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

Title

Relatore: Prof.

Tesi di laurea di
Ghitti Marco
Matr. 893986

Anno Accademico 2019/2020

Abstract

Recent advances in the AI field and the increasing computational power of HW accelerators have renewed the interest in how Machine Learning models can be used to solve abstract problems. In particolar the Deep Learning field are powerful biologically inspired models that model how the model process sensory information. Deep Learnings models achieve outstanding results in application where high dimensional data processing is needed and promise 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 target that has been used to accelerate Deep Learning applications are GPGPUs and many software frameworks are built with the intention of leveraging 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 itneresting 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 allow non HW experts to use FPGAs while allowing the programmer to prototype different models before deciding which one best fit the target application.

This thesis proposes a design flow for the implementation of Deep Learning model on FPGAs; the design flow rely 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 decide 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 translate the input C code into hardware descriptive code, ready to be deployed on FPGA.

Ringraziamenti

Ringraziamenti

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 exclusive domain of human intelligence just 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 transmit accumulated charges through synapses and once the charge is above a certain treshold the neuron fires.

Deep Learning is the evolution of classic Artificial Neural Networks. Since the brain use different structures to perform different tasks a DL model organize the network as a sequence of interconnected layers; each layer implement 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 organize neurons in a hirearchical 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 are able to 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 are able to deliver performances of orders of magnitude higher than programmable architecures, 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 make them a suitable target to implement sparse models with extremely tailored floating point precision.

An other big advantage of FPGA-based solution is the power efficiency; FPGAs maximize performance per watt of energy and the reduction of floating point precision allow 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 has been developed to overcome this problem.

This thesis proposes a design flow to ease the deployement of Deep Learning model on FPGA targets. By exploiting the Halide compilation infrastructure and the PandA-Bambu HLS framework the design flow start from the ONNX intermediate representation of a Deep Learning model and produce 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 tradeoff between memory locality, parallelism and storage granularity; this allow to find the scheule that satisfy application specific constraints such as maximum latency, minimum throughput and maximum power consumption.

The thesis is organized as follows: Chapter 2 provide an introduction to the field of Artificial Intelligence, Artificial Neural Networks and Deep Learning. Chapter 3 review 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

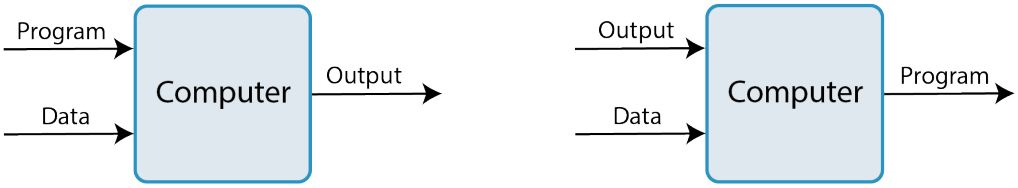
In this chapter we introduce some preliminary concepts. Section 2.1 introduce the early ideas in Artificial Intelligence and how they started advancements that led to state of the art methods. Section 2.2 introduce the Artificial Neural Network (ANN) model and how it is used in the Machine Learning context; 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 Learnign 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 machine can act humanly and solve problems that were previously considered as 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 first results were able to solve problems that were previously considered as 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 a Artificial General Intelligence really is they sparked the emerging field of Machine Learning. The Computer Science field realized that the graph based approach lack 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 algorithms in 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 use 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 produce the output by taking the program and the data as input. As comparison figure (b) show how the Machine Learning approach is foundamentally different. The program is not an input of the computer; 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 is able to learn a representation of the task to be used later as previous knowledge of the problem. Moreover, since the ML approach do not specify the kind of model that need to be used it can adapt by using the model that better fit a given task.

