MASTER OF SCIENCE
IN ENGINEERING

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts
Western Switzerland

Master of Science HES-SO in Engineering
Avenue de Provence 6
CH-1007 Lausanne

Master of Science HES-SO in Engineering

Orientation: Technologies de l'Information et la Communication (TIC)

AUTOMATIC GENERATION OF FPGA-BASED ACCELERATORS

Fait par

Lucas ELISEI

Sous la direction de
Prof. Alberto Dassatti
Dans l'institut REDS à l'HEIG-VD

Expert externe: Andrea Guerrieri, EPFL

Lausanne, HES-SO//Master, Février 2020

Abstract

In this work, our aim was to study the available technologies and toolchains for generating hardware accelerators automatically, ideally without the developer's intervention. We searched for a solution to minimize the hardware knowledge required to produce such an accelerator under normal circumstances. The main purpose of this project was to critically analyze the use of a framework (or toolchain) for generating hardware accelerators automatically, or almost.
As an example, we chose to target a Machine Learning accelerator that classifies hand-written digits.

As a first step, we listed all available technologies and compared them among several criteria such as their update frequency, their supported inputs and outputs. From this list, we reviewed each of the frameworks and HeteroCL was found to be the most interesting one to use it for the rest of this project. We then described the Machine Learning model that we used for the classification task. Then, we presented how to use HeteroCL and critically discussed its flaws and qualities.

Finally, we compared the resulting output of HeteroCL with other available solutions. The first solution was a software solution running on a CPU. The second solution was an hardware SystemVerilog module developed by a human. The third solution was the hardware accelerator generated with HeteroCL. The fourth and final solution was also the HLS code generated with the framework, but this time combined with manually code transformations.

We showed that out of the four solutions, the human-developed hardware accelerator offered the best performances. It could achieve inference in about 1 milliseconds where the software solution could do it in 7.386 milliseconds. The solutions generated with HeteroCL were less performant. The version that had been optimized with manual code transformations could perform inference in 10.858 milliseconds, and the unoptimized one in 37.866 milliseconds.

In conclusion, despite their novelty, technologies for automatically generating hardware accelerators already offer acceptable performances. They could evolve for the best in the near future and propose generated hardware solutions that compete with software and human-developed hardware solutions.

Table of contents

1 Introduction

Modern computational problems require a growing amount of computational power, thus increasing the electrical power consumption. Nowadays, Information and Communications Technologies (ICT) represent about 8% of global electricity usage and could consume up to 20% of global electricity by 2025 [1].

A famous modern computing application includes Artificial Neural Networks (ANNs). These computing systems are vaguely inspired by the biological Neural Networks that constitute animal brains. An Artificial Neural Network is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. Neurons can communicate with each other through connections that deliver signals, similarly to the synapses of a biological brain. An artificial neuron receiving a signal can then process it and further signal the neurons connected to it.
Globally, those networks can therefore serve to model complex patterns and prediction problems, given specific input information.

Recent research has showed Artificial Neural Networks capability to perform successfully well in several domains among which biology [2], image processing [3], self-driving cars [4] and many others. Most of modern NN architectures include convolutional layers and thus are called Convolutional Neural Networks (CNNs). These CNNs are specialized in analyzing visual imagery. Furthermore, there is an important need of deploying CNNs on embedded systems and mobile devices for applications such as autonomous cars [4] and medical devices [5], [6], which demand real-time and high-accuracy object recognition.

Convolutional Neural Networks, and more generally Artificial Neural Networks, consume a lot of power and require high computational loads. However, embedded systems and mobile devices usually are limited in terms of resources. They often use a battery rather than being plugged into a power source. Moreover, they are equipped with a low-power, low-frequency processor. Finally, they commonly embeds small memory amount. These limitations make the use of embedded systems for Artificial Neural Networks implementation quite difficult.

In order to reduce power consumption and accelerate computational loads, an approach could be to use specific hardware. Two possible accelerators are Graphics Processing Units (GPUs) and Field-Programmable Gate Arrays (FPGAs), which can often achieve better performance than Central Processing Units (CPUs) on certain workloads [7], [8], [9], [10]. GPUs are designed to perform floating-point operations and run software while FPGAs are integrated circuits that can be customized for a specific application. Since hardware is faster and more power-efficient than software for specific applications, FPGAs are generally a better choice than GPUs [7], [11]. Additionally, FPGAs offer true parallel processing, unlike CPUs and GPUs, and high-speed data processing.

However, hardware programming, and more specifically FPGA programming, is less user-friendly and more unwieldy than software programming. It necessitates advanced knowledge in Hardware Description Languages (HDLs), such as VHDL or SystemVerilog, as well as knowing precisely the target FPGA development board architecture. Moreover, hardware programming requires specialized tools (*e.g.* Xilinx's Vivado Design Suite [12], Intel's Quartus Prime [13]). Finally, developing hardware accelerators requires the developer to know perfectly the target of the accelerator and its application.
These limitations can be discouraging thus limiting the development of hardware accelerators for modern computing problems, like Artificial Neural Networks.

1.1 Aim and objectives

In this work, our aim is to study the available technologies and toolchains for generating hardware accelerators automatically, ideally without the developer's intervention. We are looking for a solution that seeks to reduce or even – ideally – eliminate the hardware knowledge required to produce such an accelerator under normal circumstances.

The main purpose of this project is to critically analyze the use of a framework (or toolchain) for generating hardware accelerators automatically, or almost. This critic will discuss the ease of use and the performances in terms of resources utilization and time.

These performances will be compared among:

- A solution running on a CPU
- A solution developed by a – human – programmer, with a HDL
- A solution generated with the framework
- The same solution as above, but optimized with code transformations manually applied after generation

As a testbench, the idea was to create an FPGA-based system which consists of acquiring image data from a camera connected to an FPGA board and streaming out the data through an High-Definition Multimedia Interface (HDMI) port, also present on the board.
The data passes through a Machine Learning module which performs hand-written digits recognition. This module was generated by a solution chosen among different frameworks. These several frameworks are presented later in the state of the art section, at the end of which we discuss which framework is the best candidate for this work and why. The system is described in details in subsection 1.2.

Once the hardware system had been set up, we were able create and include the Machine Learning hardware accelerator that performs hand-written digits recognition.
In order to find the framework that best met our needs and to then integrate the results, we followed the subsequent workflow:

1. **Review existing frameworks**
   We first reviewed frameworks that allow to create an hardware accelerator from an high-level programming language (*e.g.* Python) to an Hardware Description Language (*e.g.* VHDL, SystemVerilog). Out of all the framework, one was chosen as the best candidate and was then used for the rest of this project.
2. **Define, train and test a Machine Learning model**
   Before synthesizing the model into an accelerator, we defined the model used to perform hand-written digits recognition. Once we defined its architecture, we then had to prepare the training data and train the model. Finally, we tested its accuracy.
3. **Use the framework**
   The framework previously chosen in point 1. to generate an hardware accelerator of the model described in point 2. Every step is presented and documented. We completed this step with a critical analysis of the framework's generated code and utilization.
4. **Manual optimizations**
   Some optimizations were added manually to the generated accelerator, to see what tradeoffs could be reached. This resulted in a newly optimized version of the accelerator that had to be compared with the other solutions.
5. **Analyze results**
   The performance of the generated accelerator was then compared with the other solutions. The comparison was performed between a solution running on a CPU, a solution developed by a human programmer using an HDL and finally, the solution generated by the framework, and finally the solution with manual optimizations applied after-hand.
6. **Conclude**
   We highlighted the different outcomes of this project, while presenting the possible future works that can follow.

1.2 System description

This subsection explains how the system, which was used as a testbench for the accelerator generated later in this work, has been put in place.

For acquiring image data, the Pcam 5C module [14] is used. It embeds the Omnivision OV5640 5 MP color image sensor [15]. Data is transferred over a dual-lane MIPI CSI-2 interface and offers common video streaming formats such as 1080p at 30 frames per second and 720p at 60 frames per second. The camera module is connected to the FPGA board via a 15-pin Flat-Flexible Cable (FFC) which is pin-compatible with a Pcam connector.

Initially, we tried to deploy the system on a Xilinx Zynq-7000 SoC ZC702 Evaluation Kit [16]. Since the ZC702 board doesn't have a Pcam connector, we had to interface the camera module to the board by using a FMC Pcam adapter [17]. However, we had so many problems for communicating with the camera module from the board – in particular with the Inter-Integrated Circuit (I^2C) protocol – that we preferred to change the development board for a Zybo Z7-20 [18].
The reason for this change was that Digilent, the Pcam 5C module constructor, made a demonstration project available online [19]. This project targets the Zybo Z7-20 as the FPGA development board. The board's main features are presented in Appendix A.

Once data is successfully streamed from the camera module, the data must be stored into the board's Synchronous Dynamic Random Access Memory (SDRAM). To do so, we use a Video Direct Memory Access (VDMA) [20]. It provides high-bandwidth direct memory access between the board's SDRAM memory and its peripherals, such as the Pcam 5C camera module. With the VDMA set up, we simply have to stream out the data to the HDMI port.

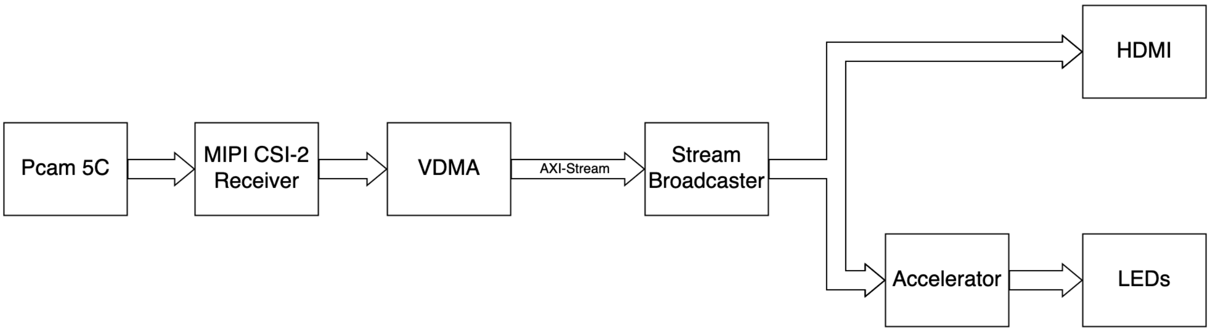


Figure 1: Block design of the testbench system.

2 State of the art

This section reviews existing frameworks that aim at translating a Machine Learning model to synthesizable hardware. This corresponds to the first objective of this project.
This review also serves as a state of the art, since it describes the most recent state in the development of this technology.

The most interesting candidate among those frameworks was chosen and used for the rest of this project. For each tool, a small description as well as the tool's objectives are presented and the following criteria are evaluated:

1. **Maintenance and Update frequency**
   It is important to know if the tool is under active development and who is responsible for the development. This also allows us to know whether we can expect potential errors to be corrected quickly, and also to have an idea of the current stage of development of the tool.
2. **License**
   The license informs us of the conditions under which the tool may be used, distributed or modified. This criteria is crucial in this project since we might want to modify the tool so it better suits our needs.
3. **Which input(s) the tool accepts**
   Does it use already existing ML description framework(s) or its own? It is preferred if the framework accepts already existing description, so it saves us the trouble to discover and learn another description framework.
4. **Which output(s) the tool targets**
   This criteria informs on the versatility and flexibility of the tool. More flexible tools will be easier to use. Therefore, the more outputs a tool supports, the more will it be an interesting candidate.
5. (Optional) **Additional remarks**

A table summarizing this state of the art is available in subsection 2.11.

2.1 DnnWeaver 2.0

DnnWeaver is an open-source framework for accelerating Deep Neural Networks (DNNs) on FPGAs. It aims to bridge the semantic gap between the high-level specifications of DNN models used by programmers and FPGA acceleration [21].

Maintenance and Update frequency

DnnWeaver is maintained by a team of six developers who are part of the Georgia Institute of Technology.

The project doesn't seem to be updated frequently. Based on its Github repository [22], some commits occurred in April 2019 but the previous ones were made in November 2018.

The fact that the project is maintained by a scholar team probably means that they have other works besides DnnWeaver and it is reflected on the commits history.

License

According to its website, DnnWeaver is licensed under the Apache License 2.0 [23], which requires preservation of the copyright notice and disclaimer but allows the user of the software the freedom to use the software for any purpose, to distribute it, to modify it and to distribute modified versions of the software, under the terms of the license, without concerns for royalties.

Input

The programmer specifies the Deep Neural Network using Caffe format.

Output
Given the input, the framework automatically generates the accelerator SystemVerilog code specialized for the given network, using hand-optimized SystemVerilog templates (included in DnnWeaver).

As of January 2020, the implemented layers are Convolution, Rectified Linear Unit (ReLU), Fully Connected Layer (InnerProduct), Pooling and Local Response Normalisation (LRN).

Remarks
The available documentation only concerns the version 1.0 of the tool, which makes it difficult to gather valid information about the current version.

A colleague has used DnnWeaver in the past and, according to him, the framework uses a mix of *Python2.7* and *Python3.6*, raising errors during utilization. The only way to fix this is to manually update the incompatible files to *Python3.6*.

Additionally, it seems that adding an unknown FPGA to the framework as an output target requires a lot of efforts.

2.2 FPGA Caffe

FPGA Caffe is a custom version of Caffe with FPGA kernels. The kernels use custom-precision floating-point arithmetic to save area and improve the throughput of the kernels, while also allowing for experimentation with different floating-point precisions and rounding for training and inference with CNNs [24].

Maintenance and Update frequency
FPGA Caffe is maintained by a team of six developers who are part of University of Toronto and University of Guelph, both based in Ontario, Canada.

The project has not been updated since December 2017, making it quite obsolete.

License
FPGA Caffe is released under the 2-Clause BSD License [25], which allows almost unlimited freedom with the software as long as the modified versions of the software include the same license.

Input
The programmer specifies the Deep Neural Network using Caffe format.

Output
The kernels target the Xilinx SDAccel OpenCL environment, thus only Xilinx FPGAs are supported.

Remarks
Only a few layers are implemented. There are forward and backward convolutions, forward and backward ReLUs, forward and backward MaxPooling, and forward and backward InnerProduct.

2.3 NVDLA

The NVIDIA Deep Learning Accelerator (NVDLA) is a free and open architecture that promotes a standard way to design Deep Learning inference accelerators [26]. With its modular architecture, NVDLA is scalable, highly configurable and designed to simplify integration and portability. The hardware supports a wide range of IoT devices. According to their website, all of the software, hardware and documentation *will* be available on GitHub [27].

Maintenance and Update frequency

The project is maintained by an internal team so it's quite professional.

NVDLA is divided in two parts: software and hardware.
The hardware part (available under the `hw` Github repository) has not been updated since April 2018.
The software part (available under the `sw` Github repository) has multiple commits in September 2019 but nothing between September 2019 and April 2019 as well as between April 2019 and August 2018 (at least publicly).
It seems that they commit changes only when a major development milestone is released.

License

NVDLA is delivered as an open-source project under the NVIDIA Open NVDLA License [28]. This license allows us to modify the tool as we please.

Input

The programmer specifies the Deep Neural Network using Caffe format.

Output

NVDLA supports two sample platforms: simulation and FPGA. These platforms are provided to observe, evaluate and test NVDLA in a minimal System-on-Chip (SoC).
The simulation platform is based on GreenSocs QBox [29]. A QEMU CPU model (`x86` or `ARMv8`) is combined with the NVDLA SystemC model to provide a register-accurate system for quick development and debugging.
The FPGA platform provides a synthesizable example of instantiating NVDLA in a real design. The FPGA model is intended for inference only, no effort has been made to optimize cycle time, design size, power consumption or performance.

The FPGA platform is Xilinx-based, thus only Xilinx FPGAs are supported.

Remarks

The documentation is well-structured and precise.

The source code is quite closed yet. Commits are pushed whenever there is a major version. And even if NVIDIA claims that the project is open-source, it seems that some code (*e.g.* the compiler) will not be released. This already causes problems because, according to Toshiba [30], some errors are raised during compilation, and we don't have information on what is making compilation fail.
However, Open Neural Network Compiler (ONNC) [31], a retargetable compilation framework, has a NVDLA backend that can compile a model into an executable NVDLA loadable file.

It also seems that people are having a hard time testing NVDLA on FPGAs [32].

The team seems to be quite responsive: we wrote them an email to get more information about the compiler source code and they replied within hours.

2.4 hls4ml

hls4ml is a package for Machine Learning inference in FPGAs. It translates traditional open-source Machine Learning models into High-Level Synthesis (HLS) language that can be configured according to the output platform [33].

Maintenance and Update frequency

hls4ml is maintained by different people around the world. Some come from the CERN, others from MIT, making it more of a community project than something professional.

The project was updated quite frequently, about 1 commit was made per week until September 2019. Since, there has been no commit made. Project might be discontinued.

License

hls4ml is licensed under the Apache License 2.0 [23].

Input
Neural Network can be specified with Keras/Tensorflow or PyTorch.

Output
Given the input, the framework generates an HLS project that can be used to produce an IP core which can be plugged into more complex designs or be used to create a kernel for CPU co-processing.

As of January 2020, only Multi-Layer Perceptron (MLP) and `Conv1D` and `Conv2D` architectures are supported.

Remarks
Unfortunately, the project is not well-documented.

2.5 TVM

TVM is an open deep learning compiler stack for CPUs, GPUs and specialized accelerators (FPGAs). It aims to close the gap between the productivity-focused deep learning frameworks, and the performance- or efficiency-oriented hardware backends [34].

Maintenance and Update frequency
The TVM stack began as research project at the SAMPL group of University of Washington. The project is now driven by an open-source community involving multiple industries and academic institutions.

The project is under active development: several commits are pushed every day.

License
The TVM stack is licensed under the Apache License 2.0 [23].

Input
Neural Network can be specified in Keras, MXNet, PyTorch, Tensorflow, CoreML and DarkNet.

Output
Given the input, TVM compiles the Deep Learning (DL) models into deployable modules on diverse hardware backends such as CPUs, GPUs and FPGAs.

Remarks
The documentation is really exhaustive and up-to-date.

A lot of tutorials are available for every thing that TVM can achieve.

A forum is available where users can ask questions which are then rapidly answered.

Versatile Tensor Accelerator (VTA) is an extension of the TVM framework that includes drivers, a Just-in-Time (JIT) runtime and an optimizing compiler stack. VTA offers a micro-architecture on which compiled TVM modules can be run. Initially, only Xilinx FPGAs were supported. However, a pull request adding Intel FPGA and Chisel support has been merged on June 5, 2019 [35].

TVM is now part of the Apache Incubator [36]. In December 2019, the "TVM and Deep Learning Compilation Conference" [37] has been organized by Apache. A lot of talkings (*e.g.* Microsoft, ARM, Xilinx) are worth a look. We believe that such an event promise a bright future for this project.

This framework has already been used in the past so we are quite comfortable with its use.

2.6 LeFlow

LeFlow is a tool that relies on LegUp [38] to map numerical computation models written in Tensorflow to synthesizable hardware [39]. It bridges Google's XLA compiler and LegUp high-level synthesis to automatically generate a SystemVerilog module.

Maintenance and Update frequency
LeFlow is maintained by three people who work at The University of British Columbia.

Even if the last update was made in February 2019, the previous one was four months before, thus we assume that it is not frequently updated.

License
LeFlow is licensed under the MIT License [40] which has only one restriction: if the project is reused within proprietary software, all copies of the licensed software must include a copy of the MIT License.

Input
Neural Network models must be specified with a customized version of Tensorflow.

Output
Since LeFlow relies on LegUp, it supports the output that LegUp supports. Those are Intel FPGAs.

Remarks
Documentation is inexistent.

Some simple examples are included in the repository.

The maintainer is quite responsive: we wrote him an email to better understand the project's structure and he replied within hours.

2.7 nGraph

nGraph is an open-source graph compiler for artificial Neural Networks. The nGraph Compiler stack provides an inherently efficient graph-based compilation infrastructure designed to be compatible with many upcoming integrated circuits while also unlocking a massive performance boost on any existing hardware targets [41].

Maintenance and Update frequency
nGraph is maintained by an artificial intelligence software company called Nervana Systems (acquired by Intel in August 2016) [42].

The Github repository is updated with several commits every day [43].

License
nGraph is licensed under the Apache License 2.0 [23].

Input
As of January 2020, nGraph takes Tensorflow 1.12, MXNet 1.3 and ONNX 1.3 as inputs for model description.

Output
nGraph currently supports Intel CPUs, Intel Neural Network Processor, NVIDIA CUDA GPUs and AMD GPUs as output.

FPGAs are set to be fully supported "in the near future" [44].

Remarks
The documentation seems to be very qualitative.

Since the project is maintained by an Intel company, it might be possible that only Intel hardware (CPUs, GPUs and FPGA) will be supported.

2.8 HeteroCL

HeteroCL is a programming infrastructure composed of a Python-based Domain-Specific Language (DSL) and a compilation flow. The HeteroCL DSL provides a clean abstraction that decouples algorithm specification from three important types of hardware customization in compute, data types and memory architectures.

Maintenance and Update frequency

The project is maintained by a team of developers of the Cornell University.

Commits are pushed frequently on the HeteroCL Github repository [45].

License

HeteroCL is licensed under the Apache License 2.0 [23].

Input

HeteroCL takes any format as input as long as weights are loadable from the input model.

Output

Cloud (AWS), Xilinx and Intel FPGAs are supported. CPUs are also supported.

Remarks

The documentation seems to be good although not complete. We assume it will improve with future releases.

It is possible to follow development roadmap by going on their respective pull request on the HeteroCL Github repository [45].

We already used this framework in the past so we are comfortable with its use.

2.9 ESP

Embedded Scalable Platforms (ESP) is an open-source platform for heterogeneous System-on-Chip (SoC) design and prototype on FPGA. It provides a flexible tile-based architecture built on a multi-plane Network-on-Chip (NoC) [46].
In addition to the architecture, ESP provides users with templates and scripts to create new accelerators from SystemC, Chisel, and C. The ESP design methodology eases the integration process by offering platform services (*i.e.* Direct Memory Access (DMA), distributed interrupt, run-time coherence selection) that hide the complexity of hardware and software integration from the accelerator designer.

Maintenance and Update frequency

ESP is maintained by the System-Level Design group at Columbia University, led by Professor Luca P. Carloni.

Commits are pushed frequently on the project's Github repository [47].

License

The project is licensed under the Apache License 2.0 [23].

Input

ESP takes SystemC, C and Keras Tensorflow as input.

Output

It generates FPGA accelerators as modules for the ESP micro-architecture, like TVM does.

Remarks

The documentation is not yet complete. A lot of tutorials are still missing, making it difficult to use this project. However, since the project seems to be quite active, we believe documentation will quickly be completed.

This framework has been put online at the beginning of January 2020, thus being too recent to be added to the list of potential candidates for this project.

2.10 Comparison

Among the listed existing frameworks, we based ourselves on the previously presented criteria to choose the most suitable candidate for this project.

Among these criteria, we decided to put a higher weight on the update frequency, the supported outputs and their reliability.

The selection will be based on several criteria: we want to focus on the update frequency and the supported outputs. The *License* criterion has been put aside because every framework is under a license allowing a lot of freedom (except maybe for NVDLA).
Also, the *Supported inputs* criterion has not been taken into account since input frameworks were found very similar and switching from one to another is quite feasible.

