strided representation. Each multi-dimensional tensor is converted into a single-dimensional representation and each load and store operation on tensor is represented as a stride access on the corresponding single-dimensional buffer.

The final step before invoking the target back-end to compile the Halide IR into the final code is to remove unrolled and vectorized loops. Unrolled loops can be simplified by performing the loop unrolling operation directly on the halide IR while vectorized loops can be removed by replacing them with dense memory accesses and operations using ramps to represent strided sets of indexes on memory buffers.

3.4.4 Deep learning applications

DL models are composed of a sequence of layers that can be expressed as a Halide algorithm by composing functions and expressions. As for other most common DL frameworks, Halide automatically differentiates feed-forward models; the programmer only needs to define the model structure, and the Automatic Differentiation return a new function that calculates the requested derivatives. The Halide Automatic Differentiation [9] system must take into consideration some optimizations to exploit as much parallelism as possible while performing gradient propagation on scatter-gather operations; since gather operations become scatter operations when differentiated and scatter operations are not easily parallelizable, the Automatic Differentiation system automatically converts scatter operations back to gather operations to allow a higher degree of exploitable parallelism in the final gradient function.

Since the increasing complexity of modern DL models is making impractical the definition by hand of a model as a Halide algorithm the programmer can provide an abstract definition with a cross-platform format like ONNX. Providing an abstract model avoids the programmer the tedious and error-prone work of defining a model as a composition of functions. Halide automatically converts the ONNX model to a Halide algorithm and returns a function that can be used to perform inference or differentiated through automatic differentiation to calculate the gradient for the backpropagation algorithm.

Chapter 4

Design flow

In this chapter we introduce the main contribution of this thesis, we describe the proposed design flow and how it is implemented as part of the Halide infrastructure. In section 4.1 we describe the motivations to use HLS tools to deploy DL models on FPGAs. We describe the main HW platform that can be used to perform inference and its drawbacks. In section 4.3 we describe how the Bambu back-end works and how it is implemented inside the Halide infrastructure. We describe the main steps and possible future improvements.

4.1 Motivations

Since the state of the art DL models are becoming increasingly more complex and demanding in terms of computational power, using CPUs and GPGPUs as inference platforms are becoming a less appealing approach. When a model is deployed the system is required to satisfy latency, throughput, and memory consumption constraints; CPUs are not suitable for high throughput applications, and power hungry GPUs have high latency due to memory transfers between host and accelerator. The use of specialized ASICs meets all 3 requirements with high throughput, low latency, and memory consumption; but such a solution is not flexible and not able to express new layers that might have not been yet invented.

A different approach is to use Field Programmable Gate Arrays (FPGA); the use of programmable HW allows the possibility to satisfy all system requirements while being able to use the system to deploy different models. The problem with FPGAs is the time required to deploy an FPGA's application; deploying a correct and optimized bitstream is not trivial and requires a long design time. To reduce that problem High-Level Synthesis tools are used; these tools take as input the abstract representation of a program, in this case, a DL model, and produce as output the optimized bitstream to be deployed. The final result does not have the same performance that a manually designed and optimized solution would have but allow fast prototyping and the use of FPGAs to deploy models without the need of being an expert in the design process.

The integration of a Bambu back-end in Halide allows the programmer to easily deploy a DL model on FPGA; the decoupling of the schedule from the algorithm definition allows the programmer to leverage the possibility to deploy multiple models in a short amount of time and find the schedule that better fit the target architecture and memory hierarchy.

4.2 Design flow

The goal of the proposed design flow is to leverage PandA - Bambu and the Halide infrastructure to deploy a Deep Learning model on FPGA while being able to optimize the sequence of scheduled operations for the target architecture.

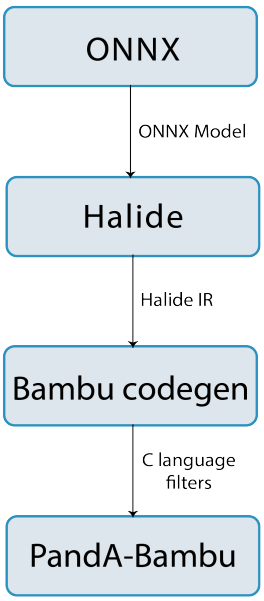


Figure 4.1: Design Flow

The starting point is a Deep Neural Network model provided by one of the most common frameworks. Since each framework represents the computational graph in different ways we use a common representation to provide a unique entry point to the proposed design flow. The Open Neural Network eXchange specification (ONNX) has been used as a common representation; using ONNX allows the portability of computational graphs between different frameworks and is already supported by the Halide infrastructure.

