Reconfigurable & embedded Digital Systems

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

FPGA-based Accelerator for Machine Learning Inference

MSE PA 2019

Student: Lucas Elisei

Supervisor: Alberto Dassatti

June 2019

Abstract

In the recent years, Machine Learning needs for computational power exponentially increased. As a result, dedicated hardware is required to accelerate execution of Machine Learning models. GPUs are currently the most widely used platform to implement Neural Networks. Nevertheless, in terms of power consumption, FPGAs are known to be more energy efficient than GPUs. Thus, numerous FPGA-based Neural Network accelerators have been proposed, targeting both HPC data-centers and embedded applications.

In this study, we are looking for frameworks that allow translating a Neural Network model to synthesizable hardware and benchmark their full development and deployment processes.

LeFlow and TVM are two frameworks identified as interesting candidates. For comparison purpose, we implement two Neural Network models: a simple MLP for classifying handwritten digits and ResNet-50.

TVM development and deployment processes are straightforward and it outputs promising results. On the other hand, LeFlow could not compile both models.

Hardware accelerators are still new so existing frameworks are still experiencing a lot of issues and don't provide an optimal solution. However, we are confident that in the near future, stable solutions will emerge.

Table of contents

1 Introduction

In the 1950s, pioneering Machine Learning (ML) research has been conducted using simple algorithms. At the time, there was not enough computational power and data available to explore further the possibilities that Machine Learning could offer and, for some years, research was quasi-inexistant.

However, in the recent years, with the Big Data era and the computational power growing exponentially, Machine Learning started to rise from the ashes and more and more companies began using it for a plethora of applications (*e.g.* image classification, object detection and speech recognition).

This performance comes at the price of a large computational cost as Convolutional Neural Networks (CNNs) require up to 38 GOP/s to classify a single frame [41]. As a result, dedicated hardware is required to accelerate their execution. Graphics Processing Units (GPUs) are the most widely used platform to implement CNNs as they offer the best performance in terms of pure computational throughput, reaching up to 11 TFLOP/s [4]. Nevertheless, in terms of power consumption, Field-Programmable Gate Arrays (FPGAs) are known to be more energy efficient than GPUs. Thus, numerous FPGA-based CNN accelerators have been proposed, targeting both High-Performance Computing (HPC) data-centers and embedded applications [2].

1.1 Study aim and objectives

In this study, we are looking for frameworks that allow translating a Neural Network (NN) model to synthesizable hardware. We aim at benchmarking the full development and deployment process of any NN and evaluating the output performances, especially the inference.

To achieve this aim, the following objectives have been identified:

1. Review existing frameworks;
2. Pick the most interesting candidates of the existing frameworks;
3. Choose two neural network models with which we will evaluate the previously picked tools;
4. Implement those two models and evaluate their convenience to do so;
5. Measure inference performance;
6. Compare results;
7. Conclude.

1.2 Introduction to Machine Learning

Machine Learning (ML) is a data analytics technique that teaches computers to learn as humans or animals: from experience. Machine Learning algorithms use computational methods to *learn* information directly from data without relying on a predetermined equation as a model [43].

Machine Learning uses two types of techniques: *supervised learning*, which trains a model on known input and output data so that it can predict future outputs; and *unsupervised learning*, which finds hidden patterns by itself in input data without output data.

Supervised learning uses classification and regression techniques to develop predictive models.
Classification techniques predict discrete responses, they classify input data into categories. Typical applications include medical imaging, speech recognition and credit scoring.
Regression techniques predict continuous responses, such as changes in temperature or fluctuations in power demand. Typical applications include electricity load forecasting and algorithmic trading.

Unsupervised learning finds hidden patterns or intrinsic structures in data. It is used to draw inferences from datasets consisting of input data without labelled responses.
Clustering is the most common unsupervised learning technique. It is used for exploratory data analysis to find hidden patterns or groupings in data. Applications for cluster analysis include gene sequence analysis, market research and object recognition.

In this study, we will talk about ML algorithms based on supervised learning only.

What is a Neural Network?

To get started, we first have to define what is a *perceptron*. A perceptron is an artificial neuron that takes one or several inputs and produces a single output, just like human neurons.

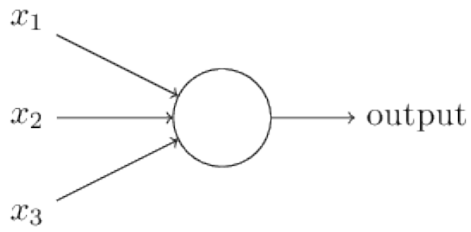


Figure 1: A perceptron with three inputs [27].

In the example shown above, the perceptron has three inputs x_1, x_2 and x_3. In general, it could have more or fewer inputs. Each input has a weight (w_1, w_2, w_3) expressing its importance relative to the others. The neuron's output, 0 or 1, is determined by whether the weighted sum $\sum_i w_i x_i$ is less than or greater than a *threshold* value. To put it in more algebraic terms:

$$\text{output} = \begin{cases} 0 & \text{if } \sum_i w_i x_i \leq \text{threshold} \\ 1 & \text{if } \sum_i w_i x_i > \text{threshold} \end{cases}$$

Obviously, the perceptron is not a complete model of human decision-making but it seems plausible that a complex network of perceptrons could make quite subtle decisions. This is what we call a Neural Network.

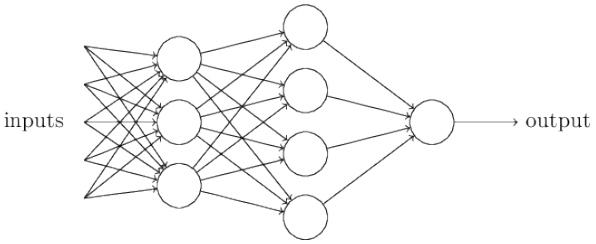


Figure 2: Simple Multi-Layer Perceptron (MLP) network [27].

A Neural Network (NN) is composed of several layers (three in the example shown above). The leftmost layer in this network is called the *input layer*, and the neurons within the layer are called *input neurons*.
The rightmost layer, or *output layer*, contains the *output neurons*, or, as in this case, a single output neuron. The middle layer is called a *hidden layer* since the neurons in this layer are neither inputs nor outputs. The network above has just a single hidden layer but some networks have multiple hidden layers.

Training, validating and testing a Neural Network

The data used to build the final model usually comes from multiple datasets. In particular, three datasets are commonly used in different stages of the creation of the model [39].

The model is initially fit on a *training dataset*. The NN is run with the training dataset and produces a result, which is then compared with the *target* (or *label*), for each input of the training dataset. Based on the result of the comparison, the parameters of the model are adjusted.

Successively, the fitted model is used to predict the responses for the observations in a second dataset, called the *validation dataset*. Validation datasets can be used for stopping the training when the error on the validation dataset increase, as this is a sign of overfitting to the training dataset.

Finally, the *test dataset* is a dataset used to provide an unbiased evaluation of the final model predictions. A test dataset is independent of the training dataset.

Inference

Once the Neural Network has been trained, it is ready for inference – to classify, recognize and process new inputs. For instance, the voice of someone talking to Siri on his iPhone is sent to an Apple server on which a CNN trained for speech recognition has been deployed.

