Vitis AI RNN

User Guide

UG1563 (v1.4.1) December 3, 2021

Xilinx is creating an environment where employees, customers, and partners feel welcome and included. To that end, we're removing non-inclusive language from our products and related collateral. We've launched an internal initiative to remove language that could exclude people or reinforce historical biases, including terms embedded in our software and IPs. You may still find examples of non-inclusive language in our older products as we work to make these changes and align with evolving industry standards. Follow this link for more information.

Revision History

The following table shows the revision history for this document.

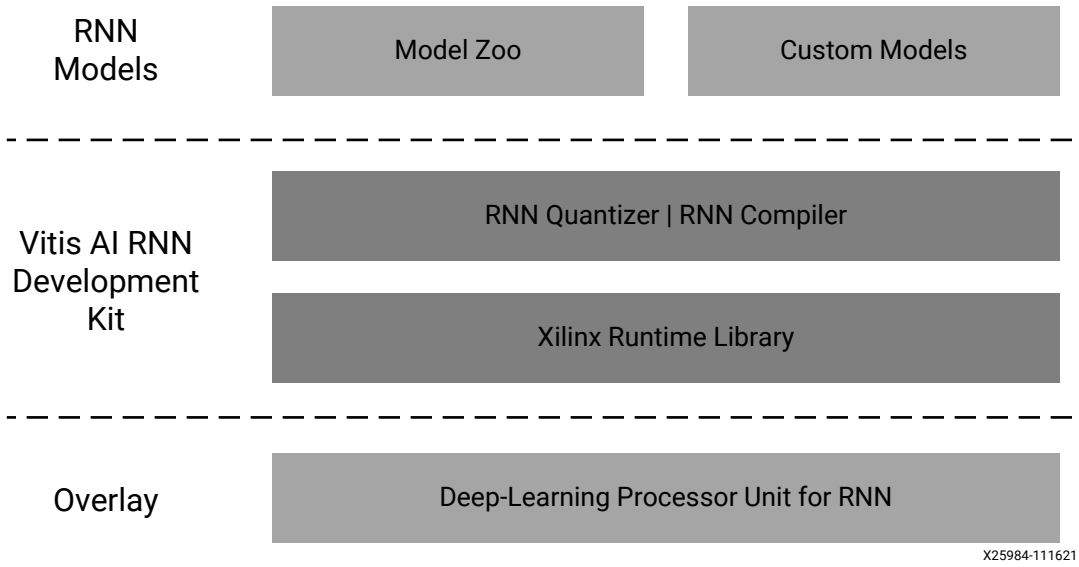
Section	Revision Summary
12/03/2021 Version 1.4.1	
Initial release	N/A

Table of Contents

Overview

The Vitis™ AI recurrent neural network (RNN) tools are a sub-module of the Vitis AI development environment focusing on implementing RNN on Xilinx® hardware platforms, including the Alveo™ accelerator cards. The tools consist of optimized IP cores, tools, libraries, models, and example designs. They are designed with high efficiency and ease-of-use in mind to unleash the full potential of AI acceleration on Xilinx FPGAs and on Adaptive Compute Acceleration Platforms (ACAPs). The RNN tools make it easy to develop RNN inference applications by abstracting the intricacies of the underlying FPGA and ACAP.

Figure 1: **Vitis AI RNN stack**



RNN Tools Overview

Deep-Learning Processor Unit for RNN

Recurrent Neural Networks (RNNs) can process sequential data of variable length and have been widely used in natural language processing, speech synthesis and recognition, and financial time series forecasting. However, RNNs are very compute-intensive and have to process input frame by frame because of their strict sequential dependency. Traditional hardware cannot achieve the ideal number in latency, especially for financial data processing, in which latency is one of the most important factors for customers.

The deep-learning processor unit (DPU) for RNN is a customized accelerator built on FPGA or ACAP devices to achieve acceleration for the RNN. It can support different types of recurrent neural networks, including RNN, gate recurrent unit (GRU), long-short term memory (LSTM), Bi-directional LSTM, and their variants. The DPU for RNN has been deployed on the Alveo U25 and the U50LV data center accelerator cards and the Versal® VCK5000 development card. The following table summarizes the features of these three RNN accelerators:

Table 1: **DPU Features for RNN on Alveo U25, U50LV Cards, and Versal VCK5000 Development Card**

Feature	DPURADR16L (U25)	DPURAHR16L (U50LV)	DPURVDRML (VCK5000)
Precision	int16	int16	Mix: int8 for GEMM on AI Engine, int16 for others
Operation Type	Matrix-Vector multiplication, element-wise multiplication and addition, sigmoid and Tanh		GEMM, Element-wise multiplication and addition, Sigmoid and Tanh, Relu, Max, Embedding (in RNN-T)
Multiplication Unit	One 32x32 Systolic Array	Seven 16x32 Systolic Arrays	40 AI Engine cores
Frequency	Freq_DSP = Freq_PL = 310 MHz	Freq_DSP = 540 MHz, Freq_PL = 270 MHz	Freq_AIE = 1.25 GHz, Freq_PL = 300 MHz
Resource Utilization	LUTs: 187,509 (35.9%) Regs: 303670 (29.0%) Block RAM: 659 (67.0%) URAM: 56 (43.8%) DSPs: 1092 (55.5%)	LUTs: 488,679 (56.1%) Regs: 1045016 (60.0%) Block RAM: 796 (59.2%) URAM: 512 (80%) DSPs: 4148 (69.7%)	LUTs: 169,163 (18.8%) Regs: 241657 (13.4%) Block RAM: 197 (20.4%) URAM: 332 (71.7%) DSPs: 82 (4.2%) AI Engine: 40 (10.0%)
Example Models	IMDB Sentiment Detection, Customer Satisfaction, Open Information Extraction		RNN-T
Quantization	RNN Quantizer v1.4.1		Manually