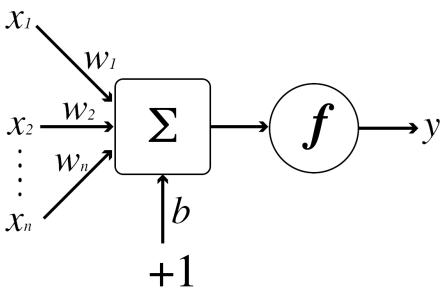
One of the most interesting models generated by the ML field is the Artificial Neural Network that is going to be presented in the next section.

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 by artificial neurons and synapses that perform nonlinear transformations of input data. In this section we present how ANNs work and how they evolved to solve more complex problems by using DL and CNN layers.

2.2.1 Vanilla neural networks

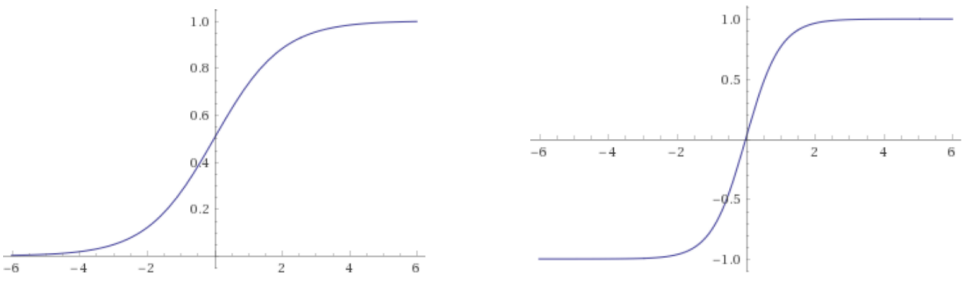
An Artificial Neural network is composed by multiple interconnected neurons. Each neuron is based on the perceptron model; a model invented in the 1950s with the goal of performing binary classification on input features.



$$y = f(b + \sum_{i=1}^{n} x_i \cdot w_i)$$

Figure 2.2: Perceptron model

To learn nonlinear representations of the input data the perceptron model use 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 usally refered as Multi-Layer Perceptron.

The MLP model use a topology named Feed Forward configuration. A FF neural network is a MLP composed by a sequence of fully connected layers; a

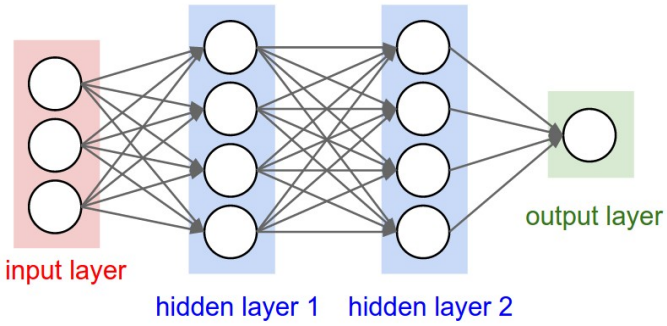


Figure 2.4: Example of Multi-layer Perceptron network

FF network always have 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 hold back by the limited computational resources availble, 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 allow the use of bigger and more complex networks leading to better performing models.

2.2.2 ANN Training

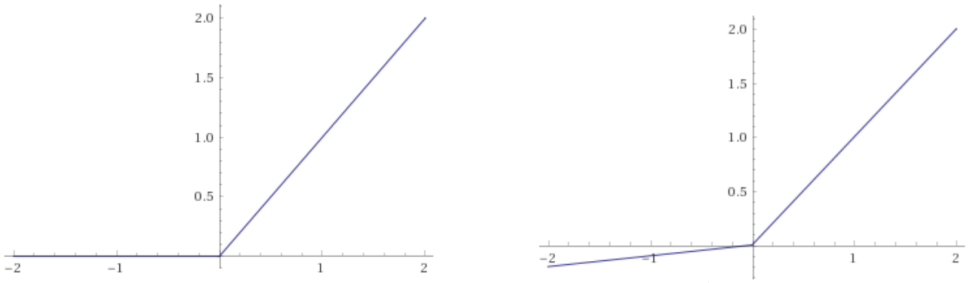
As previously stated a Machine Learning model need to be trained on a set of training data before being used with new and previously unseen samples. The most common method to train a Neural Network is by means of supervised learning using the back-propagation algorithm.

