TECHNICAL UNIVERSITY OF CRETE

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

MASTER THESIS

TensorGlue: A Framework for FPGA-based Deep Learning Design

Pavlos Giakoumakis

Committee

Professor Apostolos Dollas
Associate Professor Ioannis Papaefstathiou, AUTH
Professor Michael Zervakis

November 26, 2019

Abstract:

In a deep learning framework, the designer provides a description of the neural network architecture, in the form of a computational graph (data-flow graph). The tool is able process this graph and either run it efficiently on fixed-hardware, or generate automatically additional graphs to train the neural network. Nevertheless, this kind of formalization using computational graphs is very close to the hardware design process. The graph can be processed in many ways to not only run the described architecture on fixed-hardware, but to generate hardware designs as well.

In this work, we designed and implemented a novel framework that resembles deep learning frameworks but generates hardware designs in the form of synthesizable C++.

Keywords: Design Flow Optimization, Very High-Level Synthesis, Artificial Intelligence, Machine Learning, Deep Learning, Computational Graphs, Hardware Accelerators, Field Programmable Gate Arrays

Contents

List of Figures

List of Tables

List of Code Snippets

Chapter 1

Introduction

The available data and processing elements are becoming nearly abundant. Because both are space-oriented, we will eventually have to employ space-oriented programming paradigms for any applications that could benefit from this abundance. Dataflow computing, which is inherently space-oriented, is the most obvious paradigm and software frameworks are required to enable an efficient development. Such schemes generally tend to load the processing elements heterogeneously. At the same time, the state-of-the-art technology to produce integrated circuits is becoming more and more expensive and thus a greater limiting factor for the diversity of the computer chips that can be produced, which contrasts the needs for heterogeneous elements. On the other hand, FPGAs do not impose such limits. Hence, FPGA-based systems are the most promising platform for dataflow computing. Furthermore, with the rise of deep learning, artificial intelligence is becoming ubiquitous and deep learning frameworks - special very high-level dataflow software frameworks with advanced mathematical features - are the leading technology in this trend as these algorithms are data and resource hungry. FPGAs are already used in such applications and deep learning frameworks especially designed for FPGAs must be developed.

1.1 Deep Learning

Artificial neural networks are biologically inspired models that employ arithmetic computations organized in multiple layers designed to cope with various machine learning tasks. Deep learning is a sub-field of machine learning that utilizes deep neural networks. Deep neural networks are neural networks of many layers. Most state of the art applications use some kind of deep learning and this trend is projected to increase, primarily because of the ease of training high-performing neural networks when the data are sufficient.

1.2 Hardware Evolution and Deep Learning

The initial abrupt success of deep learning was driven from the hardware evolution, but currently - as the field emerged and grows continuously - the roles have changed and the hardware design is the one that is driven from deep learning. The corresponding market is huge and a diversity of ASIC solutions is expected to try to enter the market for addressing all kinds of problems and exploiting different approaches.

From the one hand this diversity could be seen as a negative omen for FPGAs, as there would be multiple optimized ASIC solutions on the market. However, on the other hand, FPGAs are reconfigurable and this diversity can be also seen as a possibility to thrive. First, innovation will become eventually a harder goal to reach and prototyping will grow in importance. Second, the hardware required to for a solution in a AI problem may be impossible to be foreseen until later development stages. Third, the cost to produce integrated circuits progressively increases and thus imposes significant limits to what can be implemented in an ASIC. Nevertheless, FPGA tools and infrastructure must be improved to let FPGAs be a viable solution.

1.3 Programming Paradigms

Typical software design is control-flow driven. This paradigm is perfect for systems that follow the von Neumann Architecture. In such systems contain a central control unit which decides what is the next operation to execute. Typical digital hardware design follows the register-transfer level (RTL) abstraction, in which someone models the flow of signals between registers. In RTL, the designer must cautiously handle the time and the required clock signals that synchronize the circuit. Dataflow computing is from the programmer side in the middle between these two paradigms. The designer models the flow of the data between different processing elements which somewhat resembles the RTL abstraction. However, in dataflow computing we do not explicitly model the system behavior in time and the need to model clock-related behavior is eliminated. Moreover, the programmer writes factories that instantiate and connect processing elements, which in turn act as factories that process data. Designing individual processing elements normally follows typical software and hardware design principles. The higher levels of the code also follow typical software design, except that the programmer does write factories. Thus, typical languages are perfect candidates for dataflow frameworks. Very high-level dataflow computing differs from usual dataflow computing in many aspects. The most important difference is the availability of very coarse-grained APIs and the continuous endeavor to develop new even coarser APIs. At the same time, a high-level dataflow framework is accompanied from advanced features that greatly enhance the generated design.

1.4 About this Thesis

The main purpose of this thesis is the implementation of a deep learning framework to foster research and development of FPGA-aided deep learning algorithms and applications. This framework is not designed with software development in mind. Instead, it is designed for hardware engineering. Initially, the main goal is to aid the process of directly mapping a neural network architecture to hardware resources and define the corresponding API. Hardware developers will extend this framework to improve the generated hardware and machine learning experts will use it to explore new algorithms out of the restrictions of current ASICs[1]. Hence, to achieve this, the tool through its API exposes abstracted hardware implementation details. Software engineers are now able to get hands on hardware development using a very familiar paradigm, which could have a great impact for reconfigurable devices in general in the future.

The framework is implemented in Python and HLS C++. Python is utilized for implementing most of the framework, while HLS C++ is required mainly to provide a library of kernels that can be employed from the generated code. Hence, the framework currently requires Python which is open source and the corresponding FPGA tools (currently the Xilinx Vivado Suite). The provided user code is in Python and can be processed to generate a Vivado HLS IP Core, that can compute both the described neural network and, optionally, the logic needed to update it (backward graph). All the logic is translated into a hardware module. While the generated code is HLS C++ the formalization is suitable for generating VHDL or Verilog. HLS tools have the drawback that designer is restricted. The designer cannot change the tool itself and must use specific design patterns to achieve the best results. However, the functionality that HLS tools provide is sufficient at least for this stage of the framework development. The user code can also run in CPU without any translation to hardware (without requiring Xilinx Vivado). This can be very helpful in the design phase.

1.5 Framework Goals

The goals of this framework are presented in figure 1.1:

- Maintenance reduction for applications such as deep neural networks, where a significant portion of the generated circuit may be inferred from a brief high-level specifications.
- Fast prototyping using Python scripting.
- Intuitive workflow to enhance creativity and conception.
- Truly hierarchical design flow, in which the tools escalate in complexity in a hierarchical manner.

[1] We consider all new GPUs that are specially designed to accelerate deep learning applications as ASICs too.

- Generate code that is readable and easy to interact with.
- Fast execution on fixed hardware.

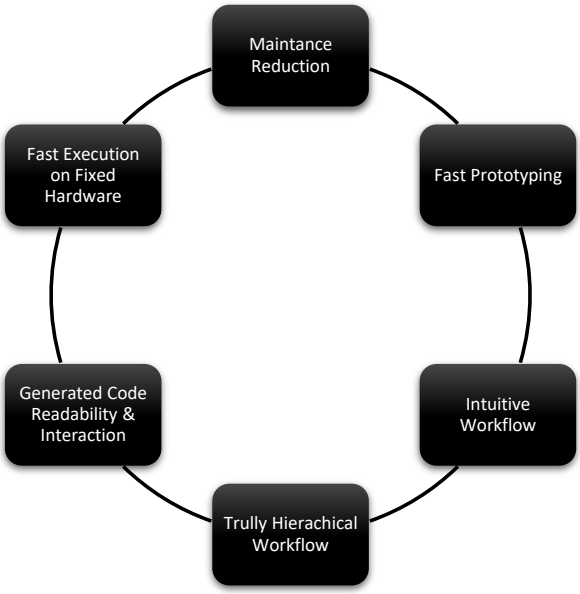

Figure 1.1: The goals of the implemented framework.

1.6 Detailed Motivation

There are numerous reasons to implement such a framework. We present some of them here. First of all, we need tools that specialize on hardware development for deep learning. Furthermore, it has been shown in the past that designing better hardware for deep learning applications is far more important and has greater impact than designing better algorithms that cannot run efficiently on current hardware. Neural networks where there many years before, but deep learning emerged only after the arrival of the powerful GPGPUs and especially CUDA. Second, it has been shown that deep learning algorithms have direct impact on the required hardware implementation and thus we need tools that consider both problems as one to better utilize FPGA resources. Third, we need deep learning frameworks that expose the hardware implementation to the designer as hyper-parameters. Current frameworks are more or less implemented and optimized to generate software not hardware. Thus, they support specific hardware and their APIs do not reflect the reconfigurable nature and the flexibility of an FPGA. Fourth, the evaluation of all current algorithms is closely tied with their performance on current hardware, especially GPUs. With such tools, by merging the processes of algorithm and FPGA-based hardware

development, we unchain the development of algorithms from current hardware. Fifth, we need a new layer of high level tools built on top of existing tool suites where a common programming language is used as a design hub for an FPGA project. An API that simplifies tasks will increase the productivity exponentially. The programming paradigm of deep learning frameworks suits perfectly for this task. The programmer describes a computational graph and this graph is optimized statically before execution, a scheme and a programming style that resembles HDL languages. In fact it may be the first time where software engineers become familiar with a programming style so close to the HDL design process. Hence, a sixth reason is to exploit this situation and help software engineers participate in hardware design. Seventh, current software-oriented frameworks have very different objectives and their APIs and the internal infrastructure changes extremely fast. These changes are more or less an additional burden for the hardware community. For example, the hardware community requires guarantees that the API will change only when it is necessary, especially when building the infrastructure behind the APIs.

Personal motivations included:

1. Get involved with deep learning.

2. Gain an intuition about the current trends in both software and hardware.

3. Get familiar with the maths behind neural networks.

4. Delve into the process of implementing a new deep learning framework.

1.7 Contribution

Contributions of this thesis include:

- To the best of our knowledge, this is the first deep learning framework intended for hardware design (and more specifically FPGA-based).

- To the best of our knowledge, this is the first framework that not only generates the hardware it was directly instructed for (forward computational graph), but it can also generates other computational elements automatically (backward computational graph).

- Currently, the framework generates the HLS C++ code and the corresponding Vivado HLS project. It is a Python library that can be used to define modules with computational graphs and testbenches, run the Python program in CPU, generate the HLS code, generate the HLS testbench, generate the test data, generate the Vivado HLS project, and finally continue to the Vivado HLS to generate the hardware module, run RTL simulation etc.

- The framework employs a unique hierarchical and structured design flow to design hardware. A high-level scripting language is used for the top levels of the design while in the bottom layers employs other lower level languages. The top-layer scripts do not work like other languages that restrict the hardware developer. They adhere to the lower levels. For example, our tool allows to provide many HLS directives on the Python if required[1].

- Simple implementation of automatic differentiation to allow easy implementation of new operators and support back-propagationon hardware. Thus, the generated module executes both inference and training tasks.

- Single-line changes can generate vastly different hardware (e.g. generate only a module for inference versus generating a module that can be trained too). We hope that by finding methods that easily generate a variety of hardware implementations we could better exploit the fine grain reconfiguration capabilities of an FPGA. Current tools take one source and generate one version of the hardware, while our vision is to create a batch of implementations out of a single source. HLS Stream and arbitrary precision types of Xilinx Vivado HLS are supported. It is very easy to add new custom data types.

- We acknowledge that the programing paradigm employed by deep learning framework somewhat resembles hardware design and we present a new design methodology for FPGAs. To the best of our knowledge it is the only HLS scheme that allows generating both optimal software and hardware. In the future it could achieve top performance in software out-of-the-box using the same source code, whereas other hardware design techniques cannot run efficiently in software.

1.8 Thesis Structure

The rest of this thesis is organized as follows. In chapter 2 we present a brief background on the filed and the related work. In chapter 3, we provide their requirement analysis for the framework and its abstract modeling. In chapter 4, we describe the framework architecture. In chapter 5, we provide information about the framework verification and we also present examples of important tasks during its use for allowing the reader to evaluate the API design. Finally, in chapter 6 we have the conclusion to this work and we also provide ideas for future work.

[1] Note that as a very high-level framework normally we prefer to model the mechanism to generate the such directives if possible

Chapter 2

Background and Related Work

2.1 Introduction

The implementation of a deep learning framework that generates hardware designs is a complex task that requires knowledge on many fields. This means a huge amount of literature to explore:

- Deep learning theory.
- Deep learning applications.
- Deep learning frameworks.
- Deep learning accelerators (GPGPUs, ASICs and FPGA-based accelerators).
- Hardware design flows.
- Hardware Description Languages.
- High Level Synthesis languages and methodologies.
- Field Programmable Gate Arrays (FPGAs), their architecture, capabilities and draw backs.

Enumerating and presenting all this work would be extraordinary and it is not in our purpose. Moreover, in this chapter we present some of the most important ideas in that relate with this thesis.

2.2 Deep Learning Background

The most substantial parts of the theory behind deep learning existed long before its emergence. A more classical approach to neural networks and is well presented in (1). Nonetheless, the book (2) is the most concise and current book about deep learning.

2.2.1 Stochastic Gradient Descent

Gradient descent (3) is a first order iterative optimization algorithm for approximating the solution to minimization problems. Its variation stochastic gradient descent is a variation is one of the most important optimization algorithms used in deep learning application. It was first used in ADALINE (4), a single layer artificial neural network. The learning rule of ADALINE was the LMS algorithm (4), which employees least mean square error as the cost function and stochastic gradient descent (**SGD**) as the optimization algorithm.

2.2.2 Back-Propagation

Back-propagation (5, 6, 7, 8, 9) can be seen as a way to generalize the LMS algorithm for multi-layer neural networks. It is a special case of automatic differentiation. Hence, as such, it is based mathematically on the chain-rule of calculus (10, 11). The back-propagation essentially is an optimized way to compute the gradients which required from optimization algorithms such as SGD to update the network parameters. A neural network is composed from several layers of computations. Each layer can be also seen as a composition of smaller elemental computations. Every elemental computation is matched with way to compute its derivatives in respect to its inputs. In back-propagation the chain rule is applied progressively in a backward manner, from the output layer of the network to the input. As an example consider the single dimension case $h(x) = f(g(r(x)))$, with $h, f, g, r \colon \mathbb{R} \to \mathbb{R}$. By applying the chain rule we obtain:

$$\frac{\partial h(x)}{\partial x} = \underbrace{\underbrace{\frac{\partial f(x)}{\partial x}\frac{\partial g(x)}{\partial x}}_{\text{Step 1.}}\frac{\partial r(x)}{\partial x}}_{\text{Step 2.}}.$$

So in this example with pack-propagation we will first compute the step 1 and then we will use the result to compute the step 2. Step 2 will employ the result of step 1 to compute the final gradient.

2.2.3 CNN Architectures

In (12), authors, among other, present the key differences of the most important deep convolutional neural network architectures. An even more synoptic list is the following:

- LeNet (1998) (13).
- AlexNet (2012) (14).
- VGGNet (2014) (15).
- GoogLeNet (2014) (16).

- ResNet (2015) (17).

The most easily noticed evolution is the increase of network depth. From our point of view, we also note that initially CNN architectures consisted of homogenous layers, a single computation. However, while they evolve progressively their layers tend become more heterogenous. This may be a hint that, at some point in the future, the layers of CNN architectures may become somewhat more complex. Hence, they may require hardware with different programming model than GPUs to run them efficiently. Alternatively, the restrictions that rise from nature of the currently available hardware may limit us from taking advantage of more heterogenous layers. Furthermore, we observe that the state of the art architecture changes very fast. Thus, the reconfigurability of FPGAs could be proven very useful, particularly with the right combination of tools.

2.3 Deep Learning Frameworks

A huge variety of frameworks that aid deep learning exists. Examples are TensorFlow (18), Torch (19), PyTorch (20), Caffe (21). These frameworks provide an API to let someone define computational graphs and enable the efficient execution of these in various hardware configurations, such as CPUs or GPGPUs. At the same time, deep learning frameworks contain automatic differentiation packages to allow conveniently support neural network training out of the box. Furthermore, most of them contain useful visualization packages and other utilities.

While all these frameworks are trying to solve the same problem in a very similar way, they have different priorities. This can result in a great difference on the features supported by each of them. Nevertheless, we aim to create a new API from the beginning, similar with other deep learning tools, but closer to the hardware designer. In FPGA design the program is mainly the hardware design itself. We do not want to generate or schedule the software to execute on a specific hardware design. Current deep learning frameworks have different design priorities and do not help in this direction. At the same time, we need a much more flexible solution in which we could change the tool in every aspect with hardware development characteristics in mind. Therefore, using one of the existing tools would not help at all. Because, these tools are very rigid to alter them in a considerable way.

2.4 Hardware Description Languages and FPGAs

The main goal of hardware description languages (**HDL**), such as Verilog and VHDL, is to model in detail digital logic circuits. These languages have a substantial role in FPGA design. However, software suites for FPGA design often support older subsets of the these languages. Moreover, in many cases tools

support different support subsets and may show a slightly different behavior. An interesting approach is MyHDL (22, 23), an open-source Python package that enables using Python as a hardware description and verification language that can generate VHDL or Verilog code. Nonetheless, the input format resembles itself an HDL language and thus we consider it as such. Furthermore, a large portion of the merits of using a high level scripting language to describe hardware are greatly diminished.

HDL languages differ from programming languages. One very important difference is that in HDLs, the designer explicitly describes the circuit behavior in time. An other is that in hardware design we mostly model the flow of data and control signals where programming languages are control-flow driven. The programing paradigm of deep learning frameworks, where the programmer provides a computational graph, is probably lying right in the middle between hardware description languages and software programming, from a programmers side. The notions of time and clock are removed but at the same time the computation is defined as a static construct: the computational graph that describes the neural network architecture, a construct that implicitly follows the dataflow paradigm.

2.5 High Level Synthesis Tools and FPGAs

Current HLS tools for FPGAs are able to generate very high-quality hardware from high level languages. Vivado HLS (24) can process C, C++ or SystemC source code and generate the hardware that implements the corresponding functionality. The tool undertakes the process of datapath and control extraction from the source code, while at the same time allows for easy interface configuration. The generated hardware can be tweaked by using special commands or pragmas. Vivado HLS supports tcl commands and scripts, which can be employed from external tools to instruct the tool. The quality of the result in many cases can be extremely good. The language is perfect for a low-level or intermediate abstraction of a design.

However, HLS programming is still very painful. The source code is too rigid and must follow a special programming style to really generate efficient hardware. It requires from the programmer to use the corresponding source language in a non-intuitive way, very different than using the same language for software design. Therefore, the language does not abstract the design well to serve as the top level of design.

Moreover, the reconfigurable nature of FPGAs require more research on the field. Vivado HLS, while it allows a much faster development cycle than traditional RTL tools, it does not really take advantage of the reconfigurability by itself. To change the hardware, you must substantially change the source, which could fast become a tedious process. As a consequence, it is nearly impossible to generate different working versions of the design in acceptable time. A new layer of very high-level tools that sits above current HLS solutions could help in the direction of better exploiting the capabilities of current HLS solutions.

2.6 Current Hardware

From our point of view, the most innovative architectures are the NVIDIA Volta GPUs (25), the Graphcore Colossus IPU (26) and the work of Wave Computing. The Volta is more than a GPU or a GPGPU. It incorporates a whole ASIC architecture to accelerate deep learning applications. Its most notable innovation is the Tensor Core, which supports mixed precision operations that have been found to be very efficient in terms of performance, area and power. Graphcore Colossus is an accelerator that is designed to hold all the model parameters on-chip (see 2.6.1). Lastly, Wave Computing is a company that designs Coarse Grained Reconfigurable Arrays (CGRAs) especially designed for deep learning applications. The neural network is fully mapped into hardware resources like FPGAs, but the architecture is much coarser and it is designed especially for such applications. We revisit the Graphcore's approach in subsection 2.6.1. In subsection 2.6.2 we talk about the current state of FPGAs.

2.6.1 Graphcore

Graphcore (26) is a semiconductor company founded in 2016 by Nigel Toon and Simon Knowles. The company develops massively parallel accelerators for machine learning. These machines - called Intelligent Processing Units (IPUs) - are optimized to execute computational graphs.

The researchers at Graphcore acknowledge some of the main characteristics of current deep learning algorithms:

1. The memory bandwidth is an important limiting factor in stateful neural network architectures and these architectures are the also the most promising for the future.

2. Computational graphs in deep learning applications are mostly static, allowing for compile-time communication optimizations.

3. The power budget of the IC is an important limiting factor that can be solved with by serializing very fast deterministic processing phases with deterministic communication phases. The determinism allows for deterministic scheduling during compile-time, and thus the power limits can be respected by saving time in processing communication.

The most important characteristics of an IPU are:

1. the complete absence of external memory and the innovative communication subsystem. The whole memory resides on chip and is distributed among thousands of processors. This means that the whole model resides in chip with a huge memory bandwidth available.

2. The communication system is not a crossbar, but instead, it is a software-controlled non-blocking system that allows for future scaling. It follows Bulk Synchronous Parallel model (27) and relies in the static nature of the graph that allows a compile-time scheduling.

3. An innovative software infrastructure framework which is even more important that the chip design itself. Poplar is a scalable graph programming framework that provides seamless interfacing to current and machine learning tools.

The first IPU to be realized is Colossus and its features will be probably these presented on table 2.1.

Process	16 nm
Number of Transistors	23.6 Bn
Chip-to-chip bandwidth	2.5TBps chip-to-ship bandwidth
Memory	300 MB on-chip
Memory Bandwidth	30TB/s
Host connection	PCIe Gen4.0 x16 with 31.5 GB/s
Cores	1216 independent IPU-Cores
Processing Performance	100 GFLOPS per IPU-Core
Parallelism	More than 7000 programs executing in parallel.
On-chip Communication	8TB/s non-blocking software controlled IPU-EXCHANGE

Table 2.1: The characteristics of Graphcore Colossus. Colossus is the the first IPU.

2.6.2 Current FPGAs

Current FPGA Architectures (28, 29) contain not only a huge amount of logic elements but also a lot of embedded memory resources. Specifically for Xilinx FPGAs, UltraRAM (30, 31) allows a good amount of in-device storage of neural network parameters (up to 500Mb of total on-chip memory). This capability seems very important for to enable the implementation of fast and low power accelerators. The cost to load the weights from an external memory is huge in area, power and performance. Thus, for deep learning applications local memory resources are probably much more important than the processing capacity of the device. Hence, this extraordinary availability of embedded memory is very important. Nevertheless, we expect to see FPGA architectural refinements due to deep neural network applications.

Recently, the trend of incorporating an FPGA and powerful processing Units on the same chip emerged significantly, a trend that also have great merits for deep learning applications. Intel designed a hybrid chip, the Intel Xeon Gold 6138P, consisting of a Intel Xeon processor and an Intel Arria 10 GX 1150 FPGA device (32). Xilinx recently designed Versal. Versal is described as an Adaptive Compute Acceleration Platform (**ACAP**) (33, 34, 35), a hybrid heterogenous compute platform that combines a scalar processor, an FPGA, and a vector processor, connected all together with a network-on-chip (**NoC**) via a

memory-mapped interface. The scalar engines comprise of a dual-core Arm Cortex-A72 Application Processing Unit (**APU**) and a dual-core Arm Cortex-R5 Real-Time Processing Unit (**RPU**). The on-chip memory resources of the programmable logic is up to 203.6 Mb of UltraRAM and 90.4 Mb of BlockRAM for the largest Versal device.

2.7 Related Tools

Maxeler Technologies (36) provides tools and infrastructure to generate hardware designs from low-level dataflow graphs and then utilize them. MaxCompiler (37, 38) is the software tool that generates the hardware. It employs an extended Java version called MaxJ that supports operator overloading and provides a set of libraries that the developer has to extend to describe the required functionality. Running the description generates the hardware design. The programming language and the API is standardized as OpenSPL (Open Spatial Programming Language) (39). MaxCompiler utilizes very similar concepts with deep learning frameworks.