Table 1: **DPU Features for RNN on Alveo U25, U50LV Cards, and Versal VCK5000 Development Card** *(cont'd)*

Feature	DPURADR16L (U25)	DPURAHR16L (U50LV)	DPURVDRML (VCK5000)
Compilation	RNN Compiler v1.4.1		Manually

Notes:

1. In the DPURVDRML, UltraRAM resources are mainly used as on-chip memory for weights. If multiple 40-core AI Engine kernels could be instantiated, this memory is shared.
2. The embedding module is customized to support the RNN-T network. This module does not support any embedding table update (size, contents).
3. Quantization and compilation for the RNN-T model are completed manually. The tools are not ready now.

DPURADR16L (Alveo U25 Card)

The Xilinx® DPURADR16L IP is a programmable engine optimized for recurrent neural networks, mainly for low latency applications. This IP is implemented on the Alveo U25 card with a single thread configuration.

The design is composed of Scheduler, Load, and Save modules for data movement between the off-chip memory and on-chip caches. It also includes a 32x32 systolic array of DSPs to perform Matrix-Vector multiplications and some other computation modules for miscellaneous operations, such as element-wise multiplication and addition and non-linear function. The scheduler is responsible for instructions fetching from the off-chip memory and distributing them to different computation units according to dependency constraints. Figure 1: Vitis AI RNN stack and Figure 2: DPURADR16L Architecture show the architecture of the kernel and the systolic array module.

Figure 2: **DPURADR16L Architecture**

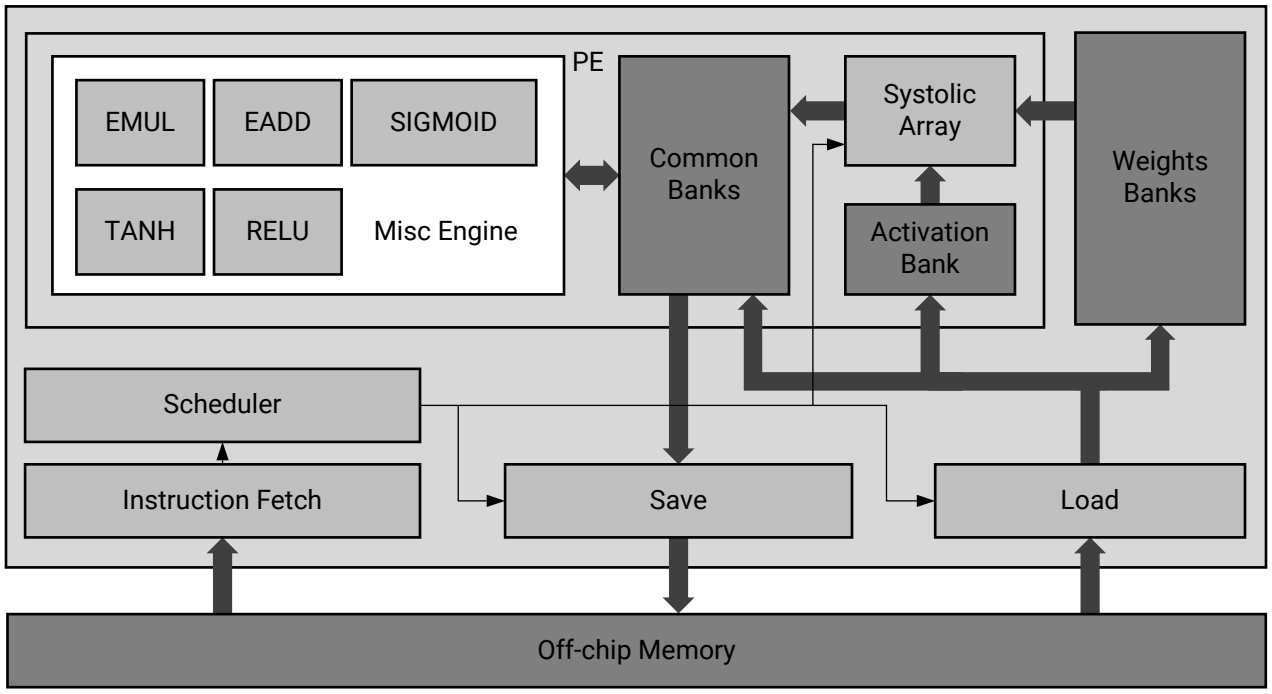
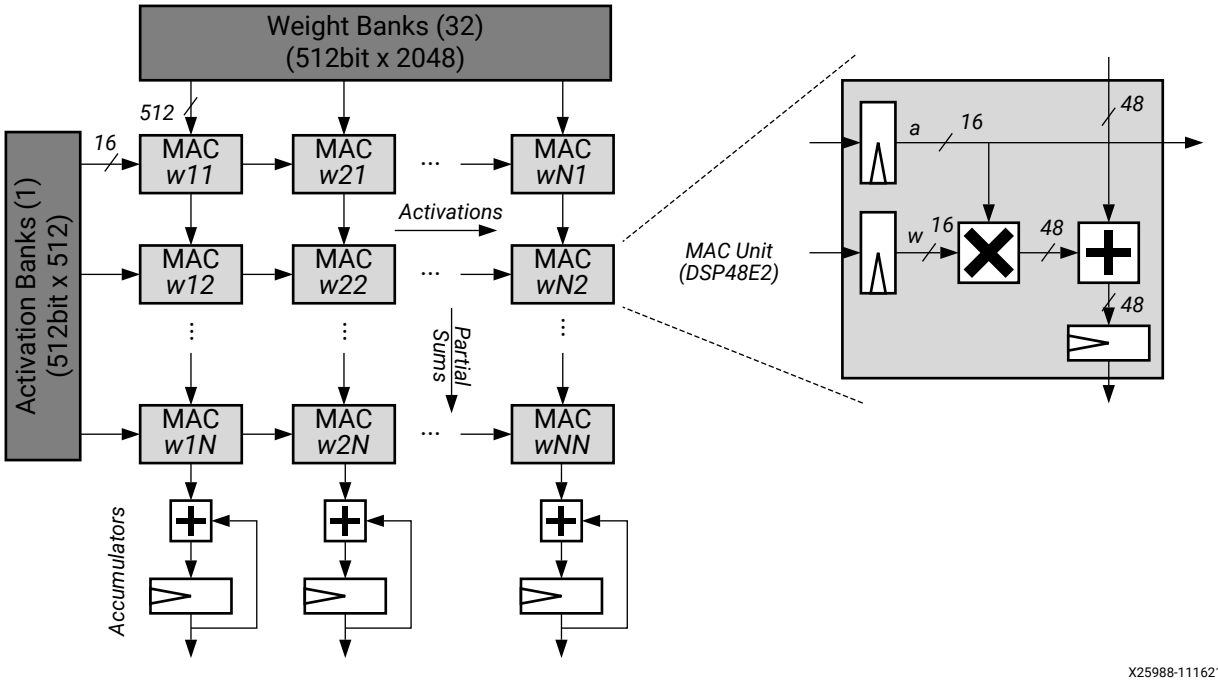