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 cal-

culated; the back-propagation algorithm need to use the chain rule to calculate the gradient of hidden layers. This lead 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. As example we can use the ReLU activation function to avoid the problem.



(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 a nonlinearity. Since the ReLU function have 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 has been proposed; one example is the Leaky ReLU function that assign a small gradient $a \leq 1$ when x is negative.

2.2.3 Deep learning and Convolutitional 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 amount of weights that a 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 type 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

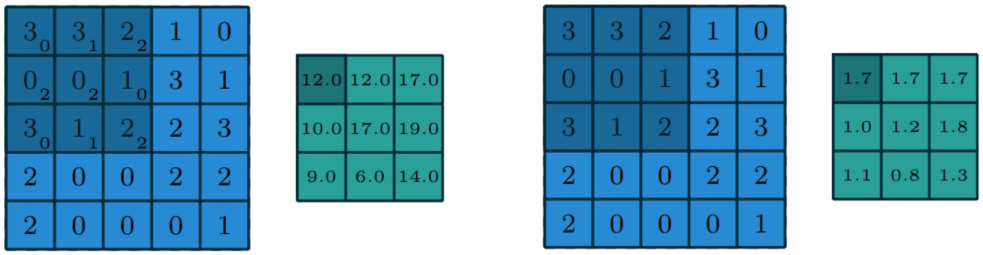
$$G(r,c) = (I \circledast h)(r,c) = \sum_{u=-L}^{L} \sum_{v=-L}^{L} I(r+u, c+v) \cdot h(-u,-v) \tag{2.1}$$

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 obtained by performing the convolution between the image and the horizontal 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 it 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). https://arxiv.org/abs/1603.07285

One interesting difference between how convolutions are tipically 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 at 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 hirearchical way to recognize features of increasing complexity. Neurons on the low level of the hirearchy recognize small features shuch as simple lines and corners; neurons at a higher levels recognize more abstract features such as more complex shapes and objects.

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

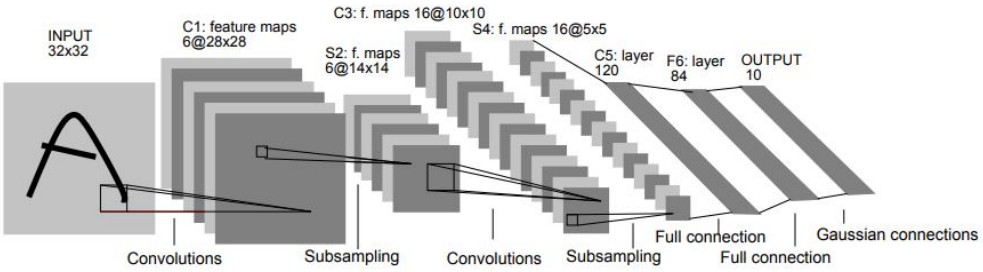


Figure 2.8: LeNet-5 architecture. Used to recognize hand written digit in 28x28 gray scale images.

performance and can exploit a limited ammount 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 choiche. The high degree of parallelism that can be exploited from different weights and the different samples of the batch allow a GPU implmentation to exploit all the available parallelism. In case of multiple GPUs available there are methods that allow to use all the available compuational 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 application where the system is required to process a stream of data where each sample depend 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 resoults in terms of final system performance but have its drawbacks. The long design time of specialized HW solutions is

impractical in situations where fast propotyping is a desirable feature of the design process. Moreover ASIC solutions are not adaptable to new models, an ASIC would be usable only with a fixed subset of models and would not be able to adapt to future models using new and yet undiscovered layers.