Supported outputs
All frameworks listed above support FPGAs as an output, except for nGraph. Thus, we eliminated it from the list of interesting candidates.

Maintenance and update frequency
FPGA Caffe has not been updated for more than a year, so it was obviously not an interesting candidate. Unfortunately, we could not retain ESP as a candidate either, because it has been released just days before the end of this work. But it might be a truly interesting solution to keep track of in the future.

Reliability
Because its compiler is still close-source and contains bugs, NVDLA didn't make it to the list of interesting candidates.
DnnWeaver 2.0 seems to be broken: some files need manual updates to be compatible with *Python3.6* so it has been pulled out of the list as well.
In one of our previous works, "FPGA-based Accelerator for Machine Learning Inference" [48], we tried to use LeFlow to synthesize a simple Multi-Layer Perceptron without any success. Our previous tests were not very conclusive, thus we preferred to rule out LeFlow as a possible candidate.

Produced output
This lefts us with three candidates: hls4ml, TVM and HeteroCL. hls4ml and HeteroCL are similar in the way that they produce HLS code that can be synthesized for both Intel and Xilinx FPGAs, while TVM produces modules that must be run on the VTA micro-architecture. This creates two groups: *HLS* and *Micro-architecture*.
The only and thus best candidate for the *Micro-architecture* group is TVM. The other two candidates are part of the *HLS code* group. Between HeteroCL and hls4ml, we believe that HeteroCL is the most suitable choice because it is well-documented and it is more frequently updated than hls4ml.
Moreover, HeteroCL has already been used in the "GNU Radio OOT module for DNN activity" [49] project, so we also are more comfortable using it.

Finally, between TVM and HeteroCL, we preferred to go with HeteroCL. Since we aim at creating an FPGA-based system, a synthesizable module is preferred over a module that can only run with a dedicated micro-architecture, thus eliminating TVM from the possible candidates.

The choice of HeteroCL as our chosen framework being made, we were able to move on to the second objective of this paper: Define, train and test a Machine Learning model.

2.11 Summary

Framework	Update frequency	Supported inputs	Supported outputs	Comments
DnnWeaver 2.0	One update every year (last in April 2019)	Caffe	SystemVerilog	Obsolete documentation, some files need to be updated manually for the framework to run.
FPGA Caffe	Not updated since December 2017	Caffe	Xilinx FPGAs	None.
NVDLA	One update every year (last in April 2019)	Caffe	CPUs, Xilinx FPGAs	Documentation is good. Project is *relatively* open-source. Seems that people are having a hard time deploying on FPGAs.
hls4ml	1 commit per week until September 2019, no commits since	Keras, PyTorch	SystemVerilog	Documentation last updated a year ago.
TVM	Several commits per day	Keras, MXNet, PyTorch, Tensorflow, CoreML, DarkNet	Modules for VTA micro-architecture	Good documentation with tutorials. A forum is available for asking questions. Project is part of the Apache Incubator. This framework has already been used for past projects so we are comfortable using it.
LeFlow	Last updated in February 2019 and November 2018	Tensorflow	Intel FPGAs	No documentation. Some examples are included.
nGraph	Several commits per day	Tensorflow, MXNet	CPUs, GPUs	FPGAs not supported yet.
HeteroCL	Several commits per week	Any as long as weights are available	HLS code	This framework has already been used for past projects, so we are comfortable using it.
ESP	Several commits per week	C, SystemC, Keras	Modules for ESP micro-architecture	None.

Table 1: State of the art summary.

3 The Machine Learning model

This section describes the second objective of this project. We first describe the Machine Learning model architecture and then present how we did prepare the dataset used for training and validating the model.

3.1 The model architecture

For this work, we chose to deploy a CNN that recognizes hand-written digits. This choice was motivated by the fact that the second solution, which is taken from the "Fixed-Point Convolutional Neural Network for Real-Time Video Processing in FPGA" paper, written by Solovyev, Kustov, Telpukhov, *et al.* [50]. This paper was the reference for the *Inference developed by a – human – developer with a HDL* solution, as stated in subsection 1.1. The Figure 2 shows its detailed architecture.
This network uses the MNIST dataset [51], which is composed of 60'000 train images and 10'000 test images.

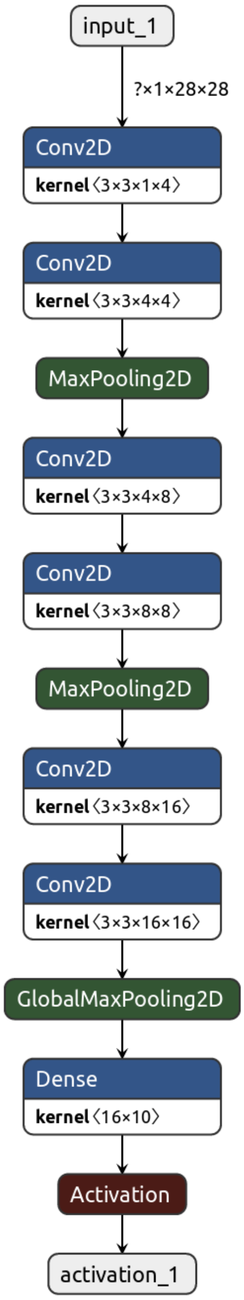
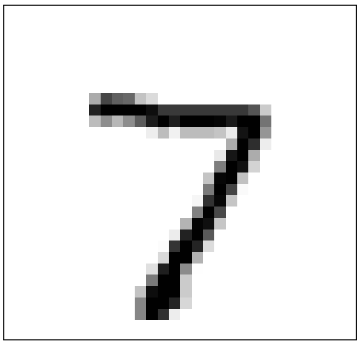


Figure 2: CNN architecture used for describing the workflow.

The model takes a grayscale (1 component per pixel) image of dimensions 28x28 as input. The data is then processed by 6 convolutional layers (each activated by the ReLU function). The last layer performs a `Dense` operation, which is finally activated by the `Softmax` function.
The output is an array of 10 values (from 0 to 9), each corresponding to the probability of being the digit in the input image.



```
[
    0.10118849, # 0
    0.07356359, # 1
    0.08777632, # 2
    0.09791587, # 3
    0.10229186, # 4
    0.08554156, # 5
    0.07353777, # 6
    0.16658820, # 7 (highest probability)
    0.09403618, # 8
    0.11756021  # 9
]
```

Figure 3: Inference example.

3.2 Preparing the dataset

We then prepared the dataset that was used to train the network. The function `load_mnist_data()` shown below does such a task.

src/python/train_nn.py

```python
def load_mnist_data(data_format='channels_first'):
    '''
    Load MNIST dataset.

    Parameters
    ----------
    data_format : str, optional
        The data format used (default is 'channels_first')

    Return
    -------
    train_x, train_y, test_x, test_y : array, array, array, array
        Training images, training labels, testing images, testing labels.
    '''
    (train_x, train_y), (test_x, test_y) = mnist.load_data()

    # Reshape data according to the specified data format
    if (data_format is 'channels_first'):
        train_x = train_x.reshape(train_x.shape[0], 1, IMG_H, IMG_W)
        test_x  = test_x.reshape(test_x.shape[0], 1, IMG_H, IMG_W)
    elif (data_format is 'channels_last'):
        train_x = train_x.reshape(train_x.shape[0], IMG_H, IMG_W, 1)
        test_x  = test_x.reshape(test_x.shape[0], IMG_H, IMG_W, 1)
    else:
        raise DataFormatException(data_format)

    # Convert to float32
    train_x = train_x.astype('float32')
    test_x  = test_x.astype('float32')

    dbg_print('train_x.shape: {}'.format(train_x.shape))
    dbg_print('test_x.shape : {}'.format(test_x.shape))

    # Convert class vectors to binary class matrices
    train_y = np_utils.to_categorical(train_y, 10)
    test_y  = np_utils.to_categorical(test_y, 10)

    # Invert images (from white on black to black on white)
    train_x = 255 - train_x
    test_x  = 255 - test_x

    return train_x, train_y, test_x, test_y
```

The line 52 loads the MNIST dataset. Train (resp. test) images are stored in the array `train_x` (resp. `train_y`) and train (resp. test) labels (*i.e.* classes) are stored in the array `train_x` (resp. `test_y`).

At lines 55 to 62, we adapt the format of the dataset. Neural Networks can have different different data layouts: NCHW or NHWC, where:

N: batch size

C: channels

H: height

W: width

Thus, if `data_format` is `channels_first` (resp. `channels_last`), NCHW (resp. NHWC) data layout will be applied on the whole dataset. Otherwise, an exception is raised. Default is NCHW.

On lines 72 and 73, we convert classes to binary representation (*e.g.* `4` becomes `0100`). This step is optional. We apply it to have the exact same network output as the one proposed by Solovyev, Kustov, Telpukhov, *et al.* [50].

Finally, lines 76 and 77 invert the MNIST colors. It is also an optional step as it depends on the data we want to train the network with. The original images provided by the MNIST dataset are white digits written on a black background. Such configuration is rare in real-life applications thus we decided to invert the colors on the whole dataset. Since MNIST images have their pixels coded with 8-bit unsigned integers, we simply have to subtract the pixel value from 255.

Figure 4: MNIST image before and after dataset preparation.

3.3 Training the network

To train the network, we used the Keras framework [52]. The first thing to do was to define and build the whole NN architecture, as described by Solovyev, Kustov, Telpukhov, *et al.* [50].

src/python/train_nn.py

```python
81  def build_model(data_format='channels_first', use_bias=False):
82      '''
83      Define network architecture.
84
85      Parameters
86      ----------
87      data_format : str, optional
88          The data format used (default is 'channels_first').
89      use_bias    : boolean, optional
90          Whether to use bias or not (default is False).
91
92      Return
93      -------
94      model       : keras.models.Model
95          The Keras model of the network.
96      '''
97      # Input is 28x28 grayscale (1 component) pixels
98      if (data_format is 'channels_first'):
99          input = Input((1, IMG_H, IMG_W))
100     elif (data_format is 'channels_last'):
101         input = Input((IMG_H, IMG_W, 1))
102     else:
103         raise DataFormatException(data_format)
104
105     conv1 = Conv2D(4, (3,3), activation='relu', padding='same', data_format=data_format, name='conv1',
        ↪  use_bias=use_bias)(input)
106     conv2 = Conv2D(4, (3,3), activation='relu', padding='same', data_format=data_format, name='conv2',
        ↪  use_bias=use_bias)(conv1)
107     pool1 = MaxPooling2D((2,2), strides=(2,2), data_format=data_format, name='pool1')(conv2)
108
109     conv3 = Conv2D(8, (3,3), activation='relu', padding='same', data_format=data_format, name='conv3',
        ↪  use_bias=use_bias)(pool1)
110     conv4 = Conv2D(8, (3,3), activation='relu', padding='same', data_format=data_format, name='conv4',
        ↪  use_bias=use_bias)(conv3)
111     pool2 = MaxPooling2D((2,2), strides=(2,2), data_format=data_format, name='pool2')(conv4)
112
113     conv5 = Conv2D(16, (3,3), activation='relu', padding='same', data_format=data_format, name='conv5',
        ↪  use_bias=use_bias)(pool2)
114     conv6 = Conv2D(16, (3,3), activation='relu', padding='same', data_format=data_format, name='conv6',
        ↪  use_bias=use_bias)(conv5)
115     pool3 = GlobalMaxPooling2D(data_format=data_format, name='pool3')(conv6)
116
117     dense1 = Dense(10, activation=None, use_bias=use_bias)(pool3)
118     output = Activation('softmax')(dense1)
119
120     model = Model(inputs=input, outputs=output)
121     model.summary(print_fn=dbg_print)
122
123     return model
```

From line 98 to line 103, the input shape is defined accordingly to the specified data layout.

The first two blocks (lines 105 to 107 and lines 109 to 111) are each composed of two 2D-Convolutional layers (with ReLU being the activation function for both) and a 2D-MaxPooling layer.

The third block (lines 113 to 115) is also composed of two 2D-Convolutional layers (with ReLU being the activation function for both) but composed of a 2D-GlobalMaxPooling layer.

Finally, a Dense layer is used to connect all neurons from the previous 2D-GlobalMaxPooling layer to the output layer. The Dense layer has the Softmax function acting as the activation function.

The execution of line 121 gives the following output:

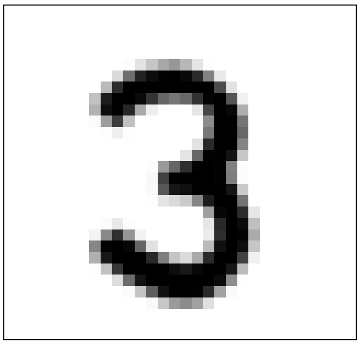
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 1, 28, 28)         0
_________________________________________________________________
conv1 (Conv2D)               (None, 4, 28, 28)         36
_________________________________________________________________
conv2 (Conv2D)               (None, 4, 28, 28)         144
_________________________________________________________________
pool1 (MaxPooling2D)         (None, 4, 14, 14)         0
_________________________________________________________________
conv3 (Conv2D)               (None, 8, 14, 14)         288
_________________________________________________________________
conv4 (Conv2D)               (None, 8, 14, 14)         576
_________________________________________________________________
pool2 (MaxPooling2D)         (None, 8, 7, 7)           0
_________________________________________________________________
conv5 (Conv2D)               (None, 16, 7, 7)          1152
_________________________________________________________________
conv6 (Conv2D)               (None, 16, 7, 7)          2304
_________________________________________________________________
pool3 (GlobalMaxPooling2D)   (None, 16)                0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                160
_________________________________________________________________
activation_1 (Activation)    (None, 10)                0
=================================================================
Total params: 4,660
Trainable params: 4,660
Non-trainable params: 0
_________________________________________________________________
```

Figure 5: Keras model summary.

This gives us more information about the output shape of each layer and their respective number of trainable parameters. The total count of trainable parameters included in the model is 4'660, which should easily fit in any FPGA design or embedded Synchronous Dynamic Random Access Memory (SDRAM).

After the network definition step came the training. This was performed by the `train_model()` function. We will not describe its code in details here but only describe what it does. The complete Python code is available in Appendix B.
The `train_model()` function first searches for an already trained model that might have been saved into a file named `mnist_weights.h5`. If this file exists, the model is loaded from there. Otherwise, the model is trained and then saved into the same file mentioned previously.
Upon completion, the function outputs the model score and accuracy:

```
Model score   : 0.07440621950239874
Model accuracy: 0.975600004196167
```



```
[
    1.1960877e-11,  # 0
    4.3369418e-13,  # 1
    7.8879259e-09,  # 2
    9.9999964e-01,  # 3 (highest)
    9.1875462e-16,  # 4
    3.9355601e-07,  # 5
    5.5602975e-16,  # 6
    3.8441872e-09,  # 7
    1.0099800e-09,  # 8
    1.9590207e-08   # 9
]
```

Figure 6: Model predictions for a non-MNIST image.

The last step in the network training was to rescale all the model weights. It is necessary if we want to use fixed-point arithmetics, which should speed-up performance. In fact, when implemented at the hardware level, floating-point calculations are slower than fixed-point calculations due to the difficulty of controlling the mantissa and the exponent of the values.

The method used for rescaling the weights is the same as the one used by Solovyev, Kustov, Telpukhov, *et al.* [50]. Basically, it consists of looking at all weights and all possible input and output values and normalizing them. The lowest value is translated to `-1.0` and the highest to `1.0`.
The function `rescale_weights()` in the file `src/python/train_nn.py` takes care of rescaling the model weights. The code is available in Appendix B.
The accuracy of the rescaled model might differ from the initial one, due to precision loss during normalization.

At this point, we had our model trained. We then moved on to actually use HeteroCL, the framework previously selected in section 2.

4 Using the framework

This section presents how to generate HLS code using HeteroCL, the framework chosen to be critically analyzed in this project.
As a first step, we explain how to install HeteroCL. Then, we show how we used the framework to generate HLS code and present some custom modifications we chose to apply to the generated HLS code. Finally, a discussion and a critical review about the use of this framework is presented.

The explanations in this section can be very technical. It is possible to go directly to subsection 4.5 for a critical analysis of the framework or to section 5 for the results analysis.

4.1 Installation

The HeteroCL framework is available on their Github repository [45]. However, we decided to not use the official version but chose instead a version of the framework that we modified especially for this project.
Indeed, some operations were not available in the official source code, so we added them in our personal fork.

Before installing the framework, one must install LLVM compiler. The tool requires version 4.0, 5.0, 6.0 or 7.0.

To install HeteroCL, simply clone our modified version and install it:

```
git clone --recursive https://github.com/faku99/heterocl.git
cd heterocl/
make
```

4.2 Generating High-Level Synthesis code

This step consists of specifying to the framework the ML model we are using. Unfortunately, HeteroCL doesn't support loading a model from a file yet, thus we had to define the model architecture once again, but this time using HeteroCL API instead of Keras'. Then, the weights of the trained model are loaded one by one and given to HeteroCL for inference.

src/python/generate_hls.py

```python
def build_mnist(input_image, w_conv1, w_conv2, w_conv3, w_conv4, w_conv5, w_conv6, w_dense1, output):
    # First convolutional layer
    conv1 = hlib.nn.conv2d_nchw(input_image, w_conv1, padding='SAME', name='conv1')
    dbg_print(conv1)
    relu1 = hlib.nn.relu(conv1, name='relu1')
    dbg_print(relu1)

    # Second convolutional layer
    conv2 = hlib.nn.conv2d_nchw(relu1, w_conv2, padding='SAME', name='conv2')
    dbg_print(conv2)
    relu2 = hlib.nn.relu(conv2, name='relu2')
    dbg_print(relu2)

    # First max pooling
    pool1 = hlib.nn.max_pool(relu2, kernel=(2,2), stride=(2,2), name='pool1')
    dbg_print(pool1)

    # Third convolutional layer
    conv3 = hlib.nn.conv2d_nchw(pool1, w_conv3, padding='SAME', name='conv3')
    dbg_print(conv3)
    relu3 = hlib.nn.relu(conv3, name='relu3')
    dbg_print(relu3)

    # Fourth convolutional layer
    conv4 = hlib.nn.conv2d_nchw(relu3, w_conv4, padding='SAME', name='conv4')
    dbg_print(conv4)
    relu4 = hlib.nn.relu(conv4, name='relu4')
    dbg_print(relu4)

    # Second max pooling
    pool2 = hlib.nn.max_pool(relu4, kernel=(2,2), stride=(2,2), name='pool2')
```

```python
    dbg_print(pool2)

    # Fifth convolutional layer
    conv5 = hlib.nn.conv2d_nchw(pool2, w_conv5, padding='SAME', name='conv5')
    dbg_print(conv5)
    relu5 = hlib.nn.relu(conv5, name='relu5')
    dbg_print(relu5)

    # Sixth convolutional layer
    conv6 = hlib.nn.conv2d_nchw(relu5, w_conv6, padding='SAME', name='conv6')
    dbg_print(conv6)
    relu6 = hlib.nn.relu(conv6, name='relu6')
    dbg_print(relu6)

    # Third max pooling
    pool3 = global_max_pool(relu6, name='pool3')
    dbg_print(pool3)

    # Output layer
    dense1 = hlib.nn.dense(pool3, w_dense1, name='dense1')
    dbg_print(dense1)

    return hlib.nn.softmax(output, dense1)
```

This code does essentially the same thing as the first snippet of code in subsection 3.3. The arguments of the `build_mnist()` function are the weights we obtained previously, during the training. They are needed by HeteroCL for generating the hardware operations, which depend on the data length and type.

src/python/generate_hls.py

```python
def build_mnist_inf(batch_size, weights, qtype1, qtype2, target=None):
    # Set placeholders
    input_image = hcl.placeholder((batch_size,1,IMG_H,IMG_W),    name='input')
    w_conv1     = hcl.placeholder(    weights['conv1'].shape,  name='w_conv1', dtype=qtype1)
    w_conv2     = hcl.placeholder(    weights['conv2'].shape,  name='w_conv2', dtype=qtype1)
    w_conv3     = hcl.placeholder(    weights['conv3'].shape,  name='w_conv3', dtype=qtype1)
    w_conv4     = hcl.placeholder(    weights['conv4'].shape,  name='w_conv4', dtype=qtype1)
    w_conv5     = hcl.placeholder(    weights['conv5'].shape,  name='w_conv5', dtype=qtype1)
    w_conv6     = hcl.placeholder(    weights['conv6'].shape,  name='w_conv6', dtype=qtype1)
    w_dense1    = hcl.placeholder(  weights['dense_1'].shape, name='w_dense1', dtype=qtype1)
    output      = hcl.placeholder((batch_size,10), name='output')

    # Create quantization scheme
    scheme = hcl.create_scheme(
        [input_image, w_conv1, w_conv2, w_conv3, w_conv4, w_conv5, w_conv6, w_dense1, output],
        build_mnist
    )

    # Quantize activation layers
    scheme.quantize(
        [build_mnist.relu1, build_mnist.relu2, build_mnist.relu3, build_mnist.relu4, build_mnist.relu5,
        ↪   build_mnist.relu6],
        qtype2
    )

    s = hcl.create_schedule_from_scheme(scheme)

    return hcl.build(s, target=target)
```

The `build_mnist_inf` quantizes the network layers weights. It is in this particular function that we can inform HeteroCL the fixed-point format we want to use for the data.
The `qtype1` argument is the quantization type for the weights, while `qtype2` is for the activation layers.
A quantization type is defined as follows:

```python
import heterocl as hcl

qtype = hcl.Fixed(32,30)
```

Here, 32 corresponds to the total number of bits and 30 the number of bits used for the mantissa. Thus, 1 bit is used for the sign and 1 bit is used for the integer part (0 or 1).

For this project, we measured the performance of the four following quantization types:

1. `hcl.Fixed(32,30)`
2. `hcl.Fixed(16,14)`
3. `hcl.Fixed(8,6)`
4. `hcl.Fixed(4,2)`

HeteroCL offers the possibility of running an hardware simulation of the model on the CPU. Table 2 shows the accuracy for each of the quantization types:

Quantization type	Accuracy
`hcl.Fixed(32,30)`	97.26%
`hcl.Fixed(16,14)`	97.28%
`hcl.Fixed(8,6)`	9.81%
`hcl.Fixed(4,2)`	9.80%

Table 2: Accuracy for each quantization types.

We observe a small loss when the model passes from using 32-bit fixed-point representation to 16-bit fixed-point representation. However, the loss is more than acceptable (0.02%). On the contrary, the loss of switching from the 32- or 16-bit representation to the 8-bit fixed-point representation is way more important and is not suitable for a real-life application. We also notice that there is no loss starting from the 4-bit fixed-point representation.
The impact on resources utilization was measured and will be discussed later.

The entirety of the source code for this subsection is available in Appendix C.

An example of generated Vivado HLS code is available in Appendix D.

4.3 Manual modifications

Depending on how data is passed to the IP, we may want to make some modifications to the previously generated Vivado HLS code. This step is optional.

In this subsection, we will show how to include the ML model's weights into the IP and how to accept the input data from the AXI-Stream protocol.

The first step is to export the weights of the trained model to C/C++ arrays so that we can *delete* the arguments of the Vivado HLS function. The following `exports_weights` function allows to export the weights in C/C++ arrays style.

src/python/generate_hls.py