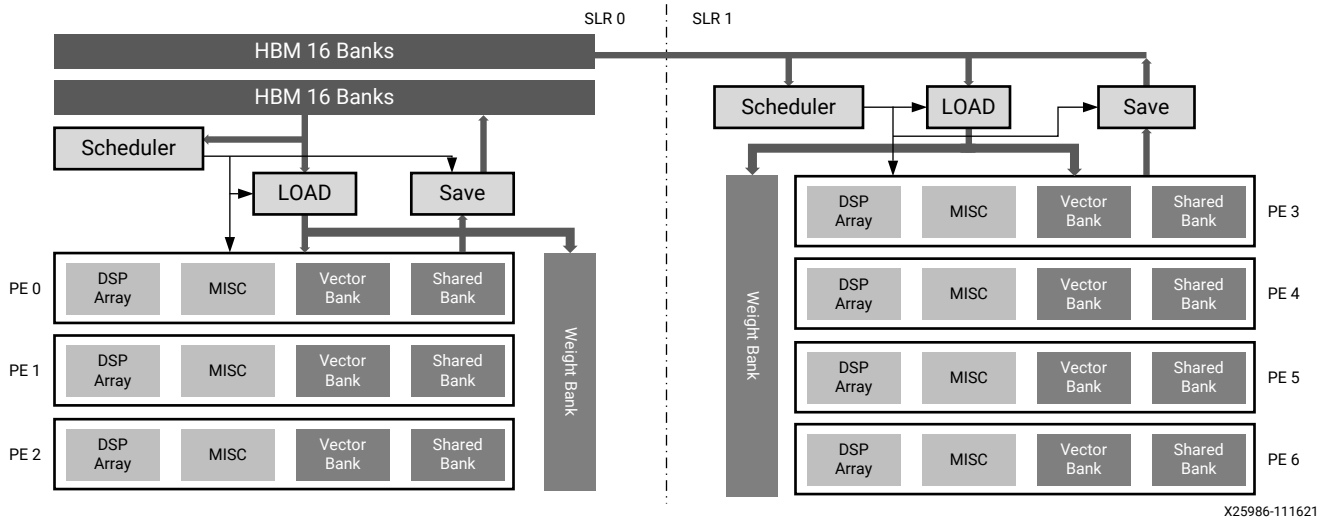
Figure 3: **Systolic Array Architecture**



DPURAHR16L (Alveo U50LV Card)

The DPURAHR16L is the design optimized for the Alveo U50LV data center accelerator card to utilize the high bandwidth of the HBMs. The DSP arrays are running at a double frequency of programmable logic, thus making the 16x32 systolic array on U50LV achieve a similar computation capacity to the U25 version but save half of DSP resources. Batches of seven inputs are supported on the Alveo U50LV card design as shown in the following figure.

