Lattice sensAI Machine Language Engine Simulator

User Guide

FPGA-UG-02228-1.0

November 2024

Disclaimers

Lattice makes no warranty, representation, or guarantee regarding the accuracy of information contained in this document or the suitability of its products for any particular purpose. All information herein is provided AS IS, with all faults, and all associated risk is the responsibility entirely of the Buyer. The information provided herein is for informational purposes only and may contain technical inaccuracies or omissions, and may be otherwise rendered inaccurate for many reasons, and Lattice assumes no obligation to update or otherwise correct or revise this information. Products sold by Lattice have been subject to limited testing and it is the Buyer's responsibility to independently determine the suitability of any products and to test and verify the same. LATTICE PRODUCTS AND SERVICES ARE NOT DESIGNED, MANUFACTURED, OR TESTED FOR USE IN LIFE OR SAFETY CRITICAL SYSTEMS, HAZARDOUS ENVIRONMENTS, OR ANY OTHER ENVIRONMENTS REQUIRING FAIL-SAFE PERFORMANCE, INCLUDING ANY APPLICATION IN WHICH THE FAILURE OF THE PRODUCT OR SERVICE COULD LEAD TO DEATH, PERSONAL INJURY, SEVERE PROPERTY DAMAGE OR ENVIRONMENTAL HARM (COLLECTIVELY, "HIGH-RISK USES"). FURTHER, BUYER MUST TAKE PRUDENT STEPS TO PROTECT AGAINST PRODUCT AND SERVICE FAILURES, INCLUDING PROVIDING APPROPRIATE REDUNDANCIES, FAIL-SAFE FEATURES, AND/OR SHUT-DOWN MECHANISMS. LATTICE EXPRESSLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS OF THE PRODUCTS OR SERVICES FOR HIGH-RISK USES. The information provided in this document is proprietary to Lattice Semiconductor, and Lattice reserves the right to make any changes to the information in this document or to any products at any time without notice.

Inclusive Language

This document was created consistent with Lattice Semiconductor’s inclusive language policy. In some cases, the language in underlying tools and other items may not yet have been updated. Please refer to Lattice’s inclusive language FAQ 6878 for a cross reference of terms. Note in some cases such as register names and state names it has been necessary to continue to utilize older terminology for compatibility.

Contents

Tables

Abbreviations in This Document

A list of abbreviations used in this document.

Abbreviation	Definition
Conv2D	2D Convolutional Layer
CPU	Central Processing Unit
FPGA	Field Programmable Gate Arrays
GPU	Graphics Processing Unit
ML	Machine Language

1. Introduction

A simulator is a tool that allows running networks in software and getting the same results as if they were run on a Lattice Semiconductor FPGA. The Simulator can be used to investigate why and where results differ between CPU or GPU networks and the FPGA.

We expect results to differ because the FPGA does not represent values with floating point numbers but uses its own custom fixed-point representations. This gives slight changes to the results of all calculations, which can add up over the course of a run. It also means that values can more easily exceed their allowed range, leading to saturation.

With the Simulator, we can analyze not only the final output of the network, but also the results of each layer individually.

The ML Engine Simulator is implemented in C++, but with wrappings that allow it to be imported as a Python package for use in Python scripts.

2. Simulator Features

2.1. Core Features

- For any given input and supported network, the simulator will produce outputs that matches the results obtained on the FPGA platform.
- The simulator can be run in both FPGA fixed-point and floating-point number modes.
- The simulator supports more than 15 different layer types.
- The simulator supports multiple input layers (experimental feature).
- The simulator supports multiple output layers.
- The simulator is fast. It can be used in real time with networks of considerable sizes
- It comes in an easy-to-use Python package that can be imported and used in any script.
- It can be used both on Windows and Linux.

2.2. Diagnostic Features

Diagnostic features are divided in three parts: Output Inspection, Output Comparison and Saturation Statistics.

2.2.1. Output Inspection

- The simulator has two run configurations: output only and layer by layer results.
- In the first configuration, only the results of the last layers of the network are returned.
- In the second configuration, the simulator returns an object that contains results for each layer individually and that provides methods to query and retrieve the output of any layer in the network.