Remarkable works and success stories

- In 2019, Yoshua Bengio, Geoffrey Hinton and Yann LeCun have been awarded the Turing Award for their conceptual and engineering breakthroughs that have made Deep Neural Networks (DNNs) a critical component of computing [7].
- Fei-Fei Li was the leading scientist and principal investigator of ImageNet [17], an image database with more than 14 million images classified over 22,000 categories. This dataset was crucial for CNNs evolution.
- Voice recognition services such as *Siri*, *OK Google* or *Amazon Echo* work well thanks to Machine Learning [12].
- Self-driving cars (*e.g.* Tesla Autopilot [14]).
- Netflix saves up to 1 billion dollars each year thanks to its AI algorithms [15].
- IBM's project Debater is the first AI system that can debate humans on complex topics [31].
- In 2017, Google created AutoML, an AI that is capable of generating its own AIs, outperforming all of its human-made counterparts [9].

2 Existing frameworks

This section reviews existing frameworks that aim at translating a Neural Network model to synthesizable hardware.

The following criteria will be evaluated for each tool:

1. Small description, tool's objectives
2. Maintenance and Update frequency – It is important to know if the tool is under active development and who is responsible for the development.
3. License – License is important since we might want to modify the tool so it better suits our needs.
4. Which input(s) the tool accepts – Does it use already existing ML description framework(s) or its own?
5. Which output(s) the tool targets – The more outputs a tool supports, the more it is an interesting candidate.
6. Remarks, if needed

Once each tool has been reviewed, we will pick the most interesting candidates and focus on evaluating their performances.

A summary of the review is available in the section 2.10.

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 [6].

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 [5], 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 reflects on the commits history.

License

According to its website, DnnWeaver is licensed under the Apache License 2.0 [3], 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 Verilog code specialised for the given network, using hand-optimised Verilog templates (included in DnnWeaver).

As of March 2019, the implemented layers are Convolution, Rectified Linear Unit (ReLU), Fully Connected Layer (InnerProduct), Pooling and Local Response Normalisation (LRN).

Remarks

The available documentation concerns the version 1.0 of the tool, which make 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 utilisation. The only fix is to update manually incompatible files to *Python3.6*.

Additionally, it seems that adding an FPGA unknown 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 [8].

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 [36], 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.

The implemented layers are: forward- and backward-Convolution, forward- and backward-ReLU, forward- and backward-MaxPooling, and forward- and backward-InnerProduct.

Remarks

None.

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 [30]. 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 [28].

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 since April 2019 but nothing 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 [29].

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 [10]. 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 optimise 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 [38], some errors are raised during compilation, and we don't have information on what is making compilation fail.

It also seems that people are having a hard time testing NVDLA on FPGAs[1].

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 [13].

[1] `https://github.com/nvdla/hw/issues/110`

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 is updated quite frequently, about 1 commit is made per week.

License

hls4ml is licensed under the Apache License 2.0 [3].

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 May 2019, only Multi-Layer Perceptron (MLP) and Conv1D/Conv2D architectures are supported.

Remarks

The documentation doesn't give a lot of information and was last updated a year ago.

2.5 TVM

TVM is an open deep learning compiler stack for CPUs, GPUs and specialised accelerators (FPGAs). It aims to close the gap between the productivity-focused deep learning frameworks, and the performance- or efficiency-oriented hardware backends [40].

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 industry 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 [3].

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 minimum deployable modules on diverse hardware backends such as CPUs, GPUs and FPGAs.

Remarks

The documentation is really good 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 question and questions are rapidly answered.

Versatile Tensor Accelerator (VTA) is an extension of the TVM framework that includes drivers, a Just-in-Time (JIT) runtime and an optimising compiler stack. This extension is currently specified in Vivado HLS C++ which is only supported by Xilinx toolchains. However, a pull request adding Intel FPGA and Chisel support is currently being reviewed [32].

2.6 LeFlow

LeFlow is a tool that relies on LegUp [20] to map numerical computation models written in Tensorflow to synthesizable hardware [18]. It bridges Google's XLA compiler and LegUp high-level synthesis to automatically generate Verilog.

Maintenance and Update frequency

LeFlow is maintained by three people who work at The University of British Columbia.

The project was last updated in February 2019, which is not that old but before that, the last update is from November 2018 so it is not updated frequently.

License

LeFlow is licensed under the MIT License [37] 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 customised version of Tensorflow.

Output

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

Remarks

Documentation is inexistant.

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 [25].

Maintenance and Update frequency

nGraph is maintained by an artificial intelligence software company called Nervana Systems (acquired by Intel in August 2016) [24].

The Github repository is updated with several commits every day [26].

License

nGraph is licensed under the Apache License 2.0 [3].

Input

As of May 2019, nGraph takes Tensorflow and MXNet as inputs for model description.

Output

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

FPGA are set to be fully supported but no time information is given.

Remarks

nGraph released a Release Candidate version on the 24th of May 2019 with documentation. Since the release was too close to the end of this study, we could not investigate more on this framework.

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 in the future.

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 customisation 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 [11].

License

HeteroCL is licensed under the Apache License 2.0 [3].

Input

HeteroCL takes PyTorch, MXNet and Keras NN specifications as inputs.

Output

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

Remarks

HeteroCL released its first version, v0.1, just days before the end of this study. Thus, we could not investigate more on this framework. It's unfortunate because it seems to be the most interesting framework since it offers more outputs than any other.

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 [11].

2.9 Comparison

Reminder: A summary of the reviewed frameworks is available at the section 2.10.

Now that the first objective (review existing frameworks) has been completed, we can move on to the second, which is picking the most interesting candidates.

We want to pick at least two candidates so that we can do a comparison of their respective performance.
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 which allows a lot of freedom (except maybe for NVDLA).
Also, the *Supported inputs* criterion has not been taken into account since input frameworks are similar and switching from one to another is quite feasible.

nGraph is not an interesting candidate yet since it doesn't support FPGAs as output yet.
Unfortunately, we could not retain HeteroCL as a candidate because it has been released just days before the end of this study. But it might be a truly interesting framework to follow in the future.
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 interesting candidates.
FPGA Caffe has not been updated for more than a year, so it obviously is not an interesting candidate.

We have three candidates left: hls4ml, TVM and LeFlow. LeFlow is the only framework to support Intel FPGAs as output so we are keeping it as an interesting candidate.
Between hls4ml and TVM, we chose to go with TVM because the update frequency is better by far as well as the documentation.

In conclusion, the two most interesting candidates to be retained are LeFlow and TVM.

2.10 Summary

Framework	Update frequency	Supported inputs	Supported outputs	Comments
DnnWeaver 2.0	One update every year (last in April 2019)	Caffe	Verilog	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	CPU, Xilinx FPGA	Documentation is good. Project is *relatively* open-source. Seems that people are having a hard time deploying on FPGAs.
hls4ml	1 commit per week	Keras, PyTorch	Synthesizable IP Core	Documentation last updated a year ago.
TVM	Several commits per day	Keras, MXNet, PyTorch, Tensorflow, CoreML, DarkNet	CPUs, GPUs, Xilinx FPGAs	Good documentation with tutorials. Have a forum for asking questions. A pull request is being reviewed for integrating Intel FPGAs support.
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	PyTorch, MXNet, Keras	FPGAs, CPUs	The framework was released just day before the end of this study thus we had no time to investigate further.

Table 1: Frameworks summary.

3 Choosing Neural Network models to evaluate

Now that we defined TVM and LeFlow as the tools we will evaluate, we have to choose which Neural Network models are suitable for implementation and evaluation.

We want to choose two NN models: a simple and a more complex one. The reason for this is so that we can evaluate tools performance on two different complexities along with the difficulty to implement such complex models.

Benchmarking tools for measuring performance of Machine Learning software frameworks already exist. They are interesting for measuring the number of operations per second required for doing inference on a fitted Neural Network.