Figure 4: **DPURAHR16L Top-Level Block Diagram**



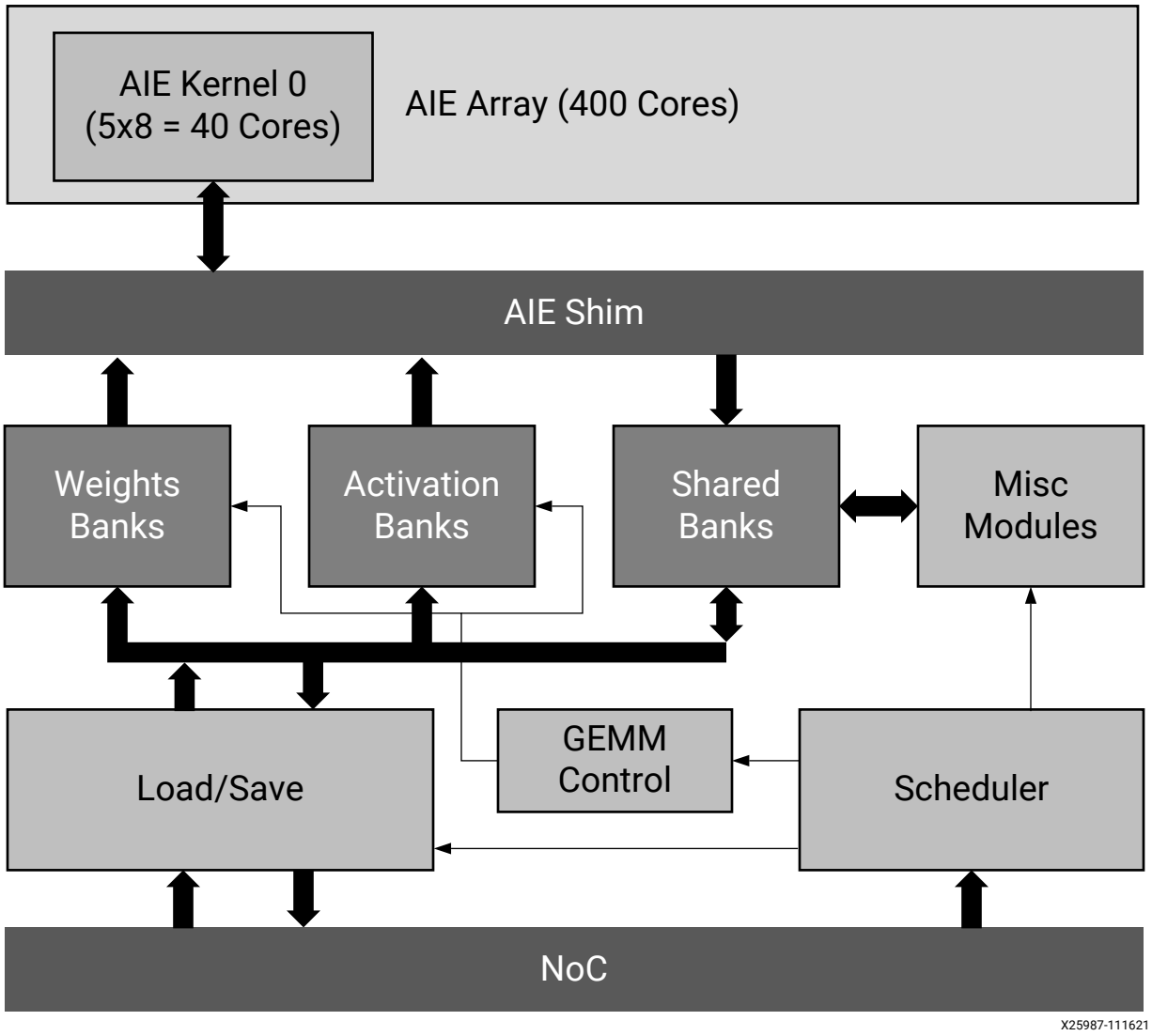
DPURVDRML (Versal VCK5000 Development Card)

The DPURVDRML is a high-performance general RNN processing engine optimized for the Versal ACAP AI Core series. Versal devices can provide superior performance or wattage over conventional FPGAs, CPUs, and GPUs. The DPURVDRML is comprised of AI Engines and PL.

The GEMM operation with a precision of int8 is deployed on the 5x8 AI Engine array. Each AI Core performs matrix-matrix multiplication of size 32x64x32. The 40-core kernel calculates GEMM of size 32x320x256. The output of GEMM is quantized to int16.

The Misc modules in the following image are composed of different modules to support different types of operations, including element-wise multiplication, addition, sigmoid, tanh, and max. Intermediate data should be represented with the precision of int16.

Figure 5: **DPURVDRML Architecture**



Register Space

The DPURADR16L on the Alveo U25 card and the DPURAHR16L on the Alveo U50LV card share the same register map. The DPURVDRML on the VCK5000 development card introduces two more registers, input_batch_stride and output_batch_stride, to describe the batching information about input and output. These kernels implement register space in the programmable logic. These registers are accessible through the AXI4-Lite interface. The following tables show these registers.

AP_CTRL

The ap_ctrl register controls and provides the kernel status. The host uses it to control the start and acknowledge the end of the kernel process. The host writes a 1 in `ap_start` and waits for both `ap_start` to be deasserted (guaranteeing the input data is fully processed) and `ap_done` to be asserted (guaranteeing the output data is fully produced). Definitions of the control register bits are listed in the following table. The address is `0x0`. The kernel can and should only be restarted after it finishes the current workload and is in the idle state.

Bit	Register	Address	Type	Description
0	ap_start	0x0	W	Asserted by host when kernel can start processing data. Cleared by kernel on handshake with ap_ready.
1	ap_done		R	Asserted by kernel when it has finished producing output data. Cleared on read by host.
2	ap_idle		R	Asserted by kernel when it is idle (deprecated).
3	ap_ready		R	Asserted by kernel when it has finished processing input data. Self-cleared immediately.
6:4	reserved			Reserved
7	auto_restart		R/W	If asserted, ap_start is held asserted by the kernel. Read/write access by the host. (Not used)
31:8	reserved			Reserved

SOFT_RESET

The soft_reset register controls the reset of the kernel. Asserting 1 to the LSB of the register resets the entire kernel logic. The reset signal is active-High. The details of the soft_reset register are shown in the following table.