2.2.2. Outputs Comparison

The simulator allows for side-by-side comparison of two simulation runs, down to the individual layer level. This feature enables users to analyze the differences in model behavior when running in different modes, such as comparing floating-point number mode to FPGA mode to understand how the model performs on hardware.

The simulator can compare the two simulation runs using aggregate metrics. One example is to determine the largest absolute difference between the results of the tenth convolution layer when running in FPGA mode versus floating-point number mode. This metric provides a quantitative measure of the discrepancy between the two implementations. Table 2.1. lists the supported aggregate metrics.

Table 2.1. Supported Aggregate Metrics

Metric Name	Metric
RMS	Root Mean Square Error
MAXE	Maximum Absolute Error
MAE	Mean Absolute Error
RMSPE	Root Mean Square Percent Error
MAXPE	Max Absolute Percent Error

Comparison can be made elementwise between layers. For example, getting a vector representing the differences between two runs of the simulator for a specific layer in the network. Vectors in element-wise comparison operations can be clipped to the range of the layer under study. This is useful when comparing between floating point number results and FPGA mode results to bring the floating-point number results into the valid range and avoid excessive differences due to saturation. Table 2.2. lists the supported element-wise metrics.

Table 2.2. Supported Element-Wise Metrics

Feature	Software Default Mode State (Programmed)
Simple	Raw difference between results and reference results = R - Ref
Absolute	Absolute differences between results and reference results = abs(R - Ref)
Relative	Fraction of the raw difference between results over reference results = (R - Ref) / Ref

2.2.3. Saturation Statistics

A results object obtained from a run of the simulator on some input can be queried to retrieve a saturation object representing saturation - a measure of clipping - on a layer-by-layer basis. For a value outside its allowed range, saturation is defined as the absolute difference between the value and the closest limit of the range.

Supported saturation statistics include the following saturation object properties as shown in Table 2.3.

Table 2.3. Saturation Statistics

Feature	Software Default Mode State (Programmed)
saturation	Total saturation for the given layer.
count	Number of times a saturation was observed while computing the layer.
avg	Average saturation in the given layer. Essentially saturation divided by count.
ad_ops	Number of addition operations performed by the layer. Estimate of amount of computation.
sat_rel_add_ops	Total saturation for the given layer divided by the number of add operations.
range	Allowed range of the values for the given layer.
sat_rel_range	Average saturation divided by the range of the layer.

Saturation statistics can be returned for a particular layer in the network as above, or for a particular layer type in the network. In that case, the statistics are the same, only the values are aggregated by layer type.

2.2.4. Weights Manipulation Features

The simulator provides a way to retrieve and change weights in a model. Layer types supporting this feature are layers with weights:

- Convolution 2D
- Depth Wise Convolution 2D
- Batch Normalization
- Dense

This feature is useful in at least two cases:

- Diagnostics — One run of the simulator can be made with a set of weights and another with another set of weights only to compare differences between the two runs on a layer-by-layer basis.
- Test set evaluation — At any time during training, it is useful to determine the error of the current model on the test set as if on FPGA. To do that we need the ability to change the weights of the model if currently loaded in the simulator.

3. Installation

3.1. Requirements

- Python versions 3.8 - 3.11

3.2. Installation Steps

The installation process for the simulator consists of installing a Python wheel. This way, it is not necessary to manually install dependencies.

To install the simulator wheel, it is strongly recommended to first create a Python environment using *venv* or *conda*. Once the environment is created, you need to activate it. Then, follow the instructions below depending on your operating system of choice.

3.2.1. Windows

To install the simulator wheel in Windows, open a command line or PowerShell prompt, activate your Python environment if not already activated, and type:

```
python -m pip install ml_engine_simulator-x.x.x-py3-none-win_amd64.whl --force-reinstall
```

Note: `x.x.x` must be replaced by your current version of the simulator.

3.2.2. Linux

To install the simulator wheel in Linux, open a shell and first install `libsz2.` This can be done, for example here on Ubuntu, by issuing these commands:

```
sudo apt-get update
sudo apt-get install libsz2
```

Then activate your Python environment if not already activated and install the simulator wheel:

```
python3 -m pip install ml_engine_simulator-x.x.x-py3-none-linux_x86_64.whl --force-reinstall
```

Note: `x.x.x` must be replaced by your current version of the simulator.