- MLPerf is a benchmark suite for measuring performance of ML software frameworks, hardware accelerators and cloud platforms [22]. It is a community-driven open-source project supported by a lot of companies.
- Deep500 is a Python library that enables fair comparison of deep learning frameworks, algorithms, libraries and techniques [1]. It is developed by a team from the Department of Computer Science of ETH Zurich.

These are really interesting benchmarking tools and we can only encourage you to go look further. However, in this paper, we focus on both implementation and output performance and using these tools would require a lot of useless efforts.

As stated before, we want to choose two Neural Network models with different complexities. Two models are widely used when introducing people to Machine Learning and Artificial Intelligence: a simple Multi-Layer Perceptron (MLP) and ResNet.

3.1 Simple Multi-Layer Perceptron

One of the simplest and common NN models is the single Multi-Layer Perceptron. We presented its basic architecture with the figure 2.

We want to create a classifier that recognizes handwritten digits using the MNIST dataset [23].

MNIST contains 70000 images of handwritten digits: 60000 for training and 10000 for testing. The images are grayscale, 28x28 pixels and centered.

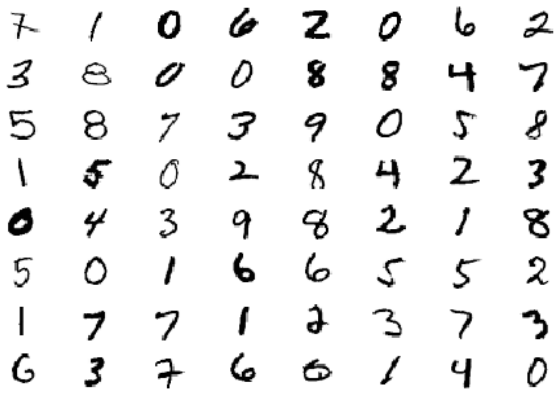


Figure 3: Random selection of MNIST digits [16].

The model is a bit more complex than the MLP presented before. This model has 4 layers:

1. the input layer, with 784 entries (one for each pixel of the image);
2. the first hidden layer, with 128 entries, uses ReLU as activation function;
3. the second hidden layer, with 64 entries, also uses ReLU as activation function;
4. the output layer, with 10 outputs, classifies the input image.

Below is a graphical representation of the model:

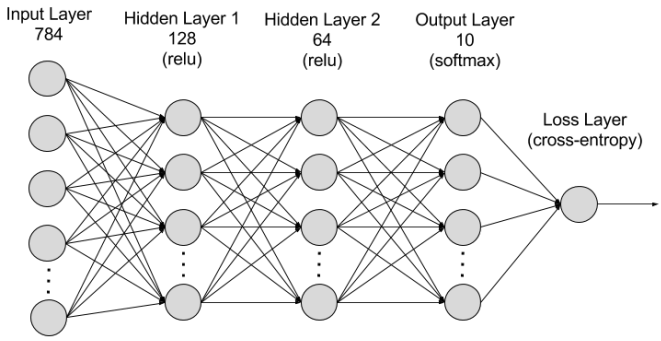


Figure 4: Simple Multi-Layer Perceptron

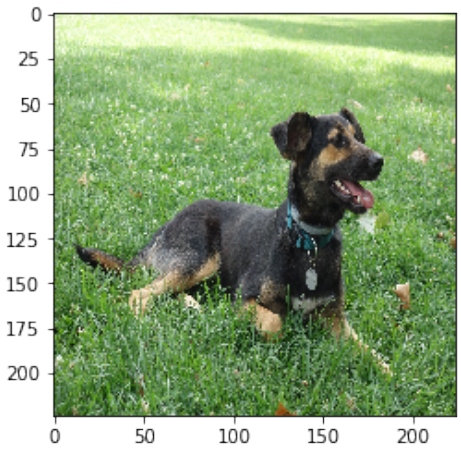
The first layer has $784 * 128$ (weigths) $+ 128$ (biases) $= 100480$ tunable parameters, the second layer has $128 * 64 + 64 = 8256$ tunable parameters and the third one has $64 * 10 + 10 = 650$ tunable parameters. Thus, the model has $100480 + 8256 + 650 = 109386$ total parameters.

3.2 ResNet-50 v1.5

Residual Neural Network (ResNet) is an artificial Neural Network. It is used to achieve human-level image classification.

To train, it uses the ImageNet dataset [17]. This dataset contains millions of images belonging to more than 20000 categories. Each image is 224x244 and centered.

The ResNet-50 v1.5 model is a modified version of the original ResNet-50 v1 model [35]. The difference between v1 and v1.5 makes the most recent version more accurate (~0.5%) but comes with a small performance drawback (~5% imgs/sec).




```
[
    ('n02106662', 'German_shepherd', 0.33396399),
    ('n02093754', 'Border_terrier',  0.29242834),
    ('n02105162', 'malinois',        0.16545239),
    ('n02096051', 'Airedale',        0.04790553),
    ('n02106550', 'Rottweiler',      0.03723163)
]
```

Figure 5: ResNet-50 example output [34].

The model has more than 25 million total tunable parameters [44], making it way more complex than the MLP model previously introduced.

4 Implementation

Now that we chose the models we want to implement and the frameworks we will use to deploy them on an FPGA, we can proceed to the implementation and benchmarking the results.

Since this research focuses on benchmarking the full development and deployment process, we will document the installation procedure, the ease of implementation and the output results.

Note: For this section, all the installation and implementation steps have been executed on a machine running Linux.

4.1 TVM

4.1.1 Installation

This section describes how to build and install the TVM packages from scratch.

It consists of two steps:

1. Build the `libtvm.so` shared library from the C++ codes
2. Install the Python packages

To get started, clone TVM's repo from Github:

```
git clone --recursive https://github.com/dmlc/tvm
```

Build the shared library

First, we need to install required packages:

```
sudo apt-get update
sudo apt-get install -y python python-dev python-setuptools gcc libtinfo-dev zlib1g-dev build-essential
 ↪  cmake
```

First, create a build directory and copy the `cmake/config.cmake` file to the directory:

```
mkdir build
cp cmake/config.cmake build
```

We also need to install LLVM. The TVM official documentation says that LLVM 4.0 or higher is required but we recommend using LLVM 8.0 since there seems to be issues with lower versions. To install LLVM, we can either build it from source (takes long time) or we can download a pre-built version from *LLVM Download Page*[1].

To let CMake know that we use LLVM, we need to modify `build/config.cmake`:

1. If you downloaded pre-built LLVM binaries, add `set(USE_LLVM /path/to/your/llvm/bin/llvm-config)`
2. If you installed LLVM system-wide, you can let CMake search for a usable version of LLVM by adding `set(USE_LLVM ON)`

We can then build TVM and its related libraries:

```
cd build
cmake ..
make -j4
```

[1] `http://releases.llvm.org/download.html`

Python packages installation

For this section, we **heavily** recommend using a *Conda environment* to ensure that no harm is done to the Python system-wide installation.

Install TVM Python packages:

```
cd python; python setup.py install --user; cd ..
cd topi/python; python setup.py install --user; cd ../..
cd nnvm/python; python setup.py install --user; cd ../..
```

Install Python dependencies:

```
pip install --user numpy decorator attrs tornado psutil xgboost
```

Everything should now be setup correctly! As you can see, TVM installation is quite straightforward and can be performed in a few minutes by experienced users, making it a really interesting framework.