To increase the performance of a forward pass of a DL model the design flow incorporates two optional steps: By using the VitisAI tools the programmer can perform pruning and quantization on the ONNX model. The pruning optimization increases the performance of the final model by reducing the amount of work that has to be done by removing the less useful connections between neurons and the quantization reduces the floating point precision of the operations. The lat-

ter is especially useful for FPGAs since the floating point precision can be easily adapted.

Given the ONNX model that needs to be deployed the Halide framework has a built-in tool to convert it to an internal generator. The programmer can then decide how to schedule the imported model by defining the schedule method of the generator, as for any other algorithm.

Halide can then proceed by converting the scheduled generator to an internal IR before feeding the optimized representation to the Bambu back-end that has been developed to provide support for FPGA targets.

4.3 The Bambu back-end

To implement a new back-end the Halide infrastructure allows the creation of target-specific codegens that take as input the Halide IR and produce the final optimized code. Each target architecture has its own back-end, that allows the Halide infrastructure to take advantage of architecture-specific features and optimizations.

The Bambu back-end takes as input the Halide IR and produces the C code to be used within the Bambu framework. The code generation is split into 3 main steps: Filter extraction, IR optimizations, and code generation. The next sections describe the three steps in detail.

4.3.1 Filters extraction

The filters extraction step takes as input the Halide IR and produces a DAG representation of the Halide lowered function. The DAG is composed of interconnected filters related to a producer-consumer relationship. Such a relationship is already extracted by the Halide infrastructure and can be easily extracted by leveraging the ProducerConsumer statement of the Halide IR.

Before proceeding with the extraction of the DAG filters the codegen prepare the Halide IR by preprocessing filters and buffer names. The ONNX model already specifies these names and the Halide IR does not check if the name starts with a number and if special characters are present. Moreover, the Halide IR does not check for clashing buffers and filters names. The codegen changes these names by substituting special characters with an underscore and by renaming buffers and filters by adding a unique identifier at the end.

After the preprocessing step of the ONNX identifiers, the next is the substitution of available compile-time information. The Halide IR consider the sizes of input and output tensors as data that are known only during runtime execution and retrieve such information when the input and output tensors are passed to the function. Since the ONNX model does not work as a general image processing pipeline, the information about input and output tensors is fixed and already present inside the ONNX representation. We can then take the values already present in the ONNX model and substitute them inside the Halide IR; this step simplifies the loop extents generated by the Halide's bound inference procedure, usually leading to loops with a constant extent.

With the unique identifiers set and the compile-time information provided by the ONNX model already incorporated inside the Halide IR, the next step is to eliminate constructs not necessary for the extraction of the DAG representation. Since the Halide IR retrieves information about the input and output tensors at runtime, the generated IR performs a sequence of checks on the input data before starting the computation. These checks involve detecting if the extends are negative and if the input tensors have the sizes different than the ones specified by the ONNX models. These checks are useless for the translation of the Halide IR and are therefore removed.

The last step before translating the Halide IR to a DAG representation between filter is the simplification of the IR. The Let statements inside the IR code are eliminated and substituted inside other constructs; this will turn useful during the IR optimization when we will push as outside as possible all stores that do not depend on the inner loop variable. Since the names of loop variables are complicated and contain several special characters they are substituted with new and unique identifiers; variable names inside the IR are substituted accordingly with the unique identifiers assigned to the corresponding loop.

Listing 4.1: Example of filter extracted from the Halide IR. The extracted code is hard to translate to c language and access the memory buffers at each iteration.

```
X_im_padded_U_sum_3[0] = 0.000000f
for (x_7, 0, 3) {
    for (x_8, 0, 3) {
        X_im_padded_U_sum_3[0] = ((float32)X_im_padded_U_sum_3[0]
          + ((float32)b2[((x_7*6) x_3)]
          * ((float32)b2[((x_8*6) + x_4)]
          * (float32)b0[((((x_2*3) + x_8)*3) + x_7)])))
    }
}
```

Another important simplification is about Allocate statements. Since the allocation of buffers with constant size is important to produce an efficient result; nonconstant buffer sizes are substituted with the maximum possible size allocated

during execution. To do so the codegen considers the expression corresponding to the allocated size and by considering the minimum and maximum values of all contained variables find and substitute the maximum allocated size of the buffer.

With the input IR containing all the available compile-time information and cleared of unused constructs, we are ready to translate the preprocessed IR to a DAG representation. The first step is the extraction of the Stages already extracted by the Halide IR. The stages are already represented ad Producer-Consumer relationships, the codegen uses these relationships and extract one stage for each producer element in the IR. The elements extracted are then connected according to Producer-Consumer relationships present in the IR. To extract the relationships when a consumer statement is found a Producer-Consumer relationship is created between the consumed stage and the innermost producer.