Moreover, a deep learning framework is in its core strongly related with the MaxCompiler due to the fact that both rely on dataflow graphs and use similar mechanisms to describe these. However, they are vastly different. The main differences are four:

1. A deep learning framework contains also many advanced arithmetic features related with neural networks that MaxCompiler lacks as it is not designed especially for neural network applications. It is not specialized for neural network prototyping and consequently all the related features and libraries are missing. The most important lacking feature is the absence of an automatic differentiation package. For fast neural network prototyping these features are essential.

2. A deep learning framework generates software for already implemented hardware while the MaxCompiler generates hardware designs.

3. MaxCompiler is organized towards a closed proprietary scheme. Therefore, it is much more complicate and restrictive. It is very strictly defined, engineered to allow describing as much computational kernels as possible and to completely replace other HLS or HDL languages from the design flow. It is engineered to ideally keep everything except the kernel description hidden as a black-box.

4. Programming OpenSPL and MaxJ can be classified at a overwhelmingly lower-level compared to deep learning frameworks. In detail, deep learning frameworks are normally stacked above software design. Software design is at a higher-level than any HLS language and OpenSPL/MaxJ can be considered as one of the lowest-level HLS languages, much lower than Vivado HLS and just a bit above RTL design.

Nonetheless, a very high-level dataflow framework that complements the ecosystem and programmatically divides the designs process into smaller sub-problems that can then be solved in the most convenient way is needed. The high-level computational graph must be defined using the high-level API, but the building blocks of this computational graph are meant to be implemented in other more convenient languages, depending on the case. For example, if a portion of the design can be done more conveniently in a C++ based HLS language, the developer must have the flexibility to swiftly isolate this portion, solve it into this exact HLS language and finally employ it into the high-level graph.

Furthermore, Maxeler software suite while it is intented for FPGA-based infrastructure, it requires specific proprietary hardware and software, designed by Maxeler. This restricts the use cases of the resulting design. A tool with the ability to generate IP cores that can be employed in other FPGA-designs or generate designs that can run in a much broader range of hardware is required.

In (40) authors present LeFlow, a tool that generates hardware modules from TensorFlow models. Their tool takes the LLVM IR code emitted from the TensorFlow's XLA compiler and then, after applying some transformations, uses the LLVM back end of LegUp (41) to generate the Verilog design. While this design flow may be useful it some cases, it has five major drawbacks:

1. It completely ignores the need for a deep learning framework with an API especially designed for use with FPGAs. The problem from mapping such a resource demanding dataflow architecture into hardware is a very complex task. The preferable solution is dependent on many parameters and can vary a lot. As an example, consider the case of a single low-capacity FPGA versus an FPGA cloud infrastructure. We need to find ways to expose conveniently these implementation details through the API.

2. The design flow will unnecessarily overburden the HLS tool, which will result in inefficiencies. Taking a problem conveniently abstracted and formulated with graphs, translate it into LLVM IR assembly language optimized to be run as software, and then pass it to a hardware tool to generate hardware can clearly become problematic. Moreover, many important choices can be done more efficiently in the initial graph format.

3. The LLVM IR assembly language is too restrictive to be the first input that an HLS will encounter. We believe that synthesizable C++ is to be preferred for serving as input language to an HLS tool.

4. The LLVM IR assembly language is not user friendly. An intermediate format such as synthesizable C++ would be a much better solution.

5. To implement this design flow you need access to the internals of the HLS tool.

In (42), authors present a tool called **hls4ml** that builds machine learning models to FPGA designs. The tool focuses on low latency and low power applications. On top of that, authors present a case study for neural network inference focusing on a classifier for jet substructure. The results seem very promising for the aforementioned for low latency and low power applications. Nevertheless, they address a very different problem than the one we are trying to cope with. We give high priority in the modeling of a deep learning problem, the productivity of the hardware design process and the corresponding functionality of the tool itself, while they address an optimization problem for a specific use case that is already modeled in an other deep learning package. In (43), authors compare several other tools that build machine learning models and generate FPGA designs. Regardless the actual achieved performance these tools and the importance of their contributions, we note the following:

- These tools are not complete High-Level Synthesis tools, as opposing with our vision. They have no modeling capacity except synthesis parameters. With such low modeling capabilities the reconfigurability of FPGAs will never be exploited effectively.
- Utilize rigid and shallow internal tool-flows tied with specific architectures and choices, essentially optimizing specific applications.
- Hardcoded logic means inherently significant advantages over ASICs as well.
- They do not really take into account the actual design flow. It may seems at first that they present an alternative design flow where the neural network is translated automatically into a hardware module. However, the only sincere way to know which tool suits your application is more-or-less to perform trials. Today, FPGA resources are plentiful even on a single device and the most important factor for is the design productivity.
- The toughest work is implemented by external deep-learning frameworks. The neural networks are modeled, trained, optimized and exported from the selected deep learning framework. Only a small step is performed by the translation tool itself.
- These tools work for inference-only, whereas the training is inherently a much more difficult task to solve and a much more important.

Chapter 3

Requirement Analysis and Modeling

In this chapter, we analyze the problem, perform a concise requirement analysis and then model the problem. The requirement analysis is presented in section 3.1, and the foundational modeling is presented in section 3.2.

3.1 Requirement Analysis

In this section, we provide a concise requirement analysis on the framework design. We first formalize the problem, then note down all the developer classes that are going to interact with the framework, and finally we provide a synoptic list of their goals with a corresponding requirement description.

3.1.1 Problem Formulation

Current frameworks are designed for running deep learning applications on GPUs and other hardwired ASICs. On the other hand, FPGAs functionality is not predefined, they have a huge range of supported hardware functionality and there is a need to design new frameworks that take this into account. Further research is also required into this direction.

The main objective of this thesis is to develop a new deep learning framework that generates FPGA-based hardware designs. The generated code is currently HLS C++, but in the future it may be extended to generate block-based designs or HDL code if it is required to. It is essentially a special HLS tool and may be employed for other tasks too, especially for numeric computations.

The framework must incorporate modules. Modules are entities that can be translated into hardware IP. Trainable modules are a special subclass of modules that automatically generate the hardware required to train any trainable parameters. Each module (or trainable module) must be able to formalize a computation with a collection of computational graphs. The building-block for these graphs are the Ops. An Op has a predefined hardware implementation that can be instantiated into the generated code. Loss functions are compu-

tational elements employed to compute the error between the prediction and the true label of the training sample.

3.1.2 Developer Classes

We can divide the developers that are going to interact with the framework into three classes:

1. Framework hardware developers.

2. Framework software developers.

3. External engineers.

Framework developers, try to develop and extend the framework. These can be further divided into framework hardware developers and framework software developers. The first try to design new operator implementations or to find new ways to improve the generated hardware. The second try to improve the API or add support for new tools, languages, etc. External engineers (engineers that use the framework), will use the framework to build new AI applications or research on new deep learning neural network architectures.

3.1.3 Short Term User Goals

For each of the aforementioned classes of users, presented in subsection 3.1.2, we can enumerate several goals. Not all goals can be formulated enough to be considered in detail. Above all, framework developers are expected to extend the framework in creative ways. These cases cannot formulated into an analysis.

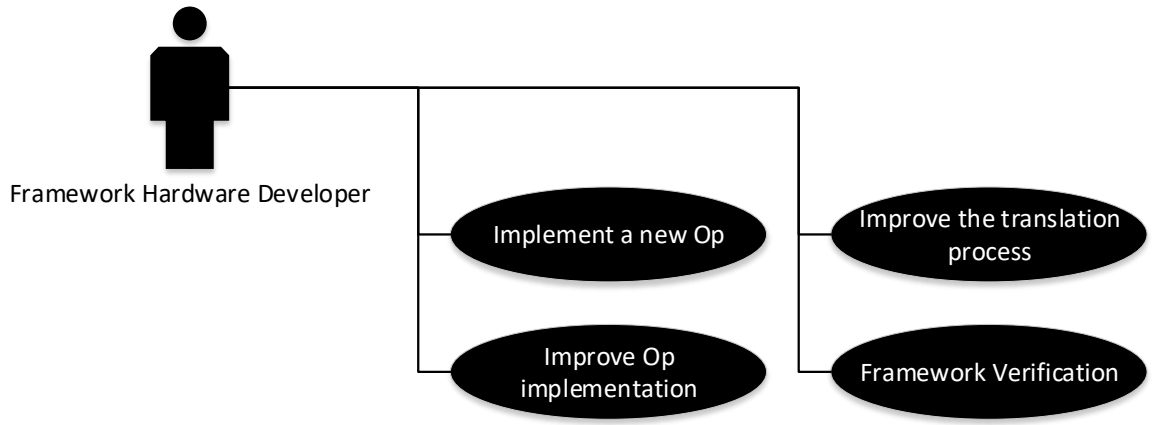

Figure 3.1: Short-term user goals of framework hardware developers.

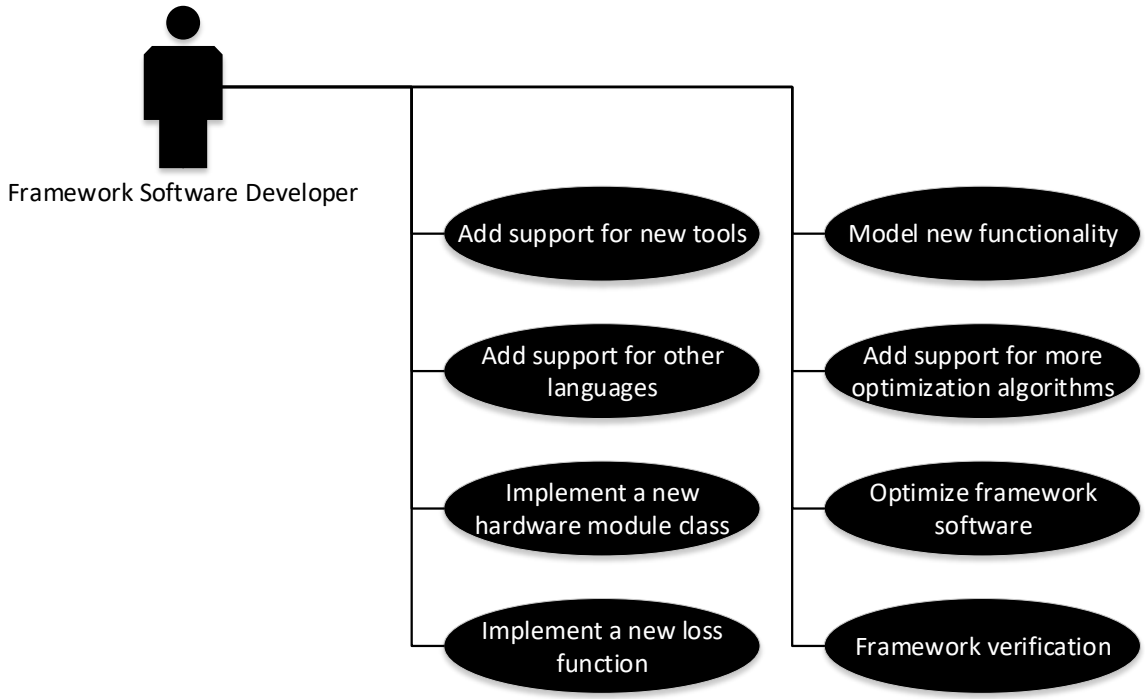

Figure 3.2: Short-term user goals of framework software developers.

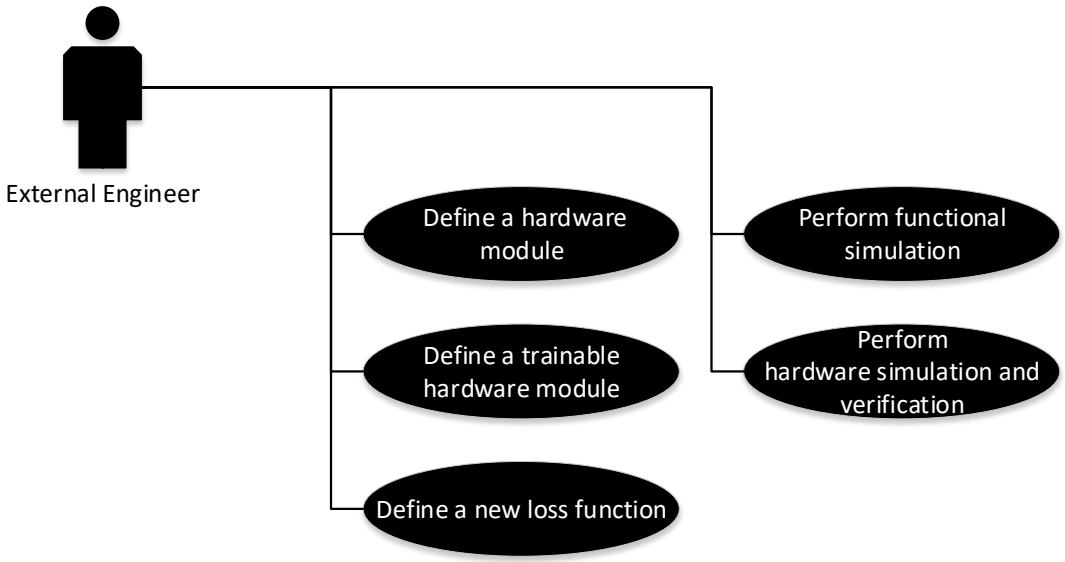

Figure 3.3: Short-term user goals of external engineers.

3.1.4 User Goal Description

In this subsection we provide a requirement description of the goals presented in subsection 3.1.3.

Add support for new tools

It must be easy to extend the framework to support new tools or adapt to tool changes. First, we will initially support only Xilinx Vivado HLS and we need to extend the framework to to support other tools and FPGA of different vendors too. Second, tools can be replaced or updated considerably and the framework must be able to easily adapt, and support these too.

Add support for new output languages

HLS languages are not standardized. Different tools support their own different languages that change between different tool versions. Furthermore, even for HDL languages, which are standardized, are not fully supported. Different tools support different subsets and different versions of these standards. Hence, it is crucial to be able to support new languages.

Add a new loss function

It must be convenient to add new loss functions. A loss function is a function with a special role during the training process. We want to implement only a subset of the framework in the beginning, and hence, we may initially have only one loss function. Therefore, we need to be possible for developers to easily extend it with more.

Model new functionality

In many cases, the framework is going to be extended in unpredictably. Therefore, in these cases, the developers will extend the framework to model new problems, or to model the same problems from different viewpoints. We need a software architecture and a programming language that will allow such modifications.

Add support for new optimization algorithms

At the beginning, we are going to support only SGD. Thus, It must be feasible to add new optimization algorithms effortlessly.

Framework verification

Framework developers will have to verify the framework. Particularly, after implementing new Ops or altering the old ones. We need ways to aid this process. First, we must be able to run the computational graph in CPU resources. Second, we must be able to use the computational graph to generate test data for the hardware verification process.

Implement new Op

Framework hardware developers have to implement new Ops (Operators). Ops are essentially hardware that can be instantiated directly in the generated code. Adding new Ops must be as simple and intuitive as it is possible.

Improve Op implementation

Framework hardware developers will occasionally update Op implementations. However, this process is a small subset of the goal "Implement new Op".

Improve the translation process

Framework hardware developers may find ways to generate better hardware. For example a better control or different IO protocols. The tool must model these, so that we could alter them without changing the framework mechanics.

Define a hardware module

External engineers will usually have to describe at least one hardware module during the design process. Each hardware module implements a set of computational graphs. The input code must follow the paradigm of deep learning frameworks. The control logic of the module must mbe modeled too, to enable easy modifications. Nevertheless, the control logic must be kept as much as hidden it is possible to keep the input code simple.

Define a trainable hardware module

Trainable modules must extend modules to also implement the following:

- Extend the API to easily declare trainable parameters.
- Generate backward graphs automatically that implement back-propagation.
- Include optimization algorithms such as SGD.
- Extend the control to enable initialization and read access to the values of the trainable parameters.

Perform functional simulation

External engineers may occasionally want to easily perform functional simulation before generating the hardware module.

Perform generated hardware verification and simulation

We need mechanisms to allow engineers to smoothly verify the generated hardware, or perform more detailed simulations to fix possible problems or unwanted behaviors.

3.1.5 Performance Indices

We want to prioritize and optimize several development tasks or use cases. For this work, we do not aim to optimize the generated hardware. Instead, these use cases constitute our performance index. These prioritized use cases are presented in the next subsection (subsection 3.1.6).

3.1.6 Design Priorities

When designing new software it is crucial to enumerate all the important use cases and prioritize these that fit better to the target philosophy. The framework implementation is currently optimized primarily for two use cases:

1. New module definition. New module definition is the most important use case. Defining a new hardware module must be very simple and intuitive, otherwise the whole idea will collapse.

2. Extend the framework to add support for a new operator. The second most important use case is to extend the framework with a new operator. Having a rigid software that it is too hard to extend will not seriously help the community in the long term.

3.2 Foundational Modeling

In this section, we present the foundational modeling upon which the framework is build. Furthermore, we describe all fundamental concepts, while in the end, we present a abstract object model of the framework with these.

3.2.1 Modeling Process

Figure 3.4 presents the modeling process in a deep learning framework that generates hardware for FPGA-based designs. As shown in the figure, the framework breaks the problem into three modeling processes. On the top, developers model deep learning architectures to to solve specific problems. On the bottom, we have the fundamental hardware process, where developers model elementary hardware building-blocks and define hardware implementation paradigms. Finally, on the center we have the core modeling process where APIs are provided to allow the other two modeling process to proceed.

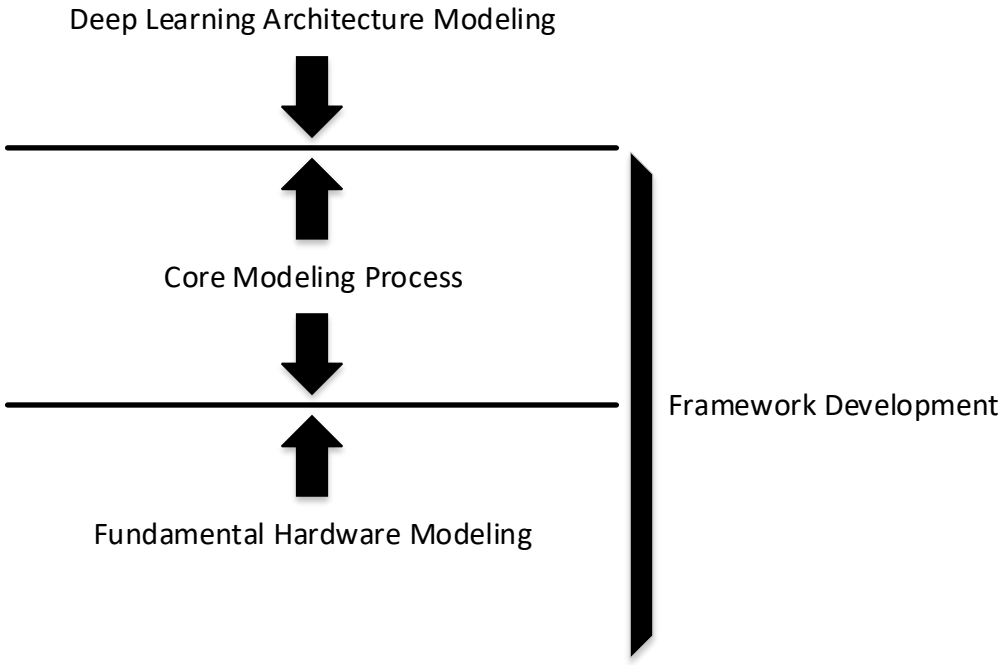

Figure 3.4: The deep learning modeling process for FPGA-based design.

3.2.2 Modules and Trainable Modules

Modules are collections of graphs that will be translated into a single hardware module. Control logic is described with a special graph, the control graph and every module class has a predefined control graph to simplify the design process. Arithmetic computations are described with a set of computational graphs. Trainable modules extend modules to also generate the hardware to train any trainable parameter. Furthermore, additional methods are implemented to automatically generate the hardware that trains these parameters. The extra computational requirements are implemented through generating additional computational graphs before translating them all into hardware resources. The control graph is slightly altered to allow the training hardware to be executed optionally.

3.2.3 Computational Graphs

For the context of this work, a computational graph is a directed acyclic graph or simply **DAG**, that describes how module outputs can be computed from a set of parameters by modeling all data dependencies and computations from these parameters to the corresponding output. Moreover, this DAG can be used to easily create valid execution plans and can be mapped into hardware. These graphs inherently can be grouped into sets of graphs to form bigger graphs that share no dependencies and could be executed simultaneously in the same period of time. To share data between different periods the only way is store the results either externally by taking the output of the module,

or internally by using stateful parameters that cache signal values for later use. The cached values can be only accessed in successive periods.

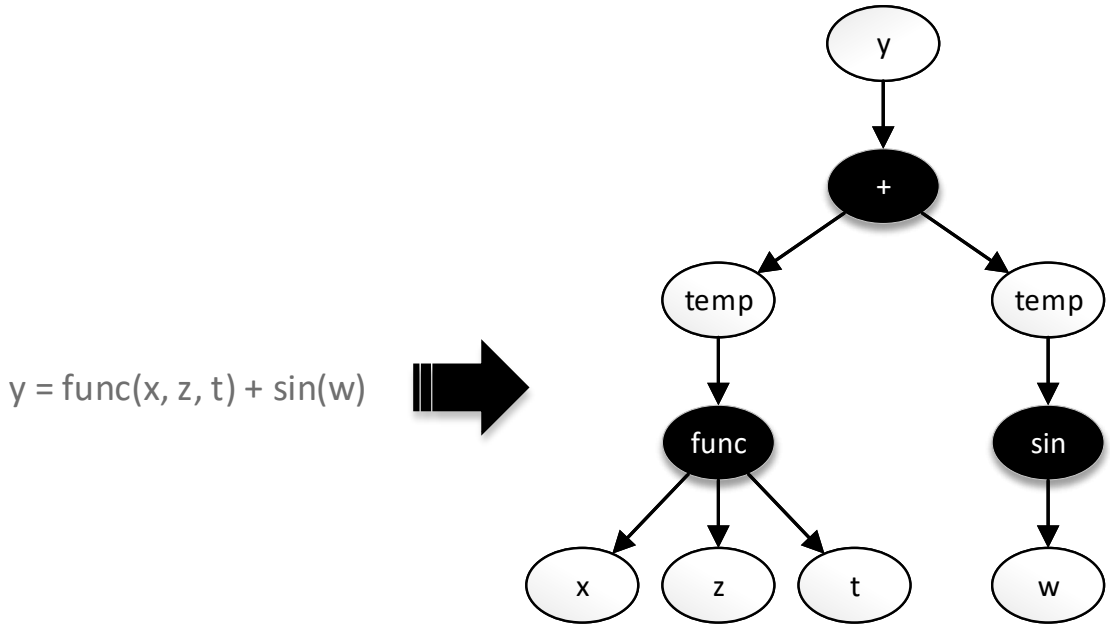

Figure 3.5: Example computational graph.

3.2.4 Forward and Backward Computational Graphs

Simple modules can be used to create hardware modules from computational graphs, for example to implement an inference-only neural network system. However, we may want to generate modules that can be trained using hardware resources. To achieve this the architecture of the neural network is processed to generate the training logic. Furthermore, the "forward" computational graph that describes the architecture is used to generate the "backward" computational graph that updates the network's parameters. These are then translated into hardware resources.

3.2.5 Operators

Operators or Ops are comprised of two parts: the actual hardware implementation, and the python class that holds the required metadata required from the framework to employ this implementation. The implementation can be an external source or a code generator.

3.2.6 Wires

The basic variable type in the framework is represented by the Wire class. We have selected this name because we usually use Wires in the framework to describe and generate hardware entities that are conceptually close with wires.

For example, most of the direct Wire instantiations are needed when interfacing hardware modules and Operators and hardware developers tend to imagine such interfacing entities as the pins and the corresponding connecting wires of integrated circuit.