3.3. Using and Importing the Simulator

To use the simulator, make sure your Python environment is activated, run Python and import the package:

```
import ml_engine_simulator.network as sim
```

When performing the steps above and regardless of your operating system, it is possible to import ml_engine_simulator.network from any directory, provided your Python environment where the simulator is installed is activated.

4. Supported Layer Types

This section enumerates the layer types supported by the simulator. Table 4.1 summarizes the layer types. Note that in this table:

- Layer refers to the type of layer
- Input Rep is the input fixed-point number representation as accepted by the layer
- Weights Rep is the fixed-point number representation used to quantize the weights
- Work Rep is the fixed-point number representation used to carry out operations
- Output Rep is the output fixed point number representation as returned by the layer
- Comments are any further comments where applicable

Representations in the Table 4.1 are denoted as:

- X.Y means *X* integer bits, *Y* fractional bits.
- Representations are either *signed* or *unsigned* as indicated in Table 4.1.
- Scale, for example a scale of 0.5, means values are further multiplied by the scale value.

Some examples:

- Unsigned 1.7 range = [0.0, 2.0) with a step size of 1/128 ➔ [0.0, 2 - 1/128]
- Signed 0.7 with 0.5 scale range = [-0.5, 0.5) with a step size of 1/256 ➔ [-0.5, 0.5 - 1/256]
- Signed 5.10 range = [-32, 32) with a step size of 1/1024 ➔ [-32, 32 - 1/1024]

Some considerations regarding representations:

- Representations are specified in a quantization file. The ones in Table 4.1 are currently in use. There should be no need to change those.
- The *Quantize* version of some layer types below can have their own representation, rather than per layer type. However, the simulator currently uses the representations in Table 4.1, which are per layer type for now.

Table 4.1. MachXO3D Feature Row Elements

Layer	Input Rep	Weights Rep	Work Rep	Output Rep	Comments
InputLayer	Unsigned 1.7	None	None	Unsigned 1.7	—
Conv2D sensAI>QuantizeConv2D	Unsigned 1.7	Signed 0.7 (scale 0.5)	Signed 16.15	Signed 5.10	—
DepthWiseConv2D sensAI>QuantizeDepthwiseConv2D	Unsigned 1.7 or Signed 5.10	Signed 0.7 (scale 0.5)	Signed 16.15	Signed 5.10	Input representation depends on previous layer
Dense sensAI>QuantizeDense	Unsigned 1.7	Signed 0.7 (scale 0.5)	Signed 24.15	Signed 5.10 or Signed 2.13	Output representation is 2.13 when Dense is followed by QuantizeActivation sigmoid, 5.10 otherwise. Support for stacked dense layers is experimental now.
MaxPooling2D	Unsigned 1.7	None	Unsigned 1.7	Unsigned 1.7	—
ReLU (activation) sensAI>QuantizeActivation (ReLU)	Unsigned 1.7 or Signed 5.10	None	Unsigned 1.7	Unsigned 1.7	Input representation is 5.10 when following Dense. It