```python
def export_weights(weights):
    import os
    import shutil
    import sys

    if (os.path.exists('weights')):
        shutil.rmtree('weights')
    os.mkdir('weights')

    for name in weights.keys():
        s = str()
        data = weights[name]
        s += 'const static float w_{}'.format(name)
        for dim in data.shape:
            s += '[{}]'.format(dim)
        s += ' = \n'
        s += '{}'.format(np.array2string(data, max_line_width=80, separator=',',
         ↪  threshold=sys.maxsize).replace('[', '{').replace(']', '}'))
        s += ';'

        file = open('weights/w_{}.h'.format(name), 'w')
        file.write(s)
        file.close()
```

This function generates a file for each entry present in the `weight` dictionary passed as the function parameter. The resulting C/C++ array is named `w_<name>`, where `<name>` is the dictionary key corresponding to the weights array. The file has the same name as what will be placed into a folder named `weights`.

After having exported of the trained weights as C/C++ arrays, we include each header file into the Vivado HLS code and modify the function signature.

mnist_fp32.cpp

```
+   #include "w_conv1.h"
+   #include "w_conv2.h"
+   #include "w_conv3.h"
+   #include "w_conv4.h"
+   #include "w_conv5.h"
+   #include "w_conv6.h"
+   #include "w_dense_1.h"

-   void default_function(float input[1][1][28][28], float w_conv1[4][1][3][3], float
 ↪   w_conv2[4][4][3][3], float w_conv3[8][4][3][3], float w_conv4[8][8][3][3], float
 ↪   w_conv5[16][8][3][3], float w_conv6[16][16][3][3], float w_dense1[16][10], float output[1][10]) {
+   void mnist_fp32(float input[1][1][28][28], output[1][10]) {
```

Noticeably, we also changed the function name. The function we discuss in this subsection is the one using 32-bit fixed-point data. All the modifications made to this function can be applied to the others data formats generated previously.

The next modification to make concerns the input protocol. At this point, the generated IP expects the address of an array containing the input image 28x28 data. However, we wanted the IP to gather the data from a Video Direct Memory Access (VDMA) offering AXI-Stream protocol for the reading part.

The first thing to do was to include the header files offered by Vivado for using AXI-Streams.

mnist_fp32.cpp

```
+   #include <hls_video.h>
```

Some definitions are then needed, as for the pixel's format and the type of the elements the AXI-Stream is composed of. Moreover, the function arguments needed to be modified.

mnist_fp32.cpp

```
+   typedef ap_axiu<32,1,1,1> pixel_t;
+   typedef hls::stream<pixel_t> stream_t;

-   void mnist_fp32(float input[1][1][28][28], float output[1][10]) {
+   void mnist_fp32(stream_t& stream_in, output[1][10]) {
```

The following snippet of code is used to synchronize the AXI-Stream. It is a bit tricky, but it basically waits for the start signal, indicating the beginning of the input image. It also waits for the end signal, which, in turn, indicates the end of a row of the input image, thus every 28 pixels.
It also puts the data received from the AXI-Stream into a `float` array.

mnist_fp32.cpp

```
+       float image[1][1][28][28];
+       pixel_t pixel;
+       union_t uni;
+       HLS_SIZE_T i, j;
+
+       bool sof = 0;
+       loop_wait_sof: while (sof == 0) {
+   #pragma HLS LOOP_TRIPCOUNT avg=0 max=0
+   #pragma HLS PIPELINE II=1
+           stream_in >> pixel;
+           sof = pixel.user.to_int();
+       }
+
+       loop_height: for (i = 0; i < 28; ++i) {
+           bool eol = 0;
+
+           loop_width: for (j = 0; j < 28; ++j) {
+   #pragma HLS LOOP_FLATTEN off
+   #pragma HLS PIPELINE II=1
+               if (sof || eol) {
+                   sof = 0;
```

```
+                   eol = pixel.last.to_int();
+               }
+               else {
+                   stream_in >> pixel;
+                   eol = pixel.last.to_int();
+               }
+
+               uni.u = pixel.data.to_uint();
+               image[0][0][i][j] = uni.f; // pixel.data.to_double();
+           }
+
+           loop_wait_eol: while (eol == 0) {
+   #pragma HLS PIPELINE II=1
+   #pragma HLS LOOP_TRIPCOUNT avg=0 max=0
+               stream_in >> pixel;
+               eol = pixel.last.to_int();
+           }
+       }
```

The last modification we made was to tell the compiler that our input uses the AXI-Stream protocol. This can be made by adding a pragma to the code.

mnist_fp32.cpp

```
+   #pragma HLS INTERFACE axis port=stream_in
```

After all those custom modifications were made, we compiled the Vivado HLS code and moved on to critically discuss the use of the HeteroCL framework.

4.4 Code analysis

This subsection presents a critic analysis about the quality of the HLS code generated by the HeteroCL framework.

The first observation we can make is about the code readability. At several points in the code, we notice things like the following:

```
float reducer84;
reducer84 = 0.000000e+00f;
```

Even if it doesn't change anything for the compiler's understanding, a better way for human comprehension would be something like `float reducer84 = 0.0f`. However, we consider it a normal behavior since it might be difficult to add a *human-readability* dimension during HLS code generation.

The second observation is about array indexes. Often, the first dimension of the arrays generated is 1.

```
float conv1[1][4][28][28];
```

This causes the generated code to have a useless *outside* loop that lowers the readability. Again, this doesn't affect the compilation, since the compiler optimizes it by erasing the *outside* loop.

The third observation is about optimizations that are made automatically by the Vivado HLS compiler. During compilation, some functions (*e.g.* `std::max()`) are inlined, small loops are automatically unrolled and array are partitioned. We consider these features being helpful and time-saving.

Finally, the most important observation, and most critical one. In subsection 4.2, we specified a type to use for the quantization and data layers. This is respected for the ReLU layers, as shown below.

mnist_fp16.cpp

```
ap_fixed<16, 2> relu1[1][4][28][28];
for (ap_int<32> args = 0; args < 1; ++args) {
    for (ap_int<32> args0 = 0; args0 < 4; ++args0) {
        for (ap_int<32> args1 = 0; args1 < 28; ++args1) {
            for (ap_int<32> args2 = 0; args2 < 28; ++args2) {
                relu1[args][args0][args1][args2] = ((ap_fixed<16, 2>)((conv1[args][args0][args1][args2]
                 ↪  < 0.000000e+00f) ? 0.000000e+00f : conv1[args][args0][args1][args2]));
            }
        }
```

```
125         }
126 }
```

On line 122, we notice that **float** values are casted to `ap_fixed<16, 2>`, which is the data representation we specified previously.

For the data layers (*e.g.* convolutional layers), calculations and storage are done in floating-point data representation.

mnist_fp16.cpp

```
102 float conv1[1][4][28][28];
103 for (ap_int<32> ff = 0; ff < 4; ++ff) {
104     for (ap_int<32> yy = 0; yy < 28; ++yy) {
105         for (ap_int<32> xx = 0; xx < 28; ++xx) {
106             float reducer84;
107             reducer84 = 0.000000e+00f;
108             for (ap_int<32> ry = 0; ry < 3; ++ry) {
109                 for (ap_int<32> rx = 0; rx < 3; ++rx) {
110                     reducer84 = ((pad_temp[0][0][(yy + ry)][(xx + rx)] * w_conv1[ff][0][ry][rx]) +
                         ↪  reducer84);
111                 }
112             }
113             conv1[0][ff][yy][xx] = reducer84;
114         }
115     }
116 }
```

Line 102 shows that calculations are stored using **float** format, even if we specified to use 16-bit fixed-point data representation. Moreover, we notice that calculations are also done with floating-point representation. This limitation might make the IP more demanding in terms of resources utilization.

We tried to understand why HeteroCL does not use the `ap_fixed<16, 2>` format for data layers. In activation layers, the fixed-point format is used but we look closely, no operations are made, only a comparison. It seems that when an operation is needed, HeteroCL prefers to use the floating-point format. Surprisingly, Vivado HLS offers an implementation for such operations. So, we assumed that the HeteroCL team did not integrate this feature in the framework yet, or have a good reason to not do so. In the case, we did not find any documentation talking about this choice.

In conclusion, the code generated by HeteroCL is understandable by a human, even if some enhancements can still be made. However, we noticed that the frameworks doesn't always respect the data format constraints it has been given during the development phase.

4.5 Use analysis

This subsection presents our critic analysis about the use of HeteroCL for generating HLS code from Python.

Looking at the HeteroCL website, we can read that "*HeteroCL provides a fully automated compilation flow from a HeteroCL program to heterogeneous compute platforms integrating CPUs and FPGAs*".

As it was previously shown, this statement is not entirely valid. Indeed, some widely used operators, such as the ReLU activation function, or also widely used parameters of the 2D-Convolutional operator, like the dilation, had to be implemented by us. Thus, the "*fully automated compilation flow*" statement is yet to be fully accurate.

Moreover, the IP generated from the code produced by HeteroCL can't be easily integrated into a system design by people with little to no experience in hardware designs. The manual modifications we presented in the subsection above are required if one wants to change the protocol used to fetch the input data. The default protocol, fetching data from an address in memory, is fine but might not be relevant in some cases, as our system design, which carries the image data captured from the camera sensor to the HDMI output port by using the AXI-Stream protocol.

In conclusion, HeteroCL still has a long way to go to propose a fully automated compilation flow for programming FPGAs that can be used by any developer with little hardware design knowledge.

4.6 Optimizations

This subsection presents the transformations manually applied to the HLS code to optimize it. Every step will be presented and all resources variations will be discussed.

At this point, apart from the customizations applied previously (*i.e.* adding the AXI-Stream protocol and including the weights into the IP), the HLS code hasn't been touched. No optimizations have been applied to the code. The Vivado HLS C++ language allows the developer to specify code optimizations by simply adding pragmas (as in the customizations made to the code in subsection 4.3).
The "Transformations of high-level synthesis codes for high-performance computing" paper, written by Fine Licht, Meierhans, and Hoefler, presents the possible transformations and optimizations that can be applied to HLS code [53]. In this work, we only applied pipeline transformations to the code.

A transformation optimizes the code and more extensively, the compiled IP. It is optimized in terms of latency (*i.e.* clock cycles). However, adding transformations like pipelining requires more resources. In the following paragraphs, we will present step by step the transformations made and the differences in both terms of clock cycles and resources utilization.
The reference IP for the transformations is the one using 16-bit fixed-point data representation. This version will now be called *FP16-opt*. This is due to the fact that the work presented in this subsection has been done after results were analyzed and discussed for the unoptimized versions generated by HeteroCL. Thus, we chose the best among these and used it as the reference IP.

The first transformation we applied to the code is the following:

```
    float pad_temp[1][1][30][30];
    pad1_h: for (ap_int<32> index_tuple = 0; index_tuple < 30; ++index_tuple) {
#pragma HLS PIPELINE
        pad1_w: for (ap_int<32> i = 0; i < 30; ++i) {
#pragma HLS PIPELINE
            pad_temp[0][0][index_tuple][i] = (((((1 <= index_tuple) && (index_tuple < 29)) && (1 <= i))
            ↪ && (i < 29)) ? image[((((i - ((i + -1) % 28)) + (index_tuple * 28)) + -29) /
            ↪ 784)][0][((((i - ((i + -1) % 28)) + (index_tuple * 28)) + -29) / 28) % 28)][((i + -1)
            ↪ % 28)] : 0.000000e+00f);
        }
    }
```

The nested loop shown in the snippet of code above is the first padding operation of the model (just before the first convolutional network). Here, we apply a pipelining transformation on both loops. This transformation increases the latency by 5966 clock cycles, the number of DSP by 13, the number of Flip-Flops used by 8'269. It also increases the number of Lookup Tables by 7'503. The BRAM usage has not been changed by this transformation. It seems that pipelining these loops only increases the latency, thus not optimizing the IP. So it has been discarded.

The second transformation happens in the first convolutional block, performed by 5 nested loops.

```
    float conv1[1][4][28][28];
    conv1_c: for (ap_int<32> ff = 0; ff < 4; ++ff) {
        conv1_h: for (ap_int<32> yy = 0; yy < 28; ++yy) {
            conv1_w: for (ap_int<32> xx = 0; xx < 28; ++xx) {
                float reducer84;
                reducer84 = 0.000000e+00f;
                conv1_kh: for (ap_int<32> ry = 0; ry < 3; ++ry) {
                    conv1_kw: for (ap_int<32> rx = 0; rx < 3; ++rx) {
#pragma HLS PIPELINE
                        reducer84 = ((pad_temp[0][0][(yy + ry)][(xx + rx)] * w_conv1[ff][0][ry][rx]) +
                        ↪  reducer84);
                    }
                }
                conv1[0][ff][yy][xx] = reducer84;
            }
        }
    }
```

The transformation shown in the snippet of code above pipelines the most nested loop. This transformation reduces the latency by 113'600 clock cycles while increasing the Flip-Flops usage by 244 and LUTs by 283.

The BRAM and DSP resources utilization have stayed untouched. This particular code transformation is interesting since it greatly reduces the latency while moderately increasing the resources utilization.
Then, we tried to pipeline the loop `conv1_kh` at line 92 but it changed the timing estimates from 8.671 to 15.210, thus decreasing the maximum frequency to 65.746 MHz. In terms of latency, the transformation decreased it by 59'575 clock cycles. Since the tradeoff between latency and maximum frequency is not worth it, we decided to revert pipelining the loop.

The third transformation consisted of pipelining the ReLU activation of the first convolutional layer. As a test, we tried to pipeline all the loops.