As shown in figure 3.6, Wires can be categorized according to what entity are modeling. We have three types of interface wires: Inputs, Outputs and Buses (bidirectional). In addition, we have Memories, Wires that represent accessible memory resources and Free Wires, which are represent memory elements that store temporary data that flow between processing elements. Interface Wires translate into interface definitions and may not utilize local device resources. On the other hand, Memories always utilize the corresponding local device memory. Free (or Simple) Wires may utilize the corresponding memory resources, but the framework may is allowed to optimize them and may get completely removed. Interface Wires and Free Wires can be marked as Streams or TempRAM (Temporary Random Access Memories). Streams employ FIFOs to cache any intermediate results. TempRAM will always utilize local resources to hold all the data. For streams the framework is allowed to optimize the size of their FIFOs[1]. Only Memories are guaranteed to hold their values across different periods. In addition, Trainable Modules we can have trainable Memories. Only trainable modules can have trainable signals. For these the framework generates the hardware that allows to train them.

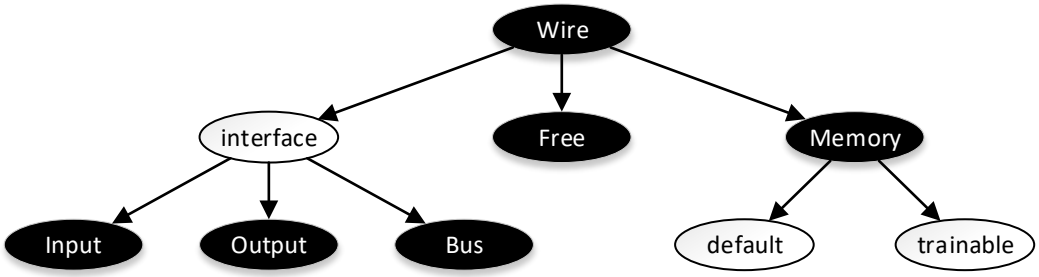

Figure 3.6: Wire categorization.

3.2.7 Conditionals

Conditionals are special nodes in the computational graphs that allow data to be processed differently depending on a specified condition. Figure 3.7 shows a graph representation of an *if-else* conditional. Wire y cat be assigned with the value of y_t or y_f depending on the condition c. Conditionals can be seen either as hardware that executes optionally and can translated differently based on the requirements and the capabilities of the target language. In an eager translation the conditional c and the two outcomes y_t or y_f could be computed in parallel and then select the correct y_t or y_f according to c. In a more relaxed scheme, we could compute c, then compute one of y_t, y_f

[1] In our current implementation we do not optimize the FIFO size.

according to c and assign the result into y. In all cases, the condition, which is also a computational graph, is computed first to decide how the processing will proceed.

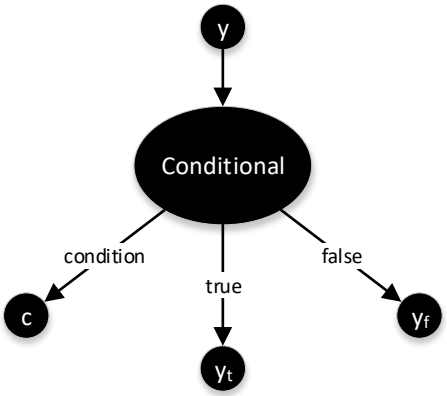

Figure 3.7: A Conditional node.

3.2.8 Commands and Execution Periods

The generated system receives commands and handles execution accordingly. Commands may span across multiple execution periods. An execution period is a slot in the scheduling scheme where a computational graph can run. Specific computational graphs are tied with specific periods to implement the desired functionality.

3.2.9 Control Graphs

Control graphs can be seen as a higher level execution plan above computational graphs that do not delve into how a specific computational graph will be executed, but instead, how to handle the synchronization between different periods, or specific auxiliary tasks. They also describe how to serve the incoming commands. It is a way to model the behavior of the module in a way that can be altered and allow module subclasses to modify or extend this behavior. Therefore, it serves the role of allowing other module classes to be build upon other more conveniently. Control graphs are composed of control nodes and control tasks. Control graphs top priority when designing new functionality for control graphs is the easiness of translation. Control graphs are designed primarily for internal tool use. Hence, normally they are not instantiated by the end designer (external framework designer).

3.2.10 Control Nodes

Control nodes are simple directives that describe the control flow that the hardware module will follow. Usually they "resemble" well-known control state-

ments, such as Case statements, or If statements.

3.2.11 Control Tasks

Many control nodes require a control task. These control nodes execute special methods of the control tasks to achieve various objectives, such as generating the logic to initialize a list of parameters or get and return their current values.

3.2.12 Graph Factories and Op Factories

Graph factories are special functions that generate some kind of computational or control graph. They are very important as many of they employed for implementing many important mechanisms. Op factories are graph factories that instantiate Ops. Their role is to simplify the Op instantiation process. Without these instantiating Ops would not resemble normal function calls and using Ops would be much less intuitive and more confusing.

3.2.13 Linker Nodes

Linker nodes allow computational graphs to employ control graphs. Like control graphs, linker nodes are normally not instantiated by the end designer (external framework designer). Up to now, Linker Nodes are not employed in our final implementation.

3.2.14 Submodules

Submodules are secondary modules that do not generate their own unit of code and usually serve as primitive graph nodes, or utility elements to mutate a graph node. Support Subsystems implement functionality for simulations, tool support etc. Supmodules are divided into control, functional and task submodules. With control and task submodules we define control graphs. Task submodules modify the functionality of control submodules. Functional submodules define computational graphs together with Wires.

Chapter 4

System Architecture and Implementation

In this chapter we present the proposed tool stack, the corresponding design flow and the implementation of the framework.

4.1 Proposed Design Flow

Figure 4.1 presents the proposed design flow. Three new stages were adhered to the HLS design flow:

1. **Design Entry**
   The process begins with a Python script. The code somewhat resembles that of TensorFlow/PyTorch. It provides a high-level functional description to the tool. This description will be used later to generate a hardware module.

2. **Python Execution**
   The design is tested with a Python testbench. If everything is confirmed that works as intended we can proceed to the next step, otherwise the designer must fix the provided Python design.

3. **Python Synthesis**
   An HLS C++ project that contains the required C++ libraries, the automatically generate C++ modules, and the generated C++ testbench. The testbench corresponds to the Python testbench and can be used to ensure that the generated hardware module continuous to function correctly.

The next steps remain the same as before. However, normally the designer will not modify the C++ code directly. Instead he/she will modify the python high-level description.

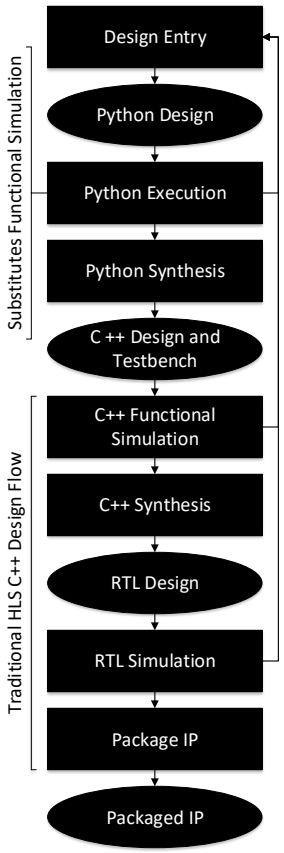

Figure 4.1: The proposed design flow. Traditional design flow is inherited but in the future is intended to become completely optional.

4.2 Proposed Tool Stack

The proposed tool stack is shown in figure 4.2. The designer describes the required functionality in high level Python code. This code is then translated into HLS C++ and an HLS project is generated (automatically). The generated project is further processed by an HLS tool. Finally, a "traditional" RTL tool is needed to synthesize the design and generate the bitstream to program the FPGA.

4.2.1 Why Python

We selected Python for the following reasons:

1. It is a simple, intuitive, and very well known scripting language.

2. Most of the current deep learning frameworks are implemented in Python or provide a Python API.

3. It supports object oriented programming and operator overloading.

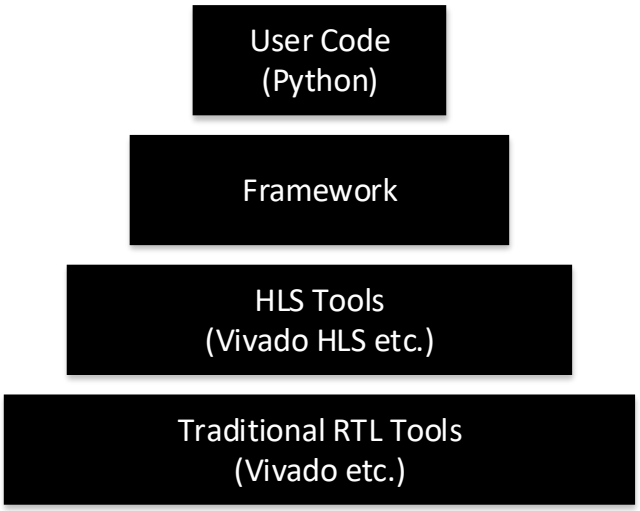

Figure 4.2: The proposed tool stack.

4. The inspection package allows to get references to local variables and this can be helpful for achieving some advanced features in some occasions.

5. It is easy to extend and maintain a Python project.

6. It is very easy to design a prototype to implement a new idea.

4.2.2 Used Schemes of Code Generation

The framework employees three different types of code generation methodologies for different occasions. First, we can just extract special library code as it is. Second, we employ templates, either by hard-coding a template into Python or by using a special template file and using it as a guide. Third, we generate code from by translating computational graphs.

4.2.3 HLS C++ and Jinja 2

The hardware implementation of the Operators is implemented in HLS C++ for Vivado. Some special operators employ Jinja 2 (44). Jinja 2 is a templating language for Python. It is a very flexible and fast Python library that allows a designer to generate text from parametric text templates. So far, we use it to dynamically generate C++ files. Moreover, some Operators need a more flexible way to get described properly that HLS C++ supports, as Vivado HLS requires many statically defined structures. An example is shown in snippets 4.1, 4.2, and 4.3. First, we write a template file. Then we can call it through Python with the parameter values depending the current case. This will generate the code, which can be stored into a distinct file later.

Code Snippet 4.1 Jinja 2 Template (example.jinja).

```
{% for i in range(num_of_outputs) %}
hls::stream<typename CFG::array_t> &output_{{i}},
{% endfor %}
```

Code Snippet 4.2 Python Code to call.

```
from jinja2 import Template

str_template = open('example.jinja').read()
template = Template(str_template, trim_blocks=True, lstrip_blocks=True)
result_text = template.render(num_of_outputs=3)
```

4.2.4 The Trade-off

During the implementation of this framework, one trade-off is almost ubiquitous: Utilization of long (software) pipelines of simple sub-tasks versus utilization of short pipelines of complex sub-tasks. Initially, the first prototypes of the framework were employing methodologies and graph abstractions that resulted on short software pipelines of complex tasks. However, we shifted to a scheme of long pipelines because there easier to conceive, maintain and extend. Moreover, while short pipelines have several advantages such as better processing speed of graphs, it is easy for the code to become that complex that is hard or even impossible to maintain. In conclusion, we have a strong preference of long-pipelines.

4.3 Framework Architecture

Figure 4.3 presents an abstract view of the architecture. The framework is implemented in Python and HLS C++. Python for the core functionality and HLS C++ for the synthesizable libraries and the Python execution libraries. Moreover, the code can be sorted into one or more of the following overlapping logical groups depending the functionality that actually implements:

- Neural network modeling support.
- Operator modeling support.
- Implemented operator models.
- Translation logic.
- Python Execution support.

Code Snippet 4.3 Generated C++ Code.

```
hls::stream<typename CFG::array_t> &output_0,
hls::stream<typename CFG::array_t> &output_1,
hls::stream<typename CFG::array_t> &output_2,
```

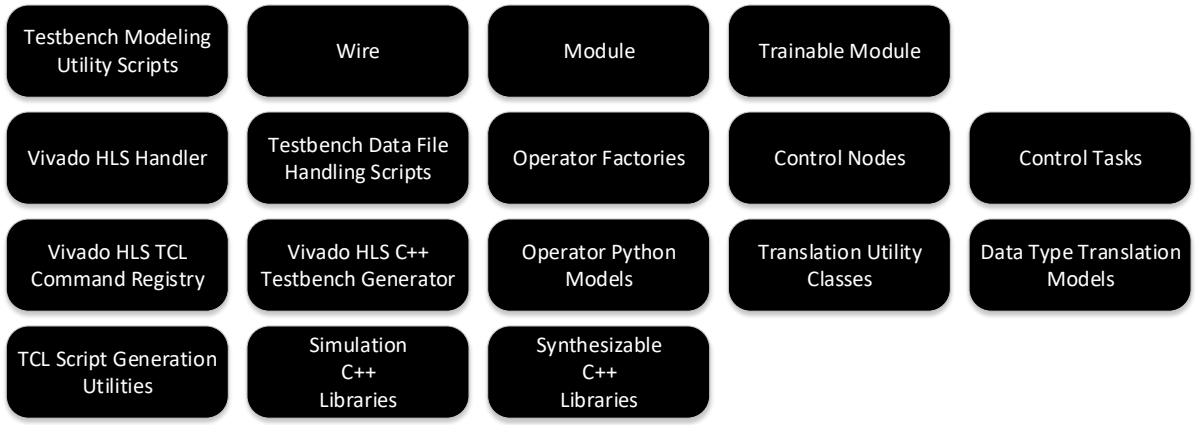

Figure 4.3: Framework Architecture.

- Vivado HLS support.
- Vivado HLS C++ language support.

These subsystems in many cases overlap especially in the Python code. During implementation the initial concepts have been refined.

In this section we describe the refined concepts presented in the chapter 3. Most concept entities are implemented with a corresponding class. However, some did not or were implemented using multiple classes. Furthermore, in addition to the previously modeled concepts we have new functionality, such as the Vivado HLS support subsystem. Hence, to the refined concepts we adhere the the description of this functionality.

The most important classes of the framework are: Wires, Modules and Trainable Modules. Wires serve as the variable type for code that defines computational graphs. Modules are classes that contain one or more computational graphs and generate a hardware module. Trainable Modules subclass Modules and generate a hardware module that is able to learn from new training data, using the back-propagation and SGD algorithms.

4.3.1 Wires

Wire is one of the most essential classes of the framework. It is the basic variable type and every operator is defined to have one or more wires as inputs and one wire as output.

Four types of Wires exist:

1. *Input* Wires serve as module inputs.
2. *Output* Wires serve as module outputs.
3. *Bus* (Bidirectional) Wires that can be used as both inputs and outputs.
4. *Memories* are all internal Wires that have a state.
5. *Free or Simple* are all internal Wires that are not Memories.

Type	**Description**
Input	A Wire that serves as a Module input.
Output	A Wire that serves as a Module output.
Bus (Bidirectional)	A Wire that serves as both, a Module input and an output.
Memory	Internal stateful Wire.
Simple or Free	Internal Wire (normally not stateful but it could generate stateful logic if required).

Table 4.1: Wire types.

Developers will normally declare Wires of Input, Output and Memory types. Bus Wires are only generated internally by the tool for tasks that require bidirectional interaction with external memory. Currently, in all module types each Memory Wire is tied with a Bus to load and save its state from an external memory.

Except connecting operators, Wires serve mostly as a data structure to hold properties of data at a specific point in the computational graph. Such properties are the shape, the data type and the number of dimensions. The most important properties and their description is presented in table 4.2.

Each wire has a data type (dtype). For normal data types, it fairly easy to define new data types. There is a special data type named *stream* which translates into *hls::stream*. Non-stream Wires translate to either scalar C++ variables or C++ arrays. Wires may have arbitrary dimensions but we concluded that the generated variable in the HLS C++ code must allowed to be only one of these types. In the case of multiple dimensions must be translated into a C++ array, this array has again only a single dimension, but it is large enough to hold the data from all its initial dimensions.

Table 4.3 presents the most important methods of a wire. The constructor is employed when a new wire is created. The properties of the wire such as the data type of the handled data (dtype), its name, or its shape can be provided as input arguments to the constructor. By providing the init value the newly created wire will considered as stateful. This means that it will be mapped into local device memory resources in the generated hardware logic. The recursive method *run* runs the computational graph that is assigned to this wire and

Property	Description
name	The name of the Wire. For Wires without a name the framework generates one automatically.
p	Pointer to another Wire or an Operator that will generate its value during execution. For Memory Wires it must be left to its default value which is None.
input	For Memory Wires only. Pointer to another Wire or an Operator that will generate its value during execution. For non-memory Wires it must be left to its default value which is None.
wireType	The type of the Wire (see table 4.1).
dtype	The data type.
value	The value evaluating the sub-graph *Wire.p* during execution of the graph on Python.
shape	The shape of the data. It is a Python tuple like NumPy, for example (10,20).
size	The total number of elements in the data structure (e.g. if the shape is (10,20) size is $10 \times 20 = 200$).
init	The data to init a Memory.
ndim	The number of dimensions.

Table 4.2: The most important properties of a Wire.

sets the property "value" with the computed result. All inputs of the computational graph must have their "value" properties initialized before execution. Many Python operators, such as *__add__* and *__sub__* are overloaded for convenience.

Furthermore, wires have some functions related with the translation to HLS C++ code specifically. The *hls_pragmas* registers HLS pragmas that will be utilized in later stages. The *get_hls_decl* returns the internal set of all registered pragmas. Moreover, *hls* is employed to translate the computational graph that this wire points to.

Initially, the wire class API was far more complex. For example, it contained two methods *get_effective_wire* and *get_effective_name* which were employed for cases where we have chains of wires in the computational graph to ensure that all connected wires in the chain are translated into the same variable in the generated code. Nevertheless, these methods have been removed and the API has been simplified in a step towards a scheme that implements different framework tasks using long software pipelines of simple and concrete sub-tasks.

Method	Description
constructor	Instantiates a new wire of the specified by the arguments properties.
eval	Runs the computational graph that this wire points to. Assigns the property named "value" to the result. Also, returns the result.
is_scalar	This method returns True if the wire has a size of 1.
overloaded operators	Special Python methods that overload operators are implemented too for convenient framework interaction.
hls_pragmas	This method registers HLS pragmas to this wire.
get_hls_pragmas	This method returns the set with all the registered to this wire HLS pragmas.
get_hls_decl	Intended for internal framework use. Generates a C++ declaration string that can be utilized directly inside the main function body.
get_hls_arg_decl	Intended for internal framework use. Generates a C++ argument declaration string that can be utilized directly in function headers.
hls	Intended for internal framework use. It initiates or proceeds the translation process of the computational graph that corresponds to this wire. Works recursively.

Table 4.3: The most important methods of a Wire.

4.3.2 Wire Configuration

Wire configuration is a special framework struct in C++ that describes the characteristics of the corresponding Wire. Hence, all wires have a wire configuration. Its schematic representation is shown in figure 4.4. Synthesizable code for Vivado HLS C++ must contain any performance critical information in a static form. Due to the limitation of the Vivado HLS C++ language this struct must be static. Nevertheless, this has other implications too. Most matrix and tensor operations require knowledge about the exact shape of the inputs. Thus, we needed to include to each wire configuration static information, about the dimensions of each wire. We concluded that the best way (if not the only one) was to generate multiple static constants and employ in the code a set of custom C++ preprocessor commands. The framework generates one struct for each wire and places it in the beginning of the generated source file. For

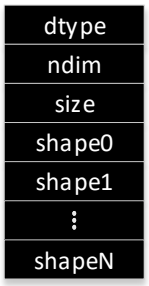

Figure 4.4: Wire configuration C++ struct.

convenience there is a predefined base template struct, that the framework extends it. Its definition is presented in snippet 4.4 and an example of the wire configuration that extends it is shown in snippet 4.5.

Code Snippet 4.4 Wire configuration HLS C++ base struct.

```
template <
  typename DataType,
  unsigned int NDim,
  unsigned int Size
>
struct WireConfiguration
{
  typedef DataType dtype;

  static const unsigned int ndim = NDim;
  static const unsigned int size = Size;
};
```

Code Snippet 4.5 Wire configuration definition example.

```
#include "wire.h"

struct MyWire_conf : WireConfiguration<float, 2, 50> {
  static const unsigned int shape0 = 5;
  static const unsigned int shape1 = 10;
};
```

4.3.3 Operators

Operators or Ops normally are comprised of two parts: the HLS C++ implementation, and the Python class that holds the metadata about it.

The HLS C++ implementation is a template class with several methods. These methods implement the hardware part of the Op. The Python code can select which method to use depending on the current characteristics of its inputs. Furthermore, the Python Op class is essentially a mix of three things:

1. Constraints on the input characteristics.

2. An abstract behavioral model, for efficient behavioral emulation (currently NumPy). This can be employed for executing the graph on Python.

3. A graph factory that models the corresponding backward graph for this node.

4. Several metadata about the Op implementation, such as the name of the header file that contains the template implementation of the Op.

The capability of easily defining and incorporating new Operators was on our top priorities. The relative diagram is shown in figure 4.5. The process be-

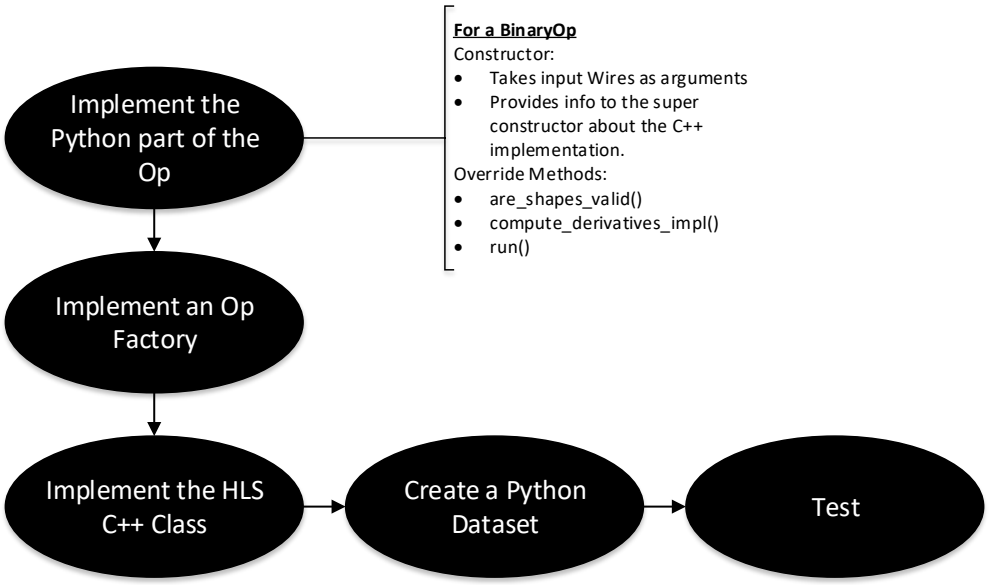

Figure 4.5: Use Case: New Op.

gins by implementing a Python class to provide metadata, derivative graphs, and execution emulation methods to the framework. Secondly, an Op factory function is required.This function simply instantiates the Op, generates an output Wire of the correct type and shape that points to this instance, and returns it. Thirdly, an HLS C++ header file must be created to implement the Op hardware. Finally, a testbench is defined and the Op is evaluated.

For convenience we have implemented the base operator classes shown in figure 4.6.

HLS C++ implementations of Operators are dominated from two facts:

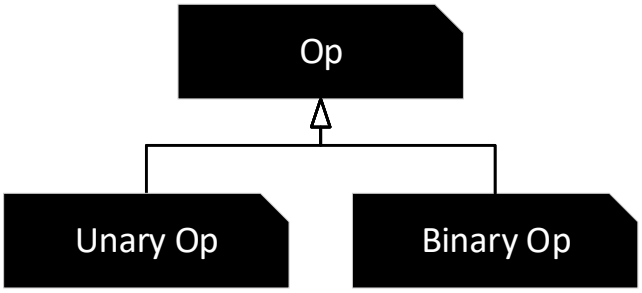

Figure 4.6: Operator classes.