When all stages have been extracted with all the Producer-Consumer relationships the codegen is ready to create the DAG representation among filters. Since it is easier for Bambu HLS to parallelize function calls instead of entire bodies of for loops we decided to split the stage code into multiple functions. To do that when the code contains loops marked as parallel the codegen split the stage into multiple filters. The result is a collection of filters connected by function calls; when a loop is meant to be parallelized the body of the loop always contains only one function call to the function intended to be executed in parallel.

Since the Halide schedule allows the execution of multiple stages at root granularity, the DAG obtained by the previous steps might have multiple filters that are not invoked by others. This introduces the necessity of having a unique entry point that executes these filters in the order specified by the Halide IR. To finalize the extracted DAG before proceeding with the optimization phase we need to introduce one additional filter meant to be used as the entry point of the computation. The codegen introduces a filter named "bambu_main_filter"; the new filter allocates the necessary buffers and invokes the functions executed at root granularity in the order specified by the Halide IR.

4.3.2 IR optimizations

Once the filters have been extracted, the program has been represented as a set of interconnected independent filters related to producer-consumer relationships. Each filter has its independent code represented as a Halide statement and the IR code must be optimized by performing a set of optimization passes over the IR code representation, such as reducing as much as possible redundant memory accesses and by avoiding recomputation of data shared by different iterations.

The first step is to extract the necessary information by visiting the IR code.

The set of variables and buffers accessed during the code execution are stored with their required data type, and if they are not allocated inside the IR code, they are marked as parameters of the filter. Dependencies between filters are extracted from the function calls generated by the stage to DAG translation, and the information about allocated buffers is stored in a map common to all filters containing the type and the size of the buffer as an expression of other variables.

A second important piece of data collected from the code IR is the kind of access performed by the filter on a specific buffer. A filter can access a buffer by reading the content of the buffer with a Load statement or by saving something on the buffer by accessing it with a store statement. The codegen collects information about the filters accessed during execution, the type of access performed, and if they are accessed multiple times.

Once the necessary information is extracted from the IR of the filter the codegen must prepare the IR for the optimization passes. The IR is currently stored as a loop nest containing all the operations in one store statement in the innermost loop; this happens because in a preprocessing step we decided to remove all let statements by substituting them into the cleaned IR. This has been done to reduce the number of parameters required by a filter but the resulting IR is not suitable to be used as a starting point. To prepare the IR for the optimizations we decided to split the store statements in multiple elementary stores that perform one operation before assigning the result to a unique variable. This is done by recursively producing a store statement for each operator, each store computes one elementary operation and stores the result in an intermediate variable. While doing that, the codegen check if the operator to be produced is already computed and stored in another variable. If that happens the codegen avoids recomputing the result and uses the already computed value.

Now that the necessary data have been retrieved and the IR has been preprocessed the IR is ready to be optimized. The first optimization step that is performed on the preprocessed IR is pushing as outside as possible each store statement while preserving the order of computation of operators that depend on the same set of variables. To do that, firstly, the codegen needs to compute the dependency of each store statement to each allocated variable inside the filter's IR; this information is then leveraged to push as outside as possible each store statement. The codegen check at every declaration if the newly declared variable satisfies the requirement of some of the store statements that still need to be placed inside the mutated IR code; if for some variable that happens, the codegen moves the store operation after the new declaration. If multiple declarations have all dependencies satisfied by the new declaration the codegen places the store statements without altering their relative order.

Listing 4.2: Example of optimized IR. The complex operation of the previous example has been split into simple operations. Intermediate results that are shared among different iterations and memory accesses has been pushed outside

```
X_im_padded_U_sum_3[0] = 0.000000f
_9[0] = (x_2*3)
_17[0] = (float32)X_im_padded_U_sum_3[0]
for (x_7, 0, 3) {
    _3[0] = (x_7*6)
    _4[0] = (_3 + x_3)
    for (x_8, 0, 3) {
        _6[0] = (x_8*6)
        _7[0] = (_6 + x_4)
        _10[0] = (_9 + x_8)
        _11[0] = (_10*3)
        _12[0] = (_11 + x_7)
        _2[0] = (float32)_17
        _5[0] = (float32)b2[_4]
        _8[0] = (float32)b2[_7]
        _13[0] = (float32)b0[_12]
        _14[0] = ((float32)_8*(float32)_13)
        _15[0] = ((float32)_5*(float32)_14)
        _16[0] = ((float32)_2 + (float32)_15)
        _17[0] = (float32)_16
    }
}
```

The previous optimization cannot be performed on store and load operations on buffers. Some operations produced by the Halide infrastructure are intended to be performed multiple times on the same index and pushing outside these operations would affect the correctness of the final program. Pairs of load and store operations on the same buffer and at the same index can be optimized by performing the memory access as outside as possible in the loop nest. The computation can then be performed on a local variable before storing the final result only at the end of the computation. To perform the optimization on the filter's IR the codegen detect, for each loop, when an inner pair of load and store operations do not depend on other variables. If that happens, the codegen load the current buffer value in a local variable, substitute the load and store operations inside the loop body with the newly declared temporary variable and store back the computed result at the end of the loop body.