Layer	Input Rep	Weights Rep	Work Rep	Output Rep	Comments
					is 1.7 to chain with Lambda otherwise.
Add sensAI>QuantizeAdd	Unsigned 1.7	None	Unsigned 1.7	Signed 5.10	We need output at 5.10 to chain with Lambda
Batch Norm	Signed 5.10	Signed 5.10	Signed 5.10	Signed 5.10	—
Lambda	Signed 5.10 or Unsigned 1.7	None	Unsigned 1.7	Unsigned 1.7	Input representation depends on previous layer
Concatenate sensAI>QuantizeConcat	Any	None	None	Same as input	Layer simply moves data
Flatten	Any	None	None	Same as input	Layer simply moves data
Slicing	Any	None	None	Same as input	Layer simply moves data
Resizing	Unsigned 1.7	Signed 0.7 (scale 0.5)	Signed 16.15	Unsigned 1.7	—
Reciprocal	Signed 5.10	None	Signed 5.10	Signed 5.10	f(x) = 1/x
Multiply	Signed 5.10 or Unsigned 1.7	None	Signed 5.10	Signed 5.10 or Unsigned 1.7	Input representation depends on previous layer Output representation is 1.7 if layer is followed by Lambda
SensAI> QuantizeActivation (sigmoid) (experimental)	Signed 2.13	None	None	Signed 5.10	—
Dropout	Any	None	None	Same as input	This layer is not considered by the simulator. It is as if it was nonexistent in the network.
Slicing	Any	None	None	Same as input	Layer simply moves data
Resizing	Unsigned 1.7	Signed 0.7 (scale 0.5)	Signed 16.15	Unsigned 1.7	—
Reciprocal	Signed 5.10	None	Signed 5.10	Signed 5.10	f(x) = 1/x
Multiply	Signed 5.10 or Unsigned 1.7	None	Signed 5.10	Signed 5.10 or Unsigned 1.7	Input representation depends on previous layer Output representation is 1.7 if layer is followed by Lambda
SensAI> QuantizeActivation (sigmoid) (experimental)	Signed 2.13	None	None	Signed 5.10	—
Dropout	Any	None	None	Same as input	This layer is not

Layer	Input Rep	Weights Rep	Work Rep	Output Rep	Comments
					considered by the simulator. It is as if it was nonexistent in the network.

5. Special Considerations

5.1. Network Structure

This section enumerates some special considerations related to network structure. If a network structure is not compatible with the simulator, it will fail loading the network with some error messages.

5.1.1. Convolution Layers Bias

If a convolution layer (Conv2D or Depth Wise Conv2D) uses a bias, it must be followed by a batch normalization layer, or the network would not load. This is because convolution layers cannot handle a bias directly. When they have a bias, the value of that bias is used to update the following batch normalization layer offset. The bias is absorbed there. If no batch normalization layer follows, the bias could not be considered.

5.1.2. Unsupported Layer Types

The simulator does not load a network that has an unsupported layer type in it.

5.1.3. Multiple Input Layers

The simulator supports multiple input layers, but this feature is experimental. It does support multiple output layers.

5.1.4. General Network Structure

The simulator will reject a network if it has structural issues such as but not limited to incompatible dimensions from layer to layer and wrong children count for certain layers.

5.2. Quantization File

5.2.1. Absence of a Quantization File

The simulator needs a quantization file to load networks. You do not need to create this file. It is distributed with the simulator and its path can be obtained by calling the Python function

```
get_default_quant_file()
```

This file describes the fixed-point number representation used by the various supported layer types.

5.3. Specific Layer Type Considerations

5.3.1. Dense Layer

The simulator currently supports stacked dense layers, but this feature is still under development and considered experimental. While it is functional, you should be aware that there may be limitations or potential issues associated with its use.

5.3.2. Multiply Layer

For an input of dimension (H, W, C), the multiply layer supports a second operand of dimension (H, W, C) or, with stacking, (H, W, 1). Any other configuration of input dimensions will result in the network not loading.

5.4. Input and Output

5.4.1. Input Dimensions

The simulator takes its input in the NHWC format for the *Run* function, see section 4.1.1.2, or HWC format for *run_one_frame*, see section 4.1.1.5

5.4.2. Output Dimensions

The simulator will return its output as a 3D tensor (numpy array) in the HWC format always. This is the case even if the expected output is a vector and not an image. For example, for a column vector of size 4 and 1 channel, the simulator will return an output with dimensions [4, 1, 1]. For a row vector, it would be [1, 4, 1].

5.4.3. Output Format

The format of the output is *float32*. This means that when comparing with output obtained from a FPGA board, FPGA results should be typecasted to *float32* for values to match.

Reference

- Lattice sensAI Stack web page
- Lattice Insights for Lattice Semiconductor training courses and learning plans

Technical Support Assistance

Submit a technical support case through www.latticesemi.com/techsupport.

For frequently asked questions, refer to the Lattice Answer Database at www.latticesemi.com/Support/AnswerDatabase.

Revision History

Revision 1.0, November 2024

Section	Change Summary
All	Initial release.

www.latticesemi.com