Bit	Register	Address	R/W	Description
0	SOFT_RESET	0x14	W	Soft reset signal, active-High.

CONF_FRAME_NUM

The conf_frame_num register indicates the number of input frames. The host configures it. The details of the conf_frame_num register are shown in the following table.

Bit	Register	Address	R/W	Description
31:0	CONF_FRAME_NUM	0x18	R/W	Indicates the sequence length of the input sample.

INSTR_ADDR

The instr_addr register is used to indicate the DDR memory address of instructions for the DPU for the RNN kernel. The default width of the DDR memory address is 64-bit, so two 32-bit registers are used to represent the instruction address. The details of the instr_addr register are shown in the following table.

Bit	Register	Address	R/W	Description
31:0	INSTR_ADDR_H	0x1C	R/W	The higher 32 bits of instructions address in DDR. It is shared with model parameter address register.
31:0	INSTR_ADDR_L	0x20	R/W	The lower 32 bits of instructions address in DDR.

MODEL_ADDR

The register model_addr is used to indicate the DDR address of the model parameters. The higher 32-bit address register has the same address 0x1C as register INSTR_ADDR_H.

Bit	Register	Address	R/W	Description
31:0	MODEL_ADDR_H	0x1C	R/W	The higher 32 bits of model parameters address in DDR. It is shared with the high instructions address register.
31:0	MODEL_ADDR_L	0x24	R/W	The lower 32 bits of model parameters address in DDR.

INPUT_ADDR

The register input_addr is used to indicate the DDR address of input activations. The details of the input_addr register are shown in the following table.

Bit	Register	Address	R/W	Description
31:0	INPUT_ADDR_H	0x28	R/W	The higher 32 bits of input address in DDR. It is shared with the high output address register.
31:0	INPUT_ADDR_L	0x2C	R/W	The lower 32 bits of input address in DDR.

OUTPUT_ADDR

The register output_addr is used to indicate the DDR address of output result. The higher 32-bit address register has the same address 0x28 as register INPUT_ADDR_H. The details of output_adddr register are shown in the following table.

Bit	Register	Address	R/W	Description
31:0	INPUT_ADDR_H	0x28	R/W	The higher 32 bits of input address in DDR. It is shared with the high input address register.
31:0	OUTPUT_ADDR_L	0x30	R/W	The lower 32 bits of output address in DDR.

INPUT_BATCH_STRIDE

The input_batch_stride register is only used by the DPURVDRML on the VCK5000 Versal development card. It describes the address step between two adjacent batches in DDR.

Bit	Register	Address	Type	Description
31:0	INPUT_BATCH_STRIDE	0x64	R/W	The address step between two adjacent batches in DDR.

OUTPUT_BATCH_STRIDE

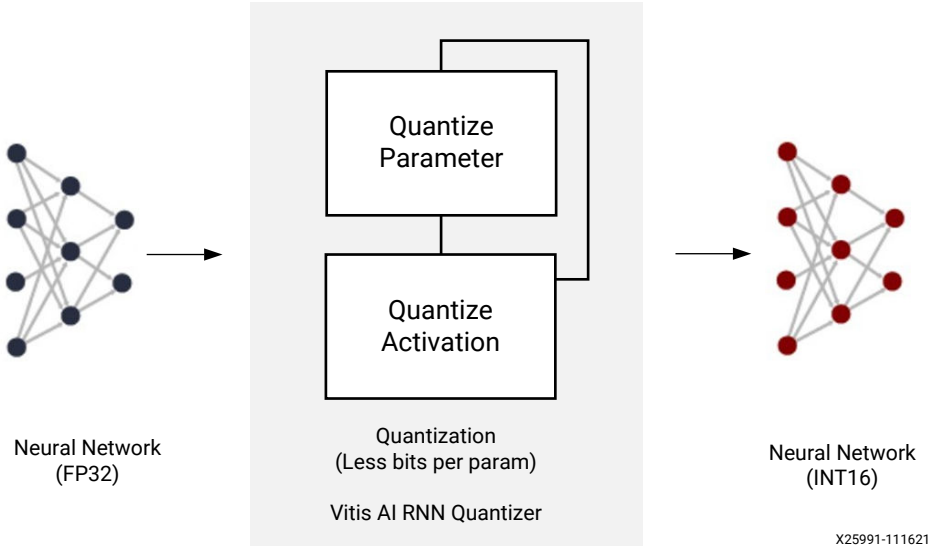
The output_batch_stride register is only used by the DPURVDRML on the VCK5000 Versal development card. It describes the address step between the output of two adjacent batches stored in DDR.

Bit	Register	Address	Type	Description
31:0	OUTPUT_BATCH_STRIDE	0x68	R/W	The address step between the output of two adjacent batches in DDR.

Vitis AI RNN Quantizer

Vitis AI RNN quantizer performs fixed-point int16 quantization for model parameters and activations. Quantizer reduces the computing complexity without losing accuracy. The quantized model requires less memory bandwidth and provides faster speed and higher power efficiency than the floating-point model.

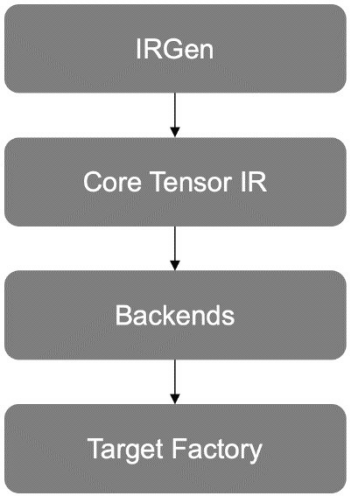
Figure 6: **Vitis AI RNN Quantizer**



Vitis AI RNN Compiler

The new RNN compiler is a modularized compiler implemented using C++ that provides better performance and user experience. It also has better abstractions with great flexibility to support more backends. It uses a unified core tensor IR to bridge the gap between the LSTM model and our diverse IPs, enabling more optimizations with less overhead, such as DDR/Bank memory planning and instruction scheduling.