Another important optimization step is the detection of queues between filters. The Bambu framework can implement FIFO connections as communication mechanisms between filters and the codegen can optimize the IR by detecting when a pair of load and store statements need to be substituted with dequeue and enqueue operations. The current implementation of this optimization considers for each filter its output connections. Before checking if the indexes of the two

operations match the connection must satisfy some preliminary constraints. For each considered connection, the starting and ending filters must have correspondingly one write and one read operations on the considered buffer. Moreover, the two operations must be parallelized under the same variables and have a matching loop nest. If all the prerequisites are satisfied, the codegen can compare the indexes of the two operations, and if they match the store and load operations are substituted with dequeue and enqueue operations.

4.3.3 Code generation

Before producing the final code the generation phase needs some preprocessing. When we extracted the information about the buffers and the variables we detected the required arguments by identifying the ones that are not declared by the filter. This mark correctly buffers and variables that need to be allocated elsewhere as parameters but there is no guarantee that they are going to be available to the invoking filter. Some parameters might be allocated in a different filter connected through multiple sequential connections in the DAG. For that reason, as preprocessing step, we need to propagate the parameters until they reach the declaring filter. The codegen iteratively considers the parameters required by the filter invocations; if they are not declared they are added as requested parameters to the caller of the filter. The step is repeated until the requested parameters reach a fixed point.

After the parameters have been propagated the next preprocessing step is to finalize the function calls. Until that point, each call has been represented by only the name of the filter that needs to be called; now that the parameters are available we can change the function call with the actual signature of the function.

Store and load operations on buffers can also be finalized; read operations on queues are changed with dequeue operations. Conversely, since store operations may write on multiple queues while also writing on an array buffer, the codegen generates the necessary enqueue and write operations.

Once the IR has been preprocessed the codegen is ready to produce the final C filters. After the inclusion of the necessary dependencies, like prototypes for invocations and vectorized operations, the codegen produces the function signature by using the data collected about the return value and the required parameters. The filter's IR is then translated to a valid C code by translating line by line the IR representation. The final code is produced by encapsulating the translation line by line of the IR in the previously created function signature.

To avoid declaring multiple times the same variable, the codegen keeps in memory

the name of previous declarations. Therefore, if a store operation assigns a result to an already declared variable the codegen generates the assignment without declaring the LHS variable.

Listing 4.3: Output C code of the example filter. The Halide IR has been translated to C code and encapsulated. Since the filter compute only one value a return statement has been added at the end of the function.

```
float X_im_padded_U_sum_3_fun_0(float b0[18], float b2[18],
       int x_2, int x_3, int x_4){
    float X_im_padded_U_sum_3 = 0x0p+0;
    X_im_padded_U_sum_3 = 0x0p+0;
    int32_t _9 = x_2 * 3;
    float _17 = X_im_padded_U_sum_3;
    for(unsigned int x_7 = (0); x_7 < (0 + 3); x_7++){
        int32_t _3 = x_7 * 6;
        int32_t _4 = _3 + x_3;
        for(unsigned int x_8 = (0); x_8 < (0 + 3); x_8++){
            int32_t _6 = x_8 * 6;
            int32_t _7 = _6 + x_4;
            int32_t _10 = _9 + x_8;
            int32_t _11 = _10 * 3;
            int32_t _12 = _11 + x_7;
            float _2 = _17;
            float _5 = b2[_4];
            float _8 = b2[_7];
            float _13 = b0[_12];
            float _14 = _8 * _13;
            float _15 = _5 * _14;
            float _16 = _2 + _15;
            _17 = _16;
        }
    }
    X_im_padded_U_sum_3 = _17;
    return X_im_padded_U_sum_3;
}
```

Since some operations can not be specified inside the Halide IR, the codegen implement other optimizations during the code generation phase. When a filter implements an add bias operation the output code is composed as a sequence of if operators, which is highly inefficient. The codegen optimizes such operation by storing the values in a static array and by accessing the right one by index.