- Different implementations depending on characteristics of its arguments, such as the number of dimensions or the data type. This is because HLS C++ is overwhelmingly static, while C++ is normally designed as a dynamic language, which makes impossible for the static-natured generated design to conveniently handle these characteristics. However, we have some cases, where the characteristics can be expressed in static way so that the HLS C++ compiler can statically infer them. For example, methods that handle stream arguments and array arguments can be correctly handled from the C++ compiler.

- Usually C++ implementations of Operators receive one Wire Configuration template argument for each Wire argument.

4.3.4 Operator Factories

Op (Operator) Factories are a special case of graph factories that help with the instantiation of new Ops and provide an easy-to-use interface for final developers. Snippet 4.6 shows the template pseudocode of a typical Op Factory. Op factory implementations currently receive two types of input arguments,

Code Snippet 4.6 Op pseudocode template.

```
1. Create the return Wire.
2. Instantiate the Op,
   Pass all input wires to the Op constructor.
3. Connect the Wire to the Op.
4. Return the Wire.
```

wires and static arithmetic values. The first step in every Factory that instantiates new Ops directly is to decide the characteristics of the result and create the corresponding Wire. Afterwards it has to instantiate the Op and finally return the result WIre. The only result is the return value, which is always a wire that points on the generated computational graph.

Most op factories are defined in "functions.py". However, loss functions, which are also op factories, are defined in "losses.py". Furthermore, these two Python modules can contain graph factories that are not op factories. Instead, it could contain graph factories that employ different ops depending on the characteristics of the provided arguments. In addition, it can utilize other op Factories or even generate and connect multiple ops, to implement a required functionality. This means that these graph factories can undertake an early optimization step if required or implement a new functionality by combining ops and other op factories.

4.3.5 Refined Control Graphs

Computational graphs can describe very complicate arithmetic operations, however there we need a way to manipulate the execution flow too. Even for the deep learning, where the computation is usually structured in only one or two rigid computational graphs (the forward and the backward graph), some control logic is required (for example to load/save the weights or initiate the processing).

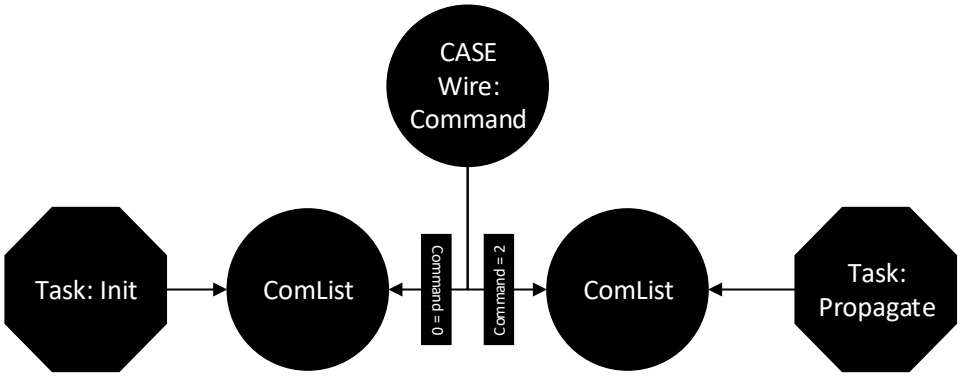

Figure 4.7: Control graph example.

Figure 4.7 presents a very simple control graph. The root is a Case node that has two ComList children nodes. Upon execution, a wire named "command" selects which one to execute. Each ComList is tied with one task and a list of Wires. The Init task instructs the system to initialize the Wire state from the DRAM, while the Get task instructs it to save the current state of the Wire to the DRAM.

4.3.6 Modules

Modules are objects, the functionality of which is described by computational and control graph ane during the translation process will generate a hardware module. Currently, each one generates a distinct HLS C++ file to be synthesized. Furthermore, each module contains a predefined control graph and a set of user defined computational graphs. The control graph is required to allow support for different types of modules. It primarily describes how to use

the computational graphs and sketches the generated control. From the other side, computational graphs describe the arithmetic work. Computational and control graphs are not executed upon their definition. Instead they are utilized to generate a hardware module that will be able to perform the described computations under the guidance of the control graph.

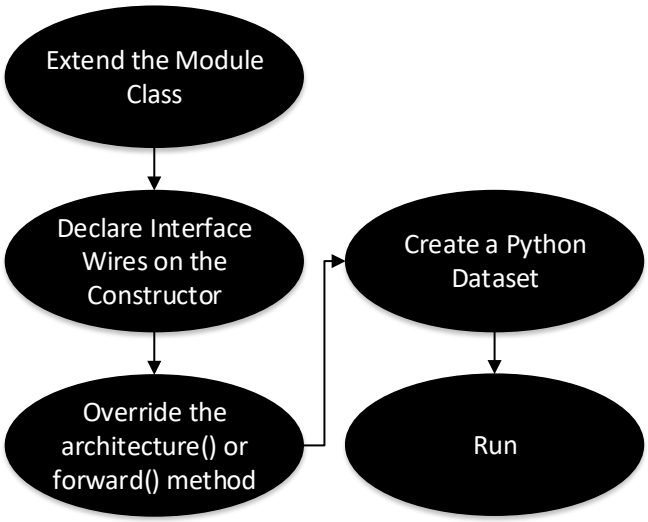

Figure 4.8: Use Case: New Module.

One of our top priorities was the ability to define easily new modules. The corresponding use case is shown in figure 4.8. Firstly, the developer extends the corresponding class. In the constructor, we must declare all wires belonging to the module's interface. Moreover, the *architecture()* method must be overridden. The role of this method is to define all the computational graphs. Finally, a Python testbench object is created in Python and the module can be tested.

Method	Description
constructor	Constructs the module and - to build the control graph - it calls *init_control*.
architecture	Constructs the computational graphs. It is abstract and hence the user must override it.
input	Declares a wire as input. IIf it is called with a string as argument it automatically grabs any local variable with that name.
output	Declares a wire as output. If it is called with a string as argument it automatically grabs any local variable with that name.
memory	Declares a memory.
eval	Runs one execution period (a single instruction to the module).
bus	Registers a wire as an bus. Intended for internal framework use currently.
get_local	Finds a variable by its name and returns it. Intended for internal framework use.

Table 4.4: The main API of the module.

Table 4.4 presents the most main API of the module. When defining a new Module the only non-computational methods that a programmer will use are *memory*, *input*, and *output*. With *input* and *output* we define the interface of the module, while *memory* declares new stateful Wires. Normally, these methods are called in the constructor and any declared wire can be used when defining a new computational graph. For stateless wires the user does not have to declare or create them, as the framework generates automatically new wires when an op is called. User defined Modules have to implement the constructor and override the method *architecture*. Inside the constructor we define any Inputs, Outputs or Memories. Methods *input* and *output* can also receive a Wire as an argument to register it accordingly as an input or output. The method *Bus* take a Wire as an argument and registers it as a bus (bidirectional). This method is currently intended for internal use only.

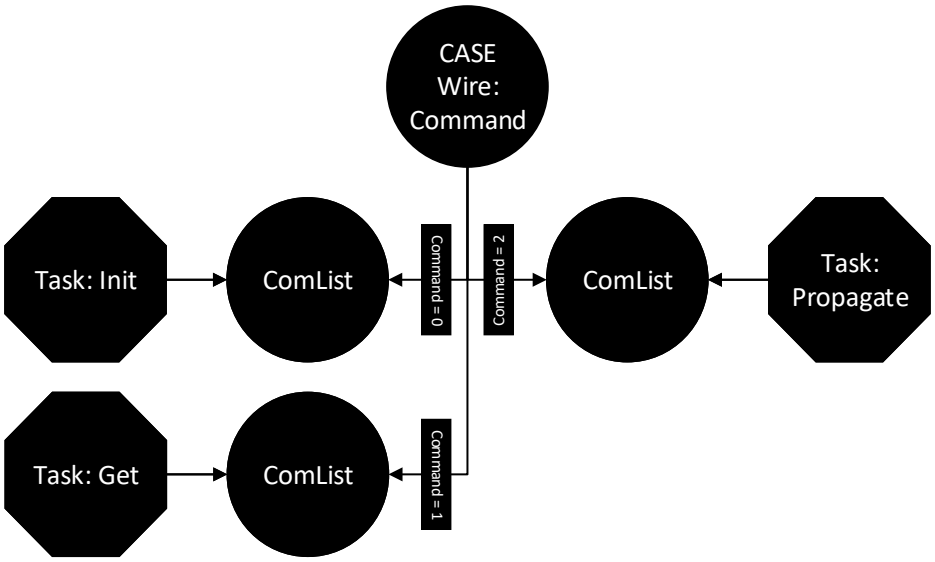

Figure 4.9: The control graph of Module class.

The control graph of a Module is shown in figure 4.9. Modules accept three types of commands:

1. *Init* commands to initialize all Memories (all wires declared with *memory*).
2. *Get* commands to get the current value of a Memory.
3. *Process* commands to execute all computational graphs.

In the control graph, the command selection corresponds to a root Case node with three children nodes, one for each command type. *Init* commands are represented by a *ComList* tied with an *Init* task. Similarly, *Get* commands are represented by a *ComList* with a *Get* task. These two tasks instruct the framework to generate hardware for loading/saving, all the wires in their ComList.

Every Memory Wire is registered in both lists, so that the application can initialize or get their state. Finally, the main processing is instructed by the *ComList* with the *Propagate* task. It is called *Propagate* because it simply calls the hls method of every Wire in the *ComList*.

4.3.7 Trainable Modules

Trainable Modules extend the Module class to generate synthesizable code that in addition to the user defined computational graphs, it contains the backward graph to train a Neural Network (back-propagation algorithm).

Trainable Module class extends Module class and implements the *architecture* method. As shown in table 4.5, User-defined Trainable Modules must extend the corresponding class, but they have to override the *forward* method. Internally, three new methods are defined:

1. Method *backward* generates the backward graph, by calling the methods *reset_derivatives* and *compute_derivatives*.

Method	Description
constructor	Constructs Module. It first employs the super constructor of the module and then alters the control graph accordingly.
architecture	This method is implemented for trainable modules by the framework. It constructs the computational graphs by utilizing the methods *forward* and *backward* to construct the forward and backward computational graphs respectively.
forward	Builds the forward computational graphs. It is abstract and hence the user must override it.
backward	Generates the backward graphs. It essentially calls *reset_derivatives* and then *compute_derivatives*.
memory	Declares a Memory. It is now extended to allow declaration of trainable memories
tmemory	Declare a Trainable Memory. It is an alias for *memory* with the argument *train=True*.
reset_derivatives	Resets all derivative graphs. Must be called to perform various initializations before calling *compute_derivatives*.
compute_derivatives	Generates the backward graphs. Must be called after calling *reset_derivatives*.

Table 4.5: The API of trainable modules (only the differences from simple modules).

2. Method *reset_derivatives* makes some simple initializations so to prepare for *computeDerivatives* execution and mainly sets pointers to backward graphs to *None*.

3. Method *compute_derivatives* is the one that actually generates all the backward graphs. To achieve it is

Method *memory* is overridden to support the declaration of Trainable Memories. Also, *tmemory* is actually an alias for the same purpose.

Figure 4.10 presents the control graph for Trainable Modules. The *process* branch is now updated to execute the forward graphs and then optionally run the backward ones. The value of *backward_flag,* decides if we will run the backward part of the execution.

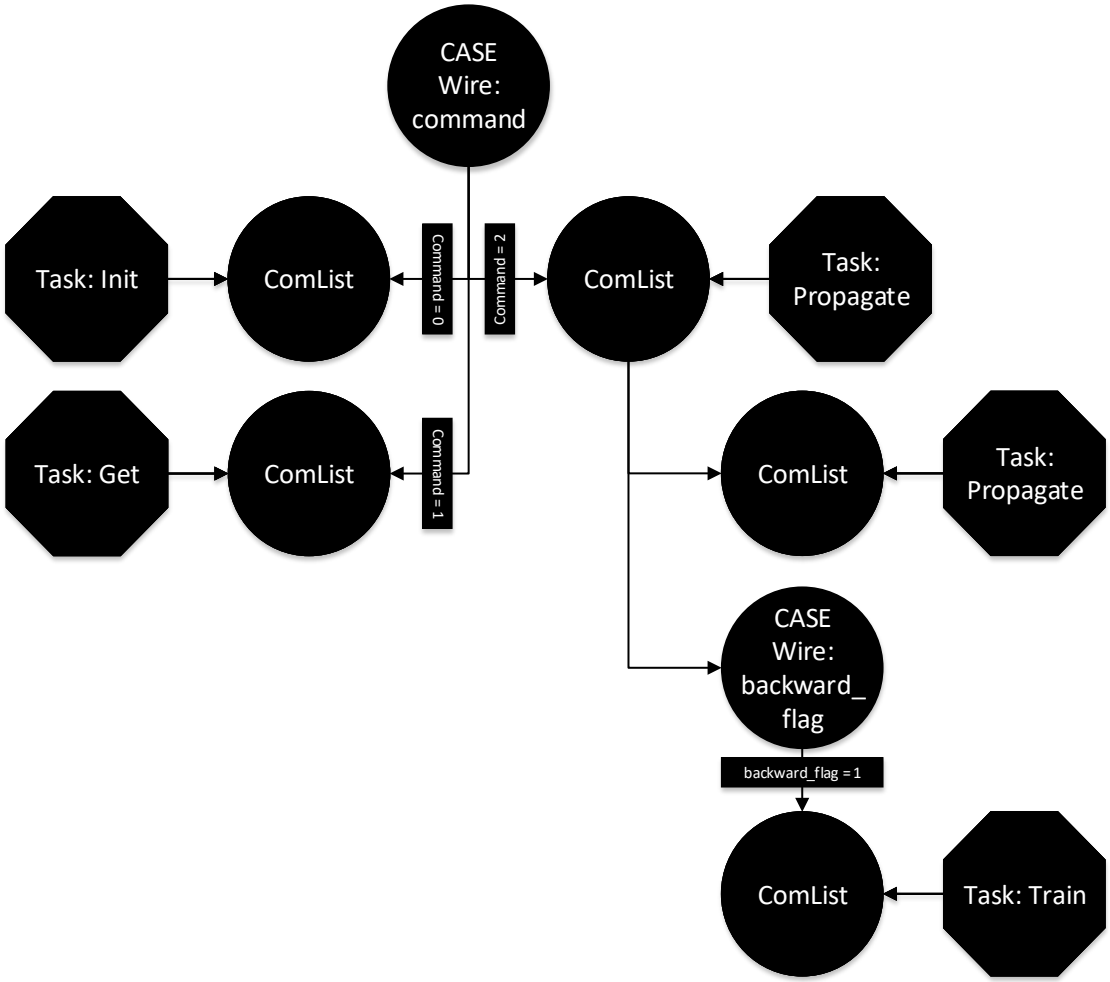

Figure 4.10: The control graph of Trainable Module class.

4.3.8 Memory Wires and Caching Mechanism

For each Memory Wire - trainable or not - another simple Wire is generated, with the same name and the same characteristics. The Memory Wire points into this simple Wire by using its "input" attribute. This representation allows for cleaner and simpler code when applying algorithms that process the graph. Furthermore, we could employ the usual pointer "p" and not "input", but that would create cycles in the graph that graph processing algorithms would require to perform complex checks to recognize when the processing must proceed to the sub-graphs or not. The use of two Wires with the same name helps on optimizations. For example, we when we use the value of a Memory Wire, there are cases when we want to utilize the value of a Memory Wire in the same period we have updated it (this happens very often when caching a Wire). In that case, we point to the simple Wire where the Memory "input" pointer indicates. This wire has the same name so the generated code will be the same, but also the main computational graph is compact and easy to process, as it can be processed by utilizing only "p" pointers. In conclusion, the complexity

of the required code is very important and this special representation helps in this directions.

Caching is a mechanism that allows the values of a Wire to be cached into a stateful Wire, so that their values can be utilized in latter periods. Initially we had implemented multiple caching strategies. However, this was far too complex and inefficient, so in the last implementation we use one caching mechanism and an additional stage in the Tanslation Pipeline, the "Memory Trimming Stage". The code is much more concrete this way.

4.3.9 Classes for Handling Vivado HLS

The framework has several classes for working with Vivado HLS, some of them are described here. First, Command Helper is a class with that implements functions that return strings of Vivado tcl commands depending on their arguments. These methods share the same name with corresponding tcl commands of Vivado HLS. Second, Command Writer defines methods for writing specific Vivado HLS commands into a TCL file. Essentially, Command Writer implements methods that wrap around the methods of Command Helper and use the same names. the same methods but now writes them in a file. Third, the class "Vivado HLS" is the central class for handling Vivado HLS projects. It is responsible for generating the correct project structure, the tcl scripts, any automatically generated testbenches and extracting any test data. To generate the testbench it utilizes another class named Test Bench Gen, which is responsible for generating the C++ testbench files.

4.3.10 Dataset and Dataset Exporter

Datasets are collections of Python Dictionaries that contain simple commands to execute. The developer that uses the framework can define a Dataset so that data can be automatically exported later for other uses such us testing the generated by the framework C++ code in Vivado HLS. Datasets can also run in Python.

Dataset Exporters have the sole role of exporting the data of a Dataset into a hard disk file structure. The corresponding structure is presented in figure 4.11. The C++ testbench that is generated by the framework (and more specifically from the TestBench Gen class) is carefully designed with the capability of interacting with these disk file structures.

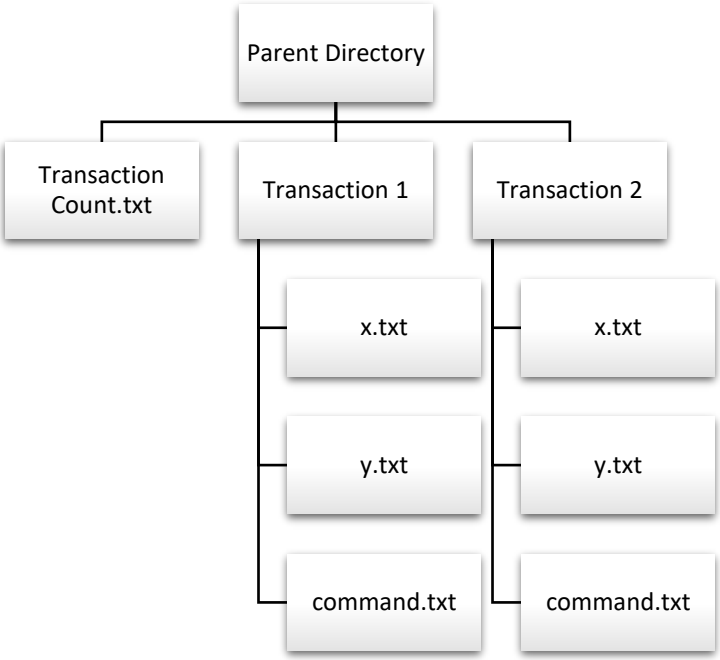

Figure 4.11: The directory structure that holds the exported Dataset.

4.4 Important Implementation Concepts

In this section, we describe many of the most important implementation concepts that need to have a compact presentation.

4.4.1 Graph Traversal

Figure 4.12 shows an example computational graph. A computational graph is comprised of wires and operations. Each wire points to the sub-graph that can be utilized to compute its value. Each operation points to its input wires. A list of all outputs is maintained for each module. Moreover, the graph can be perceived as a collection of threes that optionally share parts.

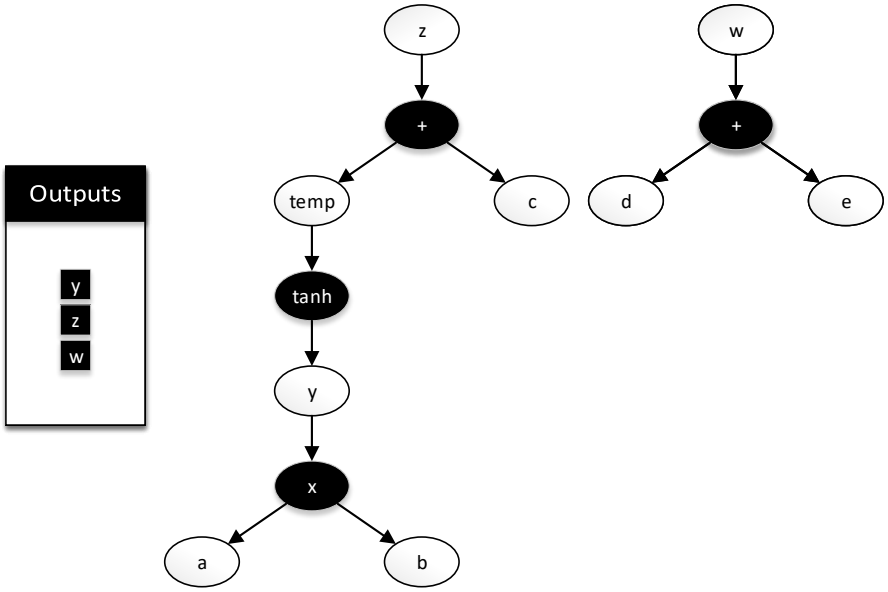

Figure 4.12: An example computational graph.

The easiest way to cope many important tasks on the framework is to process the graph as if it were a collection of trees, one for each output and use a post-order processing pattern. Intermediate results can be stored to prevent the recomputing of the shared sub-graphs. As presented in figure 4.13 the graph can be unfolded into a set of trees. Already computed sub-graphs can be seen as inputs to the the tree that we currently process. The corresponding algorithm is shown in pseudocode in snippet 4.7. Furthermore, node preprocessing (pre-order processing pattern) can be employed simultaneously for some tasks. The same algorithm can be used to process control graphs, although their structure seems different and we have only one entry point (the root of the control graph). Note that this is a very simplified version of the actual algorithm that we are using.

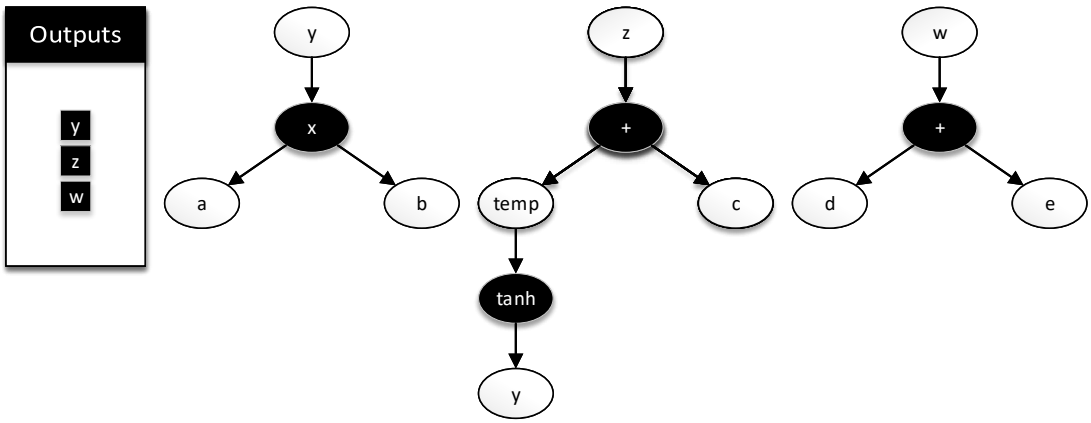

Figure 4.13: The unfolded version of the computational graph presented in figure 4.12.

Code Snippet 4.7 Simplified Graph Traversal.

```
ProcessGraph(G)
  ForEach Output of G
    Process(Output)

Process(Node)
  Node.ForcePreprocess()

  If IsAlreadyProcessed()
    Return CachedResult

  Node.Preprocess()

  ForEach Child of Node
    Process(Child)

  Node.Process()

  CacheResult()

  Return Result
```

4.4.2 Graph Updating

To make updating graphs easy we need an abstract mechanism to alter a node's pointers to its children. At the same time, we want these pointer to be accessible as class data member too, as convenience is one of our top priorities. For example, we want to be able to directly refer to a child inside the Operator methods, for example to use *self.ina* to address the first input of the Operator.