Figure 7: **Vitis AI RNN Compiler**



Getting Started

Installation and Quick Start

Build Vitis AI RNN Development Kit

The Vitis™ AI RNN is made available through the docker. If the docker is not installed on your machine, refer to the Vitis AI homepage for building this docker. The docker contains RNN quantizer, RNN compiler, and RNN runtime for cloud DPU, supporting Alveo™ U25 and U50 boards.

Vitis AI RNN docker container needs to be built from the recipe. Vitis AI RNN docker recipe depends on the Vitis AI GPU docker (xilinx/vitis-ai-gpu:latest) base docker image. Before building Vitis AI RNN docker, please follow the instructions here and build the Vitis AI GPU docker first using `./docker_build_gpu.sh`.

Run Vitis AI RNN Tool and Example

Follow these steps to run the docker image, quantize and compile the model, and process the final inference on board.

1. Run the docker image.

   ```
   ./docker_run_rnn.sh -g xilinx/vitis-ai-rnn:latest
   ```

2. In the docker image, activate the vitis-ai-rnn conda environment. The RNN quantizer is already installed there.

   ```
   conda activate vitis-ai-rnn
   ```

3. Copy the example to the docker environment and run the following relevant steps:
   - For Pytorch:
     1. Copy `example/lstm_quant_pytorch` to the docker environment. The contents in the working directory look like this.

        ```
        pretrained.pth: pretrained model for sentiment detection.
        quantize_lstm.py: python script to run quantization of the model
        run_quantizer.sh: test script to run python script
        ```

2. Run the test script.

```
cd example/lstm_quant_pytorch
sh run_quantizer.sh
```

If this quantization command runs successfully, two important files and one import sub-directory are generated in the output directory ./quantize_result.

```
Lstm_StandardLstmCell_layer_0_forward.py: converted format model
quant_info.json: quantization steps of tensors
xmodel: subdirectory that contain deployed model
```

- For TensorFlow:

1. Copy `example/lstm_quant_tensorflow` to the docker environment. The contents in the working directory as follows:

```
pretrained.h5: pretrained model for sentiment detection.
quantize_lstm.py: python script to run quantization of the model
run_quantizer.sh: test script to run python script
```

2. Run the test script.

```
cd example/lstm_quant_tensorflow
sh run_quantizer.sh
```

If this quantization command runs successfully, two important files and one import sub-directory are generated in the output directory ./quantize_result.

```
rnn_cell_0.py: converted format model
quant_info.json: quantization steps of tensors
xmodel: subdirectory that contain deployed model
```

4. Compile the xmodel.

Compile the xmodel for DPURADR16L(U25) using batch size = 1. It generates the instructions in the output file (`compiled_batch_1.xmodel`).

```
vai_c_rnn -x xmodel/ -t DPURADR16L -b 1 -o output
```

5. Activate pytorch runtime environment.

```
conda activate rnn-pytorch-1.4
```

6. Run model on DPURADR16L(U25). For more information, see VART.

***Note*:** Xilinx provides three prebuilt examples, Customer Satisfaction, IMDB Sentiment Analysis, and OpenIE. The steps in VART run the prebuilt xmodels. To run the xmodel you compiled, replace `data/compiled_batch_1.xmodel` with your output xmodel file after executing the `"TARGET_DEVICE=U25 source ./setup.sh"` command.

Run RNN-T Demo on Versal

This section shows one RNN-T model-based ASR solution on Xilinx® Versal® device VCK5000. Versal is the first adaptive compute acceleration platform (ACAP). It is a fully software-programmable heterogeneous compute platform that combines Scalar Engines, Adaptable Engines, and Intelligent Engines to achieve dramatic performance improvements over FPGA and CPU implementations among different applications. For more information, see Xilinx Versal website. RNN-T is a sequence-to-sequence model that continuously processes input samples and streams output symbols. The speech recognition model used here is a modified RNN-T model and belongs to the MLPerf Inference benchmark suite. For more information, see mlcommons inference repo.

The hardware kernel is a 40 AIE cores design. INT8 matrix-matrix multiplications are performed on AIE cores, and other functions are implemented in programmable logic (PL) with INT16 precision. The following table shows the total resource utilization for the kernel and platform. UltraRAM resources are mainly used as weights buffer and are shared if multiple kernels are instantiated.

Table 2: **Resources**

	CLB LUTs	Registers	Block RAM	UltraRAM	DSP Slices	AIE Cores
Available	899712	1799424	967	463	1968	400
Utilized	169163(18.8%)	241657(13.43%)	197(20.37%)	332(71.71%)	82(4.17%)	40(10%)

Note: For RNN-T model, the quantizer and compiler are not ready now. The process of mix-precision quantization and instruction generation are completed manually and produced for this demo. For more information, see DPU-for-RNN.

Quantizing RNN Models

XRNN is a customized accelerator built on FPGA to achieve RNNs acceleration. It can support different RNN networks, including RNN, GRU, LSTM, and bi-directional LSTM. The quantizer performs fixed-point int16 quantization for model parameters and activations. The compiler generates instructions based on the target network structure and XRNN hardware architecture.