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

An other advantage of FPGAs is the power efficency attainable; as long as the implementation manage properly the memory the use of FPGAs allow to maximize performance per watt. An other big advantage of FPGAs over other solutions is the possibility of adapting the floating point precision of each operation to a degree that would not be possible with different HW solutions. The reconfigurability allow 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 lead 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 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 a HLS tool do not have the same performance of a hand designed solution but the results are comparable. This also allow 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 make easier for developers to create and train their models. Since the DL field is realively new there is no established intermediate representation to optimize the abstract computational graph berfore producing the actual implementation; different frameworks use different representations designed to work only with one framework.

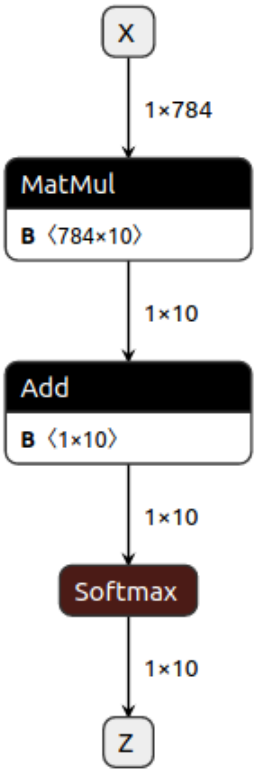


Figure 2.9: Example of ONNX model. The computational graph is represented as a set of interconnected operations.

The Open Neural Network eXchange has been created by Microsoft and Facebook with the goal of allowing the portability of computational graphs between different frameworks. The goal is to create an intermediate representa-

tion 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 common input representation for compilation stacks. Since ONNX support the translation from computational graphs of main DL frameworks to a 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.

2.4 Conclusions

In this chapter we described 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 tesis.

Chapter 3

State of the art

In this chapter we explore previously existing implementations of developement stacks used to produce stream computing applications and deep learning accelerators. This chapter also exlores 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 developement stack to deploy DL models on Xilinx HW. Section 3.4 describe Halide, a 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 require a careful management of memory transfer and HW resources. That problem can be solved by writing custom code that directly manage 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 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 take as input a program written with the StreamIt programming language, perform stream specific optimizations and produce 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 that can be leveraged by a developement 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.

3.2.1.1 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 require 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.

3.2.1.2 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 as Stream Graph and contain information about communication among filters and how to exploit the intrinsic parallelism of the specific application.

Each filter define 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.

3.2.1.3 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 allow occasional modifications to the computation

pattern but can lead to a significant performance degradation if used improperly by changing it's structure frequently.

3.2.1.4 Occasional out-of-stream communication

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

3.2.1.5 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 by 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 define the transformation performed on input data items whenever the actor is fired, the init function define 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 peak 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 apply an additional constrain on the communication among filters, each filter have only one input and one output channels; this allow the compiler to further optimize the final code and extract more parallelism from the application.

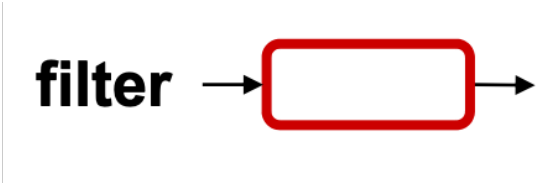


Figure 3.1: StreamIt Filter

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 allow the compiler to exploit pipeline parallelism.

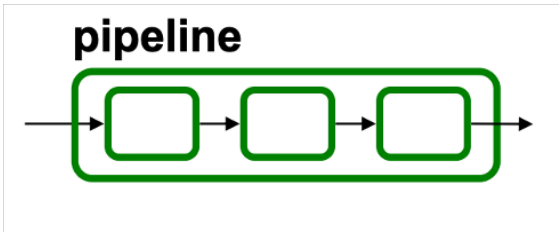


Figure 3.2: StreamIt Pipeline

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 and each filter perform a different transformation on the data item it extract task level parallelism; if the incoming data stream is load balanced among the different filters and all filters perform the same operation

it extract data level parallelism. The behavior of the SplitJoin construct is thus defined by the behavior of the splitter and the joiner blocks.