4.1.2 Board setup

Now that all the required software has been installed, we need to setup the FPGA board. Since the TVM documentation uses the Pynq-Z1 board for tutorials, the following steps target the Pynq-Z1 [33]. A microSD card with a capacity of at least 8 GB and an Ethernet cable are also required. To be able to talk to the board, make sure to assign your computer a static IP address (in this paper, we assume the computer has the IP address `192.168.1.4` and the Pynq has the IP address `192.168.2.99`).

The first step is to download the latest Pynq image (PYNQ-Z1 v2.4 at the time of writing), which can be found on the *Pynq Development Board* page[1].

To make sure the Pynq board is properly setup, power it on and try to connect to it:

```
ssh xilinx@192.168.2.99
```

Note: By default, the Pynq board uses `xilinx` as username and password.

Now that the Pynq board is setup, we need to build and deploy an RPC server so that TVM can communicate with it.

Because the direct board-to-computer connection prevents the board from directly accessing the internet, we'll need to mount the Pynq's filesystem to your computer filesystem with `sshfs`. Next, we clone the TVM Github repository into the `sshfs` mountpoint on your computer.

```
# On the computer
mkdir /path/to/mountpoint
sshfs xilinx@192.168.2.99:/home/xilinx /path/to/mountpoint
cd /path/to/mountpoint
git clone --recursive https://github.com/dmlc/tvm
cd ~
sudo umount /path/to/mountpoint
```

Now that we have cloned the TVM/VTA repository in the Pynq's filesystem, we can `ssh` into it and launch the build of the TVM-based RPC server.

```
ssh xilinx@192.168.2.99
# Build TVM runtime library
cd /home/xilinx/tvm
mkdir build
cp cmake/config.cmake build/
# Copy Pynq specific configuration
cp vta/config/pynq_sample.json build/vta_config.json
cd build
```

[1] `http://www.pynq.io/board.html`

```
cmake ..
make runtime vta -j2    # (takes ~5 min)
```

We now have to create an RPC tracker on the computer and then bind the Pynq on it:

```
# On the computer
python -m tvm.exec.rpc_tracker --host=0.0.0.0 --port=9190
# On the PYNQ board
python3 -m tvm.exec.rpc_server --tracker 192.168.2.4:9190 --key=pynq
```

You should see the following line being displayed when starting the RPC server:

```
INFO:root:RPCServer: bind to 0.0.0.0:9091
```

In order to let the RPC server run, you will need to leave the RPC server running in an `ssh` session.

4.1.3 Implementation

The file `src/run_tvm.py` contains the code required to run the implementation. In this section, we will explain step by step what this code does.

The first step is to fetch the network workload. The function `get_network()` does that. It takes the network name as input and get the workload according to the batch size, the number of classes, the number of layers, the image shape and the type of data that this image is composed of.

```
33 # Get network workload.
34 def get_network(network: str,
35                 batch_size: int,
36                 num_classes: int,
37                 num_layers: int,
38                 image_shape: tuple,
39                 dtype: str):
40     if (network == 'resnet'):
41         net, params = nnvm.testing.resnet.get_workload(batch_size=batch_size,
42                                                        num_classes=num_classes,
43                                                        num_layers=num_layers,
44                                                        image_shape=image_shape,
45                                                        dtype=dtype)
46     elif (network == 'mlp'):
47         net, params = nnvm.testing.mlp.get_workload(batch_size=batch_size,
48                                                     num_classes=num_classes,
49                                                     image_shape=image_shape,
50                                                     dtype=dtype)
51     else:
52         raise ValueError('Invalid network: {}'.format(network))
53 
54     return net, params
```

At the moment, it implements only `resnet` and `mlp` networks workloads but it can be extended to the other workloads that NNVM supports.

The function `tune_tasks()` is responsible for data layout optimisation.
Data layout optimisation converts a computational graph into one that can use better internal data layouts for execution on the target hardware. The function will not be discussed here since its implementation is not relevant.

One thing to know is that the best tuning configuration is saved into a file. Once the tuning has been done, it is not required to do it every time.

The function `tune_and_evaluate()` is the most important since it is responsible for calling all the other functions and evaluating the inference.

```
91  # Tune and evaluate network.
92  def tune_and_evaluate(tuning_opt, do_tuning=True):
93      # Get network workload.
94      print('Getting network workload...')
95      net, params = get_network(network, batch_size, num_classes, num_layers, image_shape, dtype)
96
97      data_shape = {'data': (batch_size, ) + image_shape}
98
99      # Extract tasks.
100     if (do_tuning == True):
101         print('Extracting tasks...')
102         tasks = autotvm.task.extract_from_graph(net,
103                                                 shape=data_shape,
104                                                 dtype=dtype,
105                                                 target=target,
106                                                 symbols=symbols,
107                                                 target_host=target_host)
108         print('Tuning tasks...')
109         tune_tasks(tasks, **tuning_opt)
110     else:
111         print('Extracting tasks from log file...')
112
113     # Compile kernels with history best records (using log file).
114     with autotvm.apply_history_best(tuning_opt['log_filename']):
115         print('Compiling kernels...')
116         with nnvm.compiler.build_config(opt_level=3):
117             graph, lib, params = nnvm.compiler.build(net,
118                                                      target=target,
119                                                      shape=data_shape,
120                                                      dtype=dtype,
121                                                      params=params,
122                                                      target_host=target_host)
123
124         if (env.TARGET == 'pynq'):
125             print('Uploading to PYNQ board...')
126             tmp = tempdir()
127             filename = 'net_pynq.tar'
128             lib.export_library(tmp.relpath(filename))
129
130             # Upload module to the board.
131             remote_start = time.time()
132             remote = autotvm.measure.request_remote(device_key, host=rpc_host, port=rpc_port,
                ↪  timeout=10000)
133             remote.upload(tmp.relpath(filename))
134             rlib = remote.load_module(filename)
135             remote_time = time.time() - remote_start
136             print('Module uploaded to board in {0:.2f}s'.format(remote_time))
137
138             ctx = remote.context(str(target), 0)
139             module = runtime.create(graph, rlib, ctx)
140         elif (env.TARGET == 'sim'):
141             ctx = tvm.cpu()
142             module = runtime.create(graph, lib, ctx)
143
144         # Create random data for inference.
145         data = tvm.nd.array((np.random.uniform(size=data_shape['data'])).astype(dtype))
146
147         # Upload data and parameters to target.
148         module.set_input('data', data)
149         module.set_input(**params)
150
151         # Evaluate inference.
152         print('Evaluating inference time cost...')
153         ftimer = module.module.time_evaluator('run', ctx, number=1, repeat=100)
```

On line 94, we retrieve the network workload (the `network_opt` variable will be discussed later).

Then, if the `do_tuning` parameter is set to `True`, tuning tasks are extracted from the graph on given symbols and data layout optimisation is performed.

On line 113, the best optimisations are picked and applied during compilation. Once the compilation is done

(line 116), the library containing the module is uploaded to the target device (CPU if `env.TARGET == 'sim'`, Pynq board if `env.TARGET == 'pynq'`).

Then, we create random data that fits the network input shape and send it to the device. The inference is then run multiple times (in the code, 100 times) and a mean inference time and the standard deviation is displayed.

Once the TVM module has been compiled, it is sent to Versatile Tensor Accelerator (VTA), TVM's own hardware accelerator. VTA is an open, generic and customisable deep learning accelerator design. It's an end-to-end solution that includes drivers, a JIT runtime and an optimising compiler stack based on TVM.

This generic deep learning accelerator is built around a GEMM core, which performs dense matrix multiplication at a high computational throughput.