Note: Currently, RNN toolchain only supports LSTM and OpenIE models, and the generated instructions can only be deployed in the Alveo™ U25 and U50 cards.

Running the Toolchain

Vitis™ AI RNN tool provides the simplest APIs to introduce the FPGA-friendly quantization feature. For a well-defined model, you only need to add a few lines to get a quantized model object and compile the quantized model to instructions.

Adding Quantizer APIs to Float Models

Suppose there is a trained float model and some Python scripts to evaluate the model's accuracy before quantization. The Quantizer API replaces the float module with a quantized module. The normal evaluate function encourages quantized module forwarding. Quantize calibration determines quantization steps of tensors in the evaluation process if quant_mode flag is set to calib. After calibration, evaluate the quantized model by setting quant_mode to test.

PyTorch

The `example/lstm_quant_pytorch/quantize_lstm.py` file contains an example.

1. Import the PyTorch quantizer module.

```
from pytorch_nndct.apis import torch_quantizer
```

2. Generate a quantizer with quantization and get the converted model.

```
quantizer = torch_quantizer(quant_mode=args.quant_mode,
                            module=model,
                            bitwidth=16,
                            lstm=True)
model = quantizer.quant_model
```

3. Forward a neural network with the converted model.

```
acc = test(model, DEVICE, test_loader)
```

4. Output the quantization result and deploy the model.

```
if args.quant_mode == 'calib':
    quantizer.export_quant_config()
if args.quant_mode == 'test':
    quantizer.export_xmodel(deploy_check=True)
```

TensorFlow

The `example/lstm_quant_tensorflow/quantize_lstm.py` file contains an example.

1. Import the TensorFlow quantizer module.

```
from tf_nndct.quantization.api import tf_quantizer
```

2. Generate a quantizer with quantization needed input, and the batch size of input data must be 1. Get the converted model.

```
single_batch_data = X_test[:1, ]
input_signature = tf.TensorSpec(single_batch_data.shape[1:], tf.int32)
quantizer = tf_quantizer(model,
                         input_signature,
                         quant_mode=args.quant_mode,
                         bitwidth=16)
rebuilt_model = quantizer.quant_model
```

3. Forward a neural network with the converted model.

```
output = rebuilt_model(X_test[start: end])
```

4. Output the quantization result and deploy the model. When dumping the outputs, the batch size of the input data must be 1.

```
if args.quant_mode == 'calib':
    quantizer.export_quant_config()
elif args.quant_mode == 'test':
    quantizer.dump_xmodel()
    quantizer.dump_rnn_outputs_by_timestep(X_test[:1])
```

Running Quantization and Getting the Result

Take the PyTorch version as an example.

1. Run the following command with `--quant_mode calib` to quantize the model.

```
python quant_lstm.py --quant_mode calib --subset_len 1000
```

When calibrating forward, borrow the float evaluation flow to minimize code change from the float script. If there are loss and accuracy messages displayed in the end, ignore them.

Note: Check the colorful log messages with the special keyword, "NNDCT." If this quantization command runs successfully, two important files are generated in the `./quantize_result` output directory.

- **Lstm_StandardLstmCell_layer_0_forward.py:** Converted format model
- **quant_info.json:** Quantization steps for tensors. (Keep it to evaluate the quantized model.)

2. To evaluate the quantized model, run the following command:

```
python quant_lstm.py --quant_mode test --subset_len 1000
```

3. The accuracy displayed after the command executes successfully is the right accuracy for the quantized model. The Xmodel file for the compiler is generated in the output directory, `./quantize_result/xmodel`.

```
Lstm_StandardLstmCell_layer_0_forward_int.xmodel: deployed model
```

Quantizer APIs

PyTorch APIs

The APIs are located in the `nndct/pytorch_binding/pytorch_nndct/apis.py` module.

```
class torch_quantizer():
    def __init__(self,
                 quant_mode: str, # ['calib', 'test']
                 module: torch.nn.Module,
                 input_args: Union[torch.Tensor, Sequence[Any]] = None,
                 state_dict_file: Optional[str] = None,
                 output_dir: str = "quantize_result",
                 bitwidth: int = 8,
                 mix_bit: bool = False,
                 device: torch.device = torch.device("cuda"),
                 lstm: bool = False,
                 app_deploy: str = "CV",
                 qat_proc: bool = False)
```

Class `torch_quantizer` creates a quantizer object.

Arguments

- **quant_mode:** A string that indicates which quantization mode used by the process:
  - calib: for calibration of quantization
  - test: for evaluation of the quantized model
- **module:** Float module to be quantized.

- **state_dict_file:** Float module pre-trained parameters file. If the float module has read parameters, there is no need to set the parameter.
- **output_dir:** Directory for quantization result and intermediate files. quantize_result is the default.
- **bitwidth:** Global quantization bit width. 8 is the default.
- **device:** Run model on GPU or CPU.
- **lstm:** Flag to control whether this is an LSTM model. False is the default.
- **qat_proc:** Turn on quantization-aware-training (QAT).

API Functions

- Get the quantized model.

```
@property
  def quant_model(self)
```

- Export quantization steps information for tensors to be quantized.

```
def export_quant_config(self)
```

- Export XMODEL files for compilation.

```
def export_xmodel(self, output_dir="quantize_result", deploy_check=False)
```

  - **output_dir:** Directory to save the XMODEL files. quantize_result is the default.
  - **deploy_check:** Flag to control whether to deploy simulation data.