```
    ap_fixed<16, 2> relu1[1][4][28][28];
    relu1_n: for (ap_int<32> args = 0; args < 1; ++args) {
        relu1_c: for (ap_int<32> args0 = 0; args0 < 4; ++args0) {
#pragma HLS PIPELINE
            relu1_h: for (ap_int<32> args1 = 0; args1 < 28; ++args1) {
#pragma HLS PIPELINE
                relu1_w: for (ap_int<32> args2 = 0; args2 < 28; ++args2) {
#pragma HLS PIPELINE
                    relu1[args][args0][args1][args2] = ((ap_fixed<16,
                     ↪   2>)((conv1[args][args0][args1][args2] < 0.000000e+00f) ? 0.000000e+00f :
                     ↪   conv1[args][args0][args1][args2]));
                }
            }
        }
    }
```

These transformations reduce the latency by 15'906 clock cycles while the Flip-Flops increase by 185 and the Lookup Tables by 148. The BRAM and DSP utilizations did not change. This transformation optimizes well the IP.

Since the transformations are quite the same, we will not detail them anymore. Every "*case*" has already been treated (*i.e.* padding operation, convolution, and ReLU activation). Instead, Table 4 lists all the transformations that have been applied to the generated Vivado HLS code. It also details the timings and resources difference it implies. Values that are in red mean that they are not satisfying enough (*i.e.* resources overflow, overlong timing) and that the code transformation was therefore not included.
The code including all the transformations presented in Table 4 is available in Appendix F.

As the result of all these transformations added in the Vivado HLS code generated by the HeteroCL framework, we obtained the performances presented in Table 3. These performances will be discussed in the next section.

Latency	1'242'094 clock cycles
Timing	8.742 ns
Max freq.	114.390 MHz
LUT	51'545 (96.89%)
FF	39'750 (37.36%)
BRAM	88 (31.43%)
DSP	80 (36.36%)

Table 3: Latency, timing and resources optimizations after applying transformations by hand.

We generated Vivado HLS code from a Python code describing a Neural Network model. Then, we modified the code to adapt it to our needs (*i.e.* AXI-Stream and weights included in the IP). Moreover, we made critical reviews of both the code generated by HeteroCL and its use. Finally, we applied code transformations by hand to optimize the resulting IP.
We will now discuss the performances of each of the sources mentioned previously in subsection 1.1.

Name	Cycles	Timing	BRAM	DSP	FF	LUT
pad1	+5966	8.671	0	+13	+8269	+7503
conv1_kw	-113600	8.671	0	0	+244	+283
conv1_kh	-59575	15.210	-1	0	+361	+207
relu1	-15906	8.671	0	0	+546	+355
pad2	-59644	8.717	0	-2	+137505	+363813
conv2_kw	-583537	8.671	0	0	-327	-156
pool1	-27560	8.671	0	0	+255	+195
conv3_kw	-378128	8.704	0	0	-17	+80
pad3	-10357	8.704	0	0	+1129	+534
relu3	-8074	8.704	0	0	+181	+154
pad4	-30963	8.704	0	0	+1688	+641
conv4_kw	-655664	8.704	0	0	+26	+58
relu4	-8074	8.704	0	0	+180	+154
relu2	-15906	8.704	0	0	+186	+148
pool2	-13848	8.731	0	0	+247	+196
pad5	-6298	8.742	0	0	+995	+429
pad2-OK	-57817	8.742	0	0	+1891	+761
conv5_kw	-382848	8.742	0	0	-26	+60
relu5	-4170	8.742	0	0	+177	+149
pad6	-19084	8.742	0	0	+1595	+544
conv6_kw	-659600	8.742	0	0	-17	-50
relu6	-4170	8.742	0	0	+177	+149
maxpool	-8656	8.742	0	0	+230	+130
maxpool1	-45	8.742	0	0	+126	+78
dense	-2404	15.210	0	0	+354	+308
relu2_h	0	8.742	0	0	+1	0
relu3_h	0	8.742	0	0	0	0
pad3_h	-511	8.742	0	+13	+3204	+3185
pad2_h	-1199	8.742	0	+45	+21728	+26863
pad4_h	-1023	8.742	0	+13	+5946	+9127
relu4_h	0	8.742	0	0	0	0
pad5_h	-288	8.742	+1	+6	+1453	+1471
relu5_h	0	8.742	0	0	+4	0
pad6_h	-576	8.742	0	+6	+2606	+4357
relu6_h	0	8.742	0	0	-4	0
maxpool_h	0	8.742	0	0	-5	0
softmax	-18	8.742	0	0	+4	+18
softmax1	-253	28.512	0	0	+15	-36
softmax2	-522	8.742	0	0	+128	-78
results	-27	8.742	0	0	+131	+96

Table 4: List of transformations applied by hand to the Vivado HLS code.

5 Performance results

This section presents and compares the results obtained from several sources, as mentioned previously. As a reminder, here is the list of the different sources:

- A solution running on a CPU
- A solution developed by a – human – programmer, with a HDL
- A solution generated with the framework
- The same solution as above, but optimized with code transformations manually applied after generation

In the following section, the term "hardware latency" refers to the latency in clock cycles while the term "time latency" refers to the minimum latency in seconds, which is calculated by multiplying the hardware latency by the maximum frequency.

5.1 Inference running on a CPU

The first source of results was obtained by simply running the model inference on a CPU. The code was run on the Zybo Z7-20 processor, which is a dual-core ARM Cortex-A9 processor operating at 667 MHz [18], [54].

To perform the inference, we used the TensorFlow Lite framework [55]. It is designed to run TensorFlow models on mobile, embedded and Internet of Things (IoT) devices. It allows on-device Machine Learning inference with low latency and a small binary size. Among the proposed APIs by TensorFlow Lite, we used the C++ API [56] to run the inference.
The corresponding code is available in Appendix E.

5.1.1 Accuracy

The accuracy measured for the Keras model implemented with Python (and detailed in subsection 3.3) is **97.56%**.

5.1.2 Resources utilization

The only resource we could measure for the CPU-running solution was the memory utilization during inference execution. The memory profiling was performed with the Valgrind Massif tool [57].
The Figure 7 shows the memory usage over the execution of the inference. During execution, memory usage peaks at **313.3 KiB**.

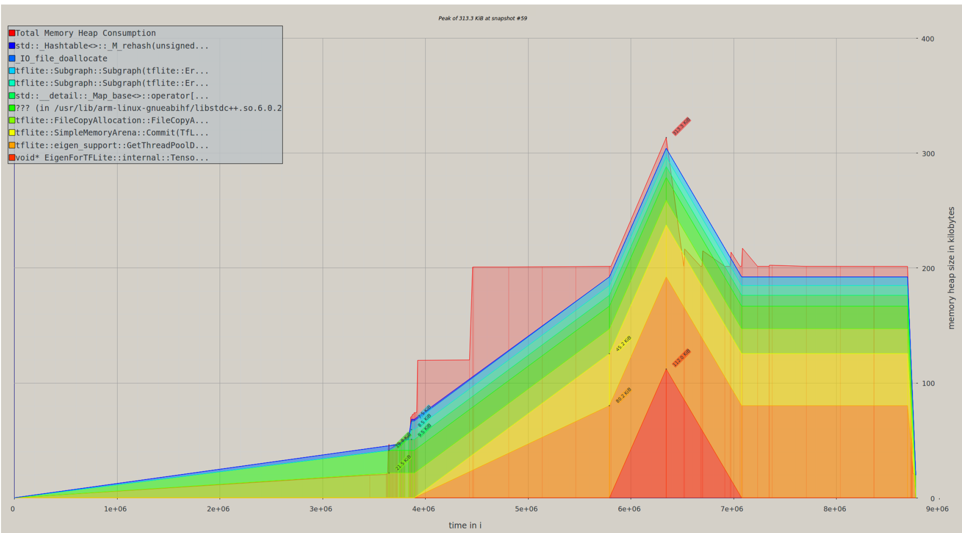


Figure 7: Inference memory usage over time.

5.1.3 Latency

To measure the inference execution time, we performed 10 inference runs and we calculated the average of these runs. The calculated average inference execution time was **7.386 milliseconds**.

The memory usage is pretty low, as well as the execution time for a system like the Zybo Z7-20. These performances allow to perform handwritten digits recognition up to 135 frames per second – which is more than enough for a real-life application – while consuming low memory.

5.2 Inference developed by a human developer

The second set of results, the inference developed by a human developer, was taken from the paper written by Solovyev, Kustov, Telpukhov, *et al.* [50].

5.2.1 Accuracy

The paper doesn't mention anything about the accuracy performed by the model. However, since we based our model and its training on the one presented in the paper, we can safely suppose that it offers about the same accuracy as the Python model we propose (97.56%) in subsection 3.3.

5.2.2 Resources utilization

About resources utilization, the "Fixed-Point Convolutional Neural Network for Real-Time Video Processing in FPGA" paper doesn't talk extensively about it and, more importantly, it uses another tool for compiling the inference SystemVerilog module. To be sure to have the same basis of comparison, we recompiled it with Vivado, the tool used in this work. The target board, defining the available resources, is the Zybo Z7-20. Its characteristics are detailed in Appendix A.

To obtain comparable results, we compiled the SystemVerilog module in four different ways, depending on the number of bits used for weight's fixed-point data representation.
After compiling the module, we obtain the following resources utilization:

Resource	**Utilization**				**Available**
	32-bit	**16-bit**	**8-bit**	**4-bit**	
LUT	4'005 (7.53%)	2'491 (4.68%)	2'554 (4.80%)	1'917 (3.60%)	53'200
FF	2'799 (2.63%)	1'562 (1.47%)	1'010 (0.95%)	706 (0.66%)	106'400
BRAM	55 (19.64%)	28 (10.00%)	14 (5.00%)	7 (2.50%)	280
DSP	40 (18.18%)	17 (7.73%)	4 (1.82%)	4 (1.82%)	220

Table 5: Resources utilization of the inference developed by a human developer.

This resources utilization summary is valid for a sequential implementation of the model. However, human implementation offers an interesting feature over automatically generated methods: parallelization. The "Fixed-Point Convolutional Neural Network for Real-Time Video Processing in FPGA" paper also presents two other implementations: one with 2 convolutional blocks parallelized and the other with 4 parallelized convolutional blocks.

Resource	**Utilization**				**Available**
	32-bit	**16-bit**	**8-bit**	**4-bit**	
LUT	5'979 (11.24%)	3'286 (6.18%)	3'565 (6.70%)	2'383 (4.48%)	53'200
FF	3'948 (3.71%)	2'096 (1.97%)	1'271 (1.19%)	823 (0.77%)	106'400
BRAM	68 (24.29%)	35 (12.50%)	18 (6.43%)	9 (3.21%)	280
DSP	74 (33.64%)	28 (12.73%)	2 (0.91%)	2 (0.91%)	220

Table 6: Resources utilization of the inference developed by a human developer with 2 convolutional blocks.

Resource	Utilization				Available
	32-bit	16-bit	8-bit	4-bit	
LUT	9'486 (17.83%)	4'811 (9.04%)	5'627 (10.58%)	3'163 (5.95%)	53'200
FF	6'267 (5.89%)	3'198 (3.01%)	1'852 (1.74%)	1'112 (1.05%)	106'400
BRAM	110 (39.29%)	56 (20.00%)	29 (10.36%)	15 (5.36%)	280
DSP	146 (66.36%)	54 (24.55%)	2 (0.91%)	2 (0.91%)	220

Table 7: Resources utilization of the inference developed by a human developer with 4 convolutional blocks.

5.2.3 Timing

Upon synthesis, the Vivado Design Suite outputs the timing estimates of the SystemVerilog module. We compiled every fixed-point data representation for 1, 2 and 4 convolutional blocks. The number of parallelized blocks didn't change the timing, thus Table 8 only shows the timings for the different data representation.

	Timing [ns]	Max freq. [MHz]
32-bit	16.293	61.376
16-bit	15.303	65.347
8-bit	14.874	67.231
4-bit	14.896	67.132

Table 8: Timings for inference developed by a human developer.

We notice that the max frequency decreases with the number of bits used for data representation.

5.2.4 Latency

According to this same paper, the number of clock cycles required to predict a digit from an image is **236'746**. Table 9 details the clock cycles needed for each stage of the inference.

Stage	Number of clock cycles
Loading input image	1'570
1st convolution: loading weights	76
1st convolution: processing	12'605
2nd convolution: loading weights	291
2nd convolution: processing	50'416
1st max pooling: processing	3'164
3rd convolution: loading weights	580
3rd convolution: processing	25'569
4th convolution: loading weights	1'155
4th convolution: processing	51'136
2nd max pooling: processing	1'623
5th convolution: loading weights	2'309
5th convolution: processing	27'009
6th convolution: loading weights	4'611
6th convolution: processing & global max pooling	54'016
Dense layer: loading weights	356
Dense layer: processing	244
Saving result	16
Total	**236'746**

Table 9: Clock cycles at each stage of FPGA-based image processing [50].

The hardware latencies detailed in Table 9 are valid only for the sequential implementation.
Table 10 shows the clock cycles that are needed when convolutional blocks are parallelized. To obtain the time latency in milliseconds, we multiply the number of clock cycles (hardware latency) by the maximum frequency presented in Table 8.

Convolutional blocks	Clock cycles	Data type	Timing [ms]
1	236'746	32-bit	3.857
		16-bit	3.623
		8-bit	3.521
		4-bit	3.527
2	125'320	32-bit	2.042
		16-bit	1.918
		8-bit	1.864
		4-bit	1.867
4	67'861	32-bit	1.106
		16-bit	1.038
		8-bit	1.009
		4-bit	1.011

Table 10: Clock cycles with parallelized convolutional blocks [50].

Table 10 shows that adding parallelized convolutional blocks reduces drastically the inference time.

5.3 Inference generated with our workflow

The third source comes from the Vivado HLS code that has been generated following the workflow described in this paper. Upon compilation, the Vivado HLS tool outputs several estimates: resources, timing and latency. In this subsection, we will present the values estimated by the tool for each IPs previously generated.

5.3.1 Accuracy

The Table 11 shows the accuracy each previously generated IP performs for handwritten digits recognition.

Generated IP	Accuracy
FP32	97.26%
FP16	97.28%
FP8	9.81%
FP4	9.80%

Table 11: Accuracy for each generated IP.

It can be noticed that allocating less than 16 bits for fixed-point data representation decreases drastically the model accuracy.
Interestingly, we notice that decreasing the bits used from 32 to 16 increases slightly the model accuracy. It shows that handwritten digits recognition doesn't require high precision for its weights.

5.3.2 Resources utilization

The manual modifications we made to the generated Vivado HLS code implies that weights are included into the IP as memory. Thus, a lot of memory resources might be used for the IP implementation when the SystemVerilog module written by Solovyev, Kustov, Telpukhov, *et al.* fetches weights from another module than the one doing the inference. Moreover, we also introduced the AXI-Stream protocol, which might change resources utilization and latency. For the sake of the good comparison, in this subsection, we will present four versions for each of the generated Vivado HLS IPs.

Table 12 presents the resources available on the Zybo Z7-20 development board. The values are interesting for comparing with the following graphs.

Resource	Available
LUT	53'200
FF	106'400
BRAM	280
DSP	220

Table 12: Amount of resources available on the Zybo Z7-20.

1. No AXI-Stream & no weights included.

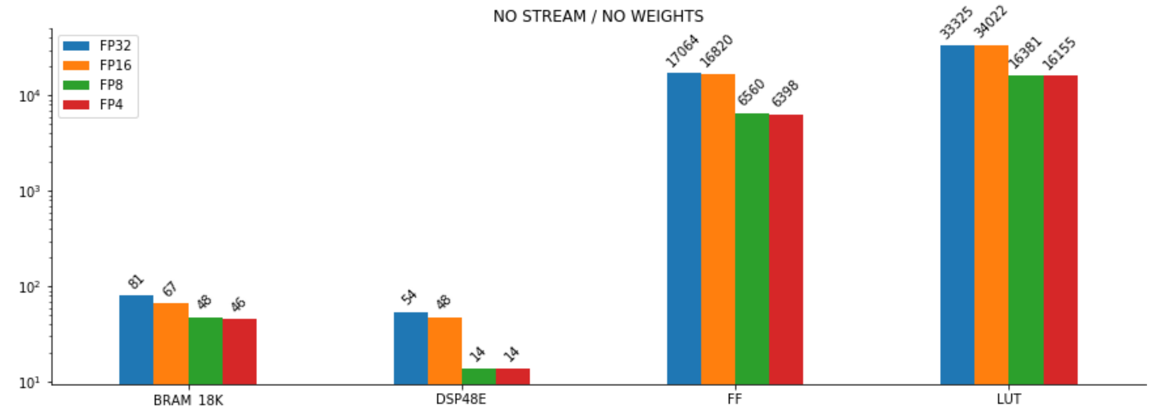


Figure 8: Resources utilization without AXI-Stream and weights not included into the IP.

Figure 8 shows that the BRAM utilization increases proportionately with the number of bits used for fixed-point data representation.
About the other resources, we notice that 32- and 16-bit use approximatively the same amount and the same goes for 8- and 4-bit fixed-point data representation.

2. No AXI-Stream & weights included.

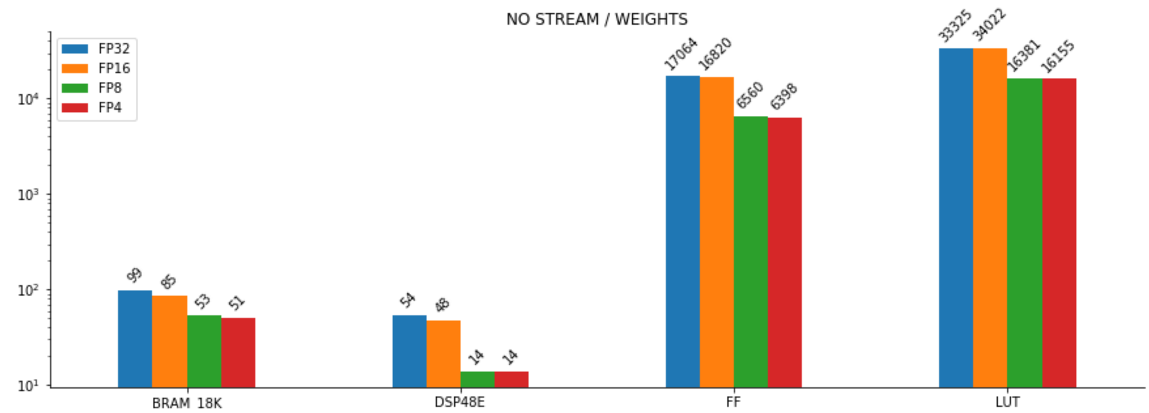


Figure 9: Resources utilization without AXI-Stream and weights included into the IP.

From the chart above, we observe that including the model's weights into the IP only increase the amount of BRAM used. The other resources are left untouched.

3. AXI-Stream & no weights included.

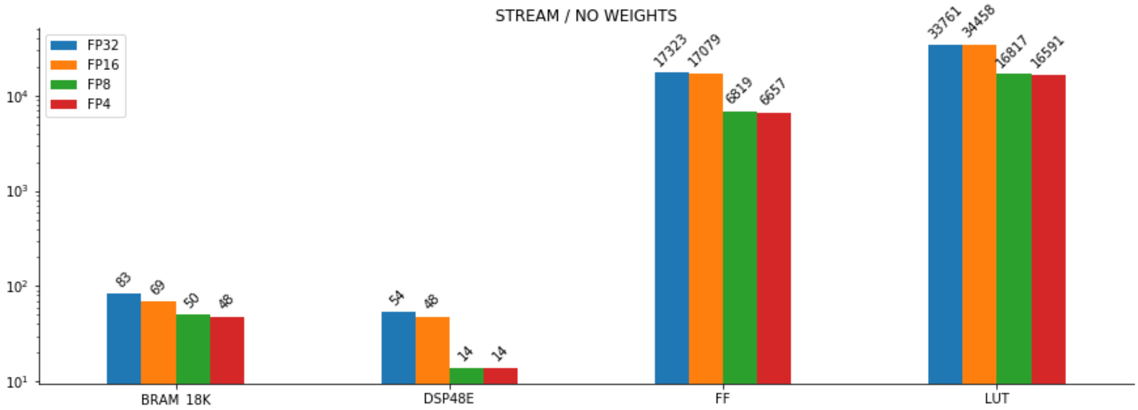


Figure 10: Resources utilization with AXI-Stream and weights not included into the IP.

Adding the AXI-Stream protocol for fetching the input data increases the utilization of BRAM, FF and LUT resources. The DSP48E resources utilization is the same with or without the AXI-Stream protocol.

4. AXI-Stream & weights included.

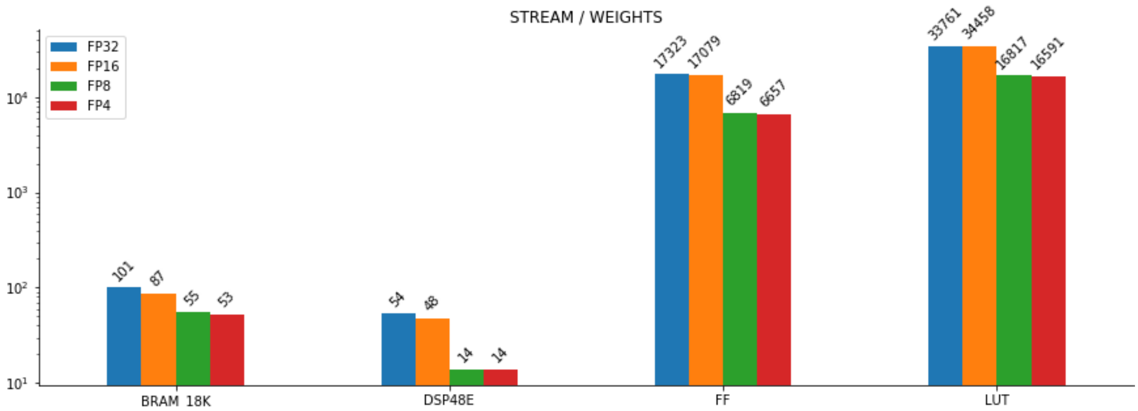


Figure 11: Resources utilization with AXI-Stream and weights included into the IP.

As noticed in the previous graphs, all resources utilization increase except for the `DSP48E`, which doesn't change either the AXI-Stream protocol or the weights are included in the IP.

From the tables above, it can be seen that the more bits are used for data formats, the more resources are used. When the weights are included into the IP, the BRAM resources utilization increases.
When AXI-Stream is used as the protocol for fetching input data, BRAM, Flip-Flops and LUT resources increase a bit.

5.3.3 Timing

Upon generation, the Vivado HLS tool also outputs the timing estimates. These timings change with the data format but do not depend on the AXI-Stream protocol or where the weights are stored. Table 13 shows the estimated timings for each of the generated IPs.

	Timing [ns]	**Max freq. [MHz]**
FP32	8.621	115.996
FP16	8.671	115.327
FP8	8.465	118.133
FP4	8.709	114.824

Table 13: Estimated timings for generated IPs.

We notice that changing the fixed-point data representation doesn't significantly change the maximum frequency (1% of variation at most).

5.3.4 Latency

The latency changes if the AXI-Stream protocol is used for the input data or not. Latency estimates will be presented for each previously generated IP.
For calculating the estimated timing in milliseconds, we take the average latency and multiply it by the maximum frequency presented in Table 13.

1. **FP32**

	Minimum	Maximum	Average	Timing [ms]
No Stream	4'570'271	4'810'211	4'690'241	40.435
Stream	4'571'254	4'811'194	4'691'224	40.443

Table 14: Latency estimates for 32-bit fixed-point IP.

2. **FP16**

	Minimum	Maximum	Average	Timing [ms]
No Stream	4'242'559	4'489'443	4'366'001	37.858
Stream	4'243'542	4'490'426	4'366'984	37.866

Table 15: Latency estimates for 16-bit fixed-point IP.

3. **FP8**

	Minimum	Maximum	Average	Timing [ms]
No Stream	3'022'067	3'215'371	3'118'719	26.400
Stream	3'023'050	3'216'354	3'119'702	26.408

Table 16: Latency estimates for 8-bit fixed-point IP.

4. **FP4**

	Minimum	Maximum	Average	Timing [ms]
No Stream	3'017'363	3'210'667	3'114'015	27.120
Stream	3'018'346	3'211'650	3'114'998	27.129

Table 17: Latency estimates for 4-bit fixed-point IP.

The tables above show that adding the AXI-Stream protocol for fetching input data increases the hardware latency by 983 clock cycles, independently of the fixed-point data representation.
The latency decreases greatly when the number of bits used for data representation is also decreased. The difference is less noticeable when passing from 8-bit to 4-bit data representation. However, it is interesting to note that even if the hardware latency decreases a bit when using 4-bit instead of 8-bit data representation, the time latency (in milliseconds) increases. This is due to the maximum frequency being lower for 4-bit data representation.

From all the results presented in this subsection, we decided to pick the IP which uses the 16-bit fixed-point data representation because it offers the best accuracy for resources utilization. It will be used for the comparison with the other solutions.

5.4 Inference generated and optimized

The final source is the result of the subsection 4.6. This actual subsection presents the performances offered by the IP with transformations applied by hand to the code generated by HeteroCL. These transformations have been only applied to the IP using 16-bit for fixed-point data representation, because it was the best candidate of the unoptimized generated solutions.

5.4.1 Accuracy

The accuracy of this solution is the same as the 16-bit unoptimized IP detailed in the previous subsection. As a reminder, it was **97.28%**.

5.4.2 Resources utilization

Adding transformations to the code such as pipelining functions increases the resources utilization. Table 18 presents the resources utilization of the optimized generated solution.

Resource	Utilization	Available
LUT	51'545 (96.89%)	53'200
FF	39'750 (37.36%)	106'400
BRAM	88 (31.43%)	280
DSP	80 (36.36%)	220

Table 18: Resources utilization for optimized generated IP.

We notice that the LUT utilization is almost overflowing. The DSP utilization is close to the double of the previous solution.

5.4.3 Timing

Upon synthesis, the Vivado Design tool outputs **8.742 nanoseconds** as the critical path delay. This means that the maximum frequency at which the IP can operate is **114.390 MHz**.

5.4.4 Latency

The main goal of applying transformations to the code was to reduce the latency. At the end, when all transformations presented in Table 4 have been applied, the hardware latency of the IP is **1'242'094** clock cycles, which is about a fourth of the solution presented previously.
If we multiply this latency by the estimated timing, we obtain a time latency of **10.858 milliseconds**.

5.5 Comparison

This subsection summarizes and compares the three solutions on their accuracy, their resources utilization, and their timings.

From the HeteroCL-generated solutions, we picked the ones using the 16-bit fixed-point data representation, as they presented the best performances over the other data formats. In order to have a consistent comparison, we will therefore use the human-developed module with 16-bit fixed-point.

In the following results, the solution named "CPU" refers to the first set of results, where the inference was run on a CPU. The solution referred by "Human-#" is the one described by Solovyev, Kustov, Telpukhov, *et al.* in the "Fixed-Point Convolutional Neural Network for Real-Time Video Processing in FPGA" paper [50]. The "#" represents the number of convolutional blocks that have been parallelized. An absence of "#" means that the value is the same regardless of whether several convolutional blocks have been parallelized or not.
"FP16" refers to the IP previously generated by HeteroCL and which uses the 16-bit fixed-point data representation.
Finally, "FP16-opt" refers to the last solution, which is FP16 but optimized with code transformations.

5.5.1 Accuracy

Solution	Accuracy
CPU	97.56%
Human	97.56%
FP16	97.28%
FP16-opt	97.28%

Table 19: Accuracy of the different solutions.

From Table 19, we notice that the accuracy doesn't change much among the proposed solutions. This first comparison shows that HeteroCL performs well at keeping quite the same accuracy even with weights fixed-point data original representation being truncated.

5.5.2 Resources utilization

Comparing the software solution and the hardware solutions is complex for resources utilization, since they are not very related. Here, we compare the memory utilization for each solution and then we compare the Zybo Z7-20 development board resources utilization for each hardware solution.

Solution	Memory [KiB]
CPU	313.3
Human-1	63.0
Human-2	78.8
Human-4	126.0
FP16	195.8
FP16-opt	198.0

Table 20: Memory utilization of the different solutions.

Table 20 shows that hardware solutions require less memory than the software one. The human-developed inference solution requires up to 5 times less than the CPU one (for the sequential version). The optimized version of FP16 doesn't require much more memory than the unoptimized version (1% more).

The next table details the complete resources utilization for all hardware solutions. As a reminder, the Zybo Z7-20 FPGA development board offers 53'200 LUTs, 106'400 FFs, 280 Block Random Access Memory (BRAM) and 220 DSP48E1 units.

Resource	Human-1	Human-2	Human-4	FP16	FP16-opt
LUT	2'491 (4.68%)	3'286 (6.18%)	4'811 (9.04%)	34'458 (64.77%)	51'545 (96.89%)
FF	1'562 (1.47%)	2'096 (1.97%)	3'198 (3.01%)	17'079 (16.05%)	39'750 (37.36%)
BRAM	28 (10.00%)	35 (12.50%)	56 (20.00%)	87 (31.07%)	88 (31.43%)
DSP	17 (7.73%)	28 (12.73%)	54 (24.55%)	48 (21.82%)	80 (36.36%)

Table 21: Resources utilization of the hardware solutions.

We notice that resources utilization, especially for LUTs and FFs, is way more higher for the IP generated with HeteroCL than for the IPs developed by a human developer. Most notably, the FP16 solution only has one convolutional block when the solution developed by Solovyev, Kustov, Telpukhov, *et al.* offers up to 4 convolutional blocks in parallel.
The optimized FP16-opt solution uses almost all of the available LUTs and consumes about the double of the FFs and DSP units than the unoptimized solution. It only requires one more BRAM.

5.5.3 Timing

Timing cannot be discussed for the software solution since it doesn't have critical path delay nor a maximum frequency at which it can operate. Thus, only the hardware solutions will be discussed in the table below.

Solution	Timing [ns]	Max freq. [MHz]
Human	15.303	65.347
FP16	8.671	115.327
FP16-opt	8.742	114.390

Table 22: Timing estimates for hardware solutions.

Table 22 shows that the IP generated by HeteroCL can operate at a frequency almost twice as big as the SystemVerilog module presented in the "Fixed-Point Convolutional Neural Network for Real-Time Video Processing in FPGA" paper [50]. We notice that both the FP16 and FP16-opt solutions have about the same maximum frequency.
This measure might be interesting, depending on the latency of both IPs, since the generated one can operate faster. This same latency is presented next.

5.5.4 Latency

Solution	Clock cycles	Latency [ms]
CPU	-	7.386
Human-1	236'746	3.623