Method	**Role Description**
get_children(self, dict)	Returns all children nodes in a list or - if dict=True - all children nodes paired with their names in a python dictionary.
set_child(self, name, value)	Utilizes the buildin method *setattr* to alter a pointer to a children node.

Table 4.6: Abstract graph API.

We need an abstract mechanism that allows accessing children nodes and altering the corresponding pointers. Fortunately, Python has a builtin method *setattr()* that allows setting an attribute by its name, for example: *setattr(op, 'ina', new_value)*. The solution is presented in table 4.6. We have two methods *get_children()* and *set_child()*The first, receives one boolean input argument:

- If it is set to False (or left unassigned), the method returns a list with pointers to all node's children.
- If it is set to True the method will return a Python dictionary with both the pointers and the corresponding attribute names so that the caller can use *set_child()* to update them.

The second, utilizes *setattr()* and updates a pointer to a child by the attribute name. While this method in most cases just calls *setattr()*, in the roots of the graph we need special handling. We decided that defining a new method would be a cleaner solution that overriding *setattr()*.

The aforementioned concepts are shown in figure 4.14. For convenience, Operator base classes come with a predefined version of *get_children()*. To register a child Operator developers must call another predefined method of Operators: *input(attribute_name)* to register the attribute name as an input.

4.4.3 Root System

We need a convenient way to alter the module's attributes while processing its graphs, especially the nodes that serve as graph roots. Root System is a Python class, designed for use by graph algorithms that require to update the roots of a graph. Its purpose is to wrap around a Module (or a Trainable Module)

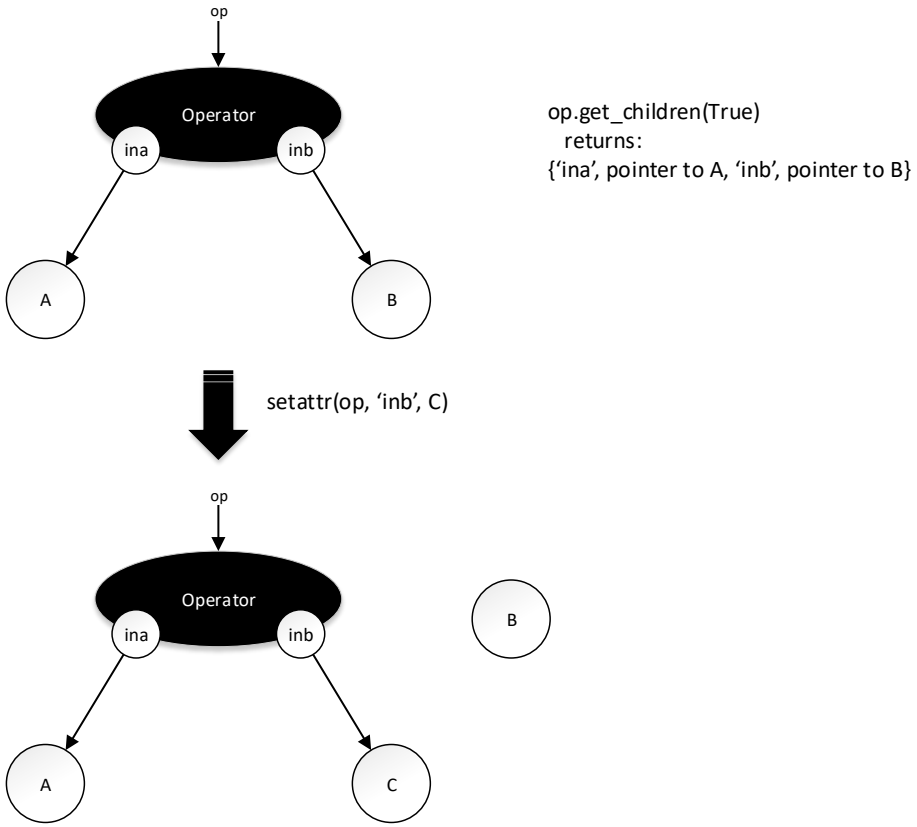

Figure 4.14: The API to alter children.

and provide the abstract Node API presented in subsection 4.4.2. Nevertheless, *get_children()* now returns a list (or dictionary) of predefined attributes of the underlying Module, not attributes of the Root System. At the same time, *set_child()* does not update the Root System attributes but the attributes of the Module. This is required because

This has several advantages versus implementing these methods on Modules:

- The algorithms can manipulate the pointers from the module to the computational graph in an abstract manner.
- We can select which attributes of the module will be considered.
- Modules have a cleaner API. The methods *get_children()* and *set_child()* on Modules normally have a very different meaning, they return the inputs of a Module, as in Operators and Wires.

4.4.4 Python Execution

Computational graphs can be utilized for efficient processing in any platform. Many Python packages and frameworks can be employed to execute the defined graph on any computer or cloud service using CPU or GPU resources.

In fact process is very efficient and can be optimized to achieve the highest possible performance on these platforms.

Such a feature can play a very important role in our Framework. Firstly, it can be very helpful in new module design because it allows to disentangle the design process from the actual hardware implementation. The hardware developer can replace the behavioral simulation with a much faster process as it will be optimized for general purpose hardware. Secondly, the framework can utilize the "Python Execution" functionality to generate datasets automatically for hardware verification.

To enable Python execution of the defined modules, the framework currently utilizes the Python NumPy library[1]. Inputs are initialized from the test data. Then we evaluate all nodes of the computational graph. The used algorithm is the same as presented in snippet 4.7. Each Operator must have a special method, "Operator.run()", that shows how to process the input data accordingly. The postprocessing step of the algorithm is set to run these methods. Executing the "Operator.run()" on the postprocessing step ensures that all subtrees have been processed and all children have data to process.

4.4.5 Python Synthesis Pipeline

Python synthesis can be a process of variable complexity depending on the implementation. However, we use an HLS language as the target language and so we prefer to push most of the complexity to the HLS C++ tool. We aim in keeping the graphs that describe the system always in a clear, easy-to-read state, so that is always easy for the developer to inspect the design and infer what is going to occur next.

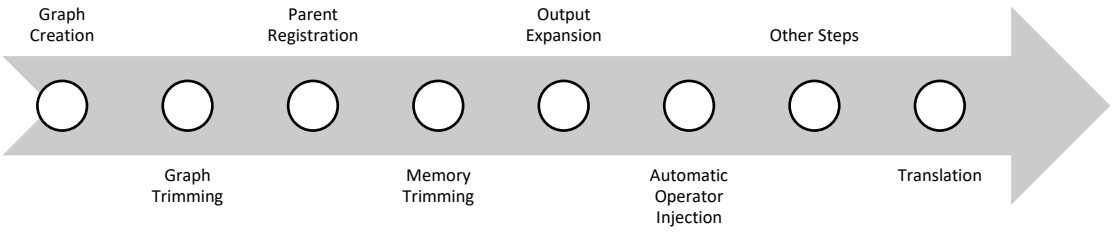

Figure 4.15: Python synthesis pipeline.

The python synthesis pipeline is presented if figure 4.15. Firstly, the framework builds the graph by calling the corresponding module methods which in their turn call the corresponding operator factories. In this phase we also generate the backward graph for Trainable Modules. Secondly, the graph is trimmed by removing unnecessary Wires. Thirdly, we register all nodes that use each

[1] In the future, it is possible to employ an other package, for instance TensorFlow.

Wire, the parent nodes. Fourthly, the framework injects automatically special operators, such as dispensers, which takes a Wire and broadcasts its values in multiple Wires. This broadcast is required for stream Wires that must feed multiple operators, because each data transmitted in a stream can be only consumed once. Finally, we translate the enhanced graph. These stages are described in detail in the subsections that follow.

4.4.6 Graph Trimming Stage

The graph may be trimmed and prepared for the next phases, otherwise we would have to contemplate more corner cases and thus next phases would be much more complex to leverage. During trimming we have to eliminate any chains of wires (wires that point into other wires). To solve the trimming problem we use the same variation of depth-first search, presented in snippet 4.7. Whenever, we find an operator, or a wire output, we perform a postprocessing step. The postprocessing occurs after every child C of a node A is fully processed. Thus, every child C has been already optimized. Hence, if the child C is a wire that points to another wire W, then the second wire W is the optimum and the node A must now point to W, not C. This is shown in figure 4.16.

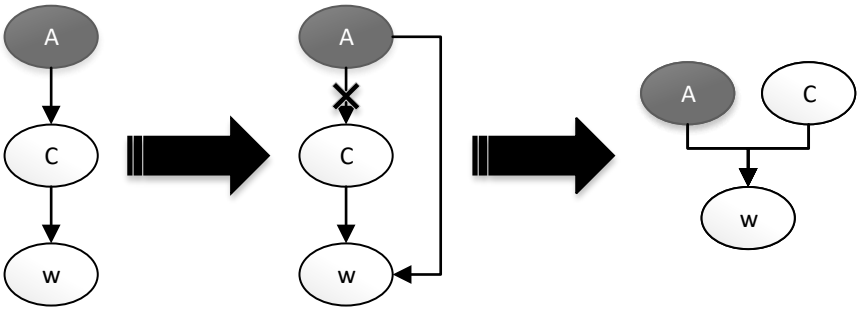

Figure 4.16: Graph trimming stage.

4.4.7 Parent Registration Stage

In some cases, we require pointers to not only one node's children, but parents too. In this step we traverse the computational graphs and keep lists of each Wire's parents, so that the next stages can use them. We run depth-first search of snippet 4.7 with the following simple forced preprocessing step: If the current node is a Wire of register the calling node as one of its parents. This naive step is enough, because the algorithm will run the forced preprocessing step once for each parent.

4.4.8 Memory Trimming Stage

In this stage, we perform optimizations related with memories. For example:

- If a Memory points into another Memory currently the two are merged, so that they generate less logic.

- If a Memory points into an output and it is not a stream then the graph is updated in a manner that the memory gets updated first and then the values are forwarded from the memory to the output. Moreover, the output will point to the input of the Memory Wire (which has the same name as the Memory Wire by convention).

- If a Memory points into an input or simple Wire that is not a stream then the graph is updated in a manner that the memory is updated first and then any further processing will utilize its values. Furthermore, any parent nodes of the Wires will be updated to point into the input of the Memory Wire (which has the same name as the Memory Wire by convention).

4.4.9 Output Expansion Stage

Output expansion Stage is a preparation step for the next stages. It helps for a cleaner operation injection phase with better name handling on the generated code. Furthermore, if an output is reused internally for further processing we have to specially handle it as interfaces have unique behavior target languages such as Vivado HLS C++. Additional Wire is needed to keep the internal version, but because interfacing Wires are indexed from the framework, we have to carefully create a Wire, replace the corresponding output Wire and make it point to the newly created Wire. This is called output expansion and we have to expand all outputs that have parent Operators. Figure 4.17 shows a minimal example of output expansion.

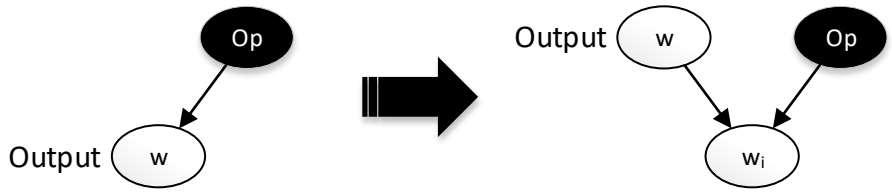

Figure 4.17: Output Expansion.

4.4.10 Operator Injection Stage

In this stage we inject operators in the computational graphs. Currently we inject two operators: Dispensers and Assignments. Dispensers are operators that create multiple copies of the data of their input Wire. Assignment Operators copy the value of a single wire to another one. While the two Operators are quite similar, both the framework code and the generated code are cleaner by separating them into two distinct Operators.

4.4.11 Dispenser Injection

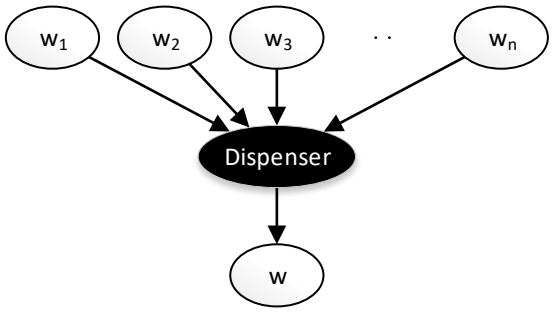

Figure 4.18: Dispenser Special Operator.

Dispenser, the graph representation of which is shown in figure 4.18, is an Operator that broadcasts the values of one source Wire into multiple destination Wires. It is instantiated automatically from the network during the dispenser injection stage. As shown in figures 4.19 and 4.20, Dispenser is injected in the following cases:

- The values streamed through a (stream) Wire are required from multiple Operators (figure 4.19).
- The values streamed through a (stream) Wire[1] output are required for further processing internally to the Module (figure 4.20).

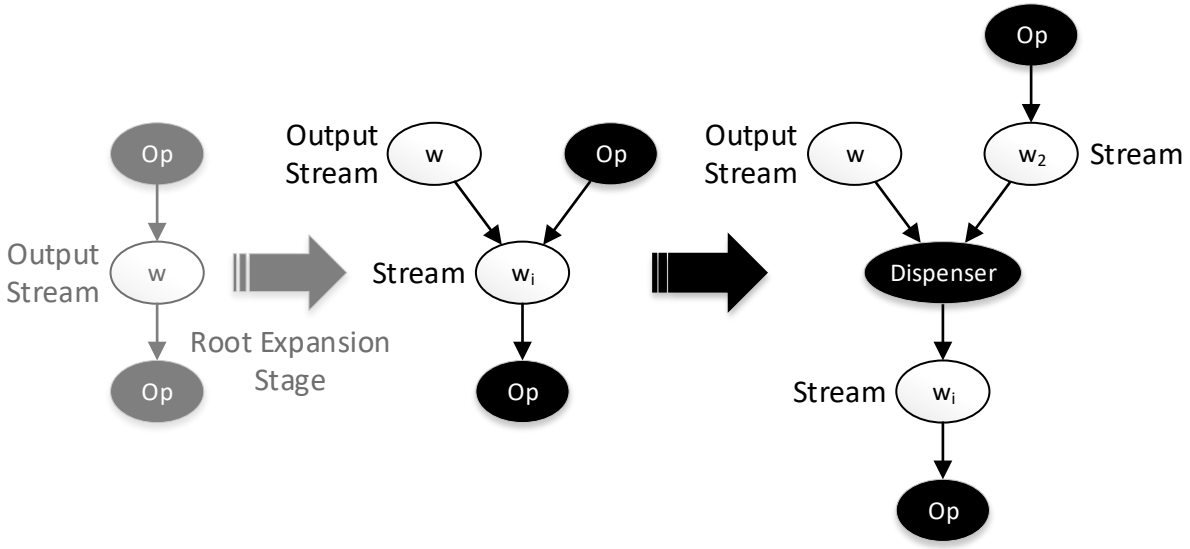

Figure 4.19: Dispenser injected to cope with an output that requires attention.

Dispenser Operator node is instantiated through its special Operator Factory. The framework generates different HLS C++ implementations of the Dispenser dynamically depending on the number and the characteristics of its

[1]For non-streams we prefer injecting Assignment Operators.

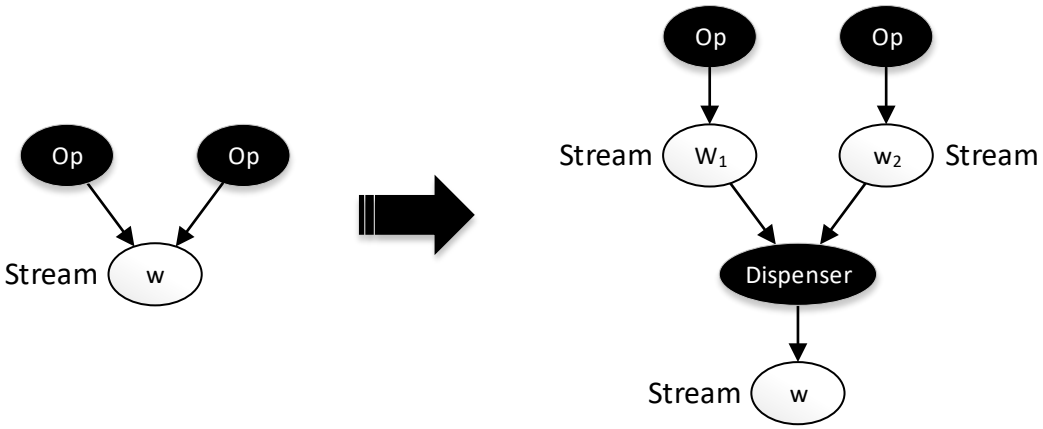

Figure 4.20: Dispenser injected to broadcast a stream of data into multiple ones.

outputs. We use Jinja 2 template language package to conveniently define the generated code using an easy-to-read template string. Note that Dispenser is the only Operator that is currently allowed to have multiple parent Wires.

4.4.12 Assignment Injection

For non-stream Wires we will instantiate multiple assignment operators, so that the generated code is easier to read. Scalar assignment injection is discussed in the next subsection. We need to keep the complexity as low as possible and the generated code easy to read and understand. In addition we have to keep our framework design clear and transparent to the developers. Hence, we need to handle the scalars differently, as programmers are used in builtin C++ scalar operators and expressions. This in turn means that we may encounter large expressions of Wires and the framework must decide where to generate an scalar-style assignment. The most suitable solution for our priorities is to generate and inject into the graph a Assignments Operator. For scalar operands the assignment will generate code that assigns a scalar C++ expression to the parent Wire. For non-stream Wires the assignment will generate code that copies the contents of the children wire to the parent. This special Operator must be generated automatically in the following cases:

1. A root node of the computational graph is scalar. An Assignment Operator is required to instruct the translation state to generate a scalar assignment.

2. A scalar Wire has multiple parent operators similar with figure 4.21, thus their values need to be reused multiple times.

3. We have encountered a Wire that is connected directly to another Wire.

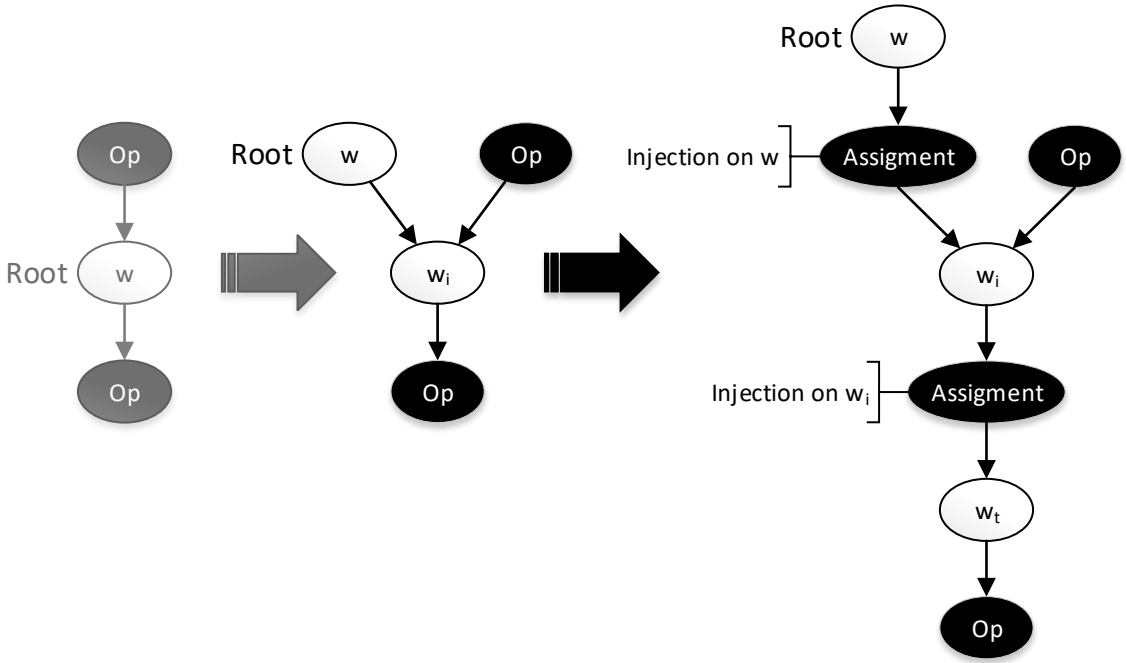

Figure 4.21: Assignment operators injected to cope with a computational graph root. The root has been expanded during the root expansion phase. Multiple assignment injections are required to handle with root nodes that have multiple parents, but due to root expansion we do not have to handle this case at all. Moreover, the combination of cases 1, 2 and 3 will handle it effectively. Note as well that injection on w_i is required only if w_i is a scalar.

4. We have encountered a scalar Wire the value of which is directly required by a non-scalar Operator. An assignment is required between the Wire and its child node to ensure that the corresponding generated variable is set, so that this variable can be later employed as an input argument by the non-scalar operator.

5. The values assigned to a non-stream root are required for further processing internally to the Module. In this case we have to insert multiple assignments (figure 4.21) after the root expansion phase. The combination of cases 1, 2 and 3 will, as well, handle this case effectively.

4.4.13 Implementation of Operator injection Stage

Implementation is greatly simplified due to the preceding phases. A set of special Operator Factories are implemented to inject Dispenser and Assignment Operators. In the highest level we have two functions:

- *dispenser_injection(wire, fix_parents, trim_extra)* and
- *assignment_injection(wire, fix_parents, trim_extra)*.

These accept three arguments: the wire that will serve as the input to the injected graph and two boolean flags. The flag *fix_parents* will update the lists of parent nodes if True. The flag *trim_extra* will ensure that no direct wire-to-wire assignments will generated from the injection. The cases described in subsections 4.4.11 and 4.4.12, after taking count the results of previous stages (especially the Root Expansion) can be rephrased into the following five cases:

1. A non-scalar root that points to another Wire which has multiple parents requires a Dispenser.
2. Any scalar root requires a single assignment injection.
3. Streams with multiple parents always require a Dispenser.
4. Scalar Wires with multiple parents require an Assignment.
5. Scalar wires with parent Operator that is non-scalar require an Assignment.
6. Any other Wire-to-Wire connections require an Assignment.

The case that a scalar output has multiple parents does not require special handling as it is handled by the combination of cases 2 and 4.

Initially, the injection phase stage examines the root nodes to detect the first two cases and handles them by calling two special operator factories accordingly. Afterwards, the stage executes the depth-first search of snippet 4.7, parametrized with a preprocessing function that detects these the rest of the cases.

4.4.14 Translation Stage and HLS Modules

The translation stage is the final step of the Python Synthesis pipeline. The generated code is divided in several parts. Jinja 2 templates are used for this purpose. The generated file is logically divided into several areas shown in figure 4.22.

A Python class have been developed to aid the translation: C++ HLS Module (in code is referred as CPPHLSModule). This class provides an API for registering, C++ header inclusions, Wire Configurations,Operator Definitions, top level HLS pragmas, as well as methods to directly write on the main body block and methods to register global and local variables. Note that global variables are utilized for memory wires and local for other non-interface wires. The API of C++ HLS Module is presented in table 4.7. A control graph, by definition, divides computational graphs in different batches. Global Wires will share their values between different batches, while Local ones will not.