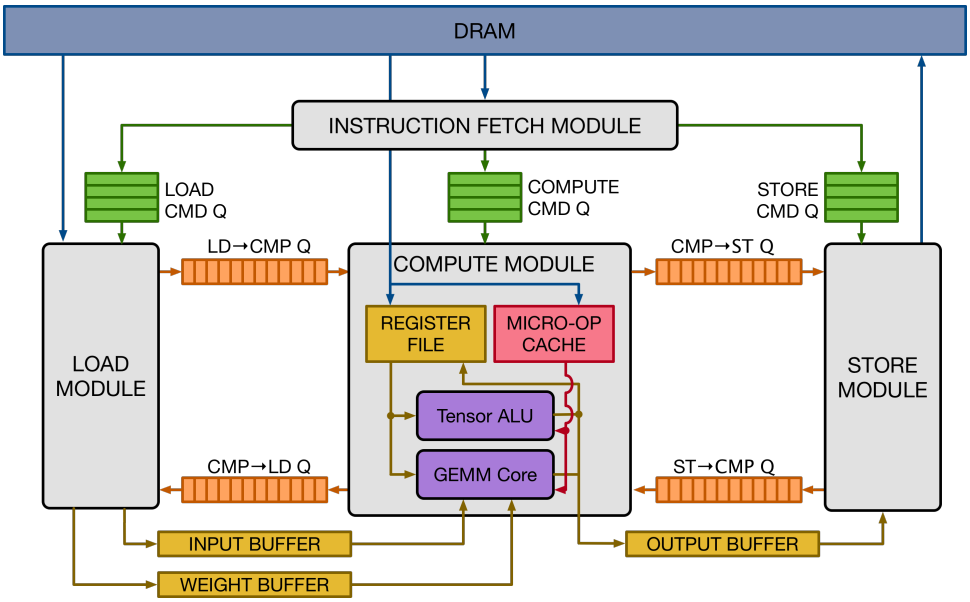


Figure 6: High-level overview of the VTA hardware organisation [42].

The figure 6 presents a high-level overview of the VTA hardware organisation. VTA is composed of four modules that communicate between each other via FIFO queues and single-writer/single-reader SRAM memory blocks, to allow for task-level pipeline parallelism. The compute module performs both dense linear algebra computation with its GEMM core, and general computation with its tensor ALU.

The VTA hardware design template offers modularity to the user, with the option to modify hardware datatypes, memory architecture, the GEMM core dimensions, hardware operators and pipelining stages.

4.1.4 Results

For comparison purposes, the model has been run on an AMD Ryzen 3 1300 Quad-Core CPU and on the Pynq board.

MLP

CPU

```
Mean inference time (std dev): 0.10 ms (0.69 ms)
```

Pynq board

```
Mean inference time (std dev): 0.39 ms (0.02 ms)
```

ResNet-50

CPU

```
Mean inference time (std dev): 107.25 ms (36.61 ms)
```

Pynq board

```
Mean inference time (std dev): 9190.38 ms (107.43 ms)
```

We see that for each model, the CPU achieves better results than the FPGA. It might be due to the fact that the Pynq board processor is running at 650 MHz while the CPU used for testing has a frequency of 3.5 GHz.

To have a meaningful comparison, we should compare the number of operations both technologies perform for the same NN model and not just the time of execution (which depends on a lot of factors). Unfortunately, we didn't find the time to measure the number of operations performed by each device.

4.1.5 Summary

TVM/VTA installation is pretty straightforward. The Python frontend is easy to use and a lot of tutorials are here to help the developer to implement what he wants.

The more the NN model is complex, the more the tensors tuning will take time and time required for loading the module on the FPGA will also increase (about 15 minutes for ResNet-50 model).

Loading the NN model on the

In conclusion, TVM is good framework to use for running inference on FPGAs and its quality will increase with future releases without any doubt.

4.2 LeFlow

4.2.1 Installation

Because LeFlow relies on the LegUp tool, the first thing is to install it. LegUp provides an official virtual machine with everything already setup. The virtual machine is available on the *LegUp Download* page[1](22.8 GB). We recommend using VirtualBox to run the virtual machine since it is free.

Once the virtual machine has been downloaded and is running, we need to install LeFlow. To do so, simply clone the repository:

```
git clone https://github.com/danielholanda/LeFlow
```

Because LeFlow makes some minor changes on Tensorflow, we need to install a modified version of Tensorflow:

```
cd LeFlow
sudo apt-get install python-pip
sudo python -m pip install --upgrade pip
cd src/tensorflow
sudo pip install tensorflow-1.6.0-cp27-cp27mu-linux_x86_64.whl --ignore-installed six
```

If you didn't download the LegUp virtual machine, you must check that the configuration variables `python_path` and `legup_examples_folder` in the `src/LeFlow` file point to the right paths.

It is also important to make the `src/LeFlow` file an executable:

```
chmod +x src/LeFlow
```

If you are interested in testing the installation, you can do so by executing the following commands:

[1] `http://legup.eecg.utoronto.ca/download.php`

```
cd test
python test_all.py --fast
```

All the tests should take less than a minute.

4.2.2 Implementation

LeFlow only needs the value of all weights and biases of the model to map them to on-chip memories.

There are two ways for retrieving weights and biases of a neural network:

1. Locally train the network
2. Download a pre-trained network

Since ResNet-50 is quite a big network, the first alternative would take a lot of time (potentially several days on a common computer).

Unfortunately, the second alternative didn't work either. The only pre-trained official networks are not compatible with the version of Tensorflow that LeFlow uses.

Given this issue, we could not implement the ResNet-50 model using LeFlow. However, a simple MLP network can be trained and implemented on an Altera DE1-SoC board.

The code used to train and implement the MLP network is in the file `src/run_leflow.py`

To generate hardware for the MLP network, we need to execute the following commands:

```
cd src/

# Generate hardware
/path/to/LeFlow/src/LeFlow run_leflow.py

# Test on Modelsim
make v -C run_leflow_files/
```

Using the Quartus software (already installed on the virtual machine), we can upload the design to the board.

For generating hardware, LeFlow relies on LegUp. LeFlow passes the C++ code (got from the Tensorflow Python description of the network) to LegUp and the latter performs the generation.

Most of the LegUp code is implemented as a target backend pass in the LLVM compiler framework. The LegUp code is logically structured according to the flow chart in the figure 7.

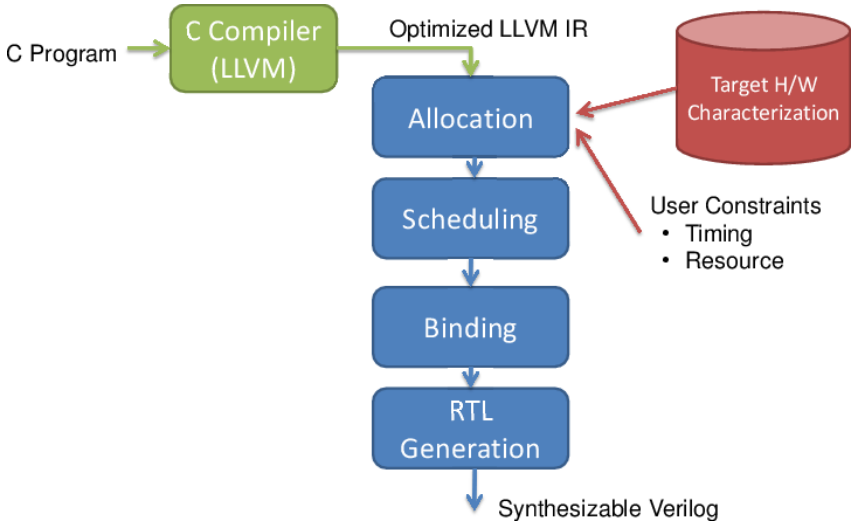


Figure 7: LegUp flow [21].