Another optimization is the optimization of ramp operations on vectorized operations. When the values of the ramp operation are known at compile-time, the codegen can directly allocate the ramp values without computing them at runtime. This is done by statically allocating the array containing the ramp values.

4.4 Parallel and vectorized operations

An important factor that influences the performance of the schedule on the target architecture is how to handle coarse and fine-grained parallelism. In the halide infrastructure, coarse-grained parallelism is represented by parallel loops that represent pieces of computation that can be performed independently without modifying the correctness of the output buffers. Fine-grained parallelism is represented by vectorized operations on vectors.

To handle coarse-grained parallelism PandA-Bambu allows the definition of for loops parallelized with the #pragma omp for directive. This allows to perform each independent iteration in parallel and on different HW resources.

Listing 4.4: Example of coarse-grained parallelism exploited by parallelizing independent iterations. The final model will perform each iteration of the for loop on different resources.

```
    #pragma omp for
    for(unsigned int x_0 = (0); x_0 < (0 + 53); x_0++){
        _conv1_2_0_fun_0(b0, conv1_2_0, b1, data_0, x_0);
    }
```

Fine-grained parallelism is handled by performing vectorized operations on small arrays. The usage of vectorized operations on simple independent operations allows to exploit the advantage of FPGA platforms on fine-grained operations.

Listing 4.5: Example of vector multiplication exploiting vectorized operations. In this example the multiply operation is performed on small arrays representing vectors of 8 elements.

```
void _Z_fun_0(float X_im_0[8], float Y_im_1[8], float Z[8]){
    int32_t _0[8] = {0, 1, 2, 3, 4, 5, 6, 7};
    float _1[8];
    load_vector<float, int32_t, 8>(_1, X_im_0, _0);
    float _2[8];
    load_vector<float, int32_t, 8>(_2, Y_im_1, _0);
    float _3[8];
    binary_op<float, float, 8, std::multiplies<float>>(_3, _1, _2);
    store_vector<float, int32_t, 8>(Z, _3, _0);
}
```

4.5 Additional generated code and files

When the codegen has produced all the necessary filters to perform the computation, we still need some additional code and files.

To be able to start the computation, the generated C filters require the definition of multiple static buffers. These buffers are generated by the Halide infrastructure; they are represented as arrays of data that do not change during the computation and are necessary to compute the correct result. In our case, since we are considering ONNX models, these buffers usually contain the weights of the Deep Learning model; without them, the application would not be able to compute the correct transformation of the input tensor.

To ease the loading procedure of these buffers, we decided to store them into dedicated .dat files. The programmer can use a handle function to load each buffer into memory without the need to know how the data are organized. Moreover, this allows the programmer to easily swap the buffer values by loading different ones, without the need to recompile the entire ONNX model.

Another file produced is the one containing handle functions for vectorized operations, named Vectorization.hpp. When vectorized operations are produced, the filter uses small arrays to store the vectors. Instead of writing each operation every time as a parallel loop over the elements of the array, the filter calls a handle operation that performs the computation on the vectors. This makes the code easier to read and reduce the size of the generated filters.

4.6 Conclusions

In this section we described the structure of the design flow. We described the steps necessary to optimize and deploy a DL models using the ONNX representation, the Halide infrastructure, and PandA-Bambu as HLS tool. We described how is the Halide Bambu codegen structured and how it works.

Chapter 5

Esperimental evaluation of results

This chapter describes how we tested the results produced by the proposed design flow. Section 5.1 presents the test suite of models used to test the results of the final design flow. Section 5.2 describes how the correctness of the back-end has been tested. Section 5.3 describes the setup used to test the performance of a set of selected models taken from the test suite; the section also presents the numerical results on a selected subset of models, taken from the test suite.

5.1 Test suite

To evaluate the results produced by the proposed design flow, we used a suite of models composed of examples taken from the literature. We used some simple operators from [10] and some other complete networks as MLP, LeNet-5, and some winners of the ImageNet competition.