Human-2	125'320	1.918
Human-4	67'861	1.038
FP16	4'366'984	37.866
FP16-opt	1'242'094	10.858

Table 23: Latency of the different solutions.

In terms of both hardware and time latencies, the table above shows that the hardware solution written by a human developer outperforms software and the other hardware solutions. We also notice the performances increase when parallelizing multiple convolutional blocks.
We observe that adding simple code transformations to the generated Vivado HLS code reduces the hardware latency by a factor of almost 4. In terms of time latency, the FP16-opt solution almost reaches the software solution.

5.5.5 Summary

From Table 19, we noticed that truncating the weights width to 16-bit from 32-bit doesn't change the accuracy much. This observation is valid for both hardware solutions.

In terms of memory usage, hardware solutions beat the software solution by a factor of two thirds. We saw that adding convolutional blocks to the human-developed hardware solution increases the memory usage but stays way under the software solution usage. However, the FP16 solution requires more memory than the human-developed, even if up to 4 convolutional blocks are incorporated in the SystemVerilog module. We also saw that adding code transformations for pipelining loops doesn't increase memory usage by a significant amount.
When comparing the resources usage of hardware solutions, the human-developed solution clearly outperforms the solution generated by HeteroCL. For some resources (*i.e.* LUT), there is a difference of more than 60%. We also saw that the optimized FP16-opt solution uses almost all of the available LUTs and consumes about the double of FFs and DSP48E1 units than the FP16 hardware solution.

When it comes to timing, HeteroCL-generated IPs outperform the human-developed solution. In fact, we saw that both unoptimized and optimized solutions can operate at a maximum frequency of almost the double of the human-developed's maximum frequency.

However, concerning time latency, the generated IP is far behind the other two solutions. The latter performs the inference in more than 37 milliseconds, when the software solution do it in about 7 milliseconds. Finally, the best of the three solutions appears to be the hardware solution developed by a human developer. It performs inference in 3.623 milliseconds with only one convolutional block and it goes down to 1 millisecond with 4 convolutional blocks. We noticed that once code transformations have been applied to the HeteroCL solution, time latency has been reduced by almost a factor 4 for the same maximum frequency, thus decreasing to 10.858 milliseconds.

Ultimately, this comparison showed that the hardware solution written by Solovyev, Kustov, Telpukhov, *et al.* performs better for almost all points, except for the maximum frequency. The software solution defends defends itself rather well on all points of comparison. Moreover, the hardware solutions *automatically* generated with the HeteroCL framework keep up in term of memory usage but not in terms of overall resources usage. They offer the greatest maximum frequency but fails to keep pace in terms of latency, thus cancelling the interest for its maximum frequency. It is interesting to note that the optimized version of the HeteroCL-generated hardware solution doesn't require much knowledge to apply code transformations while offering a latency 4 times lower than the unoptimized version. However, optimizing the IP by pipelining its loops requires a lot of resources, and we noticed that the optimized version almost overflew in terms of LUTs.

6 Conclusion

In this work, we aimed at studying the available technologies and toolchains for generating hardware accelerators automatically. We especially looked for a solution that reduces or even ideally eliminates the hardware knowledge required to produce such an accelerator under normal circumstances.
We wanted to critically discuss the use of such a toolchain and compare the resulting generated hardware accelerators with other solutions. These solutions include both a software solution and a hardware solution developed by a human developer.
As a testbench, we designed a system in which we wanted to integrate the future generated accelerator. This system captures image with a camera and outputs a predicted digit it recognizes from the camera data.

We established a state of the art of such technologies. We listed every framework or toolchain that allow creating an hardware accelerator from an high-level programming language, such as Python, to an Hardware Description Language (HDL), such as VHDL or SystemVerilog.
For each framework, we presented its functionalities and we reviewed different criteria (*i.e.* Maintenance and update frequency, licensing, supported inputs and outputs). At the end of the state of the art, we went through each criterion one by one to exclude frameworks and find the best candidate among them. HeteroCL was decided to be the best framework to integrate in this project.

With HeteroCL chosen as the framework to generate an hardware accelerator, we then had to choose the accelerator we were aiming to create to use and later critically analyze and discuss the framework. Since Machine Learning is a modern computing field and it requires a lot of computing power, we chose to implement an accelerator for hand-written digits recognition. To classify these digits, we had to describe the Machine Learning model we will use, how to prepare it for training and its performances.
The model we chose for doing the classification task uses convolutional layers. It was trained with the MNIST dataset, which is slightly modified (*i.e.* inverted colors) to better suit our needs, on 60'000 images. After the testing phase, the Machine Learning model presented an accuracy of 97.56%.

Then, we used HeteroCL, the framework chosen to automatically generate Vivado High-Level Synthesis code in order to produce an hardware accelerator. We presented how to use the framework in sequence to describe the Machine Learning model presented previously. Once HeteroCL produced the High-Level Synthesis code, we applied some manual modifications so the accelerator better fits the testbench system. Initially, the accelerator fetched the input data and the model weights from memory. We changed the input source to a stream using the AXI-Stream protocol and including the model weights directly into the IP instead of having to load them from memory.
Having used HeteroCL and given its flaws and qualities, we critically discussed the quality of the code its produces and also discussed its utilization and what a developer should expect when using it. The code is easily readable by a human even if there is a lot of space for enhancement. About its utilization, we saw that it's quite impossible to not intervene during the generation. HeteroCL can't yet adapt to the system specifications and a qualified developer must verify and modify the generated High-Level Synthesis code.
For the sake of having an additional vector of comparison, we created an optimized version of the accelerator generated with HeteroCL. To optimize it, we added some code transformations, such as pipelining loops, to decrease the IP latency.

Finally, we compared the solutions. In terms of memory usage, the human-developed hardware solution shows the best results. It requires up to 5 times less than the software solution and 3 times less than the HeteroCL-generated solutions (both optimized and unoptimized).
Resources utilization could be compared only among hardware solutions. The human-developed solution completely outperforms the solutions generated by the framework. Indeed, the human-developed solution uses at most 9.04% of the Lookup Tables resources while the unoptimized HeteroCL solution requires 64.77% and up to 96.89% for the optimized version.
In terms of timing, the solutions generated with HeteroCL offer about 115 MHz as their maximum frequency while the human-developed solution proposes 65 MHz. However, the advantage of this performance is canceled by the latency of the HeteroCL-generated accelerators. Indeed, the latency is driven by clock cycles, and the human-developed solutions clearly outperform the other solutions. When performing the inference with only one convolutional block, the human-developed solution requires 236'746 clock cycles and down to 67'861 when it has four convolutional blocks. In the mean time, HeteroCL-generated solutions need 4'366'984 clock cycles for the unoptimized version and 1'242'094 clock cycles for the optimized version. At their maximum frequency, the unoptimized version performs the inference in about 37 milliseconds, while the optimized version can do it in less than 11 milliseconds. Human-developed version can perform the inference between 3.6 and

1.0 milliseconds. The software version does it in 7.4 milliseconds.

In conclusion, HeteroCL achieves the generation of hardware accelerators. The accelerator we aimed at generating in this work was functional. However, it needed to be tuned a bit by hand to respect the system characteristics, such as fetching input data using the AXI-Stream protocol. In terms of performance, HeteroCL doesn't reach yet software and human-developed hardware solutions. However, we demonstrated that it is possible to apply some relatively simple code transformations in order to pipeline loops. These transformations allow to reduce the inference latency by almost 4 times, but require greater resources.
Despite achieving the generation of hardware accelerators, we still are far from the ideal case where the user doesn't have to intervene after the generation. And even if it was the case, there is still the need to integrate the accelerator into a design system to make use of it, and this requires hardware knowledge.
It is also important to note that HeteroCL and all the other frameworks presented in the section 2 are very new technologies that are only a few months, even weeks or days old in some cases. Given their state of development today and what they can already achieve, we are more than confident that these technologies will rapidly evolve and improve in the near future.

It is important to note that this project aimed at generating an hardware accelerator that does Machine Learning inference, which implies important operations such as 4-dimensional convolutional operations. The results obtained here are valid for complex operations but for less important computational loads (*e.g.* 2-dimensional matrix multiplication), HeteroCL might show better results than both software and human-developed hardware solutions.
Additionally, the code transformations that have been manually applied to produce an optimized version of the HLS code generated by HeteroCL, were quite simple. They only consisted of pipelining some loops. It is more than likely that better and smarter code transformations can be applied to reach even better tradeoffs.

The future development of such technologies promises to be interesting and to find applications in a wide variety of domains. These domains could be data centers (*e.g.* processing queries), autonomous cars or personal computers. For the latter, an idea would be to make use of the Non-Volatile Memory Express (NVME) interface to invoke the FPGA with a bitstream to perform a specific task, such as file-zipping, calculating SHA256 sum, and so on.

References

[1] J. Vidal, "Tsunami of data could consume one fifth of global electricity by 2025", December 11, 2017. [Online]. Available: https://www.climatechangenews.com/2017/12/11/tsunami-data-consume-one-fifth-global-electricity-2025/ (visited on December 22, 2019).

[2] T. Ching, D. S. Himmelstein, B. K. Beaulieu-Jones, A. A. Kalinin, B. T. Do, G. P. Way, E. Ferrero, P.-M. Agapow, M. Zietz, M. M. Hoffman, W. Xie, G. L. Rosen, B. J. Lengerich, J. Israeli, J. Lanchantin, S. Woloszynek, A. E. Carpenter, A. Shrikumar, J. Xu, E. M. Cofer, C. A. Lavender, S. C. Turaga, A. M. Alexandari, Z. Lu, D. J. Harris, D. DeCaprio, Y. Qi, A. Kundaje, Y. Peng, L. K. Wiley, M. H. S. Segler, S. M. Boca, S. J. Swamidass, A. Huang, A. Gitter, and C. S. Greene, "Opportunities and obstacles for deep learning in biology and medicine", *Journal of The Royal Society Interface*, vol. 15, no. 141, 2018. DOI: 10.1098/rsif.2017.0387. [Online]. Available: https://royalsocietypublishing.org/doi/abs/10.1098/rsif.2017.0387.

[3] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning", *Nature*, vol. 521, pp. 436–44, May 2015. DOI: 10.1038/nature14539.

[4] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, *et al.*, "End to end learning for self-driving cars", 2016. [Online]. Available: https://arxiv.org/pdf/1604.07316.pdf.

[5] A. Esteva, A. Robicquet, B. Ramsundar, V. Kuleshov, M. DePristo, K. Chou, C. Cui, G. Corrado, S. Thrun, and J. Dean, "A guide to deep learning in healthcare", *Nature Medicine*, vol. 25, January 2019. DOI: 10.1038/s41591-018-0316-z.

[6] A. A. Shvets, A. Rakhlin, A. A. Kalinin, and V. I. Iglovikov, "Automatic Instrument Segmentation in Robot-Assisted Surgery using Deep Learning", December 2018, pp. 624–628. DOI: 10.1109/ICMLA.2018.00100.

[7] T. P. Morgan, "How Microsoft Is Using FPGAs To Speed Up Bing Search", Sep. 3, 2014. [Online]. Available: https://www.enterpriseai.news/2014/09/03/microsoft-using-fpgas-speed-bing-search/ (visited on December 22, 2019).

[8] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello, "Hardware accelerated convolutional neural networks for synthetic vision systems", May 2010, pp. 257–260. DOI: 10.1109/ISCAS.2010.5537908.

[9] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A Performance and Energy Comparison of FPGAs, GPUs, and Multicores for Sliding-Window Applications", 2012. DOI: 10.1145/2145694.2145704. [Online]. Available: https://doi.org/10.1145/2145694.2145704.

[10] G. Van der Wal, D. Zhang, I. Kandaswamy, J. Marakowitz, K. Kaighn, J. Zhang, and S. Chai, "FPGA Acceleration for Feature Based Processing Applications", Jun. 2015. [Online]. Available: https://www.cv-foundation.org/openaccess/content_cvpr_workshops_2015/W12/papers/Wal_FPGA_Acceleration_for_2015_CVPR_paper.pdf.

[11] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno, and P. H. Jones, "Comparing Energy Efficiency of CPU, GPU and FPGA Implementations for Vision Kernels", 2019. [Online]. Available: https://arxiv.org/pdf/1906.11879.pdf.

[12] "Vivado Design Suite", [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html (visited on December 23, 2019).

[13] "Intel Quartus Prime", [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html (visited on December 23, 2019).

[14] "Pcam 5C Reference Manual", [Online]. Available: https://reference.digilentinc.com/reference/add-ons/pcam-5c/reference-manual (visited on January 11, 2020).

[15] "OV5640 Datasheet", [Online]. Available: https://cdn.sparkfun.com/datasheets/Sensors/LightImaging/OV5640_datasheet.pdf (visited on January 11, 2020).

[16] "ZC702 Evaluation Boardfor the Zynq-7000 XC7Z020 SoC - User Guide", [Online]. Available: https://www.xilinx.com/support/documentation/boards_and_kits/zc702_zvik/ug850-zc702-eval-bd.pdf (visited on January 11, 2020).

[17] "FMC Pcam Adapter Reference Manual", [Online]. Available: https://reference.digilentinc.com/reference/add-ons/fmc-pcam-adapter/reference-manual (visited on January 11, 2020).

[18] "Zybo Z7 Board Reference Manual", [Online]. Available: https://reference.digilentinc.com/_media/reference/programmable-logic/zybo-z7/zybo-z7_rm.pdf (visited on January 9, 2020).

[19] "Zybo Z7 Pcam 5C Demo", [Online]. Available: https://reference.digilentinc.com/learn/programmable-logic/tutorials/zybo-z7-pcam-5c-demo/start (visited on January 30, 2020).

[20] "AXI Video Direct Memory Access v6.2", [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_vdma/v6_2/pg020_axi_vdma.pdf (visited on January 11, 2020).

[21] "DnnWeaver v2.0", [Online]. Available: http://dnnweaver.org (visited on December 23, 2019).

[22] "DnnWeaver Github Repository", [Online]. Available: https://github.com/hsharma35/dnnweaver2 (visited on December 23, 2019).

[23] "Apache License, Version 2.0", [Online]. Available: https://opensource.org/licenses/Apache-2.0 (visited on December 23, 2019).

[24] "FPGA Caffe", [Online]. Available: https://github.com/dicecco1/fpga_caffe (visited on December 23, 2019).

[25] "The 2-Clause BSD License", [Online]. Available: https://opensource.org/licenses/BSD-2-Clause (visited on December 23, 2019).

[26] "NVDLA, NVIDIA Deep Learning Accelerator", [Online]. Available: http://nvdla.org (visited on December 23, 2019).

[27] "NVDLA GitHub", [Online]. Available: https://github.com/nvdla (visited on December 23, 2019).

[28] "NVDLA License", [Online]. Available: http://nvdla.org/license.html (visited on December 23, 2019).

[29] "GreenSocs", [Online]. Available: https://www.greensocs.com (visited on December 23, 2019).

[30] Toshiba, "Toshiba's Considerations for NVDLA-based DNN SoC Design", January 24, 2019. [Online]. Available: https://toshiba.semicon-storage.com/content/dam/toshiba-ss/ncsa/en_us/docs/white-paper/Considerations_for_NVDLA-Based_DNN_SoC_Design_Whitepaper.pdf.

[31] "ONNC", [Online]. Available: https://onnc.ai (visited on January 13, 2020).

[32] "Github Pull Request – NVDLA running on a FPGA platform", [Online]. Available: https://github.com/nvdla/hw/issues/110 (visited on December 23, 2019).

[33] "Hls4ml", [Online]. Available: https://github.com/hls-fpga-machine-learning/hls4ml (visited on December 23, 2019).

[34] "TVM", [Online]. Available: https://tvm.ai (visited on December 23, 2019).

[35] "Pull Request for Supporting Intel FPGA in VTA", [Online]. Available: https://github.com/apache/incubator-tvm/pull/3258 (visited on December 23, 2019).

[36] "The Apache Incubator", [Online]. Available: https://incubator.apache.org/ (visited on January 13, 2020).

[37] "TVM and Deep Learning Compilation Conference", [Online]. Available: https://sampl.cs.washington.edu/tvmconf/ (visited on January 13, 2020).

[38] "LegUp", [Online]. Available: http://legup.eecg.utoronto.ca (visited on December 23, 2019).

[39] "LeFlow", [Online]. Available: https://github.com/danielholanda/LeFlow (visited on December 23, 2019).

[40] "The MIT License", [Online]. Available: https://opensource.org/licenses/MIT (visited on December 23, 2019).

[41] "nGraph", [Online]. Available: https://www.ngraph.ai/ (visited on January 13, 2020).

[42] "Nervana Systems - Wikipedia", [Online]. Available: https://en.wikipedia.org/wiki/Nervana_Systems (visited on December 23, 2019).

[43] "nGraph Github Repository", [Online]. Available: https://github.com/NervanaSystems/ngraph (visited on December 23, 2019).

[44] "nGraph - Hardware and backend support", [Online]. Available: https://www.ngraph.ai/ecosystem#hardware-&-backend-support (visited on January 13, 2020).

[45] "HeteroCL Github Repository", [Online]. Available: https://github.com/cornell-zhang/heterocl (visited on December 23, 2019).

[46] "ESP - open SoC platform", [Online]. Available: https://www.esp.cs.columbia.edu/ (visited on January 8, 2020).

[47] "ESP - Github Repository", [Online]. Available: https://github.com/sld-columbia/esp (visited on January 8, 2020).

[48] L. Elisei, "FPGA-based Accelerator for Machine Learning Inference", Jun. 2019. [Online]. Available: https://faku99.github.io/files/elisei-PA.pdf.

[49] "GNU Radio OOT module for DNN activity", [Online]. Available: https://gitlab.com/librespacefoundation/sdrmakerspace/gr-dnn/-/wikis/home (visited on January 13, 2020).

[50] R. Solovyev, A. Kustov, D. Telpukhov, V. Rukhlov, and A. Kalinin, "Fixed-Point Convolutional Neural Network for Real-Time Video Processing in FPGA", *2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, January 2019. DOI: 10.1109/eiconrus.2019.8656778. [Online]. Available: http://dx.doi.org/10.1109/EIConRus.2019.8656778.

[51] Y. LeCun and C. Cortes, "MNIST handwritten digit database", 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/.

[52] "Keras Documentation", [Online]. Available: https://keras.io/ (visited on December 28, 2019).

[53] J. de Fine Licht, S. Meierhans, and T. Hoefler, "Transformations of high-level synthesis codes for high-performance computing", *CoRR, vol. abs/1805.08288*, 2018. [Online]. Available: http://htor.inf.ethz.ch/publications/img/hls-transformations.pdf.

[54] "ARM - Cortex-A9", [Online]. Available: https://developer.arm.com/ip-products/processors/cortex-a/cortex-a9 (visited on January 9, 2020).

[55] "TensorFlow Lite", [Online]. Available: https://www.tensorflow.org/lite/ (visited on January 21, 2020).

[56] "TensorFlow Lite C++ API Reference", [Online]. Available: https://www.tensorflow.org/lite/api_docs/cc (visited on January 21, 2020).

[57] "Massif: a heap profiler", [Online]. Available: https://valgrind.org/docs/manual/ms-manual.html (visited on January 21, 2020).

Acronyms

ANN Artificial Neural Network
AP SoC All Programmable System-on-Chip

BRAM Block Random Access Memory

CNN Convolutional Neural Network
CPU Central Processing Unit

DL Deep Learning
DMA Direct Memory Access
DNN Deep Neural Network
DSL Domain-Specific Language

ESP Embedded Scalable Platforms

FF Flip-Flop
FFC Flat-Flexible Cable
FPGA Field-Programmable Gate Array

GPU Graphics Processing Unit

HDL Hardware Description Language
HDMI High-Definition Multimedia Interface
HLS High-Level Synthesis

I^2C Inter-Integrated Circuit
ICT Information and Communications Technologies
IoT Internet of Things

JIT Just-in-Time

LRN Local Response Normalisation
LUT Lookup Table

ML Machine Learning
MLP Multi-Layer Perceptron

NN Neural Network
NoC Network-on-Chip
NVDLA NVIDIA Deep Learning Accelerator
NVME Non-Volatile Memory Express

ONNC Open Neural Network Compiler

ReLU Rectified Linear Unit

SDRAM Synchronous Dynamic Random Access Memory
SoC System-on-Chip

VDMA Video Direct Memory Access

VTA Versatile Tensor Accelerator

Appendices

A Zybo Z7-20 characteristics

The Zybo Z7 is a feature-rich, ready-to-use embedded software and digital circuit development board built around the Xilinx Zynq-7000 family. The Zynq family is based on the Xilinx All Programmable System-on-Chip (AP SoC) architecture, which tightly integrates a dual-core ARM Cortex-A9 processor with Xilinx 7-series FPGA logic.

The Zynq processor offers the following characteristics:

- 667 MHz dual-core Cortex-A9 processor
- Zynq-7000 (XC7Z020) FPGA
- 1 GB DDR3L
- USB, Ethernet, HDMI (in/out), Pmod, and Pcam ports
- Programmable from JTAG, Quad-SPI flash, and microSD card
- Programmable logic equivalent to Artix-7 FPGA

The FPGA XC7Z020 part has the following specifications:

- 13'300 Logic cells
- 53'200 6-input LUTs
- 106'400 Flip-Flops
- 280 18-Kb Block RAM (630 KB)
- 220 DSP Slices

Figure 12 shows the Zynq AP SoC architecture and Figure 13 presents the Zybo Z7-20 main components.

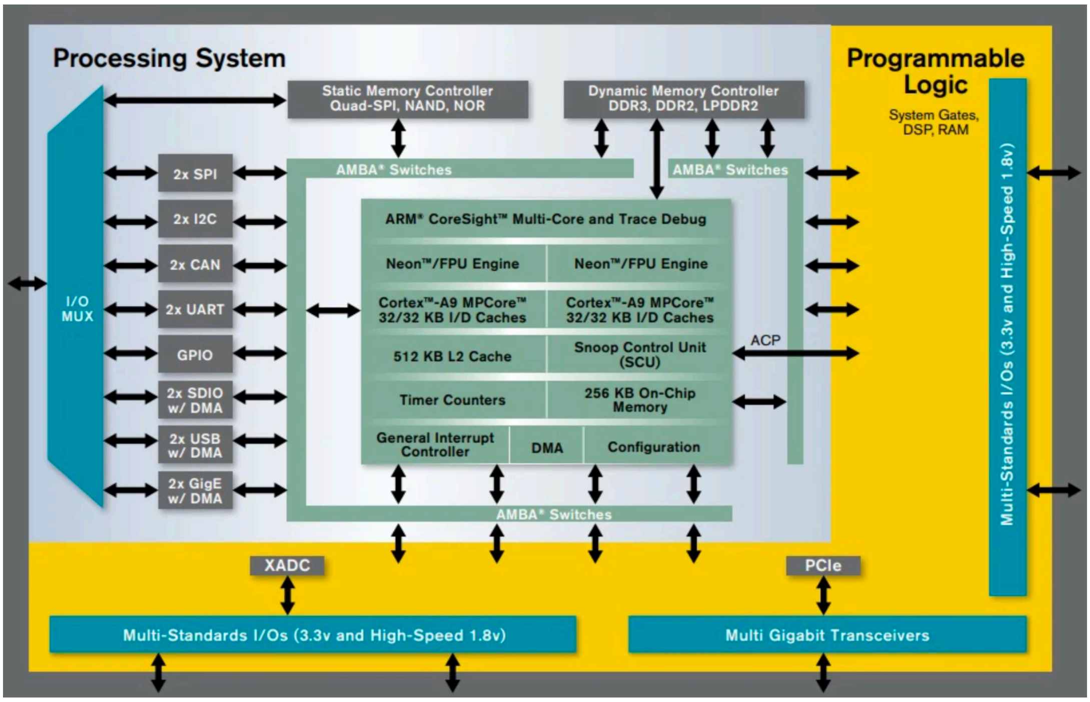


Figure 12: Zynq APSoC architecture.

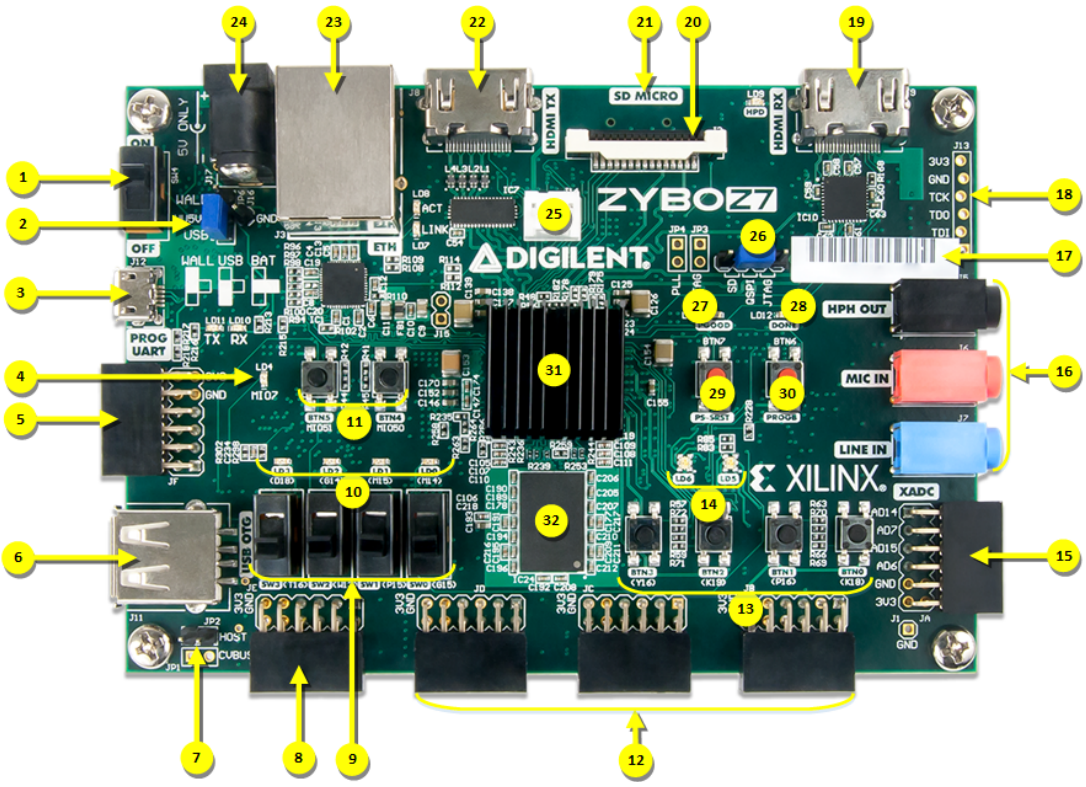


Figure 13: Zybo Z7-20.

1. Power switch
2. Power select jumper
3. USB JTAG/UART port
4. MIO user LED
5. MIO Pmod port
6. USB 2.0 Host/OTG port
7. USB Host power enable jumper
8. Standard Pmod port
9. User switches
10. User LEDs
11. MIO user buttons
12. High-speed Pmod ports
13. User buttons
14. User RGB LEDs
15. XADC Pmod port
16. Audio codec ports
17. Unique MAC address label
18. External JTAG port
19. HDMI input port
20. Pcam MIPI CSI-2 port
21. microSD connector
22. HDMI output port
23. Ethernet port
24. External power supply connector
25. Fan connector
26. Programming mode select jumper
27. Power supply good LED
28. FPGA programming done LED
29. Processor reset button
30. FPGA clear configuration button
31. Zynq-7000
32. DDR3L memory

B src/python/train_nn.py