There are five major logical steps performed in order: Allocation, Scheduling, Binding, RTL generation and producing Verilog output.

During the Allocation step, LegUp reads in a user Tcl configuration script that specifies the target device, timing constraints and HLS options. Another Tcp script that contains the FPGA device specific operation delay and area characteristics is read. This step also handles mapping LLVM instructions to unique signal names in Verilog and ensuring these names do not overlap with reserved Verilog keywords.

The next step (Scheduling) loops over each function in the program and performs HLS scheduling.

The Binding step performs bipartite weighted matching. Binding results are stored in a data structure that maps each LLVM instruction to the name of the hardware functional unit that the instruction should be implemented on.

In the RTL Generation step, each LLVM instruction is looped over and LegUp creates an RTL data structure, which represents the final circuit, based on the Scheduling and Binding information.

Finally, LegUp loops over each RTL data structure previously created and prints out the corresponding Verilog for the hardware module.

4.2.3 Results

When trying to run the file `run_leflow.py` with LeFlow, the framework outputs an error:

```
legup@legup-vm:~/LeFlow/examples/PA$ ../../src/LeFlow run_leflow.py
INFO: Creating project...
INFO: Cleaning previous files...
INFO: Generating IR from tensorflow...
INFO: Cleaning unused dumped files...
Traceback (most recent call last):
  File "../../src/LeFlow", line 200, in <module>
    run_leflow(args.file_name,tool_path)
  File "../../src/LeFlow", line 79, in run_leflow
    shutil.copy(project_folder+"ir/ir-cluster_0__XlaCompiledKernel_true__
      XlaNumConstantArgs_0__XlaNumResourceArgs_0__module-with-opt.ll",
      project_folder+project_name+"_ir_1.ll")
  File "/usr/lib/python2.7/shutil.py", line 119, in copy
    copyfile(src, dst)
  File "/usr/lib/python2.7/shutil.py", line 82, in copyfile
    with open(src, 'rb') as fsrc:
IOError: [Errno 2] No such file or directory:
  'run_leflow_files/ir/ir-cluster_0__XlaCompiledKernel_true__
  XlaNumConstantArgs_0__XlaNumResourceArgs_0__module-with-opt.ll'
```

This error is due to the fact that XLA, the Tensorflow compiler, generates vectorized instructions which are not supported by LegUp [19].

Because of this issue, we were unable to compile the simple MLP and make it run on an FPGA.

4.2.4 Summary

LeFlow is a good framework to use if you know what you are doing. The two main issues are the lack of documentation, which is inexistant; and the fact that it relies on the LegUp tool, which comes with its own limits and issues.

5 Conclusion

In this study, we looked at frameworks that allow translating a Neural Network (NN) model to synthesizable hardware: we aimed at benchmarking the full development and deployment processes and evaluating their output performances.

We started by listing and reviewing existing frameworks. We noticed that a lot of projects have been abandoned or were not developed enough to suit our needs. From these frameworks, we picked the most two interesting candidates, TVM and LeFlow, to benchmark their respective development and deployment processes.

We stated that a simple Multi-Layer Perceptron (MLP), used for handwritten digits recognition, and a more complex Deep Neural Network (DNN), ResNet-50, used for image classification, were suitable models to implement and evaluate.

The installation of the two frameworks and the implementation of the two models was explained in detail. Both frameworks installation is pretty straightforward. Implementation is very easy for both frameworks. However, TVM deployment process requires some lines of code. LeFlow deployment could not be tested since they were issues due to the tool relying on LegUp.

LeFlow has not been updated recently and its possibilities seem quite limited: LegUp does not support vectorized instructions, thus some tensors operations are not available for hardware synthesization.

TVM implements a lot of tensors operations, thus many actual NN models can be implemented with this framework. It is frequently updated so we are confident that future versions will enhance this tool. For the record, the TVM team is currently developing a new backend, Relay, which will replace the currently existing one, NNVM.

NVIDIA Deep Learning Accelerator (NVDLA) could also be an interesting framework if all their code is released publicly, and not only some parts.

nGraph seems also to be really interesting but it doesn't support FPGAs yet.

Due to the framework being released a few days prior to the end of this study, we could not review and test HeteroCL but is seems to be the most interesting one of all. We can only recommend to check it out and follow their development.

In conclusion, frameworks for translating NN models to synthesizable hardware are still new and need some improvements but we are confident that some will arise and will offer good synthesization in the near future to tackle the Big Data growing needs for computation power.

References

[1] *A Modular Benchmarking Infrastructure for High-Performance and Reproducible Deep Learning*. URL: https://arxiv.org/abs/1901.10183.

[2] *Accelerating CNN inference on FPGAs: A Survey*. URL: https://arxiv.org/abs/1806.01683.

[3] *Apache License, Version 2.0*. URL: https://opensource.org/licenses/Apache-2.0.

[4] *Can FPGAs beat GPUs in Accelerating Next-Generation Deep Neural Networks?* URL: https://jaewoong.org/pubs/fpga17-next-generation-dnns.pdf.

[5] *DnnWeaver Github Repository*. URL: https://github.com/hsharma35/dnnweaver2.

[6] *DnnWeaver v2.0*. URL: http://dnnweaver.org.

[7] *Fathers of the Deep Learning Revolution Receive ACM A.M. Turing Award*. URL: https://www.acm.org/media-center/2019/march/turing-award-2018.

[8] *FPGA Caffe*. URL: https://github.com/dicecco1/fpga_caffe.

[9] *Google's Artificial Intelligence Built an AI That Outperforms Any Made by Humans*. URL: https://futurism.com/google-artificial-intelligence-built-ai.

[10] *GreenSocs*. URL: https://www.greensocs.com.

[11] *HeteroCL Github Repository*. URL: https://github.com/cornell-zhang/heterocl.

[12] *Hey Siri, You Can Hear Better Because Of Machine Learning*. URL: https://www.analyticsindiamag.com/apple-siri-homepod-machine-learning/.

[13] *hls4ml*. URL: https://github.com/hls-fpga-machine-learning/hls4ml.

[14] *How Are Tesla Vehicles Learning To Drive By Themselves?* URL: https://insideevs.com/news/345907/tesla-cars-learning-self-driving/.

[15] *How Netflix Saves 1 Billion Dollars A Year Using AI*. URL: https://www.valuewalk.com/2016/06/netflix-how-saves-1-billion-year-ai/.

[16] *How to classify MNIST digits with different neural network architectures*. URL: https://medium.com/tebs-lab/how-to-classify-mnist-digits-with-different-neural-network-architectures-39c75a0f03e3.

[17] *ImageNet*. URL: http://www.image-net.org.

[18] *LeFlow*. URL: https://github.com/danielholanda/LeFlow.

[19] *LeFlow Github: Issue #13*. URL: https://github.com/danielholanda/LeFlow/issues/13.

[20] *LegUp*. URL: http://legup.eecg.utoronto.ca.

[21] *LegUp 4.0 Programmer's Manual*. URL: http://legup.eecg.utoronto.ca/docs/4.0/programmermanual.html.

[22] *MLPerf*. URL: https://mlperf.org.

[23] *MNIST Dataset*. URL: http://yann.lecun.com/exdb/mnist.

[24] *Nervana Systems - Wikipedia*. URL: https://en.wikipedia.org/wiki/Nervana_Systems.