5.1.1 Single operators

vecmul
The simplest models considered are vecmul operations; these operations take as input two arrays and calculate their elementwise multiplication. Two versions of this operation have been considered: a smaller one that performs the multiplication over 8 elements (01_vecmul_a) and a bigger one that uses 64 elements (01_vecmul_b).

dense
The second considered elementary operator test dense operations. Deep Learning models use dense layers in almost every model; they are always present and their memory-centric computation is well suited to be optimized by the Halide infrastructure. The model is implemented as a matrix multiplication followed by a bias add operation. As for the vecmul operations two examples have been considered; a smaller one over 8 input elements (04_dense_a) and a bigger one over 64 (05_dense_b).

softmax
Another important operation performed by Deep Learning models is the softmax operation. The softmax operation is used in networks trained to perform classification over multiple classes. The operation is fairly common and involves computing a transformation over the input that maps all elements in the 0 to 1 range. As for the other operators we used two versions of the softmax operator; the smaller compute the operation over 8 input elements (06_softmax_a) and the bigger one over 64 (07_softmax_b).

conv
Convolution operators are important for CNN models; they are compute-centric layers that perform a convolution operation over the input. The considered models use the winograd algorithm to compute the operation, which is an optimized algorithm to lower the number of matrix multiplication needed by convolutional layers. The smaller model computes the convolution over a grayscale 8x8 image (09_conv2d_a) and the bigger one performs the same operation over a grayscale 64x64 image (11_conv2d_b). Both models use 3x3 kernels.

maxpool
Another important operator for Convolutional Neural Networks is the Maxpool operator. As stated at the beginning of this thesis, this operator is used after convolutional layers to reduce the dimensionality of the input image and reduce the computational requirements of the model. As for the other operators we chose to use two different examples: a smaller that compute the operation over an 8x8 image (12_maxp_a) and a bigger one that perform the same operation over a 32x32 image (13_maxp_b). The two models have been adapted since the ones from the original paper use a type of padding not supported by the ONNX converted.

thxprlsg
The last single operator from the LeFlow paper is the thxprlsg operator. This operator performs a mix of tanh, exponential, relu, and sigmoid applied to an array with 8 elements (15_thxprlsg). This is done to test the performance of the final result on different activation functions.

These operators represent common elementary operations in Deep Learning models. Operations such as matrix multiplications, convolutions, and max-pooling operations are common to most of the CNN models. Being able to implement the most common operations independently is a requirement. If the design flow fails to implement simple operations it is not worth trying to evaluate the performance of bigger ONNX models.

5.1.2 Complete Networks

MLP
The MLP example is a standard ANN representation; the model computes a matrix multiplication over the input 784 elements followed by a bias add and a softmax operation over the output 10 elements (e1_mlp). This is the first example that composes other operators to compute a model that might be used in a real application.

LeNet-5
Another complete example that we have considered is the LeNet-5 model [8]; the model takes as input a 28x28 grayscale image and performs a sequence of simple convolutions and fully connected layers (lenet). This network goal is to recognize the handwritten digit contained in the input image.

AlexNet
The first complete model that we used is AlexNet [7] [16]. The model achieved a top-5 error rate of 15.3% on the test dataset and has approximately 60 million parameters. The computation uses all the previously tested CNN operators plus dropout layers that have been used as regularization mechanisms (i1_alexnet).

Zfnet
The second model is the Zfnet model [18] [16]. This model improved the results of AlexNet by finetuning the model by a better understanding of how convolutions work (i2_zfnet).

VGG16
The third and final model used to test the design flow is VGG16 [13] [16]. This model achieves 92.7% top-5 test accuracy in ImageNet and uses 138 million parameters (i3_vgg16).

5.2 Correctness evaluation

Once we have a design flow able to deploy an ONNX model, we need to test if the produced filters perform the correct transformation.

To do that we used the test suite of models presented in the previous chapter, we generated the output filters with the Bambu codegen, and we computed the output results by using random inputs. We repeated the same procedure with the standard x86 codegen, already implemented in the Halide infrastructure.

The testing procedure is executed by a Halide application; the makefile takes as input the name of the model to be tested and generate the code for the two targets. For every model that needed to be tested, we defined a test function; the application calls the test function to compute and check the results of the Bambu back-end with the output of the x86 back-end. If the test function finds a difference between the results of the two codegens, print an error to signal that a test has failed and that there is something wrong with the generated results. Since the output matches, even on complete models as VGG-16 and Alexnet, we are sure that the codegen can generate correct filters for the most common applications.

A more extensive procedure might involve generating random ONNX models. Even if the suite includes most of the most common Deep Learning architectures there might be edge cases. Generating random ONNX models would allow us to create a more robust codegen, by testing it in situations non considered by the test suite.

5.3 Performance Evaluation

To test the performance of the final result, we decided to use some operators and complete models taken from the test suite.

Since we are interested in Deep Learning models, and in particular in Convolutional Neural Networks, we used the two main building blocks of CNNs, which are MaxPool and Convolutional layers. These operators are the most commonly used in image processing applications, and by testing the final performance of these layers separately, the results should give an idea of the overall performance of the design flow on image processing pipelines.