```python
# Imports
import os
import numpy as np

# Set backend to Tensorflow
os.environ['KERAS_BACKEND'] = 'tensorflow'
# Disable GPU and use CPU
os.environ['CUDA_VISIBLE_DEVICES'] = '0'
# Disable Tensorflow logging
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

from PIL import Image

import tensorflow as tf
from keras.datasets import mnist
from keras.layers import Activation, Input, Dense, GlobalAveragePooling2D, GlobalMaxPooling2D, Conv2D, MaxPooling2D
from keras.models import Model
from keras.optimizers import Adam
from keras.utils import np_utils

import matplotlib.pyplot as plt

# Constants
DEBUG = False
IMG_H = 28
IMG_W = 28
WEIGHTS_PATH = 'mnist_weights.h5'
RESCALED_WEIGHTS_PATH = 'mnist_weights_rescaled.h5'

class DataFormatException(Exception):
    def __init__(self, data_format):
        super('Wrong data_format specified: {}'.format(data_format))

def dbg_print(msg):
    if (DEBUG is True):
        print('[DEBUG] {}'.format(msg))

def load_mnist_data(data_format='channels_first'):
    '''
    Load MNIST dataset.

    Parameters
    ----------
    data_format : str, optional
        The data format used (default is 'channels_first')

    Return
    -------
    train_x, train_y, test_x, test_y : array, array, array, array
        Training images, training labels, testing images, testing labels.
    '''
    (train_x, train_y), (test_x, test_y) = mnist.load_data()

    # Reshape data according to the specified data format
    if (data_format is 'channels_first'):
        train_x = train_x.reshape(train_x.shape[0], 1, IMG_H, IMG_W)
        test_x  = test_x.reshape(test_x.shape[0], 1, IMG_H, IMG_W)
    elif (data_format is 'channels_last'):
        train_x = train_x.reshape(train_x.shape[0], IMG_H, IMG_W, 1)
        test_x  = test_x.reshape(test_x.shape[0], IMG_H, IMG_W, 1)
    else:
        raise DataFormatException(data_format)

    # Convert to float32
    train_x = train_x.astype('float32')
    test_x  = test_x.astype('float32')

    dbg_print('train_x.shape: {}'.format(train_x.shape))
    dbg_print('test_x.shape : {}'.format(test_x.shape))
```

src/python/train_nn.py

```python

    # Convert class vectors to binary class matrices
    train_y = np_utils.to_categorical(train_y, 10)
    test_y  = np_utils.to_categorical(test_y, 10)

    # Invert images (from white on black to black on white)
    train_x = 255 - train_x
    test_x  = 255 - test_x

    return train_x, train_y, test_x, test_y

def build_model(data_format='channels_first', use_bias=False):
    '''
    Define network architecture.

    Parameters
    ----------
    data_format : str, optional
        The data format used (default is 'channels_first').
    use_bias    : boolean, optional
        Whether to use bias or not (default is False).

    Return
    -------
    model       : keras.models.Model
        The Keras model of the network.
    '''
    # Input is 28x28 grayscale (1 component) pixels
    if (data_format is 'channels_first'):
        input = Input((1, IMG_H, IMG_W))
    elif (data_format is 'channels_last'):
        input = Input((IMG_H, IMG_W, 1))
    else:
        raise DataFormatException(data_format)

    conv1 = Conv2D(4, (3,3), activation='relu', padding='same', data_format=data_format, name='conv1', use_bias=use_bias)(input)
    conv2 = Conv2D(4, (3,3), activation='relu', padding='same', data_format=data_format, name='conv2', use_bias=use_bias)(conv1)
    pool1 = MaxPooling2D((2,2), strides=(2,2), data_format=data_format, name='pool1')(conv2)

    conv3 = Conv2D(8, (3,3), activation='relu', padding='same', data_format=data_format, name='conv3', use_bias=use_bias)(pool1)
    conv4 = Conv2D(8, (3,3), activation='relu', padding='same', data_format=data_format, name='conv4', use_bias=use_bias)(conv3)
    pool2 = MaxPooling2D((2,2), strides=(2,2), data_format=data_format, name='pool2')(conv4)

    conv5 = Conv2D(16, (3,3), activation='relu', padding='same', data_format=data_format, name='conv5', use_bias=use_bias)(pool2)
    conv6 = Conv2D(16, (3,3), activation='relu', padding='same', data_format=data_format, name='conv6', use_bias=use_bias)(conv5)
    pool3 = GlobalMaxPooling2D(data_format=data_format, name='pool3')(conv6)

    dense1 = Dense(10, activation=None, use_bias=use_bias)(pool3)
    output = Activation('softmax')(dense1)

    model = Model(inputs=input, outputs=output)
    model.summary(print_fn=dbg_print)

    return model

def train_model(predict_image=False, data_format='channels_first') -> Model:
    '''
    Build and train the neural network model, then save into the file defined by
    the `WEIGHTS_PATH` constant.
    If the file already exists, load it.

    Parameters
    ----------
    predict_image   : boolean, optional
        Whether to predict class from `test_image.png` or not (default is False).

    Return
```

```python
    ------
    model   : keras.models.Model
        The trained model.
    '''
    # Build Keras model
    model = build_model(data_format=data_format)

    # Check if model has already been trained
    if (os.path.isfile(WEIGHTS_PATH) is False):
        print('Weights file ({}) could not be found. Initiating model training...'.format(WEIGHTS_PATH))
        # Load MNIST dataset
        train_x, train_y, test_x, test_y = load_mnist_data(data_format=data_format)

        # Training parameters
        learning_rate = 0.001
        optimizer = Adam(lr=learning_rate)
        loss='categorical_crossentropy'
        metrics=['accuracy']
        batch_size = 128
        epochs = 10

        model.compile(optimizer=optimizer, loss=loss, metrics=metrics)

        model.fit(train_x, train_y, batch_size=batch_size, epochs=epochs, verbose=1, validation_data=(test_x, test_y))
        score = model.evaluate(test_x, test_y, verbose=0)

        print('Model score   :', score[0])
        print('Model accuracy:', score[1])

        model.save(WEIGHTS_PATH)
        print('Model saved into {}'.format(WEIGHTS_PATH))
    else:
        print('Weights file exists. Loading model...')
        model.load_weights(WEIGHTS_PATH)

    if (predict_image is True):
        test_img = np.asarray(Image.open('test_image.png'), dtype='float32')
        test_img = np.reshape(test_img, (1,1,IMG_H,IMG_W))

        preds = model.predict(test_img, batch_size=1)
        print('predictions: {}'.format(preds))
        print('predicted  : {} (expected: 3)'.format(np.argmax(preds, axis=1)))

    return model

def rescale_weights(model: Model, data_format='channels_first') -> Model:
    # Coefficient to make safe gap for found range to prevent overflow. Lower = less safe. Higher = more rounding error.
    GAP_COEFF = 1.1

    # Check if weights have already been rescaled
    if (os.path.isfile(RESCALED_WEIGHTS_PATH) is False):
        print('Rescaled weights file ({}) could not be found. Initiating weights rescaling...'.format(RESCALED_WEIGHTS_PATH))
        # Get all images from MNIST dataset
        train_x, _, test_x, _ = load_mnist_data(data_format=data_format)
        x = np.concatenate((train_x, test_x))

        # Initialize reduction rate
        reduction_rate = 1.0

        # Iterate over layers
        for layer in model.layers:
            dbg_print('Getting min and max value for layer \'{}\''.format(layer.name))

            weights = layer.get_weights()
            # Check there are weights available
            if (len(weights) > 0):
                # Extract submodel from original model
                submodel = Model(inputs=model.inputs, outputs=layer.output)
                submodel.summary(print_fn=dbg_print)
```

```python
                preds = submodel.predict(x)

                # We're looking for the most out and about the scales. Weights should not exceed 1.0 including.
                coeff = GAP_COEFF * max(abs(preds.min()), abs(preds.max()), abs(weights[0].min()), abs(weights[0].max()))

                dbg_print(' Submodel shape   : {}'.format(preds.shape))
                dbg_print(' Min preds value  : {}, max: {}'.format(preds.min(), preds.max()))
                dbg_print(' Min weights value: {}, max: {}'.format(weights[0].min(), weights[0].max()))
                dbg_print(' Reduction coeff  : {}'.format(coeff))

                # Rescale weights
                layer.set_weights(weights / coeff)
                reduction_rate = reduction_rate * coeff
            else:
                dbg_print('-> No weights available')
            # For better readability
            dbg_print('')

        model.save_weights(RESCALED_WEIGHTS_PATH)
        print('Rescaled weights saved into {}'.format(RESCALED_WEIGHTS_PATH))
        print('Reduction rate: {}'.format(reduction_rate))
    else:
        print('Rescaled weights file exists. Loading model...')
        model.load_weights(RESCALED_WEIGHTS_PATH)

    return model

def main():
    data_format = 'channels_last'

    print('=== MODEL TRAINING ===')
    model = train_model(predict_image=False, data_format=data_format)

    print('=== WEIGHTS RESCALING ===')
    model = rescale_weights(model, data_format=data_format)

    model.save('mnist_model_rescaled.h5')

    _, _, test_x, test_y = load_mnist_data(data_format=data_format)

    labels = np.argmax(test_y, axis=1)
    # predictions = np.argmax(model.predict(test_x), axis=1)
    # score = np.sum(np.equal(predictions, labels))
    predictions = np.zeros(labels.shape)
    times = np.zeros(labels.shape)

    import time
    for i in range(len(test_x)):
        image = test_x[i].reshape(1,28,28,1)

        time_start = time.perf_counter()
        preds = model.predict(image)
        time_end = time.perf_counter()

        predictions[i] = np.argmax(preds, axis=1)
        times[i] = time_end - time_start

    score = np.sum(np.equal(predictions, labels))
    print('\n-> Testing accuracy: {}%'.format((score / len(test_x) * 100.0)))
    print('-> Inference time: {} ms'.format(np.average(times) * 1000))

if __name__ == '__main__':
    main()
```

C src/python/generate_hls.py

```python
# pylint: disable=no-member,undefined-variable

# Imports
import h5py
import numpy as np

import heterocl as hcl
import hlib
import heterocl.tvm as tvm

from keras.datasets import mnist
from keras.utils import np_utils

from collections import OrderedDict

# Constants
DEBUG = False
IMG_H = 28
IMG_W = 28
WEIGHTS_FILE = 'mnist_weights_rescaled.h5'

# Let every operation be floating-point
hcl.init(hcl.Float())

def dbg_print(msg):
    if (DEBUG is True):
        print('[DEBUG] {}'.format(msg))

class DataFormatException(Exception):
    def __init__(self, data_format):
        super('Wrong data_format specified: {}'.format(data_format))

# TODO: Include in hlib
def global_max_pool(data, name='global_max_pool'):
    assert len(data.shape) == 4, 'only support 4-dim global pooling'
    batch, channels, height, width = data.shape

    kernel = (height,width)
    stride = (1,1)
    max_pool = hlib.nn.max_pool(data, kernel, stride)

    return hcl.compute(
        (batch, channels),
        lambda i,c: max_pool[i,c,0,0],
        dtype=data.dtype,
        name=name,
        attrs=OrderedDict([('app_name', tvm.make.StringImm('global_max_pool'))])
    )

def load_mnist_data(data_format='channels_first'):
    '''Load MNIST dataset.
    Parameters
    ----------
    data_format : str, optional
        The data format used (default is 'channels_first')
    '''
    (train_x, train_y), (test_x, test_y) = mnist.load_data()

    # Reshape data according to the specified data format
    if (data_format is 'channels_first'):
        train_x = train_x.reshape(train_x.shape[0], 1, IMG_H, IMG_W)
        test_x  = test_x.reshape(test_x.shape[0], 1, IMG_H, IMG_W)
    elif (data_format is 'channels_last'):
        train_x = train_x.reshape(train_x.shape[0], IMG_H, IMG_W, 1)
        test_x  = test_x.reshape(test_x.shape[0], IMG_H, IMG_W, 1)
    else:
        raise DataFormatException(data_format)

    # Convert to float32
    train_x = train_x.astype('float32')
```

```
    test_x  = test_x.astype('float32')

    dbg_print('train_x.shape: {}'.format(train_x.shape))
    dbg_print('train_y.shape: {}'.format(train_y.shape))
    dbg_print('test_x.shape : {}'.format(test_x.shape))
    dbg_print('test_y.shape : {}'.format(test_y.shape))

    # Convert class vectors to binary class matrices
    train_y = np_utils.to_categorical(train_y, 10)
    test_y  = np_utils.to_categorical(test_y, 10)

    # Invert images (from white on black to black on white)
    train_x = 255 - train_x
    test_x  = 255 - test_x

    return train_x, train_y, test_x, test_y

def get_weights() -> dict:
    # Initialize weights dict
    weights = dict()

    # Open weights file
    f = h5py.File(WEIGHTS_FILE, 'r')

    # Iterate over layers (reshape is needed because of format divergence between HeteroCL and Keras weights)
    weights['conv1']   = np.asarray(    f['conv1']['conv1']['kernel:0'], dtype='float32').transpose(3,2,0,1)#.reshape(  (4,1,3,3))
    weights['conv2']   = np.asarray(    f['conv2']['conv2']['kernel:0'], dtype='float32').transpose(3,2,0,1)#.reshape(  (4,4,3,3))
    weights['conv3']   = np.asarray(    f['conv3']['conv3']['kernel:0'], dtype='float32').transpose(3,2,0,1)#.reshape(  (8,4,3,3))
    weights['conv4']   = np.asarray(    f['conv4']['conv4']['kernel:0'], dtype='float32').transpose(3,2,0,1)#.reshape(  (8,8,3,3))
    weights['conv5']   = np.asarray(    f['conv5']['conv5']['kernel:0'], dtype='float32').transpose(3,2,0,1)#.reshape((16, 8,3,3))
    weights['conv6']   = np.asarray(    f['conv6']['conv6']['kernel:0'], dtype='float32').transpose(3,2,0,1)#.reshape((16,16,3,3))
    weights['dense_1'] = np.asarray(f['dense_1']['dense_1']['kernel:0'], dtype='float32')

    dbg_print('weights:')
    for key in weights.keys():
        dbg_print(' {}: {}'.format(key, weights[key].shape))

    return weights

def build_mnist(input_image, w_conv1, w_conv2, w_conv3, w_conv4, w_conv5, w_conv6, w_dense1, output):
    # First convolutional layer
    conv1 = hlib.nn.conv2d_nchw(input_image, w_conv1, padding='SAME', name='conv1')
    dbg_print(conv1)
    relu1 = hlib.nn.relu(conv1, name='relu1')
    dbg_print(relu1)

    # Second convolutional layer
    conv2 = hlib.nn.conv2d_nchw(relu1, w_conv2, padding='SAME', name='conv2')
    dbg_print(conv2)
    relu2 = hlib.nn.relu(conv2, name='relu2')
    dbg_print(relu2)

    # First max pooling
    pool1 = hlib.nn.max_pool(relu2, kernel=(2,2), stride=(2,2), name='pool1')
    dbg_print(pool1)

    # Third convolutional layer
    conv3 = hlib.nn.conv2d_nchw(pool1, w_conv3, padding='SAME', name='conv3')
    dbg_print(conv3)
    relu3 = hlib.nn.relu(conv3, name='relu3')
    dbg_print(relu3)

    # Fourth convolutional layer
    conv4 = hlib.nn.conv2d_nchw(relu3, w_conv4, padding='SAME', name='conv4')
    dbg_print(conv4)
    relu4 = hlib.nn.relu(conv4, name='relu4')
```

```
    dbg_print(relu4)

    # Second max pooling
    pool2 = hlib.nn.max_pool(relu4, kernel=(2,2), stride=(2,2), name='pool2')
    dbg_print(pool2)

    # Fifth convolutional layer
    conv5 = hlib.nn.conv2d_nchw(pool2, w_conv5, padding='SAME', name='conv5')
    dbg_print(conv5)
    relu5 = hlib.nn.relu(conv5, name='relu5')
    dbg_print(relu5)

    # Sixth convolutional layer
    conv6 = hlib.nn.conv2d_nchw(relu5, w_conv6, padding='SAME', name='conv6')
    dbg_print(conv6)
    relu6 = hlib.nn.relu(conv6, name='relu6')
    dbg_print(relu6)

    # Third max pooling
    pool3 = global_max_pool(relu6, name='pool3')
    dbg_print(pool3)

    # Output layer
    dense1 = hlib.nn.dense(pool3, w_dense1, name='dense1')
    dbg_print(dense1)

    return hlib.nn.softmax(output, dense1)

def build_mnist_inf(batch_size, weights, qtype1, qtype2, target=None):
    # Set placeholders
    input_image = hcl.placeholder((batch_size,1,IMG_H,IMG_W),    name='input')
    w_conv1     = hcl.placeholder(    weights['conv1'].shape,  name='w_conv1', dtype=qtype1)
    w_conv2     = hcl.placeholder(    weights['conv2'].shape,  name='w_conv2', dtype=qtype1)
    w_conv3     = hcl.placeholder(    weights['conv3'].shape,  name='w_conv3', dtype=qtype1)
    w_conv4     = hcl.placeholder(    weights['conv4'].shape,  name='w_conv4', dtype=qtype1)
    w_conv5     = hcl.placeholder(    weights['conv5'].shape,  name='w_conv5', dtype=qtype1)
    w_conv6     = hcl.placeholder(    weights['conv6'].shape,  name='w_conv6', dtype=qtype1)
    w_dense1    = hcl.placeholder(  weights['dense_1'].shape, name='w_dense1', dtype=qtype1)
    output      = hcl.placeholder((batch_size,10), name='output')

    # Create quantization scheme
    scheme = hcl.create_scheme(
        [input_image, w_conv1, w_conv2, w_conv3, w_conv4, w_conv5, w_conv6, w_dense1, output],
        build_mnist
    )

    # Quantize activation layers
    scheme.quantize(
        [build_mnist.relu1, build_mnist.relu2, build_mnist.relu3, build_mnist.relu4, build_mnist.relu5,
         ↪   build_mnist.relu6],
        qtype2
    )

    s = hcl.create_schedule_from_scheme(scheme)

    return hcl.build(s, target=target)


def export_weights(weights):
    import os
    import shutil
    import sys

    if (os.path.exists('weights')):
        shutil.rmtree('weights')
    os.mkdir('weights')

    for name in weights.keys():
        s = str()
        data = weights[name]
        s += 'const static float w_{}'.format(name)
        for dim in data.shape:
            s += '[{}]'.format(dim)
```

```
        s += ' = \n'
        s += '{}'.format(np.array2string(data, max_line_width=80, separator=',',
        ↪   threshold=sys.maxsize).replace('[', '{').replace(']', '}'))
        s += ';'

        file = open('weights/w_{}.h'.format(name), 'w')
        file.write(s)
        file.close()


def get_accuracy(weights, qtype1, qtype2):
    batch_size = 1
    accuracy = 0

    print('Using qtype1={}, qtype2={}'.format(qtype1, qtype2))

    f = build_mnist_inf(batch_size, weights, qtype1, qtype2)

    # Prepare numpy weights for testing
    w_conv1_hcl  = hcl.asarray(  weights['conv1'], dtype=qtype1)
    w_conv2_hcl  = hcl.asarray(  weights['conv2'], dtype=qtype1)
    w_conv3_hcl  = hcl.asarray(  weights['conv3'], dtype=qtype1)
    w_conv4_hcl  = hcl.asarray(  weights['conv4'], dtype=qtype1)
    w_conv5_hcl  = hcl.asarray(  weights['conv5'], dtype=qtype1)
    w_conv6_hcl  = hcl.asarray(  weights['conv6'], dtype=qtype1)
    w_dense1_hcl = hcl.asarray(weights['dense_1'], dtype=qtype1)

    _, _, test_x, test_y = load_mnist_data()

    for i in range(len(test_x) // batch_size):
        label = np.argmax(test_y[i*batch_size:(i+1)*batch_size], axis=1)
        input_image_np = test_x[i*batch_size:(i+1)*batch_size]
        input_image_hcl = hcl.asarray(input_image_np)
        output_hcl = hcl.asarray(np.zeros((batch_size,10)))

        f(input_image_hcl,
            w_conv1_hcl, w_conv2_hcl, w_conv3_hcl, w_conv4_hcl, w_conv5_hcl, w_conv6_hcl, w_dense1_hcl,
            output_hcl)

        prediction = np.argmax(output_hcl.asnumpy(), axis=1)
        accuracy += np.sum(np.equal(prediction, label))

    print("-> Testing accuracy: {}%".format(accuracy / len(test_x) * 100.0))