[25] *nGraph*. URL: https://ngraph.nervanasys.com.

[26] *nGraph Github Repository*. URL: https://github.com/NervanaSystems/ngraph.

[27] Michael A. Nielsen. *Neural Networks and Deep Learning*. 2018. URL: http://neuralnetworksanddeeplearning.com.

[28] *NVDLA GitHub*. URL: https://github.com/nvdla.

[29] *NVDLA License*. URL: http://nvdla.org/license.html.

[30] *NVDLA, NVIDIA Deep Learning Accelerator*. URL: http://nvdla.org.

[31] *Project Debater*. URL: https://www.research.ibm.com/artificial-intelligence/project-debater/.

[32] *Pull Request for Supporting Intel FPGA in VTA*. URL: https://github.com/dmlc/tvm/pull/1694.

[33] *PYNQ*. URL: http://www.pynq.io.

[34] *ResNet-50 Example*. URL: https://www.kaggle.com/gaborfodor/resnet50-example.

[35] *ResNet-50 v1.5*. URL: https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/Classification/RN50v1.5.

[36] *The 2-Clause BSD License*. URL: https://opensource.org/licenses/BSD-2-Clause.

[37] *The MIT License*. URL: https://opensource.org/licenses/MIT.

[38] *Toshiba's Considerations for NVDLA-based DNN SoC Design*. URL: https://toshiba.semicon-storage.com/content/dam/toshiba-ss/ncsa/en_us/docs/white-paper/Considerations_for_NVDLA-Based_DNN_SoC_Design_Whitepaper.pdf.

[39] *Training, validation and test sets*. URL: https://en.wikipedia.org/wiki/Training,_validation,_and_test_sets.

[40] *TVM*. URL: https://tvm.ai.

[41] *Very Deep Convolutional Networks for Large-Scale Image Recognition*. URL: https://arxiv.org/abs/1409.1556.

[42] *VTA: An Open, Customizable Deep Learning Acceleration Stack*. URL: https://tvm.ai/2018/07/12/vta-release-announcement.html.

[43] *What is Machine Learning?* URL: https://www.mathworks.com/discovery/machine-learning.html.

[44] *Wide Residual Networks*. URL: https://arxiv.org/abs/1605.07146.

Acronyms

AI Artificial Intelligence 1

BDT Boosted Decision Trees 5

CNN Convolutional Neural Network 2

DNN Deep Neural Network 2, 3
DSL Domain-Specific Language 8

FPGA Field-Programmable Gate Array 3–5, 7, 8, 13

HLS High-Level Synthesis 5

JIT Just-in-Time 6

LRN Local Response Normalisation 3

ML Machine Learning 1
MLP Multi-Layer Perceptron 5, 10, 11

NN Neural Network 1, 2, 5, 10, 11
NVDLA NVIDIA Deep Learning Accelerator 4

PBQP Partitioned Boolean Quadratic Programming 2

ReLU Rectified Linear Unit 3, 11
ResNet Residual Neural Network 11
RTL Register Transfer Language 4

SoC System on Chip 4

VTA Versatile Tensor Accelerator 6, 7

Appendices

A Authentication

I, Lucas Elisei, hereby declare having realised this work alone and not having used any other resources than those quoted in the bibliography.

Par la présente, je soussigné, Lucas Elisei, déclare avoir réalisé seul ce travail et ne pas avoir utilisé d'autres sources que celles citées dans la bibliographie.

Date	Signature

Lucas Elisei

B src/run_leflow.py

```
################################################################################
# File: leflow.py
#
# Created by: Lucas Elisei <lucas.elisei@master.hes-so.ch>
# Created on: 26.05.2019
#
# LeFlow implementation for a MLP 784(Input)-128(HiddenLayer1)-64(HiddenLayer2)
#   -10(Output) model.
################################################################################

# Import libraries.
import numpy as np
import tensorflow
from tensorflow.examples.tutorials.mnist import input_data

import sys
sys.path.append('../../src')
import processMif as mif

# Load MNIST dataset.
mnist_data = input_data.read_data_sets('MNIST_data/', one_hot=True)

# Number of inputs and outputs.
num_input  = 784    # 28*28 pixels
num_output = 10     # 0-9 digits

# Number of neurons for each hidden layers.
num_layers_0 = 128
num_layers_1 = 64

# Initialisation parameters.
learning_rate_init = 0.001
regularizer_rate   = 0.1

# Placeholder for input data.
input_x = tensorflow.placeholder(tensorflow.float32, shape=(None, num_input), name="input_x")
input_y = tensorflow.placeholder(tensorflow.float32, shape=(None, num_output), name="input_y")
# Dropout layer.
keep_prob = tensorflow.placeholder(tensorflow.float32)

# Initialise weights and biases to 0.
weights_0 = tensorflow.Variable(tensorflow.zeros([num_input, num_layers_0]))
biases_0  = tensorflow.Variable(tensorflow.zeros([num_layers_0]))

weights_1 = tensorflow.Variable(tensorflow.zeros([num_layers_0, num_layers_1]))
biases_1  = tensorflow.Variable(tensorflow.zeros([num_layers_1]))

weights_2 = tensorflow.Variable(tensorflow.zeros([num_layers_1, num_output]))
biases_2  = tensorflow.Variable(tensorflow.zeros([num_output]))

# Define hidden layers.
hidden_output_0   = tensorflow.nn.relu(tensorflow.matmul(input_x, weights_0) + biases_0)
hidden_output_0_0 = tensorflow.nn.dropout(hidden_output_0, keep_prob)

hidden_output_1   = tensorflow.nn.relu(tensorflow.matmul(hidden_output_0_0, weights_1) + biases_1)
hidden_output_1_1 = tensorflow.nn.dropout(hidden_output_1, keep_prob)

predicted_y       = tensorflow.matmul(hidden_output_1_1, weights_2) + biases_2

# Define the loss function.
loss = tensorflow.reduce_mean(tensorflow.nn.softmax_cross_entropy_with_logits_v2(logits=predicted_y,
 ↪  labels=input_y))   \
    + regularizer_rate * (tensorflow.reduce_sum(tensorflow.square(biases_0)) +
     ↪  tensorflow.reduce_sum(tensorflow.square(biases_1)))

# Variable learning rate.
learning_rate = tensorflow.train.exponential_decay(learning_rate_init, 0, 5, 0.85, staircase=True)

# Adam optimizer for finding the correct weight.
optimizer = tensorflow.train.AdamOptimizer(learning_rate)
```

```
train_step = optimizer.minimize(loss, var_list=[weights_0, weights_1, weights_2,
                                                biases_0, biases_1, biases_2])

# Initialisation
init = tensorflow.global_variables_initializer()

with tensorflow.Session() as session:
    epochs = 1000
    session.run(init)

    for _ in range(epochs):
        batch_x, batch_y = mnist_data.train.next_batch(100)
        session.run(train_step, feed_dict={input_x: batch_x, input_y: batch_y})

    correct_prediction = tensorflow.equal(tensorflow.argmax(input_y, 1), tensorflow.argmax(input_y, 1))
    accuracy = tensorflow.reduce_mean(tensorflow.cast(correct_prediction, tensorflow.float32))
    net_accuracy = session.run(accuracy, feed_dict={input_x: mnist_data.test.images, input_y:
     ↪  mnist_data.test.labels})

    print('The accuracy over the MNIST dataset is {:.2f}%'.format(net_accuracy * 100))

    # Generating hardware.
    with tensorflow.device('device:XLA_CPU:0'):
        y = tensorflow.matmul(hidden_output_1_1, weights_2)[0] + biases_2
    session.run(y, {input_x: [mnist_data.test.images[123]]})

    # Creating memories for testing.
    test_image = 123
    mif.createMem([biases_0.eval(), weights_0.eval(), biases_1.eval(), weights_1.eval(),
     ↪  biases_2.eval(), weights_2.eval(), mnist_data.test.images[test_image]])

    # Print expected result.
    print('Expected result: {}'.format(str(np.argmax(mnist_data.test.labels[test_image]))))
```