To test more complete examples we used one other model from the design suite, which is MLP. Since MLP is a simple Artificial Neural Network, we used its model to test the performance on simple networks that do not incorporate convolutional operations. Its structure is well suited to test the final performance of sequences of dense and nonlinear transformations.

5.3.1 Experimental setup

The design flow relies on 3 components to generate final RTL Description: The ONNX IR to provide a common entry point, the Halide infrastructure to optimize the computations, and Bambu HLS to produce the final implementation.

The models used to test the final results have been generated by using ONNX version 1.5.0. The models have been generated by using basic ONNX features since what matters is to provide a common entry point for Halide.

The last release of the Halide infrastructure (2019/08/27) has been used to implement the Bambu backend and optimize the ONNX IR. To translate the ONNX representation we used the ONNX converter app already implemented in the Halide master branch. Even if the Halide app is not able to convert some models from the ONNX representation we are confident that the support is going to be expanded in the future. The project is currently under development and the GitHub master branch receives commits on a daily basis.

Finally, Bambu 0.9.6 was used to translate the C filters, the output of the Bambu backend, to HDL code.

5.3.2 Numerical results

We ran the selected models by using two different approaches: By computing and storing all stages at root granularity, and by using the CPU x86 autoscheduler. The models were compiled from the ONNX representation to the abstract representation through the Halide infrastructure with its Bambu-backend. The results were then synthesized on a Zynq-7000 SoC with gcc and -O3 optimization.

Benchmark	Area	Slack	Cycles	DSPs	Frequency	Registers	Slice
maxpool_8	988	3,76	556	0	88,96	878	381
maxpool_32	988	3,76	556	0	88,96	878	381
conv2d_8x8	12.353	-0,822	442.209	2	63,20	13.702	5.925
conv2d_64x64	16.919	-1,798	18.093.466	2	59,53	15.267	7.750
MLP	6.932	-1,2	57.648	9	61,72	5.048	2.601

Table 5.1: Timing and area estimates targeting a Zynq-7000 SoC with autoscheduler

Benchmark	Area	Slack	Cycles	DSPs	Frequency	Registers	Slice
maxpool_8	832	0,35	82	0	68,25	728	339
maxpool_32	1.025	0,481	2.113	0	68,87	685	349
conv2d_8x8	3.121	-0,554	2.019.778	2	64,29	3.769	1.464
conv2d_64x64	3.234	-0,707	129.261.690	2	63,66	3.824	1.528
MLP	3.904	-0,872	107.958	9	63,00	3.359	1.449

Table 5.2: Timing and area estimates targeting a Zynq-7000 SoC without autoscheduler

It is not straight forward to compare the results; sometimes the performance is better when using the x86 autoscheduler and sometimes a simple compute root approach leads to better results. The best solution is to try different solutions and select the one that best suits the constraints of the target-specific application.

Even if the results do not provide a clear and unique approach they show the impact that the schedule can have on the final implementation. The performance of a model with a carefully managed schedule can easily outperform a schedule that follows a naive heuristic.

5.4 Conclusions

In this section, we presented the test suite used to test the proposed design flow. We addressed the problem of ensuring that the produced abstract representation is correct, and that leads to high-performance implementation.

In the next section, we are going to highlight some of the limitations of the current implementation and present some possible future development directions to expand the capabilities of the current design flow.

Chapter 6

Conclusions and future developments

This chapter confronts the goals that were set at the beginning of the thesis with the actual result. We outline the advantages and disadvantages of the final solution while discussing possible future developments to improve the performance of the current design.

6.1 Design flow

At the beginning of the thesis, we discussed the main advantages of deploying Deep Learning models on FPGA. We discussed how the characteristics of FPGAs make them ideal targets to deploy sparse models with custom precision while being energy efficient. The drawbacks of FPGAs as HW solution were presented as two-fold: the first problem is related to the long design required to produce an RTL design and deploy the model; this also does not allow fast prototyping of different solution which is a desirable characteristic of the development process. The second problem is related to the performance of the final result; just deploying the model as a sequence of transformations on tensors is highly inefficient. The target would be required to store buffers with the size of an entire tensor and all the operations would require loading off-chip memory leading to very poor data locality. Since memory transfer is usually responsible for most of the power consumption, continuous communication with the off-chip memory would defeat the purpose of using FPGAs in an application where the power consumption is a requirement of the final application [1].

The solution to the first problem is the proposed design flow itself. The proposed stack eases the design process by only requiring the programmer to convert the computational graph of the Deep Learning model to its ONNX IR representation. The model can then be optimized by performing weight pruning and quantization; the Halide infrastructure optimize and translate the ONNX model to a software representation and the PandA-Bambu framework translate the software representation to Verilog/VHDL code.