The method *hls* initiates the Python synthesis and results in a HLS C++ file. All the aforementioned methods are for internal usage and the framework user will not have to interfere with them. The methods *globalWire* and *localWire* help in Wire declaration. Their role is tied with the notion of control graphs The

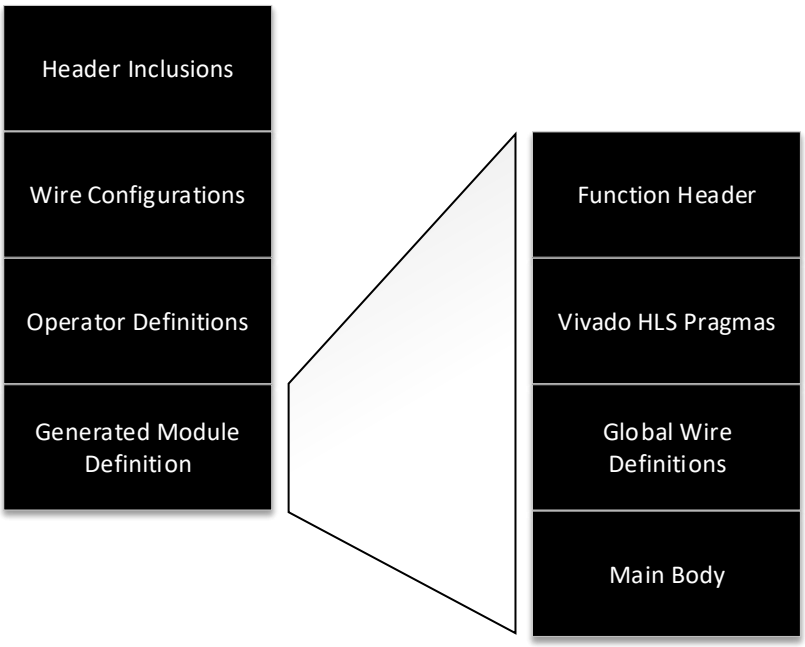

Figure 4.22: Generated Vivado HLS C++ source file structure.

use of control flow graphs helps for even simpler translation process, because it essentially instructs the framework how to structure the output HLS program. The main algorithm for translation is also the one presented in snippet 4.7. The process is performed by the hls methods that all modules and sub-modules have implemented. Initially, the hls method of the module is called, this calls the hls methods of the nodes of the control graph. The nodes of the control graph call the hls methods of computational graphs or the hls methods of tasks.

Method	**Description**
constructor	Constructs a Vivado HLS Module. It receives a module (or trainable module) as input.
hls	Translates the module to a C++ source file. Calls the *hls_cpp_str* and then saves the string in the adequate file.
hls_wire_conf	Registers a wire configuration string.
hls_op_def	Registers an Op definition.
hls_interface_pragma	Registers a Vivado HLS pragma.
hls_local_wire	Registers a local wire.
hls_global_wire	Registers a global wire.
hls_indent_incr	Increases indentation for the output source. To be used with *hls_put* and *hls_putline*.
hls_indent_decr	Decreases indentation for the output source. To be used with *hls_put* and *hls_putline*.
hls_put	Puts a string into main code block.
hls_putline	Puts a string and a newline character into the main code block.
hls_decl	Returns the header of the C++ function that implements the hardware module.
hls_cpp_str	Translates the module into string for later use.
hls_fix_interface	Performs some required steps

Table 4.7: The API of Vivado HLS module.

4.4.15 Backpropagation

Implementing backpropagation in a software package that receives as instructions computational graphs means that we have to add a package that generates automatically a special computational graph as well. These generated graph is called backward graph and shows how to compute the first-order derivatives efficiently. Note that the derivatives are required from optimization algorithms to train all the network parameters that the network designer marked as trainable.

Backward Graph Factory is a special Graph Factory, implemented to process the forward graph and generate the backward graph[1]. This idea is shown in figure 4.23. The framework treats the set of all already defined graphs that

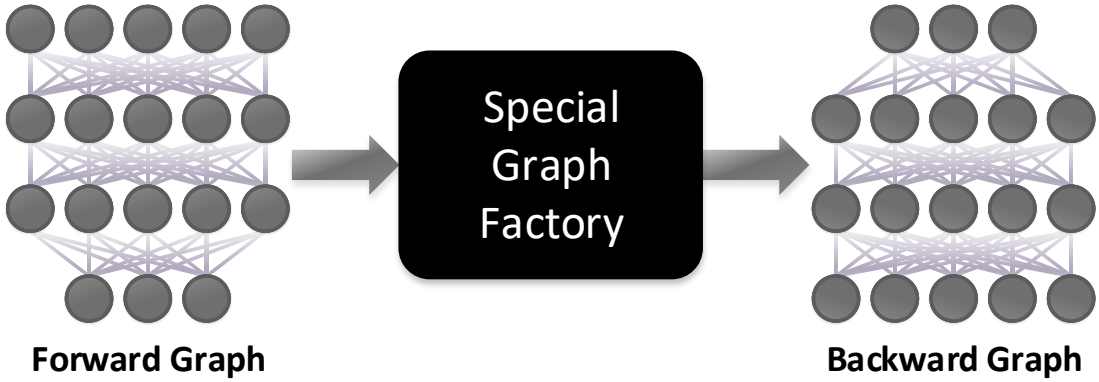

Figure 4.23: Backpropagation implementation as a special Graph Factory, the Backward Graph Factory. It should be noted that the graph representations are extremely abstracted. Especially the backward graph.

drive an output as the forward graph. The backward graph is somehow indistinguishable, excepting the fact that normally backward graph is responsible not to drive an output, but to train the network and drive Trainable Memory Wires with their next values. Backward Graph Factory employs simple Graph Factories implemented in each operator. These generate the computational graph for the portion of the chain rule that corresponds to the Operator. Furthermore, it takes into account the partial derivatives of the Operator outputs with respect to its inputs.

In more detail, backpropagation exploits the chain rule to compute the derivatives. For example, consider the scalar expression:

$$y = w_2 x_3 = w_3(x_2 + w_2) = w_3(x_1 b_1 + w_2).$$

A representation using a low level computational graph of two elemental operators - one for addition, one for multiplication - would require three computational nodes. Schematically the graph and the corresponding partial deriva-

[1] An alternative would be to employ an other software package with automatic differentiation already implemented. Furthermore, we could transform the graph in a format acceptable by some other package, compute the derivatives, and finally, transform back to our framework's graph model. Nevertheless, this implies that we would have to cope with all operators implemented in the used package.

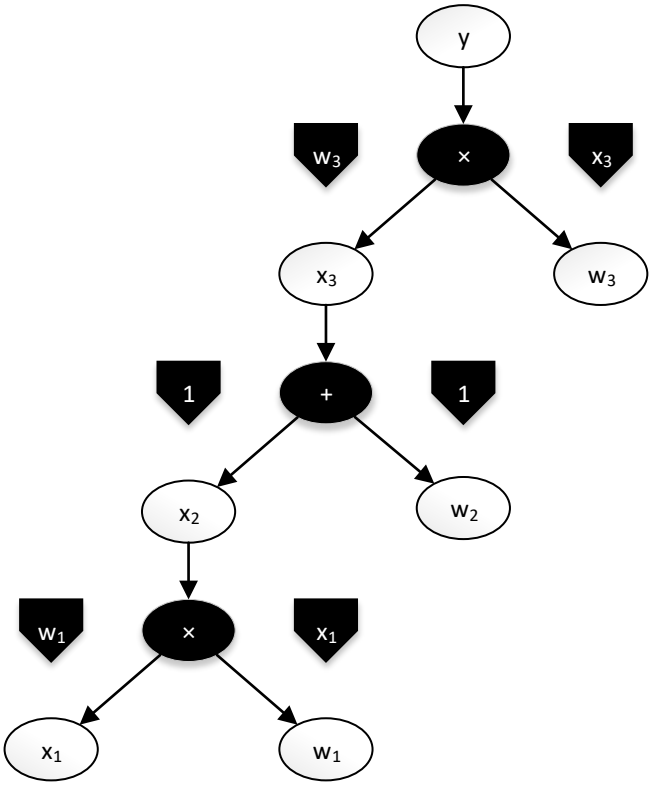

Figure 4.24: Computational graph example with all partial derivatives shown on the arrows next to the corresponding Operators.

tives are shown in figure 4.24. The arrows contain the derivatives of its Operators. In more detail these derivatives are: $\frac{\partial y}{\partial w_3} = x_3$, $\frac{\partial y}{\partial x_3} = w_3$, $\frac{\partial x_3}{\partial w_2} = 1$, $\frac{\partial x_3}{\partial x_2} = 1$, $\frac{\partial x_2}{\partial w_1} = x_1$, $\frac{\partial x_2}{\partial x_1} = w_1$. Assume an error function $\epsilon(y)$ and let $e_z = \frac{de}{dz}$ for any wire z in the graph. Note that in this example we do not have dependencies between the inputs of operators. By applying the chain rule progressively we obtain for example:

1. First, we obtain the graph to compute e_{w_3} and e_{x_3}:

$$e_{w_3} = \frac{de}{dy}\frac{dy}{dw_3} = e_y x_3, e_{x_3} = \frac{de}{dy}\frac{dy}{dx_3} = e_y w_3.$$

2. Second, we extend this graph for e_{w_2} and e_{x_2}.

$$e_{w_2} = \frac{de}{dx_3}\frac{dx_3}{dw_2} = e_{x_3} \cdot 1 = e_{x_3}, e_{x_2} = \frac{de}{dx_3}\frac{dx_3}{dx_2} = e_{x_3} \cdot 1 = e_{x_3}.$$

3. Finally, we extend the graph for e_{w_1} and e_{x_1}.

$$e_{w_1} = \frac{de}{dx_2}\frac{dx_2}{dw_1} = e_{x_2} x_1, e_{x_1} = \frac{de}{dx_2}\frac{dx_2}{dx_1} = e_{x_2} w_1.$$

Error-related derivative information flow from the error function to all other Wires that participate in the computation of the error function, in a way opposite to the forward graph. Each Operator has to extend the backward graph so that ensures that this flow can proceed from its output to its inputs. Moreover, it receives the graph to compute the derivatives of an error function with respect to the operators outputs and extends the graph to describe how to compute the partial derivatives of the error function with respect to the Operator's inputs. This implies one application of the chain rule with one substitution of the corresponding partial derivative of the operator, plus optimizations for each input. The application of the chain rule is embedded into each operator as a relatively simple Graph Factory. to allow easy optimizations depending on the Operator for the case of multidimensional Wires. The backward graph for our example is presented in figure 4.25.

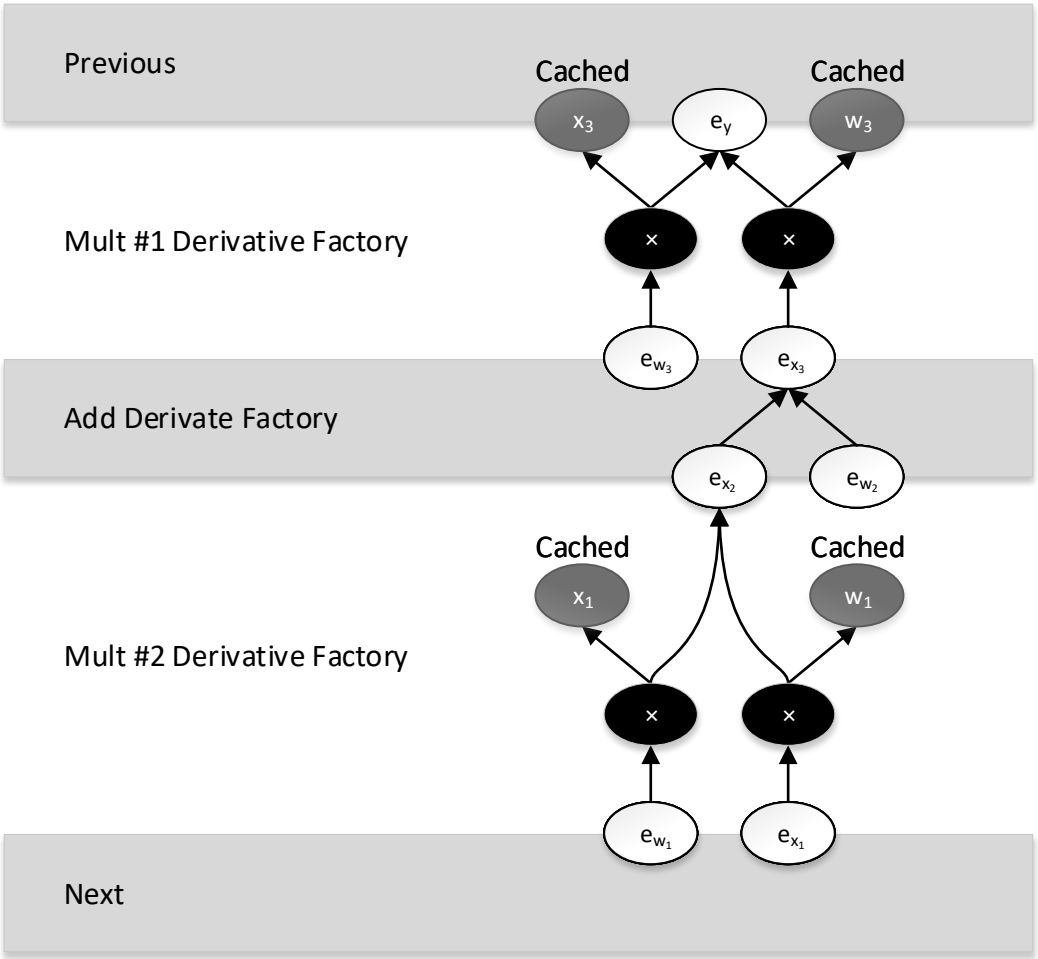

Figure 4.25: The generated backward graph for the example of figure 4.24.

If we have a dependence from one input to another we have a slight difference. In this case, the total derivative is different from the partial derivative and we normally have to use the chain rule. For example assume a function

$f(x,u)$ with $u = g(x)$ the total derivative $\frac{df}{dx}$ is:

$$\frac{df}{dx} = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial u}\frac{du}{dx} = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial u}\frac{dg}{dx}.$$

Thus we can temporarily ignore the dependence, proceed as before, and create Add (or Sum) Operators for more than one dependency paths. An example is shown in figure 4.26. To generate the graph that computes e_x we have to

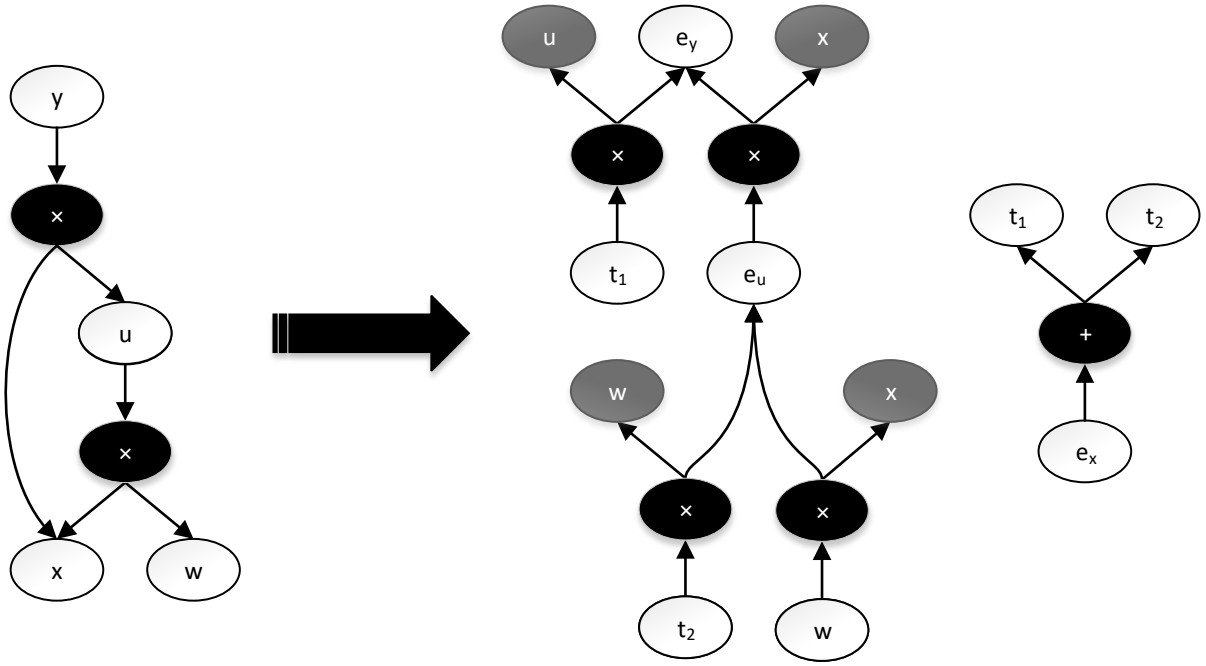

Figure 4.26: Backpropagation example with input dependency. The generated backward graph is shown in the right of the arrow.

temporarily ignore the dependence of u from x, process the graph as before and finally, employ an Add Operator to sum t_1 and t_2 together.

The algorithm that generates the backward graph is presented in snippet 4.8.

Code Snippet 4.8 Backward Graph Factory Algorithm.

```
1. Initialize all derivative pointers to None:
  For each Module Output that has an error function assigned:
    Run depth-first search from it and assign the
    'e' attribute of all Wires encountered to None.
2. Generate the graph:
  For each Module Output that has an error function assigned:
    Run depth-first search with a preprocessing step
    that calls the corresponding derivative factory
    of each encountered Operator.
```

Chapter 5

Framework Validation and Evaluation

5.1 Introduction

In this chapter we validate and evaluate the framework. We examine many different aspects of the tools. The verification is performed by examining the generated HLS C++ code and the it's synthesizability. The key points of the framework's evaluation is its usability and extensibility. At this stage we do not aim for generating the best performing hardware module. We aim primarily to design a tool that other developers will use and improve.

The rest of this chapter is organized as follows. In section 5.2 we present the framework validation methodology. Section 5.3 we analyze the installation procedure and the prerequisites. In section 5.4 we evaluate various framework-related tasks of high priority.

5.2 Framework Validation

The framework was developed in a incremental manner from the early beginning. As consequence, the correctness of many many important subsystems has been verified in different circumstances. For every new important update new testing code has been written to test the new functionality, while older testing code that remained compliant at the time has been reexamined. Nevertheless, we have to note that this constant updating with new features combined the pursue for extendability can is some occasions expose the framework into flaws of the newest code. Hence, methods for faster or even automatic testing must be explored at some point.

Testing and verification consecrated around the following:

- Op Verification: New Ops must be verified for expelling metadata inaccuracies, bugs in the HLS C++ implementation, Op factory flaws, differentiation graph factory flaws, Python execution method misses.

- Automatic differentiation: Methods that generate the graphs that implement the back-propagation algorithm must be verified due to their impor-

tance. This functionality is partially checked during verifying the correctness of an Op.

- Generated code synthesizability: During the development of new features the synthesizability of generated code must be checked periodically for small flaws that occasionally may raise.
- Python Execution and Simulation Subsystem: Python execution methods are utilized very often even during verifying the framework itself, so it is critical to function as expected.

5.3 Evaluation of Installation and Prerequisites

Prerequisites are very important when deciding to use a new tool. Open source and easy installable prerequisites are generally preferred. Requiring other tools that require explicit licenses is considered a big disadvantage. At the same time a tedious installation procedure can play a considerable negative role.

All the framework's prerequisites are shown in table 5.1. The only proprietary

Prerequisite	**Role Description**
Python 3	The framework is written in Python 3.
Pip	Required to install the framework and all the Python prerequisites.
NumPy	It is the main Python scientific package and the framework relies on it for many tasks (Python execution, random data generation etc.).
Jinja 2	It is a Python package that provides support for Jinja 2 templating language. We use it to generate C++ code.
tqdm	It is a python Package for displaying progress bars on a terminal.
Xilinx Vivado Suite	Required to translate the synthesizable C++ to RTL and perform tasks from the traditional RTL workflow (RTL Simulation, Synthesis, Map, PAR, BitGen). The framework can generate code and Vivado HLS projects even if Vivado is not installed, but it cannot proceed any further.
Git	Required for interaction with the framework's repository.
Setuptools & Wheels	Python packages - required only for development and distribution.

Table 5.1: Framework Prerequisites

software and the only that requires explicit license is the Xilinx Vivado Suite.

The framework is able to generate synthesizable C++ code without Vivado, but cannot complete the workflow and generate the hardware IP or bitfile. Furthermore, the framework can be developed and improved without Vivado.

Installation is extremely easy as the framework is distributed as other Python packages. 1. First, install Python 3 and ensure that Pip is installed. 2. Second, get the framework's distribution. 3. Third (optional step), configure the file "configuration.py" and set the path of Vivado bin folder. For development it is recommended to clone the git repository, install Setuptools and Wheels, and finally install from the downloaded local git repository using the command:

pip install -e <path>

5.4 Task-based Evaluation

The framework is designed with some priorities. For example the ability to extend the framework or define new modules easily and intuitive. Initially, we have to create a new Python module (a *.py file) inside the cloned project. As shown in snippet 5.1, this file must import the framework main module and the numpy library for modeling the test data (if required).

Code Snippet 5.1 Script beginning

```
import tensorglue as tg
import numpy as np
```

5.4.1 TensorGlue Module Definition

To define a new module the user must extend the Module class and implement two methods:

1. the constructor *__init__()*, and
2. the method *architecture()*.

These are presented in listings 5.2, and 5.3 respectively.

In the constructor we specify the directions of the interface's wires, and any stateful wires. We peaked a "loosely typed" scheme and do not specify the type and the dimensions of the interface to allow cleaner, faster and more flexible constructors. In the future we will also support strict definitions. Stateful wires will be always translated into FPGA local memory resources. For initialization we use NumPy arrays (here we generate random data and zeros using NumPy). In method *architecture()* we have to provide a computational graph that describes how the output can be computed from the inputs and memory elements (stateful wires). Here, the graph is simple to make easier the comparison with the generated code and differences with trainable modules. Defining

Code Snippet 5.2 Custom module constructor.

```
class CustomModule(tg.Module):
    def __init__(self, x, y):

        super().__init__()

        self.input('x')
        self.output('y')

        # Memories
        self.memory('W', np.random.randn(10,20),
            dtype=tg.float)
        self.memory('b', np.zeros(20), dtype=tg.float)
```

Code Snippet 5.3 Custom module architecture.

```
    def architecture(self):

        n = tg.matvecmul(self.x, self.W) + self.b
        self.y = tg.softmax(n)
```

the graph resembles normal arithmetic operations and functions. These do not actually execute the computations, but they create the corresponding graph nodes. However, this similarity can greatly help in framework API familiarization and code comprehension. Nevertheless, as with all deep learning frameworks, problems may also arise from such similarity. For example using normal Python control flow statements is counter-intuitive for software developers. Using them in graph definitions rather resembles the *generate* statements of VHDL language. Finally, Local variables, such as *n* here, can be used for convenience but they are simple pointers to graphs.

Code Snippet 5.4 Custom module instantiation.

```
x = tg.Wire('x', shape=[10], dtype=tg.float)
y = tg.Wire('y', shape=[20], dtype=tg.float)

# Module instantiation
modeinst = CustomModule(x, y)
```

Snippet 5.4 shows how the module is instantiated. We need to instantiate a module to be able to generate the HLS project. One wire must be declared for each input and each output.

5.4.2 TensorGlue Trainable Module Definition

To define a new trainable module we have to extend the TrainableModule class. The constructor, which is presented in snippet 5.5, works the same way as simple modules, however we can now have trainable signals. In trainable mod-

Code Snippet 5.5 Custom trainable module constructor.

```
class CustomTrainableModule(tg.TrainableModule):
    def __init__(self, x, y):

        super().__init__()

        self.input('x')
        self.output('y')

        # Trainable sitgals
        self.tmemory('W', np.random.randn(10,20),
            dtype=tg.float)
        self.tmemory('b', np.zeros(20), dtype=tg.float)
```

ules we implement the *forward()* method instead of *architecture*. An example implementation of the method is shown in snippet 5.6. As we see, the function

Code Snippet 5.6 Example implementation of the *forward* method in a trainable module.

```
    def forward(self):

        n = tg.matvecmul(self.x, self.W) + self.b
        self.y = tg.softmax(n)
```

code for this example is identical with the *architecture* method of the simple module defined in snipped 5.3. Trainable modules extend simple modules and have an internal implementation of the *architecture()* method that calls the *forward* method and then generates automatically the backward graph for back-propagation.