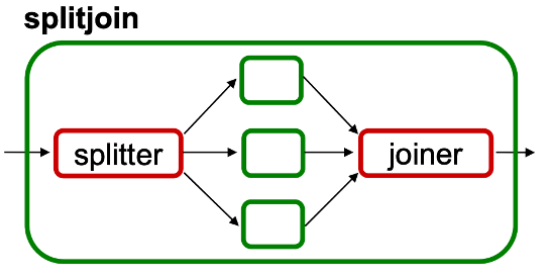


Figure 3.3: StreamIt SplitJoin

Feedback loop Allow the creation of loops in the computation.

Wrap a main 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 splitted 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 allow transformations on items that are sent along the feedback connection.

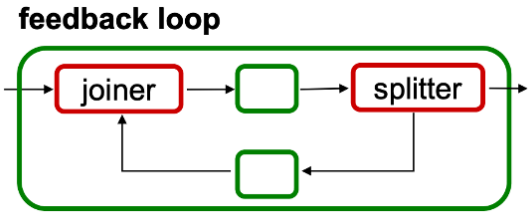


Figure 3.4: StreamIt Feedback Loop

3.2.3 Compilation Infrastructure

The StreamIt compilation infrastructure is designed to start from the high level abstraction of a StreamIt program and produce the final application to be de-

ployed. The programmer describe 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 optimize the StreamIt program for the streaming context and produce the final optimized code for the target back end.

3.2.3.1 Stream Graph Scheduling

One of the most important pieces of 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 represent how different actors interact; the StreamIt compiler use 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 allow the program to process the stream of input items while maintaining constant the number of live items on each communication channel.

3.2.3.2 Partitioning

Given the Stream Graph and the steady state schedule of the application the compiler need to adapt the computation to the granularity of the target architecture. The Stream Graph compiled from the StreamIt program do 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 use 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. In order to create the new stream graph, the compiler must implement Fusion and Fission operations. Fusion merge 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 in order 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 estimate 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 merge the least demanding ones.

3.2.3.3 Layout

Given the work balanced nodes from the partitioning phase, the StreamIt compiler need 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.

3.2.3.4 Communication scheduler

When the layout phase has mapped filters to physical computation nodes the only remaining abstraction that need 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 is a development stack for AI inference on Xilinx's HW; it is a complete development stack to optimize and deploy a pretrained model on FPGA without the need of knowing any implementation detail about the underlying HW.

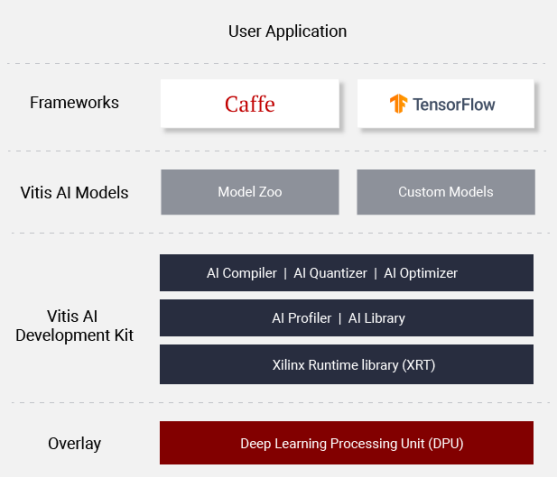


Figure 3.5: Vitis AI stack

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 optimize 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 have 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 prune 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 a 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 perform HW depentant 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, do not allow for fast prototyping of different scheduling solutions and the programmer has a constant focus on implementation details. Halide 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 allow to separately define the algorithm that need 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 allow 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 describe what the programmer want to compute without defining how the final result should be computed; this guarantee 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 represent 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 by 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 refer 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 from others; this is actually 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 suffer of 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 maximize locality by recomputing every value each time but require an amount of work that grow exponentially with the number of stages in the pipeline. A third approach is to allocate large buffers while performing fine-grained computation, this allow 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.