The second problem is addressed by the scheduling of the Halide infrastructure. One of the main characteristics of halide is the separation of the algorithm definition and scheduling. The scheduling allows the programmer to optimize the computation by finding the right trade-off among data locality, parallelism, and buffer sizes. Halide also allows to automate the process by using an autoscheduler [2],

6.2 Future developments and current limitations

The main current limitation is the lack of an autoscheduler for FPGA targets. Other autoschedulers are already available (Current only for x86 CPUs) but they do not take into consideration the HW resource of the target FPGA and try to predict characteristics that make sense only in the CPU case (Such as the number of page fault costs) [2]. Another disadvantage of currently available autoschedulers is the fact that they do not take into consideration the HW limitations of

the target; for complex models, the autoscheduler easily produces a schedule with buffer sizes that exceed the available memory. Using an autoscheduler that takes into consideration the HW resources and limitations would result in a significant improvement in terms of ease of use and final quality of results.

Another limitation is the simplicity of how the back-end extract queue channels between filters. The current implementation use queues only in the most simple case; single read/write access, parallelization under the same variables, same loop nest, and matching read/write indexes. A more complete implementation should be able to extract queues between filters that do not have such static and strict requirements.

Pruning and quantization operations are currently performed by relying on the VitisAI tools and thus only limited to Xilinx Hardware. Developing other tools able to optimize DL models for generic targets and not only for Xilinx HW would allow the programmer to optimize models without using other resources.

In terms of support for Deep Learning models, the current implementation support models implemented by the ONNX to Halide converter implemented inside the Halide infrastructure and some types of layers are not supported. That limitation affects the type of models that can be deployed by using the proposed design flow, but Halide is currently under development and will improve the support in the future.

List of Figures

List of Tables

Bibliography

[1] "A Survey of FPGA Based Deep Learning Accelerators: Challenges and Opportunities". In: *CoRR* abs/1901.04988 (2019). Withdrawn. arXiv: 1901.04988. URL: http://arxiv.org/abs/1901.04988.

[2] Andrew Adams et al. "Learning to optimize halide with tree search and random programs". In: *ACM Transactions on Graphics* 38 (July 2019), pp. 1–12. DOI: 10.1145/3306346.3322967.

[3] *Convolutional Neural Networks for Visual Recognition*. URL: https://cs231n.github.io/convolutional-networks/.

[4] Vincent Dumoulin and Francesco Visin. *A guide to convolution arithmetic for deep learning*. 2016. arXiv: 1603.07285 [stat.ML].

[5] Michael I. Gordon, Bill Thies, and Saman Amarasinghe. "Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2006). San Jose, CA*. Oct. 2006. URL: https://www.microsoft.com/en-us/research/publication/exploiting-coarse-grained-task-data-pipeline-parallelism-stream-programs/.

[6] Michael I. Gordon et al. "A Stream Compiler for Communication-Exposed Architectures". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2002). San Jose, CA*. Aug. 2002. URL: https://www.microsoft.com/en-us/research/publication/stream-compiler-communication-exposed-architectures/.

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Neural Information Processing Systems* 25 (Jan. 2012). DOI: 10.1145/3065386.

[8] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[9] Tzu-Mao Li et al. "Differentiable programming for image processing and deep learning in halide". In: *ACM Transactions on Graphics* 37 (July 2018), pp. 1–13. DOI: 10.1145/3197517.3201383.

[10] Daniel H. Noronha, Bahar Salehpour, and Steven J. E. Wilton. *LeFlow: Enabling Flexible FPGA High-Level Synthesis of Tensorflow Deep Neural Networks*. 2018. arXiv: 1807.05317 [cs.LG].

[11] *ONNX: the open ecosystem for interchangeable AI models*. URL: https://onnx.ai/.

[12] Jonathan Ragan-Kelley et al. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: vol. 48. June 2013, pp. 519–530. DOI: 10.1145/2499370.2462176.

[13] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2014. arXiv: 1409.1556 [cs.CV].

[14] William Thies, Michal Karczmarek, and Saman Amarasinghe. "StreamIt: A Language for Streaming Applications". In: vol. 2304. June 2002. DOI: 10.1007/3-540-45937-5_14.

[15] William Thies et al. "StreamIt: A Compiler for Streaming Applications". In: (Aug. 2002).

[16] Stylianos I. Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. *Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions*. 2018. arXiv: 1803.05900 [cs.CV].

[17] VitisAi. URL: https://github.com/Xilinx/Vitis-AI.

[18] Matthew D Zeiler and Rob Fergus. *Visualizing and Understanding Convolutional Networks*. 2013. arXiv: 1311.2901 [cs.CV].