tinyHLS

Template-Based, Layer-Oriented High Level Synthesis for CNNs

tinyHLS userguide

Version v0.9

Documentation

The online documentation of the project (a.k.a. the **user guide**) is available on GitHub-pages: https://github.com/Fraunhofer-IMS/tinyHLS

Table of Contents

Chapter 1. Application Scenario

Toolbox Description

tinyHLS is a compact hardware compiler developed by Fraunhofer IMS, which is an useful toolbox for embedded systems engineers and data scientists responsible for developing and deploying edge AI solutions. It processes tensorflow.keras neural networks and translates them into a dedicated co-processor unit which is suitable for integration with embedded processing systems. The generated outputs are platform independent and consist entirely of synthesizable and technology-agnostic Verilog HDL code. The co-processor is wrapped within an AHB-lite interface to allow easy integration into custom processor systems. Figure 1 shows the simplified overview of tinyHLS.

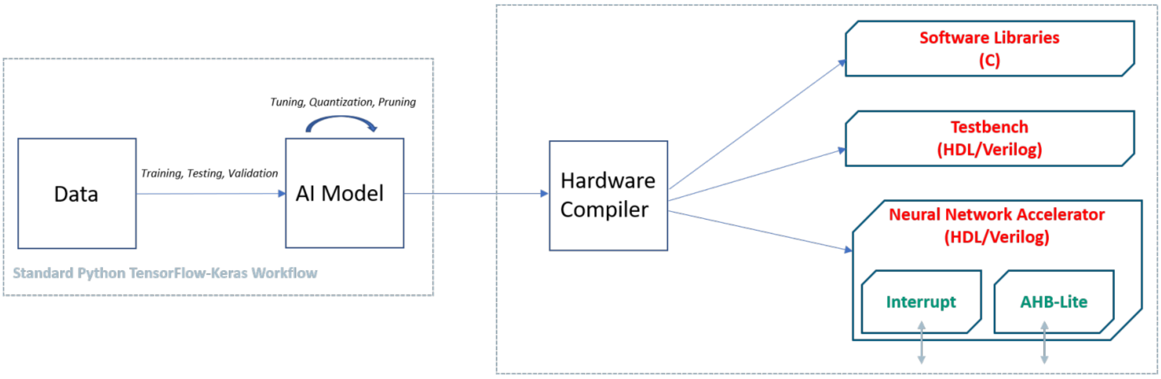
Tuning, Quantization, Pruning
Data
Training, Testing, Validation
AI Model
Standard Python TensorFlow-Keras Workflow
Hardware Compiler
Software Libraries (C)
Testbench (HDL/Verilog)
Neural Network Accelerator (HDL/Verilog)
Interrupt
AHB-Lite

Figure 1: tinyHLS overview

Motivation

- **Real-time**: real-time processing is crucial and requires a constant flow of receiveing, processing and exporting data to maintain real-time insights.
- **Resource constraints**: edge devices have limited resources (size, throughput, energy budget) and limited power supply compared to cloud servers.
- **Time-to-market**: designing and implementing specialized hardware accelerators by hand is time-consuming and requires specialized expertise.

Features

- **Template-based generation**: tinyHLS leverages templates to create hardware accelerators, enabling easy extension and customization for different AI models and tasks. This significantly reduces the time and effort required to develop hardware accelerators, allowing implementers to focus on refining their AI models.
- **Technology- and vendor-Agnostic**: the accelerators generated by tinyHLS are compatible with various deployment platforms, ensuring flexibility and broad applicability. This technology-independent nature ensures the accelerators can be deployed across various platforms without modification.
- **AHB-lite-compatible Interface** provides seamless integration with embedded processing systems, such as the AIRISC RISC-V processor. This specialized interface allows the accelerator

to be efficiently integrated into edge device as memory-mapped peripheral.

- **Software HALs** offers hardware abstraction layers to control the accelerators from software. Developers can use the provided HALs, templates, and wrappers to create applications that control and interact with the accelerators. That simplified the development process.
- **Supported CNN Layers**: current support includes conv1D, maxpool1D, globalaveragepool1D, dense, linear and ReLU activation functions. For example, a 1D convolutional neural network (CNN) with ReLU activations can be trained as a base model and then compiled into a dedicated hardware accelerator using tinyHLS.
- **Performance and efficiency**: the generated accelerators provide substantial speedups, meeting the real-time processing requirements of edge AI applications. Benchmarking has shown that the hardware accelerator can achieve a 75x speedup compared to the pure-software implementation, significantly
- **Scalability and adaptability**: The template-based approach allows for easy extension to support other types of neural networks, making tinyHLS adaptable to future requirements. This scalability ensures that tinyHLS remains a valuable tool as AI models and requirements evolve.

Chapter 2. Usage

2.1. Installation

To install tinyHLS, these instructions should be followed step by step,the steps can also be seen in the Github:

Step 1: Install conda environment

It is recommended to use the framework in a conda environment. The official website is available to download and install conda on your computer or server.