    # Get Vivado HLS code
    f = build_mnist_inf(batch_size, weights, qtype1, qtype2, target='vhls')

    file = open('hls_fp{}.cpp'.format(qtype2.bits), 'w')

    file.write('/**\n')
    file.write(' * Testing accuracy: {}\n'.format(float(accuracy / float(len(test_x)))))
    file.write(' */\n\n')
    file.write(f)

    file.close()

def main():
    weights = get_weights()

    if False:
        qtypes = [hcl.Fixed(32,30), hcl.Fixed(16,14), hcl.Fixed(8,6), hcl.Fixed(4,2)]

        for qtype in qtypes:
            get_accuracy(weights, hcl.Float(), qtype)

    else:
        export_weights(weights)

if __name__ == '__main__':
    main()
```

D src/python/hls_fp32.cpp

```cpp
/**
 * Testing accuracy: 0.9726
 */

#include <ap_int.h>
#include <ap_fixed.h>
#include <math.h>

void default_function(float input[1][1][28][28], float w_conv1[4][1][3][3], float w_conv2[4][4][3][3], float w_conv3[8][4][3][3], float w_conv4[8][8][3][3], float w_conv5[16][8][3][3], float w_conv6[16][16][3][3], float w_dense1[16][10], float output[1][10]) {
  float pad_temp[1][1][30][30];
  for (ap_int<32> index_tuple = 0; index_tuple < 30; ++index_tuple) {
    for (ap_int<32> i = 0; i < 30; ++i) {
      pad_temp[0][0][index_tuple][i] = (((((1 <= index_tuple) && (index_tuple < 29)) && (1 <= i)) && (i < 29)) ? input[((((i - ((i + -1) % 28)) + (index_tuple * 28)) + -29) / 784)][0][(((((i - ((i + -1) % 28)) + (index_tuple * 28)) + -29) / 28) % 28)][((i + -1) % 28)] : 0.000000e+00f);
    }
  }
  float conv1[1][4][28][28];
  for (ap_int<32> ff = 0; ff < 4; ++ff) {
    for (ap_int<32> yy = 0; yy < 28; ++yy) {
      for (ap_int<32> xx = 0; xx < 28; ++xx) {
        float reducer36;
        reducer36 = 0.000000e+00f;
        for (ap_int<32> ry = 0; ry < 3; ++ry) {
          for (ap_int<32> rx = 0; rx < 3; ++rx) {
            reducer36 = ((pad_temp[0][0][(yy + ry)][(xx + rx)] * w_conv1[ff][0][ry][rx]) + reducer36);
          }
        }
        conv1[0][ff][yy][xx] = reducer36;
      }
    }
  }
  ap_fixed<32, 2> relu1[1][4][28][28];
  for (ap_int<32> args = 0; args < 1; ++args) {
    for (ap_int<32> args0 = 0; args0 < 4; ++args0) {
      for (ap_int<32> args1 = 0; args1 < 28; ++args1) {
        for (ap_int<32> args2 = 0; args2 < 28; ++args2) {
          relu1[args][args0][args1][args2] = ((ap_fixed<32, 2>)((conv1[args][args0][args1][args2] < 0.000000e+00f) ? 0.000000e+00f : conv1[args][args0][args1][args2]));
        }
      }
    }
  }
  float pad_temp1[1][4][30][30];
  for (ap_int<32> not_zero = 0; not_zero < 4; ++not_zero) {
    for (ap_int<32> index_tuple1 = 0; index_tuple1 < 30; ++index_tuple1) {
      for (ap_int<32> i1 = 0; i1 < 30; ++i1) {
        pad_temp1[0][not_zero][index_tuple1][i1] = (((((1 <= index_tuple1) && (index_tuple1 < 29)) && (1 <= i1)) && (i1 < 29)) ? ((float)relu1[(((((i1 - ((i1 + -1) % 28)) + (index_tuple1 * 28)) + (not_zero * 784)) + -29) / 3136)][((((((i1 - ((i1 + -1) % 28)) + (index_tuple1 * 28)) + (not_zero * 784)) + -29) / 784) % 4)][((((((i1 - ((i1 + -1) % 28)) + (index_tuple1 * 28)) + (not_zero * 784)) + -29) / 28) % 28)][((i1 + -1) % 28)]) : 0.000000e+00f);
      }
    }
  }
  float conv2[1][4][28][28];
  for (ap_int<32> ff1 = 0; ff1 < 4; ++ff1) {
    for (ap_int<32> yy1 = 0; yy1 < 28; ++yy1) {
      for (ap_int<32> xx1 = 0; xx1 < 28; ++xx1) {
        ap_fixed<32, 2> reducer37;
        reducer37 = ((ap_fixed<32, 2>)0);
        for (ap_int<32> rc = 0; rc < 4; ++rc) {
          for (ap_int<32> ry1 = 0; ry1 < 3; ++ry1) {
            for (ap_int<32> rx1 = 0; rx1 < 3; ++rx1) {
              reducer37 = ((ap_fixed<32, 2>)(((ap_fixed<65, 5>)(((ap_fixed<64, 34>)((ap_fixed<32, 2>)pad_temp1[0][rc][(yy1 + ry1)][(xx1 + rx1)])) * ((ap_fixed<64, 34>)((ap_fixed<32, 2>)w_conv2[ff1][rc][ry1][rx1])))) + ((ap_fixed<65, 5>)reducer37)));
            }
```

```cpp
          }
        }
        conv2[0][ff1][yy1][xx1] = ((float)reducer37);
      }
    }
  }
  ap_fixed<32, 2> relu2[1][4][28][28];
  for (ap_int<32> args3 = 0; args3 < 1; ++args3) {
    for (ap_int<32> args01 = 0; args01 < 4; ++args01) {
      for (ap_int<32> args11 = 0; args11 < 28; ++args11) {
        for (ap_int<32> args21 = 0; args21 < 28; ++args21) {
          relu2[args3][args01][args11][args21] = ((ap_fixed<32, 2>)((conv2[args3][args01][args11][args21] < 0.000000e+00f) ? 0.000000e+00f : conv2[args3][args01][args11][args21]));
        }
      }
    }
  }
  float pool1[1][4][14][14];
  for (ap_int<32> i2 = 0; i2 < 1; ++i2) {
    for (ap_int<32> c = 0; c < 4; ++c) {
      for (ap_int<32> h = 0; h < 14; ++h) {
        for (ap_int<32> w = 0; w < 14; ++w) {
          float reducer38;
          reducer38 = -1.000000e+00f;
          for (ap_int<32> ra27 = 0; ra27 < 2; ++ra27) {
            for (ap_int<32> ra28 = 0; ra28 < 2; ++ra28) {
              reducer38 = std::max(((float)relu2[i2][c][((h * 2) + ra27)][((w * 2) + ra28)]), reducer38);
            }
          }
          pool1[i2][c][h][w] = reducer38;
        }
      }
    }
  }
  float pad_temp2[1][4][16][16];
  for (ap_int<32> not_zero1 = 0; not_zero1 < 4; ++not_zero1) {
    for (ap_int<32> index_tuple2 = 0; index_tuple2 < 16; ++index_tuple2) {
      for (ap_int<32> i3 = 0; i3 < 16; ++i3) {
        pad_temp2[0][not_zero1][index_tuple2][i3] = (((((1 <= index_tuple2) && (index_tuple2 < 15)) && (1 <= i3)) && (i3 < 15)) ? pool1[((((((i3 - ((i3 + -1) % 14)) + (index_tuple2 * 14)) + (not_zero1 * 196)) + -15) / 784)][((((((i3 - ((i3 + -1) % 14)) + (index_tuple2 * 14)) + (not_zero1 * 196)) + -15) / 196) % 4)][((((((i3 - ((i3 + -1) % 14)) + (index_tuple2 * 14)) + (not_zero1 * 196)) + -15) / 14) % 14)][((i3 + -1) % 14)] : 0.000000e+00f);
      }
    }
  }
  float conv3[1][8][14][14];
  for (ap_int<32> ff2 = 0; ff2 < 8; ++ff2) {
    for (ap_int<32> yy2 = 0; yy2 < 14; ++yy2) {
      for (ap_int<32> xx2 = 0; xx2 < 14; ++xx2) {
        float reducer39;
        reducer39 = 0.000000e+00f;
        for (ap_int<32> rc1 = 0; rc1 < 4; ++rc1) {
          for (ap_int<32> ry2 = 0; ry2 < 3; ++ry2) {
            for (ap_int<32> rx2 = 0; rx2 < 3; ++rx2) {
              reducer39 = ((pad_temp2[0][rc1][(yy2 + ry2)][(xx2 + rx2)] * w_conv3[ff2][rc1][ry2][rx2]) + reducer39);
            }
          }
        }
        conv3[0][ff2][yy2][xx2] = reducer39;
      }
    }
  }
  ap_fixed<32, 2> relu3[1][8][14][14];
  for (ap_int<32> args4 = 0; args4 < 1; ++args4) {
    for (ap_int<32> args02 = 0; args02 < 8; ++args02) {
      for (ap_int<32> args12 = 0; args12 < 14; ++args12) {
        for (ap_int<32> args22 = 0; args22 < 14; ++args22) {
```

```
          relu3[args4][args02][args12][args22] = ((ap_fixed<32, 2>)((conv3[args4][args02][args12][args22] < 0.000000e+00f) ? 0.000000e+00f : conv3[args4][args02][args12][args22]));
        }
      }
    }
  }
  float pad_temp3[1][8][16][16];
  for (ap_int<32> not_zero2 = 0; not_zero2 < 8; ++not_zero2) {
    for (ap_int<32> index_tuple3 = 0; index_tuple3 < 16; ++index_tuple3) {
      for (ap_int<32> i4 = 0; i4 < 16; ++i4) {
        pad_temp3[0][not_zero2][index_tuple3][i4] = (((((1 <= index_tuple3) && (index_tuple3 < 15)) && (1 <= i4)) && (i4 < 15)) ? ((float)relu3[(((((i4 - ((i4 + -1) % 14)) + (index_tuple3 * 14)) + (not_zero2 * 196)) + -15) / 1568)][((((((i4 - ((i4 + -1) % 14)) + (index_tuple3 * 14)) + (not_zero2 * 196)) + -15) / 196) % 8)][((((((i4 - ((i4 + -1) % 14)) + (index_tuple3 * 14)) + (not_zero2 * 196)) + -15) / 14) % 14)][((i4 + -1) % 14)]) : 0.000000e+00f);
      }
    }
  }
  float conv4[1][8][14][14];
  for (ap_int<32> ff3 = 0; ff3 < 8; ++ff3) {
    for (ap_int<32> yy3 = 0; yy3 < 14; ++yy3) {
      for (ap_int<32> xx3 = 0; xx3 < 14; ++xx3) {
        ap_fixed<32, 2> reducer40;
        reducer40 = ((ap_fixed<32, 2>)0);
        for (ap_int<32> rc2 = 0; rc2 < 8; ++rc2) {
          for (ap_int<32> ry3 = 0; ry3 < 3; ++ry3) {
            for (ap_int<32> rx3 = 0; rx3 < 3; ++rx3) {
              reducer40 = ((ap_fixed<32, 2>)(((ap_fixed<65, 5>)(((ap_fixed<64, 34>)((ap_fixed<32, 2>)pad_temp3[0][rc2][(yy3 + ry3)][(xx3 + rx3)])) * ((ap_fixed<64, 34>)((ap_fixed<32, 2>)w_conv4[ff3][rc2][ry3][rx3])))) + ((ap_fixed<65, 5>)reducer40)));
            }
          }
        }
        conv4[0][ff3][yy3][xx3] = ((float)reducer40);
      }
    }
  }
  ap_fixed<32, 2> relu4[1][8][14][14];
  for (ap_int<32> args5 = 0; args5 < 1; ++args5) {
    for (ap_int<32> args03 = 0; args03 < 8; ++args03) {
      for (ap_int<32> args13 = 0; args13 < 14; ++args13) {
        for (ap_int<32> args23 = 0; args23 < 14; ++args23) {
          relu4[args5][args03][args13][args23] = ((ap_fixed<32, 2>)((conv4[args5][args03][args13][args23] < 0.000000e+00f) ? 0.000000e+00f : conv4[args5][args03][args13][args23]));
        }
      }
    }
  }
  float pool2[1][8][7][7];
  for (ap_int<32> i5 = 0; i5 < 1; ++i5) {
    for (ap_int<32> c1 = 0; c1 < 8; ++c1) {
      for (ap_int<32> h1 = 0; h1 < 7; ++h1) {
        for (ap_int<32> w1 = 0; w1 < 7; ++w1) {
          float reducer41;
          reducer41 = -1.000000e+00f;
          for (ap_int<32> ra29 = 0; ra29 < 2; ++ra29) {
            for (ap_int<32> ra30 = 0; ra30 < 2; ++ra30) {
              reducer41 = std::max(((float)relu4[i5][c1][((h1 * 2) + ra29)][((w1 * 2) + ra30)]), reducer41);
            }
          }
          pool2[i5][c1][h1][w1] = reducer41;
        }
      }
    }
  }
  float pad_temp4[1][8][9][9];
  for (ap_int<32> not_zero3 = 0; not_zero3 < 8; ++not_zero3) {
    for (ap_int<32> index_tuple4 = 0; index_tuple4 < 9; ++index_tuple4) {
      for (ap_int<32> i6 = 0; i6 < 9; ++i6) {
```

```
        pad_temp4[0][not_zero3][index_tuple4][i6] = (((((1 <= index_tuple4) && (index_tuple4 < 8)) && (1 <= i6)) && (i6 < 8)) ? pool2[(((((i6 - ((i6 + -1) % 7)) + (index_tuple4 * 7)) + (not_zero3 * 49)) + -8) / 392)][((((((i6 - ((i6 + -1) % 7)) + (index_tuple4 * 7)) + (not_zero3 * 49)) + -8) / 49) % 8)][((((((i6 - ((i6 + -1) % 7)) + (index_tuple4 * 7)) + (not_zero3 * 49)) + -8) / 7) % 7)][((i6 + -1) % 7)] : 0.000000e+00f);
      }
    }
  }
  float conv5[1][16][7][7];
  for (ap_int<32> ff4 = 0; ff4 < 16; ++ff4) {
    for (ap_int<32> yy4 = 0; yy4 < 7; ++yy4) {
      for (ap_int<32> xx4 = 0; xx4 < 7; ++xx4) {
        float reducer42;
        reducer42 = 0.000000e+00f;
        for (ap_int<32> rc3 = 0; rc3 < 8; ++rc3) {
          for (ap_int<32> ry4 = 0; ry4 < 3; ++ry4) {
            for (ap_int<32> rx4 = 0; rx4 < 3; ++rx4) {
              reducer42 = ((pad_temp4[0][rc3][(yy4 + ry4)][(xx4 + rx4)] * w_conv5[ff4][rc3][ry4][rx4]) + reducer42);
            }
          }
        }
        conv5[0][ff4][yy4][xx4] = reducer42;
      }
    }
  }
  ap_fixed<32, 2> relu5[1][16][7][7];
  for (ap_int<32> args6 = 0; args6 < 1; ++args6) {
    for (ap_int<32> args04 = 0; args04 < 16; ++args04) {
      for (ap_int<32> args14 = 0; args14 < 7; ++args14) {
        for (ap_int<32> args24 = 0; args24 < 7; ++args24) {
          relu5[args6][args04][args14][args24] = ((ap_fixed<32, 2>)((conv5[args6][args04][args14][args24] < 0.000000e+00f) ? 0.000000e+00f : conv5[args6][args04][args14][args24]));
        }
      }
    }
  }
  float pad_temp5[1][16][9][9];
  for (ap_int<32> not_zero4 = 0; not_zero4 < 16; ++not_zero4) {
    for (ap_int<32> index_tuple5 = 0; index_tuple5 < 9; ++index_tuple5) {
      for (ap_int<32> i7 = 0; i7 < 9; ++i7) {
        pad_temp5[0][not_zero4][index_tuple5][i7] = (((((1 <= index_tuple5) && (index_tuple5 < 8)) && (1 <= i7)) && (i7 < 8)) ? ((float)relu5[(((((i7 - ((i7 + -1) % 7)) + (index_tuple5 * 7)) + (not_zero4 * 49)) + -8) / 784)][((((((i7 - ((i7 + -1) % 7)) + (index_tuple5 * 7)) + (not_zero4 * 49)) + -8) / 49) % 16)][((((((i7 - ((i7 + -1) % 7)) + (index_tuple5 * 7)) + (not_zero4 * 49)) + -8) / 7) % 7)][((i7 + -1) % 7)]) : 0.000000e+00f);
      }
    }
  }
  float conv6[1][16][7][7];
  for (ap_int<32> ff5 = 0; ff5 < 16; ++ff5) {
    for (ap_int<32> yy5 = 0; yy5 < 7; ++yy5) {
      for (ap_int<32> xx5 = 0; xx5 < 7; ++xx5) {
        ap_fixed<32, 2> reducer43;
        reducer43 = ((ap_fixed<32, 2>)0);
        for (ap_int<32> rc4 = 0; rc4 < 16; ++rc4) {
          for (ap_int<32> ry5 = 0; ry5 < 3; ++ry5) {
            for (ap_int<32> rx5 = 0; rx5 < 3; ++rx5) {
              reducer43 = ((ap_fixed<32, 2>)(((ap_fixed<65, 5>)(((ap_fixed<64, 34>)((ap_fixed<32, 2>)pad_temp5[0][rc4][(yy5 + ry5)][(xx5 + rx5)])) * ((ap_fixed<64, 34>)((ap_fixed<32, 2>)w_conv6[ff5][rc4][ry5][rx5])))) + ((ap_fixed<65, 5>)reducer43)));
            }
          }
        }
        conv6[0][ff5][yy5][xx5] = ((float)reducer43);
      }
    }
  }
  ap_fixed<32, 2> relu6[1][16][7][7];
  for (ap_int<32> args7 = 0; args7 < 1; ++args7) {
    for (ap_int<32> args05 = 0; args05 < 16; ++args05) {
      for (ap_int<32> args15 = 0; args15 < 7; ++args15) {
```

```
        for (ap_int<32> args25 = 0; args25 < 7; ++args25) {
          relu6[args7][args05][args15][args25] = ((ap_fixed<32, 2>)((conv6[args7][args05][args15][args25] < 0.000000e+00f) ? 0.000000e+00f : conv6[args7][args05][args15][args25]));
        }
      }
    }
  }
  float max_pool[1][16][1][1];
  for (ap_int<32> i8 = 0; i8 < 1; ++i8) {
    for (ap_int<32> c2 = 0; c2 < 16; ++c2) {
      float reducer44;
      reducer44 = -1.000000e+00f;
      for (ap_int<32> ra31 = 0; ra31 < 7; ++ra31) {
        for (ap_int<32> ra32 = 0; ra32 < 7; ++ra32) {
          reducer44 = std::max(((float)relu6[i8][c2][ra31][ra32]), reducer44);
        }
      }
      max_pool[i8][c2][0][0] = reducer44;
    }
  }
  ap_fixed<32, 2> pool3[1][16];
  for (ap_int<32> i9 = 0; i9 < 1; ++i9) {
    for (ap_int<32> c3 = 0; c3 < 16; ++c3) {
      pool3[i9][c3] = ((ap_fixed<32, 2>)max_pool[i9][c3][0][0]);
    }
  }
  float dense1[1][10];
  for (ap_int<32> i10 = 0; i10 < 1; ++i10) {
    for (ap_int<32> j = 0; j < 10; ++j) {
      float reducer45;
      reducer45 = 0.000000e+00f;
      for (ap_int<32> ra33 = 0; ra33 < 16; ++ra33) {
        reducer45 = ((((float)pool3[i10][ra33]) * w_dense1[ra33][j]) + reducer45);
      }
      dense1[i10][j] = reducer45;
    }
  }
  float compute6;
  float reducer46;
  reducer46 = -1.000000e+00f;
  for (ap_int<32> ra34 = 0; ra34 < 10; ++ra34) {
    reducer46 = std::max(dense1[0][ra34], reducer46);
  }
  compute6 = reducer46;
  float compute7;
  float reducer47;
  reducer47 = 0.000000e+00f;
  for (ap_int<32> ra35 = 0; ra35 < 10; ++ra35) {
    reducer47 = ((float)(exp(((double)(dense1[0][ra35] - compute6))) + ((double)reducer47)));
  }
  compute7 = reducer47;
  float update3;
  for (ap_int<32> j1 = 0; j1 < 10; ++j1) {
    output[0][j1] = ((float)(exp(((double)(dense1[0][j1] - compute6))) / ((double)compute7)));
  }
}
```

E TensorFlow Lite C++ inference

```
#include <stdio.h>
#include <time.h>

#include "tensorflow/lite/interpreter.h"
#include "tensorflow/lite/kernels/register.h"
#include "tensorflow/lite/model.h"
#include "tensorflow/lite/optional_debug_tools.h"

using namespace tflite;

#define TFLITE_MINIMAL_CHECK(x)                               \
    if (!(x)) {                                               \
        fprintf(stderr, "Error at %s:%d\n", __FILE__, __LINE__); \
        exit(1);                                              \
    }

timespec diff(timespec start, timespec end) {
    timespec temp;
    if ((end.tv_nsec - start.tv_nsec) < 0) {
        temp.tv_sec = end.tv_sec - start.tv_sec - 1;
        temp.tv_nsec = 1000000000 + end.tv_nsec - start.tv_nsec;
    } else {
        temp.tv_sec = end.tv_sec - start.tv_sec;
        temp.tv_nsec = end.tv_nsec - start.tv_nsec;
    }

    return temp;
}

int main(int argc, char* argv[]) {
    timespec time1, time2;

    if (argc != 2) {
        fprintf(stderr, "minimal <tflite model>\n");
        return 1;
    }
    const char* filename = argv[1];

    // Load model
    std::unique_ptr<tflite::FlatBufferModel> model =
        tflite::FlatBufferModel::BuildFromFile(filename);
    TFLITE_MINIMAL_CHECK(model != nullptr);

    // Build the interpreter
    tflite::ops::builtin::BuiltinOpResolver resolver;
    InterpreterBuilder builder(*model, resolver);
    std::unique_ptr<Interpreter> interpreter;
    builder(&interpreter);
    TFLITE_MINIMAL_CHECK(interpreter != nullptr);

    // Allocate tensor buffers.
    TFLITE_MINIMAL_CHECK(interpreter->AllocateTensors() == kTfLiteOk);

    // Input tensor is 21
    // Output tensor is 0

    // Fill input buffers with black pixels
    float* input = interpreter->typed_input_tensor<float>(21);
    for (int i = 0; i < 784; ++i) {
        input[i] = 0.0f;
    }

    // Run inference and measure time
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &time1);
    TFLITE_MINIMAL_CHECK(interpreter->Invoke() == kTfLiteOk);
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &time2);

    // Read output buffers
    // Expected output is 0.1 for each output neuron
    float *output = interpreter->typed_output_tensor<float>(0);
```