An other 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 allow the programmer to combine in different ways different features for each variable and function; this make 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 have a built in autoscheduler to automatically find a schedule; the current version require extensive optimization to find a schedule that is as good as one manually created by a 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 use 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 need to specify how the computation is performed, Halide takes case of all implementation details and generate efficient code.

3.4.3 Halide Compiler

The Halide compiler take as input both the algorithm definition and the schedule and produce a cross platform internal representation. The internal represen-

tation is then optimized by performing target independent optimizations and compiled into the final code of the target back-end.

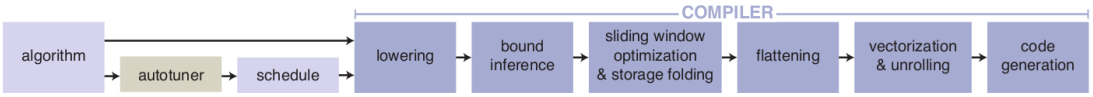


Figure 3.6: Halide stack

The first step ot 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 start from the output function and recursively proceeds backward toward input functions. The process is complete once all functions has been lowered.

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

By knowing the extend 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 by a sequence of layers that can be expressed as an Halide algorithm by composing functions and expressions. As for other most common DL frameworks Halide automatically differentiate feed forward models; the programmer only need to define the model structure and the Automatic Differentiation return a new function that calculate the requested derivatives. The Halide Automatic Differentiation 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 become scatter operations when differentiated and scatter operations are not easily parallelizable, the Automatic Differentiation system automatically convert 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 are making impractical the definition by hand of a model as an Halide algorithm the programmer can provide an abstract definition with a cross-platform format like ONNX. Providing an abstract model avoid to the programmer the tedious and error prone work of defining a model as composition of functions. Halide automatically convert the ONNX model to an Halide algorithm and return a function that can be used to perform inference or differentiated through automatic differentiation to calculate the gradient for the backpropagation algorithm.

Chapter 4

Proposed 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 their 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 is 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 specialized ASICs meet 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 allow 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 require 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 models, and produce as output the optimized bitstream to be deployed. The final result do 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 of the design process.

The integration of a Bambu back-end in Halide allow the programmer to easily deploy a DL model on FPGA; the decoupling of the schedule from the algorithm definition allow 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.

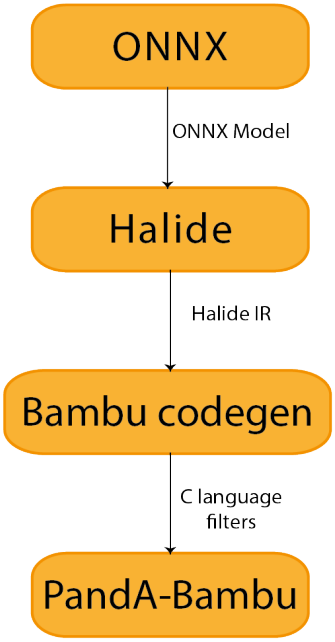


Figure 4.1: Design Flow

The starting point is a Deep Neural Network model provided from one of the most common frameworks. Since each framework represent the computational graph in different ways we to 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 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 models the design flow incorporate two optional steps: By using the VitisAI tools the programmer can perform pruning and quantization on the ONNX model. The pruning optimization increase 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 reduce the floating point precision of the operations. The latter is especially useful for FPGAs since the floating point precision can be easily adapted.