Step 2: Install Python

Conda treats Python the same way as any other package. Hence, it is possible to install Python easily by conda. The corresponding website: managing Python

Step 3: Clone the tinyHLS repository

Clone the tinyHLS im Github auf your computer or server

Step 4: Create the Virtual Environment tinyHLS

```
$ conda env create -f environment.yml
```

Step 5: Activate the virtual environment

```
$ conda activate tinyHLS
$ pip install
```

2.2. Use

A model can be trained or loaded using the standard workflow in Python TensorFlow Keras. The first part of the translation process is, to convert the weights of the user CNN into Verilog includes by running the following commands:

```
$ tinyhls.extract_weights()
$ tinyhls.convert_weights_to_hex()
$ tinyhls.create_verilog_includes()
```

Second, the model is translated into HDL and a testbench with examplary values is generated by the following commands:

```
$ tinyhls.translate_model()
$ tinyhls.create_testbench()
```

Figure 2 shows the workflow using tinyHLS. The five functions are executed in sequence, and the resulting verilog files can be generated.

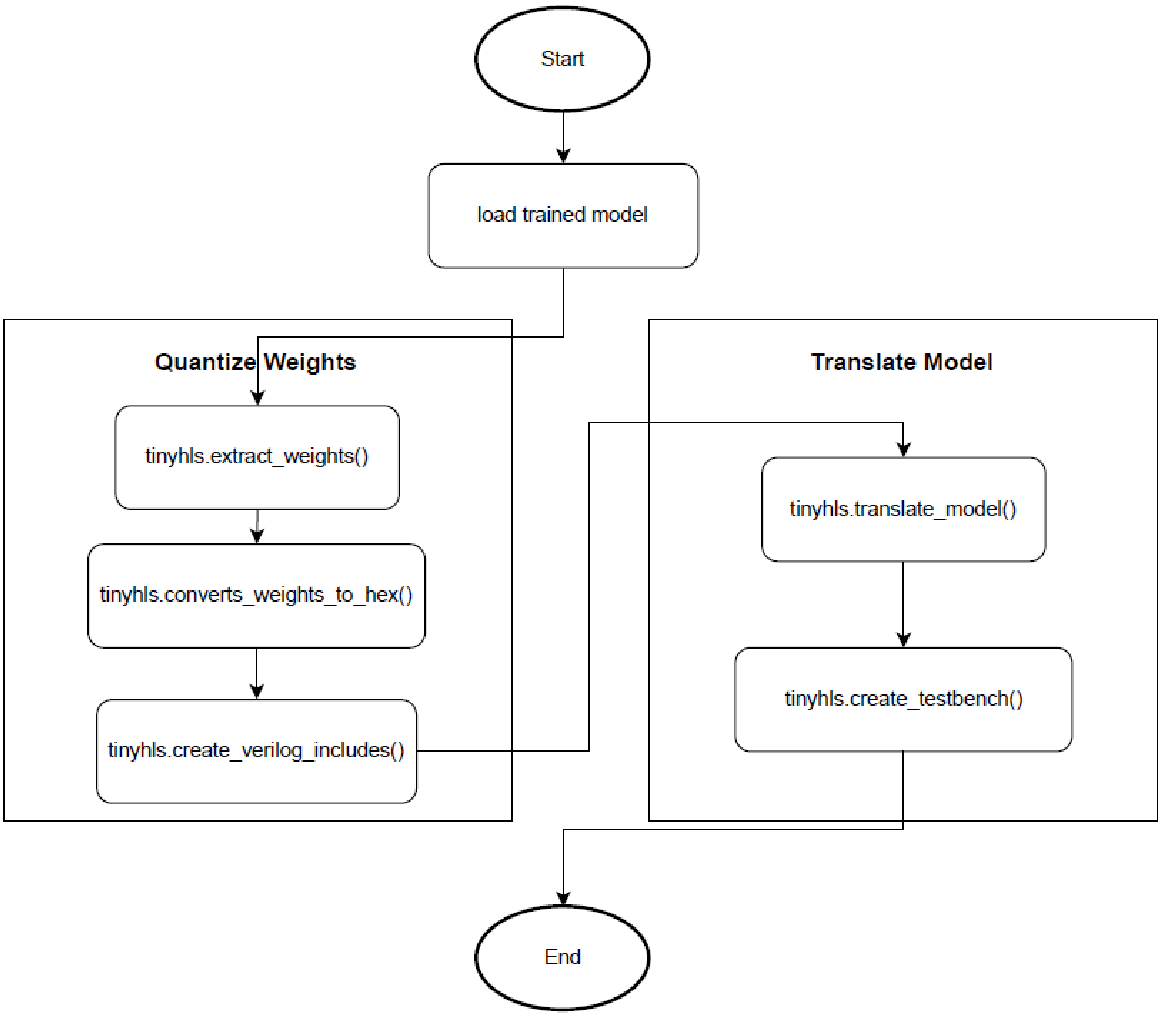
Start
load trained model
Quantize Weights
tinyhls.extract_weights()
tinyhls.converts_weights_to_hex()
tinyhls.create_verilog_includes()
Translate Model
tinyhls.translate_model()
tinyhls.create_testbench()
End

Figure 2: Translation flowchart using tinyHLS

Detailed descriptions of each function are given below.