To train the module we need an optimization algorithm and a loss function. The first is internally defined as SGD, but we intent to support more algorithms in the future. The second is defined during module instantiation as shown in snippet 5.7. Each output has its own loss function.

5.4.3 Framework Tesbench Definition

An example testbench is presented in snippet 5.8. The data are generated with the aid of NumPy, and then passed to the framework through a Python

Code Snippet 5.7 Custom trainable module instantiation.

```
# Interface Connections
x = tg.Wire('x', shape=[10], dtype=tg.float)
y = tg.Wire('y', shape=[20], dtype=tg.float)

# The loss function
y.loss = tg.losses.crossentropyloss

# Module instantiation
modinst = CustomTrainableModule(x, y)
```

dictionary. Here we add three commands to the testbench. The first is *addInit()* which adds an initialization command that will initialize all stateful signals to their initial values. The second is an simple *add()* command which takes a Python dictionary as argument that describes the case to test. Command '2' means to execute the graphs. We do not need to specify the expected values if we want just to test the generated HLS hardware module. The Python simulation will be employed to generate the expected output values if these are not specified explicitly in the testbench case. Nevertheless, someone could simulate the module in Python or provide specify explicitly the output values for the Vivado HLS simulation.

5.4.4 Arbitrary Precision Types and Streams

To create a Wire of an arbitrary fixed-point precision type one could simply type:

Code Snippet 5.9 Arbitrary precision type example.

```
    # i is the number of bits for the integer part
    # d the number of bits for the decimal part
    x = tg.Wire('x', shape=[10], dtype=tg.fixed(i, d))
```

To create a stream one could use for example:

Code Snippet 5.10 Stream type example.

```
    x = tg.Wire('x', shape=[10], dtype=tg.stream(tg.float))
```

Code Snippet 5.8 Example testbench.

```
tb = tg.Testbench(testadd_inst)

# Initialization command
tb.addInit()

# Custom input data
x = np.random.uniform(0, 1, 10)

# One Python dictionary per case
# The expected correct outputs are computed in
# Python automatically if missing
io_dict = {'x':x, 'command':2}

# Adds the test case to the testbench
tb.add(io_dict)

# Read stateful sitgal state
tb.addGet()
```

5.4.5 Generating Vivado HLS C++ Project

With a module already defined it is very easy to generate the Vivado HLS project. We have only to run the code shown in snippet 5.11. The snippet is only comprised of two lines. The first line instantiates a handler for Vivado HLS

Code Snippet 5.11 Commands to generate the Vivado HLS project.

```
tb.hls()
```

related tasks. The second creates the project files with all the module code, simulation files, and simulation data.

5.4.6 Op Definition

To define a new Op we require three things:

1. The HLS C++ code that implements the Op.
2. A python class that can be used in computational graphs.
3. An Op Factory to enable easy instantiation.

All the three tasks are very straighforward to be implemented. The C++ code is simply a class template with one template parameter for each input or output. The python class very simple too. The developer must extend one

class of Ops and provide some metadata (such as the name C++ header file, or the allowed shapes for the Op inputs). Nevertheless, if this Op is intented for use with TrainableModules, then we also have to implement another special method too. This method is also an Op Factory that essentially defines the backward graph for this Op. Finally, we have to implement an Op Factory in "functions.py" that helps achieving easy and intuitive instantiation of the Op.

5.5 Performance Evaluation

To evaluate the performance of the Framework we run several tests. The actual performance of the generated hardware is a very important factor that needs dedicated research and work that can be conducted only after a serious portion of the framework infrastructure is already implemented and fully functional. For this reason, we have decided to not clutter the framework code with heavy optimization modifications and leave a significant portion of optimizations as future work. Nevertheless, we have included a basic set of optimizations, such as simple "HLS PIPELINE" directives on loops. In this section we evaluate briefly the current performance characteristics of the framework. This section is intended to map the behavior of the framework so that any latter work can be compared-with and any improvement can be presented.

5.5.1 Testing Script

To map the behavior of the system we have created a small script (about 55 lines). Inside we first define a custom parametric feed-forward Neural Network. The module extends the Module class of the framework (so it is essentially a Python class) and accepts as parameters the data type, the number of layers and the size of the input and output vectors. Then in a for-loop we iterate over different configurations, generate instances of the module and instruct the framework to synthesize in python and Vivado HLS. The framework is also provided with different destination directories for each configuration. Furthermore, the framework is instructed to time Python and C++ Synthesis. After running the script, we have multiple projects in distinct directories, synthesized, with a file time.txt with time statistics inside. The easiness to conduct such experiments is quite noticeable. It is the result of this current work that prioritizes developer friendliness and very fast prototyping. It is very uncommon for hardware development tools to be governed from such elasticity. In a second series of evaluation tests we ran the same configurations, but with Trainable Modules instead so that the Python Synthesis will create the logic for training the neural network parameters.

5.5.2 Testing Configurations

Our configuration to present an early performance behavior of the system is shown in the following tables. The network is a feed-forward neural network with ReLu activation functions in the hidden layers and softmax applied in the output layer. All combinations of the characteristics presented in the table are examined. For example, one combination is: float data types, 64 input size, 64 output size, 7 layers. The input size is set equal to output size to allow stacking multiple layers easily. The target device is xc7k160tfbg484-1, the target clock period is 4 ns and the module interface is selected to be m_axi. We currently present results for 32 bit floats, but the framework can support any data type that is supported from Vivado HLS and thus fixed point arithmetic is supported too. We used Vivado 2019.1 for these tests.

Data types	float
Layer Size	8, 64, 128
Layers	1, 4, 7, 10

Table 5.2: Testing Configurations

5.5.3 Synthesis Execution Time

The tables that follows presents the time required for Python and C++ synthesis for each combination. All time values are in seconds. We can clearly see that Python synthesis is orders of magnitude less time consuming that C++ synthesis as expected. Enabling training by utilizing Training Modules increases the required processing during Synthesis, but show a similar behavior. This justifies continuing using Python for all the framework's code excepting the HLS C++ kernel libraries or code generation templates. In the future, when we implement a more complicate optimization strategy, the numbers will likely increase. Nevertheless, the coarse grain nature of the building blocks and the complementary to C++ Synthesis role, more or less ensure that they will remain much smaller than the corresponding numbers of C++ Synthesis. The tests run on a machine with Intel 7700K CPU and 16GB RAM. The OS was CentOS 7.6 and we used Python 3.7 (Anaconda 2019.07). The TensorGlue was installed with pip in development mode.

	Number of Layers			
	1	4	7	10
Python Synthesis	0.030	0.027	0.017	0.022
C++ Synthesis	12.147	20.806	29.036	40.022
Total	12.177	20.833	29.053	40.044

Table 5.3: Synthesis Elapsed Time with 8 Neurons per layer.

	Number of Layers			
	1	4	7	10
Python Synthesis	0.025	0.027	0.018	0.225
C++ Synthesis	12.059	20.395	30.365	38.621
Total	12.084	20.422	30.383	38.846

Table 5.4: Synthesis Elapsed Time with 64 Neurons per layer.

	Number of Layers			
	1	4	7	10
Python Synthesis	0.028	0.0155	0.021	0.028
C++ Synthesis	12.922	18.803	26.687	38.876
Total	12.950	18.818	26.708	38.904

Table 5.5: Synthesis Elapsed Time with 128 Neurons per layer.

	Number of Layers			
	1	4	7	10
Python Synthesis	0.037369	0.032046	0.047269	0.065149
C++ Synthesis	14.207448	34.233570	57.302809	85.935359
Total	14.244817	34.265616	57.350078	86.000509

Table 5.6: Synthesis Elapsed Time with 8 Neurons per layer and training enabled.

	Number of Layers			
	1	4	7	10
Python Synthesis	0.031168	0.032550	0.048550	0.113427
C++ Synthesis	14.119601	31.540804	57.067342	86.105353
Total	14.150769	31.573354	57.115892	86.218780

Table 5.7: Synthesis Elapsed Time with 64 Neurons per layer and training enabled.

	Number of Layers			
	1	4	7	10
Python Synthesis	0.015981	0.085570	0.088197	0.069962
C++ Synthesis	14.944453	32.882714	57.293712	88.165618
Total	14.960433	32.968284	57.381909	88.235579

Table 5.8: Synthesis Elapsed Time with 128 Neurons per layer and training enabled.

5.5.4 Device Utilization and Latency

In this subsection we present the device utilization and the achieved latency results, from the C++ Synthesis of the generated modules. Most of the generated modules achieved the target clock period. Moreover, the same results presented here, are provided in more detail in appendix B in tables. Module interfacing employs s_axi and m_axi which also require resources to support. We did not try to tweak any generated design at all. As expected trainable modules generates more logic and it is far more expensive. In two cases - 7 and 10 layers with 128 neurons in each layer - the generated design requires far more BRAM_18K resources that the selected device support. We also run tests for fixed point arithmetic of 16 bits, where the generated code utilizes the "ap_fixed" datatype of Vivado HLS. Furthermore, 4 bits for the integer part and 12 for the decimal. The following tables present the results for 10 layers with the same characteristics as previously.

Figures 5.1 presents the BRAM utilization for each test configuration. As ex-

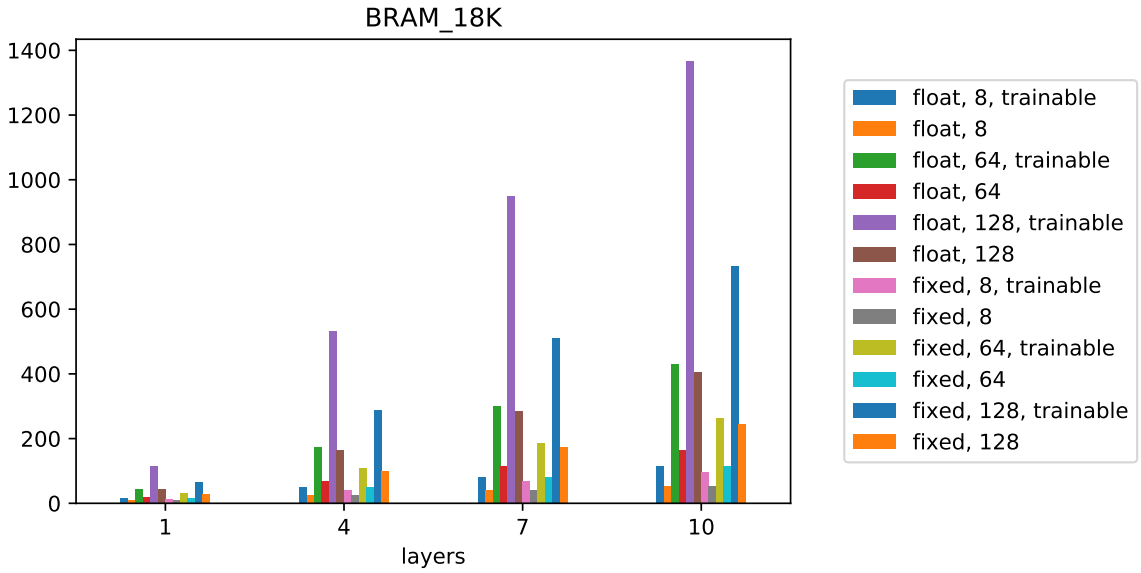

Figure 5.1: Python and Vivado C++ Synthesis results, BRAM utilization.

pected, the required resources scale almost linearly as the number of layers

increases. Nevertheless, the actual rate of the increase depends on the other characteristics of the configuration. Trainable Modules require far more BRAM resources that simple inference-only Modules. At the same time, utilizing float (32-bit) data types requires much more resources that fixed (16-bit) types. Figure 5.2 shows the DSP utilization. While the DSP utilization scales, the utilization

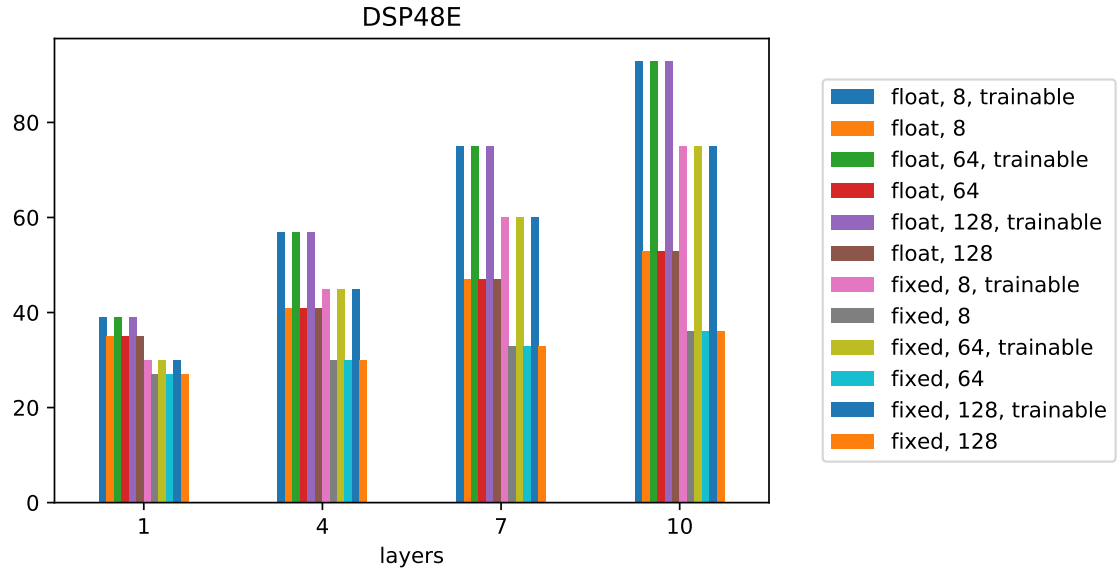

Figure 5.2: Python and Vivado C++ Synthesis results, DSP48E utilization.

is low and could possibly increase with small optimizations on the kernels. Figure 5.3 presents the utilization of flip-flop resources. The LUT utilization is shown

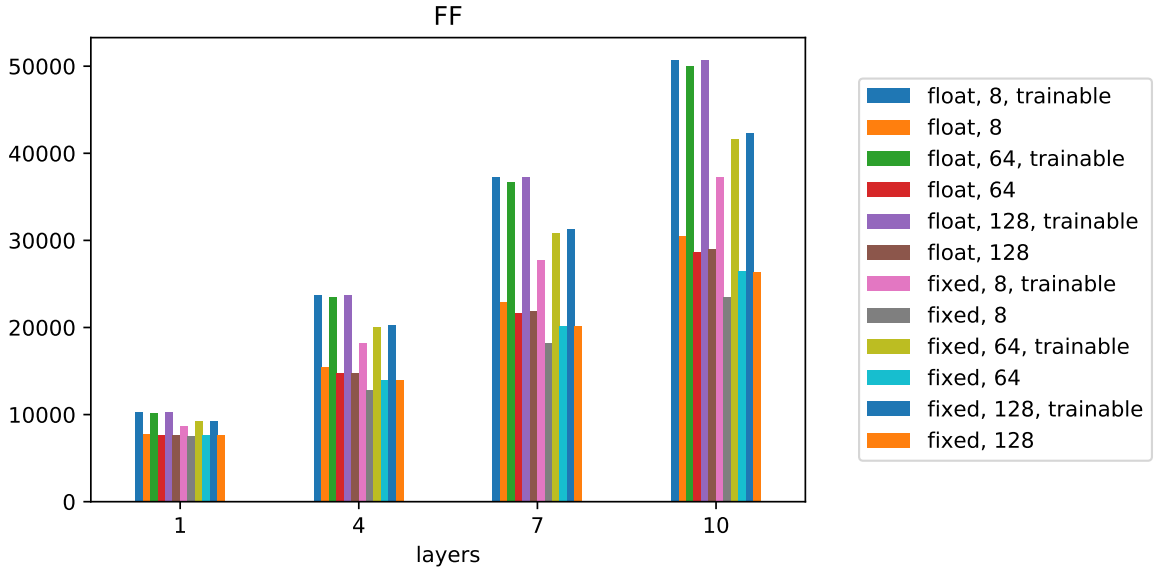

Figure 5.3: Python and Vivado C++ Synthesis results, FF utilization.

in figure 5.4.

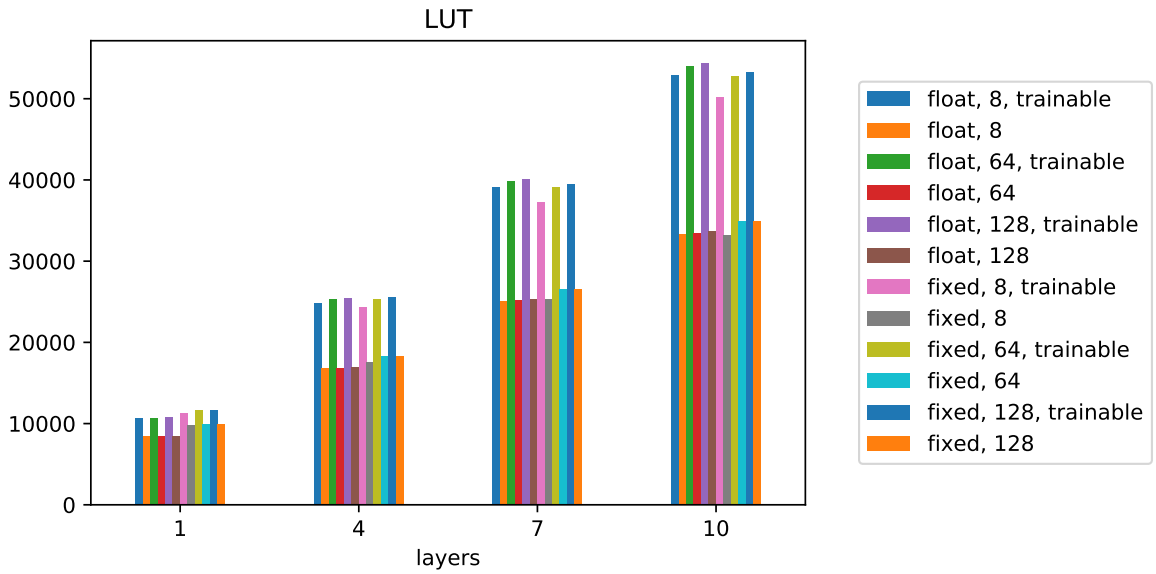

Figure 5.4: Python and Vivado C++ Synthesis results, LUT utilization.

Figure 5.5 shows the corresponding achieved latency. Note that the latency increases considerably for Trainable Modules. This is expected as training requires far more computations. Float data types require more latency as well due to inherent latency of float operations.

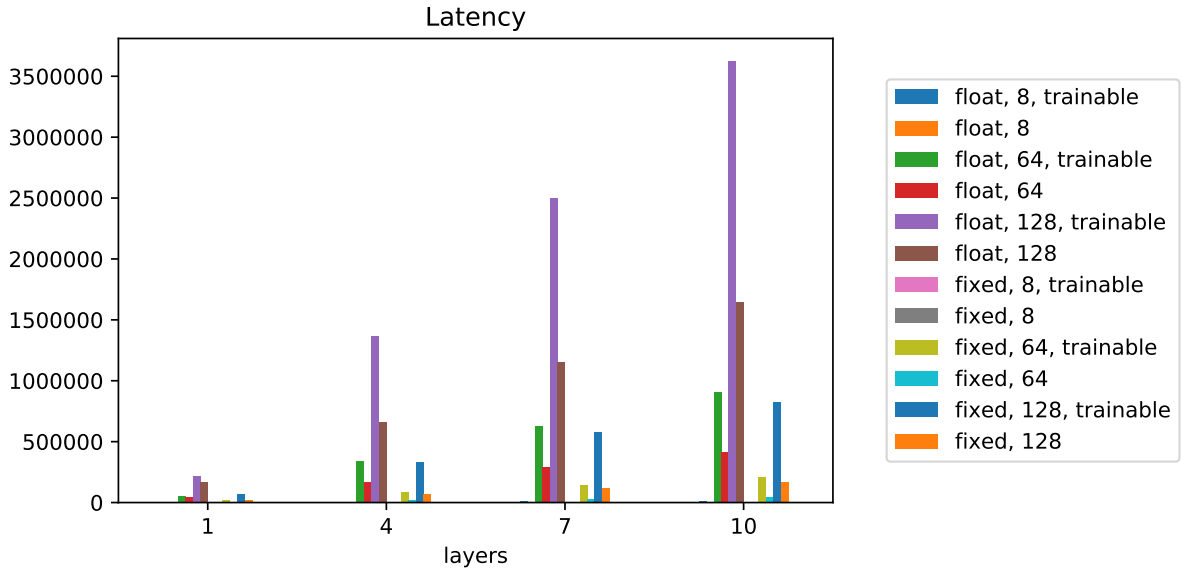

Figure 5.5: Python and Vivado C++ Synthesis results, latency results.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Reconfigurability of FPGAs may be an exotic property for deep learning research, but it currently remains more or less unexplored, unexploited and degraded because of the lack of the corresponding tools. In this work, we have implemented the first deep learning framework for FPGA-based hardware design. We have acknowledged that designing such a framework is a three-part modeling process. On the top developers that use the framework try to model the solutions for the problems they try to solve. On the bottom, developers we have the fundamental hardware modeling process. In the center, the core modeling process where APIs are designed to provide the required functionality for the other two processes.

Moreover, this framework could be seen as an HLS tool. Its greatest difference from other HLS tools is its extendability. We have endeavored to keep this framework as simple as it was possible, so that it could be effectively extendable. All the internals of the tool are exposed to the designer and its easy to extend it. Most (if not all) HLS tools lack this property but from our point of view it is crucial to have.

6.2 Future Work

This work was designed to be extended further in many different ways. From the one hand, extendability of the framework was a top priority. From the other, one of its primary goals is to foster research. Some ideas for future work include:

- Allow modules to employ other modules on their computational graphs. The framework was designed with this kind of modularity in mind. Moreover, its first versions included this functionality, but we temporarily disable it and stalled its development in order to push for other features.
- Extend the framework to be able to generate block-level designs. While currently we can generate IP cores using Vivado HLS, we would like to be able to generate the full system design automatically.

- Research on graph partitioning, either to generate block-level designs comprising of multiple generated IPs, or to utilize multi-FPGA infrastructures.
- Employ partial reconfiguration in order to schedule in time the computational graph by reconfiguring the hardware device. Moreover, a computational graph can be broken up into smaller sub-graphs. These subgraphs can be then translated into different hardware modules and scheduled accordingly.
- Implement more Ops, Optimizations algorithms, and loss functions.
- Research for DNN architectures and algorithms that take advantage of the characteristics of FPGA. Moreover, many of the current DNN architectures and algorithms are more or less state-of-the-art depending on their GPU performance.
- Support a carefully designed form of eager execution that may be helpful for even efficient design flow.
- Design a high-level API resembling Keras as much as it is possible, but specially designed and extended to address hardware related concepts.
- Replace NumPy with TensorFlow wherever it is possible to allow running the same code in top performance in non FPGA infrastructures.
- Support handling control flow statements in a way similar to TensorFlow Autograph.
- Implement a runtime system that will manage and even automatically handle FPGA infrastructure programming.