Given the ONNX model that need to be deployed the Halide framework have a built-in tool to convert it to a 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 a 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 allow the creation of target specific codegens that take as input the Halide IR and produce the final optimized code. Each target architecture have it's own back-end, that allow the Halide infrastructure to take advantage of architecture specific features and optimizations.

The Bambu back-end take as input the Halide IR and produce the C code to be used within the bambu framework. The code generation is split in 3 main steps: Filters extraction, IR optimizations and code generation. In the next sections describe the three steps in details.

4.3.1 Filters extraction

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

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)])))
    }
}
```

Before proceding with the extraction of the DAG filters the codegen clean the IR representation by removing unused constructs, renaming variables and buffers to avoid name conflicts and by substituting into the IR values that are already known at compile time such as the dimensions of the input and output tensors. This improves the bound inference analysis of the Halide compiler by reducing the number of variables and increasing the number of fixed size loops inside the Halide IR.

Once the IR has been preprocessed the producer consumer filters can be extracted. Each filter of the DAG representation is going to be fully described by the IR code of the filter and the connections with other filters. Other information about buffers and variables usage are stored in order to make easier the work of subsequent steps.

When the schedule store multiple functions at root granularity the resulting DAG have multiple filters without incoming dependencies. This introduce a entry point problem when trying to generate the final code. To solve such problem a new base filter is always introduced; the bambu_main_filter work as entry point for the computation and is used to store the necessary buffers and call the filters with root store granularity.

4.3.2 IR optimizations

Once the filters has been extracted the program has been represented as a set of interconnected independent filters related by producer consumer relationships. Each filter have its own independent code represented as a Halide statement.

The IR code of each filter can be optimized by performing a set of basic 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 operation that is performed is the IR preparation for subsequent optimization passes. Each IR operation is split into multiple statements that perform basic operations such as single operators and call to external functions. Such operation also improve the readability of the generated code.

The first optimization pass performed is pushing each store as outside as possible of the loop nest. This is performed by computing for each store the dependency on other variables and by pushing an operation outside a for loop only if the operation do not depend on the variable of the loop. This allow the final code to avoid recomputation of data shared among different iterations inside the filter execution. The second optimization pass performed is the fix of multiple read writes on the same buffer inside the filter code. The Halide IR represent the computation by operating directly on the buffer memory. This is not an optimized approach for the code to be generated. When a filter code access a buffer location on the same index inside the same loop the IR is optimized by pushing as outside as possible the read and write memory access inside the IR. The previous memory operations are replaced with access to local variables.

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
    }
}
```

A third optimization pass is the elimination of trivial buffers of size of one element. When a filter operate on a buffer of single size the buffer is replaced with a variable and is passed to the called filter as a single variable. When the buffer with a size of one element is instead the output buffer the filter return the computed value as the return value of the function representing the filter execution.

An other important optimization is the detection of situations when producer consumer operations allow the usage of FIFO queues to send data between filters. If such a situation holds, the memory access on a buffer can be replaced with a FIFO queue. FIFO queues are already implemented and optimized inside the Bambu framework; using such a communication channel can improve the memory management of processed values among filters. At the moment FIFO queues are detected in the trivial case of single read and write accesses on matching indexes and loop nests. A future work might improve the detection of FIFO channel by detecting sliding window optimizations

Once the IR optimization passes has been performed on all filters the DAG is ready to be translated into the C function to be feed to the Bambu framework.

4.3.3 Code generation

After the optimization step the codegen is ready to produce the final code to be used with the Bambu framework. The work is carried out by simply visiting the IR and translating line by line the IR operations to corresponding C operations. To avoid loss of floating point precision when writing constants we decided to write the value of each constant as hexfloat values.

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;
}
```

The result is the set of final filters written in C language and ready to be deployed.

4.4 Parallel and vectorized operations

An important factor that influence 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 allow the definition of for loops parallelized with the #pragma omp for directive. This allow to perform each independent iteration in parallel 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 allow 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 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 work.

List of Figures

List of Tables

Bibliography