1. Extracting the weights: The weights and biases of input model for each layer are extracted in floating-point format and saved as text files in the output directory.

```
tinyhls.extract_weights(model,destination_path)
```

About arguments:

`model` is an instance of `keras.models.Model`.

`destination_path` is a string denoting the output directory.

2. Converting into hex: The extracted weights and biases will be converted into hex format and saved in hex files. `tinyhls.convert_weights_to_hex()` is used for extracted weights and `tinyhls.convert_bias_to_hex()` is used for extracted biases.

```
tinyhls.convert_weights_to_hex(source_path, destination_path, txt_files,
BIT_WIDTH, INT_WIDTH)
```

```
tinyhls.convert_bias_to_hex(source_path, destination_path, txt_files, BIT_WIDTH,
INT_WIDTH)
```

About arguments:

`source_path` is the string denoting the location of text files of extracted weights.

`destination_path` is a string denoting the output directory for generated hex files.

`txt_files` is an array containing the name of text files where weights (or biases) are stored.

`BIT_WIDTH` is an integer denoting the expected total quantized bit width.

`INT_WIDTH` is an integer denoting the integer bit width for converting floating point weights (or biases) into a fixed point data format.

3. Generating Verilog files of weights and biases: The hex files will be read. The weights and biases will be assigned to a particular array location. Then the corresponding Verilog files will be generated.

```
tinyhls.create_verilog_includes(source_path, destination_path, model_json,
BIT_WIDTH)
```

About arguments:

`source_path` is the string denoting the location of hex files.

`destination_path` is a string denoting the output directory for generated verilog files.

`model_json` is a JSON string of the neural network architecture.

`BIT_WIDTH` is the integer denoting the expected total quantized bit width.

4. Generating HDL Code: HDL code will be generated for all layers and saved as verilog file.

```
tinyhls.translate_model(model_arch, param_path, output_path, quantization,
file_name="tinyhls_cnn", fast=False, use_relative_path=True)
```

About arguments:

`model_arch` is the JSON string.

`param_path` is the string denoting the location of generated Verilog files.

`output_path` is a string denoting the output directory where the generated Verilog file can be saved.

`quantization` is a dictionary containing the quantization parameters of the weights. It is used to specify the bit-width and fractional bits for the fixed-point representation in the hardware description language (HDL) code. For example, `quantization` is defined as `{'total': 32, 'int': 10, 'frac': 22}` in the test example. That means, the total number of bits used for the fixed-point representation is 32 bits, the number of bits used for the integer part of the fixed-point representation is 10 bits and the number of bits used for the fractional part of the fixed-point representation is 22 bits.

`file_name="tinyhls_cnn"` is the name of the output Verilog file. By default it is `tinyhls_cnn`.

`fast=False` is a Boolean argument denoting a configuration option to control the performance or resource usage of the HDL generation process. By default it is `False`, which means the generated HDL prioritizes resource efficiency. By `True` the generated HDL will prioritize speed over resource usage.

`use_relative_path` is a boolean argument denoting if the files included in the generated Verilog file should use relative paths or absolute paths.By default it it `True`.

5. Generating testbench: A testbench will be generated for the CNN.

```
tinyhls.create_testbench(model_arch, quantization, clk_period, destination_path,
file_name="tinyHLS_tb")
```

About arguments:

`model_arch` ist the JSON String.

`quantization` is the dictionary containing the quantization parameters of the weights.

`clk_period` is an integer which shows the basic time delay in generated Verilog file.

destination_path is a string denoting the output directory.

file_name is the name of the output Verilog file. By default it is tinyhls_tb.

2.3. Test Example

An example of a small convolutional neural network (CNN) is provided for test in the test.py file. It consists of nine cells that effectively showcase the capabilities of tinyHLS.

1. Import the necessary packages.
2. Build a convolutional neural network (CNN) model and test the model. The model contains various layers stacked on top of each other. The output of one layer feeds into the input of the next layer. Two convolution layers (conv and conv2) are added.
3. Save the architecture of the model and save it into a Python dictionary, which is used as argument for tinyhls.
4. Output the prediction of each convolution layer and the prediction of the CNN model.
5. Extract the weights and biases with `tinyhls.extract_weights()`.
6. Convert the extracted weights and biases into hex format with `tinyhls.convert_weights_to_hex()` and `tinyhls.convert_bias_to_hex()`.
7. Generate Verilog files for extracted weights and biases with `tinyhls.create_verilog_includes()`.
8. Generate Verilog files for all layers with `tinyhls.translate_model()`.
9. Generate the testbench for the convolutional neural network (CNN) model with `tinyhls.create_testbench()`.

2.4. Requirements

Please see environment.yml for a detailed list of python packages.

Chapter 3. Legal

Contact

tinyHLS
Fraunhofer IMS
Duisburg, Germany
tinyhls@ims.fraunhofer.de

Acquisition

tinyHLS
https://github.com/Fraunhofer-IMS/tinyHLS

License

This project is provided under the GPL-3.0 license. A special exception was included, that applies particularly for the outputs of the framework. Outputs of this project can be used under the SHL-2.1 license.