```
    printf("Output: [");
    for (int i = 0; i < 10; ++i) {
        printf(" %.5f", output[i]);
    }
    printf("]\n");
    printf("Inference time: %ld s %ld ns\n", diff(time1, time2).tv_sec, diff(time1, time2).tv_nsec);

    return 0;
}
```

F vivado_hls/mnist_fp16-opt/src/mnist_fp16.cpp

```
/**
 * Testing accuracy: 0.9728
 */

#include <ap_int.h>
#include <ap_fixed.h>
#include <float.h>
#include <math.h>

#include <ap_axi_sdata.h>
#include <hls_video.h>

#include "w_conv1.h"
#include "w_conv2.h"
#include "w_conv3.h"
#include "w_conv4.h"
#include "w_conv5.h"
#include "w_conv6.h"
#include "w_dense_1.h"


typedef ap_axiu<32,1,1,1> pixel_t;
typedef hls::stream<pixel_t> stream_t;

typedef union {
    unsigned int    u;
    float           f;
} union_t;


void mnist_fp16_opt(stream_t& stream_in, ap_uint<4>& result) {
#pragma HLS INTERFACE ap_ctrl_hs port=return
#pragma HLS INTERFACE axis port=stream_in
#pragma HLS INTERFACE ap_vld port=result

    float preds[1][10];
    float image[1][1][28][28];
    pixel_t pixel;
    union_t uni;
    HLS_SIZE_T i, j;

    bool sof = 0;
    loop_wait_sof: while (sof == 0) {
#pragma HLS LOOP_TRIPCOUNT avg=0 max=0
#pragma HLS PIPELINE II=1
        stream_in >> pixel;
        sof = pixel.user.to_int();
    }

    loop_height: for (i = 0; i < 28; ++i) {
        bool eol = 0;

        loop_width: for (j = 0; j < 28; ++j) {
#pragma HLS LOOP_FLATTEN off
#pragma HLS PIPELINE II=1
            if (sof || eol) {
                sof = 0;
                eol = pixel.last.to_int();
            }
            else {
                stream_in >> pixel;
                eol = pixel.last.to_int();
            }

            uni.u = pixel.data.to_uint();
            image[0][0][i][j] = uni.f;
        }

        loop_wait_eol: while (eol == 0) {
#pragma HLS PIPELINE II=1
```

```
#pragma HLS LOOP_TRIPCOUNT avg=0 max=0
            stream_in >> pixel;
            eol = pixel.last.to_int();
        }
    }


    float pad_temp[1][1][30][30];
    pad1_h: for (ap_int<32> index_tuple = 0; index_tuple < 30; ++index_tuple) {
#pragma HLS PIPELINE
        pad1_w: for (ap_int<32> i = 0; i < 30; ++i) {
#pragma HLS PIPELINE
            pad_temp[0][0][index_tuple][i] = (((((1 <= index_tuple) && (index_tuple < 29)) && (1 <= i)) && (i < 29)) ? image[((((i - ((i + -1) % 28)) + (index_tuple * 28)) + -29) / 784)][0][(((((i - ((i + -1) % 28)) + (index_tuple * 28)) + -29) / 28) % 28)][((i + -1) % 28)] : 0.000000e+00f);
        }
    }
    float conv1[1][4][28][28];
    conv1_c: for (ap_int<32> ff = 0; ff < 4; ++ff) {
        conv1_h: for (ap_int<32> yy = 0; yy < 28; ++yy) {
            conv1_w: for (ap_int<32> xx = 0; xx < 28; ++xx) {
                float reducer84;
                reducer84 = 0.000000e+00f;
                conv1_kh: for (ap_int<32> ry = 0; ry < 3; ++ry) {
                    conv1_kw: for (ap_int<32> rx = 0; rx < 3; ++rx) {
#pragma HLS PIPELINE
                        reducer84 = ((pad_temp[0][0][(yy + ry)][(xx + rx)] * w_conv1[ff][0][ry][rx]) + reducer84);
                    }
                }
                conv1[0][ff][yy][xx] = reducer84;
            }
        }
    }
    ap_fixed<16, 2> relu1[1][4][28][28];
    relu1_n: for (ap_int<32> args = 0; args < 1; ++args) {
        relu1_c: for (ap_int<32> args0 = 0; args0 < 4; ++args0) {
#pragma HLS PIPELINE
            relu1_h: for (ap_int<32> args1 = 0; args1 < 28; ++args1) {
#pragma HLS PIPELINE
                relu1_w: for (ap_int<32> args2 = 0; args2 < 28; ++args2) {
#pragma HLS PIPELINE
                    relu1[args][args0][args1][args2] = ((ap_fixed<16, 2>)((conv1[args][args0][args1][args2] < 0.000000e+00f) ? 0.000000e+00f : conv1[args][args0][args1][args2]));
                }
            }
        }
    }


    float pad_temp1[1][4][30][30];
    pad2_c: for (ap_int<32> not_zero = 0; not_zero < 4; ++not_zero) {
// #pragma HLS PIPELINE
        for (ap_int<32> index_tuple1 = 0; index_tuple1 < 30; ++index_tuple1) {
            for (ap_int<32> i1 = 0; i1 < 30; ++i1) {
#pragma HLS PIPELINE
                pad_temp1[0][not_zero][index_tuple1][i1] = (((((1 <= index_tuple1) && (index_tuple1 < 29)) && (1 <= i1)) && (i1 < 29)) ? ((float)relu1[((((((i1 - ((i1 + -1) % 28)) + (index_tuple1 * 28)) + (not_zero * 784)) + -29) / 3136)][((((((i1 - ((i1 + -1) % 28)) + (index_tuple1 * 28)) + (not_zero * 784)) + -29) / 784) % 4)][((((((i1 - ((i1 + -1) % 28)) + (index_tuple1 * 28)) + (not_zero * 784)) + -29) / 28) % 28)][((i1 + -1) % 28)]) : 0.000000e+00f);
            }
        }
    }
    float conv2[1][4][28][28];
    conv2_c: for (ap_int<32> ff1 = 0; ff1 < 4; ++ff1) {
        for (ap_int<32> yy1 = 0; yy1 < 28; ++yy1) {
            for (ap_int<32> xx1 = 0; xx1 < 28; ++xx1) {
                ap_fixed<16, 2> reducer85;
                reducer85 = ((ap_fixed<16, 2>)0);
```

```
                for (ap_int<32> rc = 0; rc < 4; ++rc) {
                    for (ap_int<32> ry1 = 0; ry1 < 3; ++ry1) {
                        for (ap_int<32> rx1 = 0; rx1 < 3; ++rx1) {
#pragma HLS PIPELINE
                            reducer85 = ((ap_fixed<16, 2>)(((ap_fixed<33, 5>)(((ap_fixed<32, 18>)((ap_fixed<16, 2>)pad_temp1[0][rc][(yy1 + ry1)][(xx1 + rx1)])) * ((ap_fixed<32, 18>)((ap_fixed<16, 2>)w_conv2[ff1][rc][ry1][rx1])))) + ((ap_fixed<33, 5>)reducer85)));
                        }
                    }
                }
                conv2[0][ff1][yy1][xx1] = ((float)reducer85);
            }
        }
    }
    ap_fixed<16, 2> relu2[1][4][28][28];
    relu2_n: for (ap_int<32> args3 = 0; args3 < 1; ++args3) {
        relu2_c: for (ap_int<32> args01 = 0; args01 < 4; ++args01) {
            for (ap_int<32> args11 = 0; args11 < 28; ++args11) {
#pragma HLS PIPELINE
                for (ap_int<32> args21 = 0; args21 < 28; ++args21) {
#pragma HLS PIPELINE
                    relu2[args3][args01][args11][args21] = ((ap_fixed<16, 2>)((conv2[args3][args01][args11][args21] < 0.000000e+00f) ? 0.000000e+00f : conv2[args3][args01][args11][args21]));
                }
            }
        }
    }
    float pool1[1][4][14][14];
    pool1_n: for (ap_int<32> i2 = 0; i2 < 1; ++i2) {
        pool1_c: for (ap_int<32> c = 0; c < 4; ++c) {
            for (ap_int<32> h = 0; h < 14; ++h) {
                for (ap_int<32> w = 0; w < 14; ++w) {
                    float reducer86;
                    reducer86 = -1.000000e+00f;
                    for (ap_int<32> ra63 = 0; ra63 < 2; ++ra63) {
#pragma HLS PIPELINE
                        for (ap_int<32> ra64 = 0; ra64 < 2; ++ra64) {
#pragma HLS PIPELINE
                            reducer86 = std::max(((float)relu2[i2][c][((h * 2) + ra63)][((w * 2) + ra64)]), reducer86);
                        }
                    }
                    pool1[i2][c][h][w] = reducer86;
                }
            }
        }
    }


    float pad_temp2[1][4][16][16];
    pad3_c: for (ap_int<32> not_zero1 = 0; not_zero1 < 4; ++not_zero1) {
        for (ap_int<32> index_tuple2 = 0; index_tuple2 < 16; ++index_tuple2) {
#pragma HLS PIPELINE
            for (ap_int<32> i3 = 0; i3 < 16; ++i3) {
#pragma HLS PIPELINE
                pad_temp2[0][not_zero1][index_tuple2][i3] = (((((1 <= index_tuple2) && (index_tuple2 < 15)) && (1 <= i3)) && (i3 < 15)) ? pool1[((((((i3 - ((i3 + -1) % 14)) + (index_tuple2 * 14)) + (not_zero1 * 196)) + -15) / 784)][((((((i3 - ((i3 + -1) % 14)) + (index_tuple2 * 14)) + (not_zero1 * 196)) + -15) / 196) % 4)][((((((i3 - ((i3 + -1) % 14)) + (index_tuple2 * 14)) + (not_zero1 * 196)) + -15) / 14) % 14)][((i3 + -1) % 14)] : 0.000000e+00f);
            }
        }
    }
    float conv3[1][8][14][14];
    conv3_c: for (ap_int<32> ff2 = 0; ff2 < 8; ++ff2) {
        for (ap_int<32> yy2 = 0; yy2 < 14; ++yy2) {
            for (ap_int<32> xx2 = 0; xx2 < 14; ++xx2) {
                float reducer87;
                reducer87 = 0.000000e+00f;
                for (ap_int<32> rc1 = 0; rc1 < 4; ++rc1) {
```

```
                    for (ap_int<32> ry2 = 0; ry2 < 3; ++ry2) {
                        for (ap_int<32> rx2 = 0; rx2 < 3; ++rx2) {
#pragma HLS PIPELINE
                            reducer87 = ((pad_temp2[0][rc1][(yy2 + ry2)][(xx2 + rx2)] * w_conv3[ff2][rc1][ry2][rx2]) + reducer87);
                        }
                    }
                }
                conv3[0][ff2][yy2][xx2] = reducer87;
            }
        }
    }
    ap_fixed<16, 2> relu3[1][8][14][14];
    relu3_n: for (ap_int<32> args4 = 0; args4 < 1; ++args4) {
        relu3_c: for (ap_int<32> args02 = 0; args02 < 8; ++args02) {
            for (ap_int<32> args12 = 0; args12 < 14; ++args12) {
#pragma HLS PIPELINE
                for (ap_int<32> args22 = 0; args22 < 14; ++args22) {
#pragma HLS PIPELINE
                    relu3[args4][args02][args12][args22] = ((ap_fixed<16, 2>)((conv3[args4][args02][args12][args22] < 0.000000e+00f) ? 0.000000e+00f : conv3[args4][args02][args12][args22]));
                }
            }
        }
    }


    float pad_temp3[1][8][16][16];
    pad4_c: for (ap_int<32> not_zero2 = 0; not_zero2 < 8; ++not_zero2) {
        for (ap_int<32> index_tuple3 = 0; index_tuple3 < 16; ++index_tuple3) {
            for (ap_int<32> i4 = 0; i4 < 16; ++i4) {
#pragma HLS PIPELINE
                pad_temp3[0][not_zero2][index_tuple3][i4] = (((((1 <= index_tuple3) && (index_tuple3 < 15)) && (1 <= i4)) && (i4 < 15)) ? ((float)relu3[((((((i4 - ((i4 + -1) % 14)) + (index_tuple3 * 14)) + (not_zero2 * 196)) + -15) / 1568)][((((((i4 - ((i4 + -1) % 14)) + (index_tuple3 * 14)) + (not_zero2 * 196)) + -15) / 196) % 8)][((((((i4 - ((i4 + -1) % 14)) + (index_tuple3 * 14)) + (not_zero2 * 196)) + -15) / 14) % 14)][((i4 + -1) % 14)]) : 0.000000e+00f);
            }
        }
    }
    float conv4[1][8][14][14];
    conv4_c: for (ap_int<32> ff3 = 0; ff3 < 8; ++ff3) {
        for (ap_int<32> yy3 = 0; yy3 < 14; ++yy3) {
            for (ap_int<32> xx3 = 0; xx3 < 14; ++xx3) {
                ap_fixed<16, 2> reducer88;
                reducer88 = ((ap_fixed<16, 2>)0);
                for (ap_int<32> rc2 = 0; rc2 < 8; ++rc2) {
                    for (ap_int<32> ry3 = 0; ry3 < 3; ++ry3) {
                        for (ap_int<32> rx3 = 0; rx3 < 3; ++rx3) {
#pragma HLS PIPELINE
                            reducer88 = ((ap_fixed<16, 2>)(((ap_fixed<33, 5>)(((ap_fixed<32, 18>)((ap_fixed<16, 2>)pad_temp3[0][rc2][(yy3 + ry3)][(xx3 + rx3)])) * ((ap_fixed<32, 18>)((ap_fixed<16, 2>)w_conv4[ff3][rc2][ry3][rx3])))) + ((ap_fixed<33, 5>)reducer88)));
                        }
                    }
                }
                conv4[0][ff3][yy3][xx3] = ((float)reducer88);
            }
        }
    }
    ap_fixed<16, 2> relu4[1][8][14][14];
    relu4_n: for (ap_int<32> args5 = 0; args5 < 1; ++args5) {
        relu4_c: for (ap_int<32> args03 = 0; args03 < 8; ++args03) {
            for (ap_int<32> args13 = 0; args13 < 14; ++args13) {
#pragma HLS PIPELINE
                for (ap_int<32> args23 = 0; args23 < 14; ++args23) {
#pragma HLS PIPELINE
                    relu4[args5][args03][args13][args23] = ((ap_fixed<16, 2>)((conv4[args5][args03][args13][args23] < 0.000000e+00f) ? 0.000000e+00f : conv4[args5][args03][args13][args23]));
```

```
                }
            }
        }
    }
    float pool2[1][8][7][7];
    pool2_n: for (ap_int<32> i5 = 0; i5 < 1; ++i5) {
        pool2_c: for (ap_int<32> c1 = 0; c1 < 8; ++c1) {
            for (ap_int<32> h1 = 0; h1 < 7; ++h1) {
                for (ap_int<32> w1 = 0; w1 < 7; ++w1) {
                    float reducer89;
                    reducer89 = -1.000000e+00f;
                    for (ap_int<32> ra65 = 0; ra65 < 2; ++ra65) {
                        for (ap_int<32> ra66 = 0; ra66 < 2; ++ra66) {
#pragma HLS PIPELINE
                            reducer89 = std::max(((float)relu4[i5][c1][((h1 * 2) + ra65)][((w1 * 2) +
                            ↪   ra66)]), reducer89);
                        }
                    }
                    pool2[i5][c1][h1][w1] = reducer89;
                }
            }
        }
    }


    float pad_temp4[1][8][9][9];
    pad5_c: for (ap_int<32> not_zero3 = 0; not_zero3 < 8; ++not_zero3) {
        for (ap_int<32> index_tuple4 = 0; index_tuple4 < 9; ++index_tuple4) {
#pragma HLS PIPELINE
            for (ap_int<32> i6 = 0; i6 < 9; ++i6) {
#pragma HLS PIPELINE
                pad_temp4[0][not_zero3][index_tuple4][i6] = (((((1 <= index_tuple4) && (index_tuple4 <
                ↪   8)) && (1 <= i6)) && (i6 < 8)) ? pool2[((((((i6 - ((i6 + -1) % 7)) + (index_tuple4
                ↪   * 7)) + (not_zero3 * 49)) + -8) / 392)][((((((i6 - ((i6 + -1) % 7)) +
                ↪   (index_tuple4 * 7)) + (not_zero3 * 49)) + -8) / 49) % 8)][((((((i6 - ((i6 + -1) %
                ↪   7)) + (index_tuple4 * 7)) + (not_zero3 * 49)) + -8) / 7) % 7)][((i6 + -1) % 7)] :
                ↪   0.000000e+00f);
            }
        }
    }
    float conv5[1][16][7][7];
    conv5_c: for (ap_int<32> ff4 = 0; ff4 < 16; ++ff4) {
        for (ap_int<32> yy4 = 0; yy4 < 7; ++yy4) {
            for (ap_int<32> xx4 = 0; xx4 < 7; ++xx4) {
                float reducer90;
                reducer90 = 0.000000e+00f;
                for (ap_int<32> rc3 = 0; rc3 < 8; ++rc3) {
                    for (ap_int<32> ry4 = 0; ry4 < 3; ++ry4) {
                        for (ap_int<32> rx4 = 0; rx4 < 3; ++rx4) {
#pragma HLS PIPELINE
                            reducer90 = ((pad_temp4[0][rc3][(yy4 + ry4)][(xx4 + rx4)] *
                            ↪   w_conv5[ff4][rc3][ry4][rx4]) + reducer90);
                        }
                    }
                }
                conv5[0][ff4][yy4][xx4] = reducer90;
            }
        }
    }
    ap_fixed<16, 2> relu5[1][16][7][7];
    relu5_n: for (ap_int<32> args6 = 0; args6 < 1; ++args6) {
        relu5_c: for (ap_int<32> args04 = 0; args04 < 16; ++args04) {
            for (ap_int<32> args14 = 0; args14 < 7; ++args14) {
#pragma HLS PIPELINE
                for (ap_int<32> args24 = 0; args24 < 7; ++args24) {
#pragma HLS PIPELINE
                    relu5[args6][args04][args14][args24] = ((ap_fixed<16,
                    ↪   2>)((conv5[args6][args04][args14][args24] < 0.000000e+00f) ? 0.000000e+00f :
                    ↪   conv5[args6][args04][args14][args24]));
                }
            }
        }
    }
```

```
    float pad_temp5[1][16][9][9];
    pad6_c: for (ap_int<32> not_zero4 = 0; not_zero4 < 16; ++not_zero4) {
        for (ap_int<32> index_tuple5 = 0; index_tuple5 < 9; ++index_tuple5) {
            for (ap_int<32> i7 = 0; i7 < 9; ++i7) {
#pragma HLS PIPELINE
                pad_temp5[0][not_zero4][index_tuple5][i7] = (((((1 <= index_tuple5) && (index_tuple5 <
                 ↪  8)) && (1 <= i7)) && (i7 < 8)) ? ((float)relu5[((((((i7 - ((i7 + -1) % 7)) +
                 ↪  (index_tuple5 * 7)) + (not_zero4 * 49)) + -8) / 784)][((((((i7 - ((i7 + -1) % 7))
                 ↪  + (index_tuple5 * 7)) + (not_zero4 * 49)) + -8) / 49) % 16)][((((((i7 - ((i7 + -1)
                 ↪  % 7)) + (index_tuple5 * 7)) + (not_zero4 * 49)) + -8) / 7) % 7)][((i7 + -1) % 7)])
                 ↪  : 0.000000e+00f);
            }
        }
    }
    float conv6[1][16][7][7];
    conv6_c: for (ap_int<32> ff5 = 0; ff5 < 16; ++ff5) {
        for (ap_int<32> yy5 = 0; yy5 < 7; ++yy5) {
            for (ap_int<32> xx5 = 0; xx5 < 7; ++xx5) {
                ap_fixed<16, 2> reducer91;
                reducer91 = ((ap_fixed<16, 2>)0);
                for (ap_int<32> rc4 = 0; rc4 < 16; ++rc4) {
                    for (ap_int<32> ry5 = 0; ry5 < 3; ++ry5) {
                        for (ap_int<32> rx5 = 0; rx5 < 3; ++rx5) {
#pragma HLS PIPELINE
                            reducer91 = ((ap_fixed<16, 2>)(((ap_fixed<33, 5>)(((ap_fixed<32,
                             ↪  18>)((ap_fixed<16, 2>)pad_temp5[0][rc4][(yy5 + ry5)][(xx5 + rx5)])) *
                             ↪  ((ap_fixed<32, 18>)((ap_fixed<16, 2>)w_conv6[ff5][rc4][ry5][rx5])))) +
                             ↪  ((ap_fixed<33, 5>)reducer91)));
                        }
                    }
                }
                conv6[0][ff5][yy5][xx5] = ((float)reducer91);
            }
        }
    }
    ap_fixed<16, 2> relu6[1][16][7][7];
    relu6_n: for (ap_int<32> args7 = 0; args7 < 1; ++args7) {
        relu6_c: for (ap_int<32> args05 = 0; args05 < 16; ++args05) {
            for (ap_int<32> args15 = 0; args15 < 7; ++args15) {
#pragma HLS PIPELINE
                for (ap_int<32> args25 = 0; args25 < 7; ++args25) {
#pragma HLS PIPELINE
                    relu6[args7][args05][args15][args25] = ((ap_fixed<16,
                     ↪  2>)((conv6[args7][args05][args15][args25] < 0.000000e+00f) ? 0.000000e+00f :
                     ↪  conv6[args7][args05][args15][args25]));
                }
            }
        }
    }
    float max_pool[1][16][1][1];
    max_pool_n: for (ap_int<32> i8 = 0; i8 < 1; ++i8) {
        max_pool_c: for (ap_int<32> c2 = 0; c2 < 16; ++c2) {
            float reducer92;
            reducer92 = -1.000000e+00f;
            for (ap_int<32> ra67 = 0; ra67 < 7; ++ra67) {
#pragma HLS PIPELINE
                for (ap_int<32> ra68 = 0; ra68 < 7; ++ra68) {
#pragma HLS PIPELINE
                    reducer92 = std::max(((float)relu6[i8][c2][ra67][ra68]), reducer92);
                }
            }
            max_pool[i8][c2][0][0] = reducer92;
        }
    }
    ap_fixed<16, 2> pool3[1][16];
    max_pool_convert0: for (ap_int<32> i9 = 0; i9 < 1; ++i9) {
        max_pool_convert1: for (ap_int<32> c3 = 0; c3 < 16; ++c3) {
#pragma HLS PIPELINE
            pool3[i9][c3] = ((ap_fixed<16, 2>)max_pool[i9][c3][0][0]);
        }
    }
```

```


    float dense1[1][10];
    dense_n: for (ap_int<32> i10 = 0; i10 < 1; ++i10) {
        dense_c: for (ap_int<32> j = 0; j < 10; ++j) {
            float reducer93;
            reducer93 = 0.000000e+00f;
            for (ap_int<32> ra69 = 0; ra69 < 16; ++ra69) {
                reducer93 = ((((float)pool3[i10][ra69]) * w_dense_1[ra69][j]) + reducer93);
            }
            dense1[i10][j] = reducer93;
        }
    }

    float compute14;
    float reducer94;
    reducer94 = -1.000000e+00f;
    softmax_0: for (ap_int<32> ra70 = 0; ra70 < 10; ++ra70) {
#pragma HLS PIPELINE
        reducer94 = std::max(dense1[0][ra70], reducer94);
    }
    compute14 = reducer94;
    float compute15;
    float reducer95;
    reducer95 = 0.000000e+00f;
    softmax_1: for (ap_int<32> ra71 = 0; ra71 < 10; ++ra71) {
        reducer95 = ((float)(exp(((double)(dense1[0][ra71] - compute14))) + ((double)reducer95)));
    }
    compute15 = reducer95;
    float update7;
    softmax_2: for (ap_int<32> j1 = 0; j1 < 10; ++j1) {
#pragma HLS PIPELINE
        preds[0][j1] = ((float)(exp(((double)(dense1[0][j1] - compute14))) / ((double)compute15)));
    }

    ap_uint<4> index = 0;
    float pred = FLT_MIN;
    results: for (ap_uint<4> i = 0; i < 10; ++i) {
#pragma HLS PIPELINE
        if (preds[0][i] > pred) {
            index = i;
            pred = preds[0][i];
        }
    }

    result = index;
}
```