C src/run_tvm.py

```
################################################################################
# File: tvm.py
#
# Created by: Lucas Elisei <lucas.elisei@master.hes-so.ch>
# Created on: 21.05.2019
################################################################################

# Imports
import numpy as np
import os
import random
import sys
import time

from matplotlib import pyplot as plt
from PIL import Image

# TVM imports
import tvm
import nnvm.compiler
import nnvm.testing
import vta
import vta.testing

from tvm import autotvm
from tvm.autotvm.tuner import RandomTuner
from tvm.contrib.util import tempdir
import tvm.contrib.graph_runtime as runtime

# Get VTA environment.
env = vta.get_env()

# Get network workload.
def get_network(network: str,
                batch_size: int,
                num_classes: int,
                num_layers: int,
                image_shape: tuple,
                dtype: str):
    if (network == 'resnet'):
        net, params = nnvm.testing.resnet.get_workload(batch_size=batch_size,
                                                       num_classes=num_classes,
                                                       num_layers=num_layers,
                                                       image_shape=image_shape,
                                                       dtype=dtype)
    elif (network == 'mlp'):
        net, params = nnvm.testing.mlp.get_workload(batch_size=batch_size,
                                                    num_classes=num_classes,
                                                    image_shape=image_shape,
                                                    dtype=dtype)
    else:
        raise ValueError('Invalid network: {}'.format(network))

    return net, params

# Tasks tuning.
def tune_tasks(tasks,
               measure_option,
               n_trial=1000,
               early_stopping=None,
               log_filename='tuning.log',
               use_transfer_learning=True):

    # create tmp log file
    tmp_log_file = log_filename + ".tmp"
    if os.path.exists(tmp_log_file):
        os.remove(tmp_log_file)

    for i, tsk in enumerate(reversed(tasks)):
        prefix = "[Task %2d/%2d] " % (i+1, len(tasks))
```

```
        # Create tuner.
        tuner_obj = RandomTuner(tsk)

        if use_transfer_learning:
            if os.path.isfile(tmp_log_file):
                tuner_obj.load_history(autotvm.record.load_from_file(tmp_log_file))

        # do tuning
        tuner_obj.tune(n_trial=min(n_trial, len(tsk.config_space)),
                       early_stopping=early_stopping,
                       measure_option=measure_option,
                       callbacks=[
                           autotvm.callback.progress_bar(n_trial, prefix=prefix),
                           autotvm.callback.log_to_file(tmp_log_file)])

    # pick best records to a cache file
    autotvm.record.pick_best(tmp_log_file, log_filename)
    os.remove(tmp_log_file)

# Tune and evaluate network.
def tune_and_evaluate(tuning_opt, do_tuning=True):
    # Get network workload.
    print('Getting network workload...')
    net, params = get_network(network, batch_size, num_classes, num_layers, image_shape, dtype)

    data_shape = {'data': (batch_size, ) + image_shape}

    # Extract tasks.
    if (do_tuning == True):
        print('Extracting tasks...')
        tasks = autotvm.task.extract_from_graph(net,
                                                shape=data_shape,
                                                dtype=dtype,
                                                target=target,
                                                symbols=symbols,
                                                target_host=target_host)
        print('Tuning tasks...')
        tune_tasks(tasks, **tuning_opt)
    else:
        print('Extracting tasks from log file...')

    # Compile kernels with history best records (using log file).
    with autotvm.apply_history_best(tuning_opt['log_filename']):
        print('Compiling kernels...')
        with nnvm.compiler.build_config(opt_level=3):
            graph, lib, params = nnvm.compiler.build(net,
                                                     target=target,
                                                     shape=data_shape,
                                                     dtype=dtype,
                                                     params=params,
                                                     target_host=target_host)

        if (env.TARGET == 'pynq'):
            print('Uploading to PYNQ board...')
            tmp = tempdir()
            filename = 'net_pynq.tar'
            lib.export_library(tmp.relpath(filename))

            # Upload module to the board.
            remote_start = time.time()
            remote = autotvm.measure.request_remote(device_key, host=rpc_host, port=rpc_port,
             ↪  timeout=10000)
            remote.upload(tmp.relpath(filename))
            rlib = remote.load_module(filename)
            remote_time = time.time() - remote_start
            print('Module uploaded to board in {0:.2f}s'.format(remote_time))

            ctx = remote.context(str(target), 0)
            module = runtime.create(graph, rlib, ctx)
        elif (env.TARGET == 'sim'):
            ctx = tvm.cpu()
            module = runtime.create(graph, lib, ctx)
```

```
        # Create random data for inference.
        data = tvm.nd.array((np.random.uniform(size=data_shape['data'])).astype(dtype))

        # Upload data and parameters to target.
        module.set_input('data', data)
        module.set_input(**params)

        # Evaluate inference.
        print('Evaluating inference time cost...')
        ftimer = module.module.time_evaluator('run', ctx, number=1, repeat=100)
        prof_res = np.array(ftimer().results) * 1000    # Convert to milliseconds.

        print('Mean inference time (std dev): {:.2f} ms ({:.2f} ms)'.format(np.mean(prof_res),
         ↪  np.std(prof_res)))

# Constants
network = 'mlp'
batch_size = 1
num_classes = 10
num_layers = 50
image_shape = (1, 28, 28)
dtype = 'float32'
symbols = (nnvm.sym.conv2d, nnvm.sym.dense)

# Compilation variables.
if (env.TARGET == 'sim'):       # CPU
    device      = 'vtacpu'
    target      = 'llvm'
    target_host = 'llvm'
    tuning_opt = {
        'log_filename': '{}_{}.log'.format(network, env.TARGET),

        'n_trial': 10,
        # 'early_stopping': 80,

        'measure_option': autotvm.measure_option(
            builder=autotvm.LocalBuilder(),
            runner=autotvm.LocalRunner(number=10, repeat=10, min_repeat_ms=1000)
        )
    }

elif (env.TARGET == 'pynq'):    # Pynq board
    device      = 'vta'
    target      = 'llvm -device=arm_cpu -target=armv7-none-linux-gnueabihf'
    target_host = 'llvm -mtriple=armv7-none-linux-gnueabihf -mcpu=cortex-a9 -mattr=+neon'
    rpc_host    = '0.0.0.0'
    rpc_port    = 9191
    device_key  = 'pynq'
    tuning_opt = {
        'log_filename': '{}_{}.log'.format(network, env.TARGET),

        'n_trial': 10,
        #'early_stopping': 80,

        'measure_option': autotvm.measure_option(
            builder=autotvm.LocalBuilder(),
            runner=autotvm.RPCRunner(
                key=device_key, host=rpc_host, port=rpc_port,
                number=10, repeat=10,
                timeout=10,
            ),
        ),
    }

else:                           # Unknown target
    raise ValueError('Unknown target: {}'.format(env.TARGET))

# Define how many threads TVM can use.
num_threads = 4
os.environ['TVM_NUM_THREADS'] = str(num_threads)
```

```
tune_and_evaluate(tuning_opt, do_tuning=True)
```