Appendix A

Input and Generated Code

A.1 Input Code

Listing A.1: TensorGlue custom Module.

```
import tensorglue as tg
import numpy as np

tg.set_vivado_path('/opt/Xilinx/Vivado/2019.1/bin')

class CustomModule(tg.Module):

    def __init__(self, x, y):

        super().__init__()

        self.input('x')
        self.output('y')

        self.memory(f'W', np.random.randn(20, 10), dtype=tg.
   float)
        self.memory(f'b', np.zeros(20), dtype=tg.float)

    def architecture(self):

        n = tg.matvecmul(self.W, self.x) + self.b

        self.y = tg.softmax(n)


x = tg.Wire('x', shape=(10,), dtype=tg.float)
y = tg.Wire('y', shape=(20,), dtype=tg.float)

inst = CustomModule(x, y)
```

Listing A.2: TensorGlue custom Trainable Module.

```
import tensorglue as tg
import numpy as np

tg.set_vivado_path('/opt/Xilinx/Vivado/2019.1/bin')

class CustomTrainableModule(tg.TrainableModule):

    def __init__(self, x, y):

        super().__init__()

        self.input('x')
        self.output('y')

        W = self.tmemory(f'W', np.random.randn(20, 10), dtype=
   tg.float)
        b = self.tmemory(f'b', np.zeros(20), dtype=tg.float)


    def forward(self):

        n = tg.matvecmul(self.W, self.x) + self.b

        self.y = tg.softmax(n)


x = tg.Wire('x', shape=(10,), dtype=tg.float)
y = tg.Wire('y', shape=(20,), dtype=tg.float)
y.loss = tg.losses.crossentropyloss

inst = CustomModule(x, y)

inst.hls()
```

A.2 Generated Code

Listing A.3: The generated code for a TensorGlue Module.

```
// Includes
#include "softmax.h"
#include <string.h>
#include "add.h"
#include "matvecmul.h"
```

```
// Op Configurations
struct x_conf : ArrayConfiguration<float, 1, 10> {
  static const unsigned int shape0 = 10;
};

struct command_conf : ArrayConfiguration<unsigned int, 0, 1> {
};

struct y_conf : ArrayConfiguration<float, 1, 20> {
  static const unsigned int shape0 = 20;
};

struct W_conf : ArrayConfiguration<float, 2, 200> {
  static const unsigned int shape0 = 20;
  static const unsigned int shape1 = 10;
};

struct b_conf : ArrayConfiguration<float, 1, 20> {
  static const unsigned int shape0 = 20;
};

struct namelesswire_0_conf : ArrayConfiguration<float, 1, 20>
   {
  static const unsigned int shape0 = 20;
};

struct namelesswire_1_conf : ArrayConfiguration<float, 1, 20>
   {
  static const unsigned int shape0 = 20;
};

// Op Definitions
static MatVecMul<namelesswire_0_conf, W_conf, x_conf>
   MatVecMul_namelesswire_0;
static Add<namelesswire_1_conf, namelesswire_0_conf, b_conf>
   Add_namelesswire_1;
static Softmax<y_conf, namelesswire_1_conf> Softmax_y;

void CustomModule(float x[10], unsigned int command, float y
   [20], float W_port[200], float b_port[20])
{
    #pragma HLS INTERFACE s_axilite port=x
    #pragma HLS INTERFACE s_axilite port=command
    #pragma HLS INTERFACE s_axilite port=y
    #pragma HLS INTERFACE m_axi port=W_port
    #pragma HLS INTERFACE m_axi port=b_port
```

```
    // Global Wires
    static float b[20];
    static float W[200];

  switch (command) {
  case 0:
    // Case "init"
    memcpy(W, W_port, 200 * sizeof(float));
    memcpy(b, b_port, 20 * sizeof(float));

    break;
  case 1:
    // Case "get"
    memcpy(W_port, W, 200 * sizeof(float));
    memcpy(b_port, b, 20 * sizeof(float));

    break;
  case 2:
    // Case "process"

    float namelesswire_0[20];

    MatVecMul_namelesswire_0.run(namelesswire_0, W, x);

    float namelesswire_1[20];

    Add_namelesswire_1.run(namelesswire_1, namelesswire_0, b);
    Softmax_y.run(y, namelesswire_1);



    break;
  }


}
```

Listing A.4: The generated code for a TensorGlue Trainable Module.

```
// Includes
#include "softmax.h"
#include "outer.h"
#include <string.h>
#include "matvecmul.h"
#include "scale.h"
#include "sub.h"
```

```
#include "add.h"
#include "copy.h"

// Op Configurations
struct learning_rate_conf : ArrayConfiguration<float, 0, 1> {
};

struct command_conf : ArrayConfiguration<unsigned int, 0, 1> {
};

struct x_conf : ArrayConfiguration<float, 1, 10> {
  static const unsigned int shape0 = 10;
};

struct y_label_conf : ArrayConfiguration<float, 1, 20> {
  static const unsigned int shape0 = 20;
};

struct backward_flag_conf : ArrayConfiguration<unsigned int,
   0, 1> {
};

struct y_conf : ArrayConfiguration<float, 1, 20> {
  static const unsigned int shape0 = 20;
};

struct y_cached_conf : ArrayConfiguration<float, 1, 20> {
  static const unsigned int shape0 = 20;
};

struct W_conf : ArrayConfiguration<float, 2, 200> {
  static const unsigned int shape0 = 20;
  static const unsigned int shape1 = 10;
};

struct x_cached_conf : ArrayConfiguration<float, 1, 10> {
  static const unsigned int shape0 = 10;
};

struct b_conf : ArrayConfiguration<float, 1, 20> {
  static const unsigned int shape0 = 20;
};

struct namelesswire_0_conf : ArrayConfiguration<float, 1, 20>
   {
  static const unsigned int shape0 = 20;
```

```
};

struct namelesswire_1_conf : ArrayConfiguration<float, 1, 20>
    {
  static const unsigned int shape0 = 20;
};

struct namelesswire_4_conf : ArrayConfiguration<float, 1, 20>
    {
  static const unsigned int shape0 = 20;
};

struct W_error_conf : ArrayConfiguration<float, 2, 200> {
  static const unsigned int shape0 = 20;
  static const unsigned int shape1 = 10;
};

struct namelesswire_8_conf : ArrayConfiguration<float, 2, 200>
     {
  static const unsigned int shape0 = 20;
  static const unsigned int shape1 = 10;
};

struct namelesswire_10_conf : ArrayConfiguration<float, 1, 20>
     {
  static const unsigned int shape0 = 20;
};

// Op Definitions
static Copy<x_cached_conf, x_conf> Copy_x_cached;
static MatVecMul<namelesswire_0_conf, W_conf, x_cached_conf>
    MatVecMul_namelesswire_0;
static Add<namelesswire_1_conf, namelesswire_0_conf, b_conf>
    Add_namelesswire_1;
static Softmax<y_cached_conf, namelesswire_1_conf>
    Softmax_y_cached;
static Copy<y_conf, y_cached_conf> Copy_y;
static Sub<namelesswire_4_conf, y_cached_conf, y_label_conf>
    Sub_namelesswire_4;
static Outer<W_error_conf, namelesswire_4_conf, x_cached_conf>
     Outer_W_error;
static Scale<namelesswire_8_conf, learning_rate_conf,
    W_error_conf> Scale_namelesswire_8;
static Add<W_conf, W_conf, namelesswire_8_conf> Add_W;
static Scale<namelesswire_10_conf, learning_rate_conf,
    namelesswire_4_conf> Scale_namelesswire_10;
```

```
static Add<b_conf, b_conf, namelesswire_10_conf> Add_b;

void CustomTrainableModule(float learning_rate, unsigned int
   command, float x[10], float y_label[20], unsigned int
   backward_flag, float y[20], float b_port[20], float W_port
   [200])
{
    #pragma HLS INTERFACE s_axilite port=learning_rate
    #pragma HLS INTERFACE s_axilite port=command
    #pragma HLS INTERFACE s_axilite port=x
    #pragma HLS INTERFACE s_axilite port=y_label
    #pragma HLS INTERFACE s_axilite port=backward_flag
    #pragma HLS INTERFACE s_axilite port=y
    #pragma HLS INTERFACE m_axi port=b_port
    #pragma HLS INTERFACE m_axi port=W_port

    // Global Wires
    static float x_cached[10];
    static float y_cached[20];
    static float b[20];
    static float W[200];

  switch (command) {
  case 0:
    // Case "init"
    memcpy(b, b_port, 20 * sizeof(float));
    memcpy(W, W_port, 200 * sizeof(float));

    break;
  case 1:
    // Case "get"
    memcpy(b_port, b, 20 * sizeof(float));
    memcpy(W_port, W, 200 * sizeof(float));

    break;
  case 2:
    // Case "process"
    Copy_x_cached.run(x_cached, x);

    float namelesswire_0[20];

    MatVecMul_namelesswire_0.run(namelesswire_0, W, x_cached);

    float namelesswire_1[20];

    Add_namelesswire_1.run(namelesswire_1, namelesswire_0, b);
```

```
    Softmax_y_cached.run(y_cached, namelesswire_1);
    Copy_y.run(y, y_cached);



    switch (backward_flag) {
    case 0:
      // Case "backward"

      float namelesswire_4[20];

      Sub_namelesswire_4.run(namelesswire_4, y_cached, y_label
  );

      float W_error[200];

      Outer_W_error.run(W_error, namelesswire_4, x_cached);

      float namelesswire_8[200];

      Scale_namelesswire_8.run(namelesswire_8, learning_rate,
  W_error);
      Add_W.run(W, W, namelesswire_8);

      float namelesswire_10[20];

      Scale_namelesswire_10.run(namelesswire_10, learning_rate
  , namelesswire_4);
      Add_b.run(b, b, namelesswire_10);

      break;
    }

    break;
  }


}
```

Appendix B

Measurements in Detail

B.1 Device Utilization and Latency

	Number of Layers			
	1	4	7	10
BRAM_18K	9 (1%)	24 (3%)	39 (6%)	54 (8%)
DSP48E	35 (5%)	41 (6%)	47 (7%)	53 (8%)
FF	7777 (3%)	15388 (7%)	22933 (11%)	30478 (15%)
LUT	8409 (8%)	16782 (16%)	25078 (24%)	33271 (32%)
Latency	843	2898	4953	7008
Interval	843	2898	4953	7008
Estimated Clock (ns)	3.5 ± 0.5	3.5 ± 0.5	4.087 ± 0.5	3.907 ± 0.5

Table B.1: Python and Vivado C++ Synthesis results for 8 Neurons per layer and training disabled.

	Number of Layers			
	1	4	7	10
BRAM_18K	20 (3%)	68 (10%)	116 (17%)	164 (25%)
DSP48E	35 (5%)	41 (6%)	47 (7%)	53 (8%)
FF	7644 (3%)	14697 (7%)	21684 (10%)	28671 (14%)
LUT	8437 (8%)	16867 (16%)	25220 (24%)	33470 (33%)
Latency	41836	165187	288538	411889
Interval	41836	165187	288538	411889
Estimated Clock (ns)	3.5 ± 0.5	3.5 ± 0.5	4.087 ± 0.5	3.907 ± 0.5

Table B.2: Python and Vivado C++ Synthesis results for 64 Neurons per layer and training disabled.

	Number of Layers			
	1	4	7	10
BRAM_18K	44 (6%)	164 (25%)	284 (43%)	404 (62%)
DSP48E	35 (5%)	41 (6%)	47 (7%)	53 (8%)
FF	7675 (3%)	14806 (7%)	21871 (10%)	28936 (14%)
LUT	8461 (8%)	16945 (16%)	25352 (25%)	33656 (33%)
Latency	165484	657859	1150234	1642609
Interval	165484	657859	1150234	1642609
Estimated Clock (ns)	3.5 ± 0.5	3.5 ± 0.5	4.087 ± 0.5	3.907 ± 0.5

Table B.3: Python and Vivado C++ Synthesis results for 128 Neurons per layer and training disabled.

	Number of Layers			
	1	4	7	10
BRAM_18K	16 (2%)	49 (7%)	82 (12%)	115 (17%)
DSP48E	39 (6%)	57 (9%)	75 (12%)	93 (15%)
FF	10271 (5%)	23759 (11%)	37216 (18%)	50704 (25%)
LUT	10605 (10%)	24846 (24%)	39109 (38%)	52940 (52%)
Latency	1145	5998	10784	15569
Interval	1145	5998	10784	15569
Estimated Clock (ns)	3.5 ± 0.5	4.584 ± 0.5	4.625 ± 0.5	4.153 ± 0.5

Table B.4: Python and Vivado C++ Synthesis results for 8 Neurons per layer and training enabled.

	Number of Layers			
	1	4	7	10
BRAM_18K	43 (6%)	172 (26%)	301 (46%)	430 (66%)
DSP48E	39 (6%)	57 (9%)	75 (12%)	93 (15%)
FF	10164 (5%)	23454 (11%)	36713 (18%)	50003 (24%)
LUT	10695 (10%)	25242 (24%)	39811 (39%)	53948 (53%)
Latency	54517	342288	625960	909631
Interval	54517	342288	625960	909631
Estimated Clock (ns)	3.5 ± 0.5	4.584 ± 0.5	4.625 ± 0.5	4.153 ± 0.5

Table B.5: Python and Vivado C++ Synthesis results for 64 Neurons per layer and training enabled.

	Number of Layers			
	1	4	7	10
BRAM_18K	115 (17%)	532 (81%)	**949 (146%)**	**1366 (210%)**
DSP48E	39 (6%)	57 (9%)	75 (12%)	93 (15%)
FF	10229 (5%)	23744 (11%)	37228 (18%)	50743 (25%)
LUT	10735 (10%)	25426 (25%)	40139 (39%)	54420 (53%)
Latency	215349	1364048	2496360	3628671
Interval	215349	1364048	2496360	3628671
Estimated Clock (ns)	3.5 ± 0.5	4.584 ± 0.5	4.625 ± 0.5	4.153 ± 0.5

Table B.6: Python and Vivado C++ Synthesis results for 128 Neurons per layer and training enabled.

	Number of Layers			
	1	4	7	10
BRAM_18K	9 (1%)	24 (3%)	39 (6%)	54 (8%)
DSP48E	27 (4%)	30 (5%)	33 (5%)	36 (6%)
FF	7486 (3%)	12832 (6%)	18181 (8%)	23530 (11%)
LUT	9789 (9%)	17589 (17%)	25341 (24%)	33173 (32%)
Latency	189	468	747	1026
Interval	189	468	747	1026
Estimated Clock (ns)	3.5 ± 0.5	3.5 ± 0.5	3.5 ± 0.5	3.5 ± 0.5

Table B.7: Python and Vivado C++ Synthesis results for fixed16, 8 Neurons per layer and training disabled.

	Number of Layers			
	1	4	7	10
BRAM_18K	16 (2%)	49 (7%)	82 (12%)	115 (17%)
DSP48E	27 (4%)	30 (5%)	33 (5%)	36 (6%)
FF	7632 (3%)	13896 (6%)	20160 (9%)	26424 (13%)
LUT	9898 (9%)	18235 (17%)	26524 (26%)	34893 (34%)
Latency	4392	17112	29832	42552
Interval	4392	17112	29832	42552
Estimated Clock (ns)	3.5 ± 0.5	3.5 ± 0.5	3.5 ± 0.5	3.5 ± 0.5

Table B.8: Python and Vivado C++ Synthesis results for fixed16, 64 Neurons per layer and training disabled.

	Number of Layers			
	1	4	7	10
BRAM_18K	28 (4%)	100 (15%)	172 (26%)	244 (37%)
DSP48E	27 (4%)	30 (5%)	33 (5%)	36 (6%)
FF	7660 (3%)	13906 (6%)	20152 (9%)	26398 (13%)
LUT	9922 (9%)	18268 (18%)	26566 (26%)	34944 (34%)
Latency	16872	66840	116808	166776
Interval	16872	66840	116808	166776
Estimated Clock (ns)	3.5 ± 0.5	3.5 ± 0.5	3.5 ± 0.5	3.5 ± 0.5

Table B.9: Python and Vivado C++ Synthesis results for fixed16, 128 Neurons per layer and training disabled.

	Number of Layers			
	1	4	7	10
BRAM_18K	14 (2%)	41 (6%)	68 (10%)	95 (14%)
DSP48E	30 (5%)	45 (7%)	60 (10%)	75 (12%)
FF	8706 (4%)	18219 (8%)	27701 (13%)	37233 (18%)
LUT	11319 (11%)	24267 (23%)	37239 (36%)	50133 (49%)
Latency	453	1712	2904	4095
Interval	453	1712	2904	4095
Estimated Clock (ns)	3.5 ± 0.5	3.5 ± 0.5	3.5 ± 0.5	3.5 ± 0.5

Table B.10: Python and Vivado C++ Synthesis results for fixed16, 8 Neurons per layer and training enabled.

BRAM_18K	30 (4%)	108 (16%)	186 (28%)	264 (40%)
DSP48E	30 (5%)	45 (7%)	60 (10%)	75 (12%)
FF	9233 (4%)	20042 (9%)	30820 (15%)	41629 (20%)
LUT	11624 (11%)	25346 (24%)	39092 (38%)	52760 (52%)
Latency	17037	83498	145860	208221
Interval	17037	83498	145860	208221
Estimated Clock (ns)	3.5 ± 0.5	3.5 ± 0.5	3.5 ± 0.5	3.5 ± 0.5

Table B.11: Python and Vivado C++ Synthesis results for fixed16, 64 Neurons per layer and training enabled.

	Number of Layers			
	1	4	7	10
BRAM_18K	66 (10%)	288 (44%)	510 (78%)	732 (112%)
DSP48E	30 (5%)	45 (7%)	60 (10%)	75 (12%)
FF	9296 (4%)	20318 (10%)	31309 (15%)	42331 (20%)
LUT	11664 (11%)	25530 (25%)	39420 (38%)	53232 (52%)
Latency	66701	330538	577988	825437
Interval	66701	330538	577988	825437
Estimated Clock (ns)	3.5 ± 0.5	3.5 ± 0.5	3.5 ± 0.5	3.5 ± 0.5

Table B.12: Python and Vivado C++ Synthesis results for fixed16, 128 Neurons per layer and training enabled.

Bibliography

(1) H. B. Demuth, M. H. Beale, O. De Jess, and M. T. Hagan, *Neural Network Design*, 2nd ed. USA: Martin Hagan, 2014.

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

(3) A. L. Cauchy, "Méthode générale pour la résolution de systèmes d'équations simultanées," *Compte rendu des séances de l'académie des sciences*, p. 536–538, Jul. 1847.

(4) B. Widrow and M. E. Hoff, "Adaptive Switching Circuits," in *1960 IRE WESCON Convention Record, Part 4*. New York: IRE, 1960, pp. 96–104.

(5) P. Werbos, "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Science," Ph.D. dissertation, Harvard University, Cambridge, MA, USA, Jan 1974.

(6) D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Representations by Back Propagating Errors," *Nature*, vol. 323, pp. 533–536, 10 1986.

(7) D. Parker, *Learning-logic: Casting the Cortex of the Human Brain in Silicon*, ser. Technical report: Center for Computational Research in Economics and Management Science. Massachusetts Institute of Technology, Center for Computational Research in Economics and Management Science, 1985.

(8) Y. LeCun, "Une procédure d'apprentissage pour réseau a seuil asymmetrique (a Learning Scheme for Asymmetric Threshold Networks)," in *Proceedings of Cognitiva 85*, Paris, France, 1985, pp. 599–604.

(9) D. E. Rumelhart, J. L. McClelland, and C. PDP Research Group, Eds., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. Cambridge, MA, USA: MIT Press, 1986.

(10) G. W. Leibniz, "Memoir using the chain rule," 1676, (Cited in TMME 7:2&3 pp. 321-332, 2010).

(11) G. de l'Hôpital, "Analyse des infiniment petits, pour l'intelligence des lignes courbes," 1696, Paris: L'Imprimerie Royale.

(12) M. Z. Alom, T. M. Taha, C. Yakopcic, S. Westberg, M. Hasan, B. C. V. Esesn, A. A. S. Awwal, and V. K. Asari, "The history began from alexnet: A comprehensive survey on deep learning approaches," *CoRR*, vol. abs/1803.01164, 2018. (Online). Available: http://arxiv.org/abs/1803.01164

(13) Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

(14) A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. USA: Curran Associates Inc., 2012, pp. 1097–1105. (Online). Available: http://dl.acm.org/citation.cfm?id=2999134.2999257

(15) K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014. (Online). Available: http://arxiv.org/abs/1409.1556

(16) C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014. (Online). Available: http://arxiv.org/abs/1409.4842

(17) K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. (Online). Available: http://arxiv.org/abs/1512.03385

(18) M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. (Online). Available: https://www.tensorflow.org/

(19) R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning."

(20) A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

(21) Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

(22) J. Decaluwe, "Myhdl: A python-based hardware description language," *Linux J.*, vol. 2004, no. 127, pp. 5–, Nov. 2004. (Online). Available: http://dl.acm.org/citation.cfm?id=1029015.1029020

(23) Myhdl: Design hardware with python. (Online). Available: https://http://www.myhdl.org/

(24) Xilinx Inc., "Vivado Design Suite User Guide: High-Level Synthesis - UG902 (v2018.2)," Jul. 2018. (Online). Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug902-vivado-high-level-synthesis.pdf

(25) NVIDIA Corporation Inc., "NVIDIA TESLA V100 GPU ARCHITECTURE," Aug. 2017. (Online). Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

(26) Graphcore. (Online). Available: https://www.graphcore.ai/

(27) L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990. (Online). Available: http://doi.acm.org/10.1145/79173.79181

(28) Xilinx Inc., "DS890 (v3.6) - UltraScale Architecture and Product Data Sheet: Overview," 2018. (Online). Available: https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf

(29) "Stratix 10 GX/SX Device Overview," 2018. (Online). Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/s10-overview.pdf

(30) Xilinx Inc., "UG573 (v1.9) - UltraScale Architecture Memory Resources," 2018. (Online). Available: https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf

(31) ——, "WP477 (v1.0) - UltraRAM: Breakthrough Embedded Memory Integration on UltraScale+ Devices," 2016. (Online). Available: https://www.xilinx.com/support/documentation/white_papers/wp477-ultraram.pdf

(32) Intel Corporation, "Intel Xeon Processor Scalable Family Datasheet, Volume One: Electrical," May 2018. (Online). Available: https://www.intel.com/content/www/us/en/processors/xeon/scalable/xeon-scalable-datasheet-vol-1.html

(33) Xilinx Inc., "WP505 (v1.0) - Versal: The First Adaptive Compute Acceleration Platform (ACAP)," Oct. 2018. (Online). Available: https://www.xilinx.com/support/documentation/white_papers/wp505-versal-acap.pdf

(34) ——, "WP506 (v1.0.2) - Xilinx AI Engines and Their Applications," Oct. 2018. (Online). Available: https://www.xilinx.com/support/documentation/white_papers/wp506-ai-engine.pdf

(35) ——, "DS950 (v1.0) - Versal Architecture and Product Data Sheet: Overview," Oct. 2018. (Online). Available: https://www.xilinx.com/support/documentation/data_sheets/ds950-versal-overview.pdf

(36) Maxeler Technologies, Inc. (Online). Available: http://maxeler.com/

(37) Maxeler Technologies, Inc., "MaxCompiler, White paper," Feb. 2011. (Online). Available: https://www.maxeler.com/media/documents/MaxelerWhitePaperMaxCompiler.pdf

(38) ——, "Programming MPC Systems, White paper," Jun. 2013. (Online). Available: https://www.maxeler.com/media/documents/MaxelerWhitePaperProgramming.pdf

(39) ——, "OpenSPL: Revealing the Power of Spatial Computing," Dec. 2013. (Online). Available: http://www.openspl.org/wp-content/uploads/OpenSPL-WP1.pdf

(40) D. H. Noronha, B. Salehpour, and S. J. E. Wilton, "Leflow: Enabling flexible FPGA high-level synthesis of tensorflow deep neural networks," *CoRR*, vol. abs/1807.05317, 2018. (Online). Available: http://arxiv.org/abs/1807.05317

(41) A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: High-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 33–36. (Online). Available: http://doi.acm.org/10.1145/1950413.1950423

(42) J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, and Z. Wu, "Fast inference of deep neural networks in fpgas for particle physics," 2018.

(43) S. I. Venieris, A. Kouris, and C.-S. Bouganis, "Toolflows for mapping convolutional neural networks on fpgas: A survey and future directions," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 56:1–56:39, Jun. 2018. (Online). Available: http://doi.acm.org/10.1145/3186332

(44) Jinja 2. (Online). Available: http://jinja.pocoo.org/