TensorFlow APIs

The APIs are located in the `nndct/tensorflow/tf_nndct/quantization/api.py` and `quantizer.py` modules.

```
def tf_quantizer(model,
                 input_signature,
                 quant_mode: int = 0,
                 output_dir: str = "quantize_result",
                 bitwidth: int = 8)
```

Quantizes the LSTM model.

Arguments

- **model:** Float module to be quantized.
- **input_signature:** Input tensor with the same shape as real input of float module to be quantized, but the values can be random numbers.

- **quant_mode:** A string that indicates which quantization mode the process is using:
  - calib: for calibration of quantization
  - test: for evaluation of quantized model
- **output_dir:** Directory for quantization result and intermediate files. The default value is quantize_result.
- **bitwidth:** Global quantization bit width. The default value is 8.

API Functions

- Get the quantized model.

```
@property
  def quant_model(self)
```

- Export quantization steps information for tensors to be quantized.

```
def export_quant_config(self)
```

- Export XMODEL files for compilation.

```
def dump_xmodel(self)
```

- Deploy simulation data of the quantized model. `Input` is the input data that feed in the quantized model.

```
def dump_rnn_outputs_by_timestep(self, inputs)
```

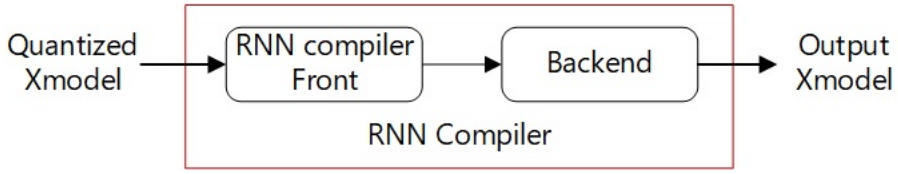
Vitis RNN Compiler

As part of the Xilinx® Vitis™ AI ecosystem, the RNN compiler compiles quantized RNN models and maps these models to highly optimized instruction sequences for RNN DPU IPs. Currently, the RNN compiler only supports standard LSTM models.

RNN Compilation Flow

RNN compilation flow uses XMODEL format as the unified interface for the quantizer versus the compiler and the compiler versus run time. The RNN compiler accepts the quantized XMODEL(s) as input and generates another XMODEL output as the RNN run-time input when compilation is complete. The core components of the compiler are the RNN compiler frontend and backend:

Figure 8: **RNN Compilation Flow**

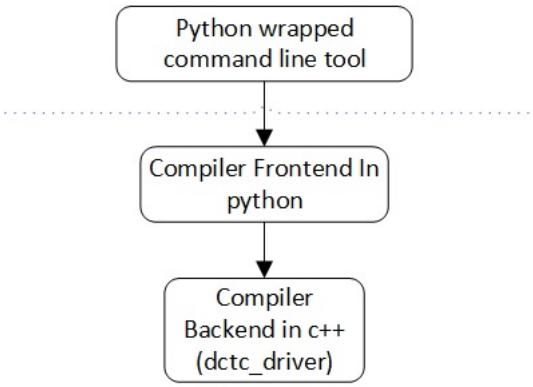


- **Frontend:** The frontend parses the XMODEL files into JSON files for backend usage.
- **Backend:** The backend parses the JSON files into Tensor immediate representation and performs multiple target-specific optimizations, including off-chip and on-chip memory planning for aggressive memory reuse and efficient instruction scheduling to achieve better parallelism. The backend also generates hardware instruction and generates an XMODEL as ouput. The generated XMODEL contains all the necessary metadata and hardware instructions. The RNN runtime uses it for on-board inference purposes.

RNN Compiler Usage

VAI_C_RNN Command Tool

Figure 9: **RNN Compiler Stack**



Because the front-end and back-end of the RNN compilers are implemented as independent components using different programming languages, a Python wrapped command-line tool named `vai_c_rnn` is provided to enable a unified user interface for the RNN compiler. The wrapper invokes the front-end and back-end independently for the whole compilation flow, and so, the internal workings of the compiler implementation are invisible to the end-user.

VAI_C_RNN Options

Table 3: **Command Options**

Parameters	Description	Notes
`-x, --xmodel_path`	Quantized XMODEL input path	The quantization tool may output multiple XMODEL files, so for this option, only the parent path for these XMODEL files should be provided.
`-t, --target`	Target device name: DPURADR16L, DPURAHR16L	DPURADR16L (U25) DPURAHR16L (U50LV)
`-b, --batch`	Batch size, supported range is1-4	For DPURADR16L (U25), only batch =1 is supported, while for DPURAHR16L (U50LV), only batch=3 and batch=4 is supported.
`-o, --output_dir`	Output directory of the XMODEL	
`-v`	Version	

Supported Targets and Operators

For details about the supported targets and operators in RNN compiler, refer to the following tables.

Table 4: **RNN DPU Targets**

	DPURADR16L (U25)	DPURAHR16L (U50LV)
Supported batch size	Batch1	Batch3, batch4
Maximum batch size	Batch=1	Batch=7

DPURADR16L (U25) only supports batch=1, DPURAHR16L (U50LV) can support batch 3 and 4 at the same time.

Table 5: **Operators Support by the RNN DPU**

Operator Type	Description
Matmul	Matrix-vector multiplication
Mul	Multiplication; only two operands are supported
Add/Sub	Addition and subtraction; only two operands are supported
Tanh/Sigmoid	Activation operators; only one operand is support

Compilation Example

To compile a sentiment RNN model for an Alveo™ U50 data center accelerator card with batch=4 engine, follow these steps:

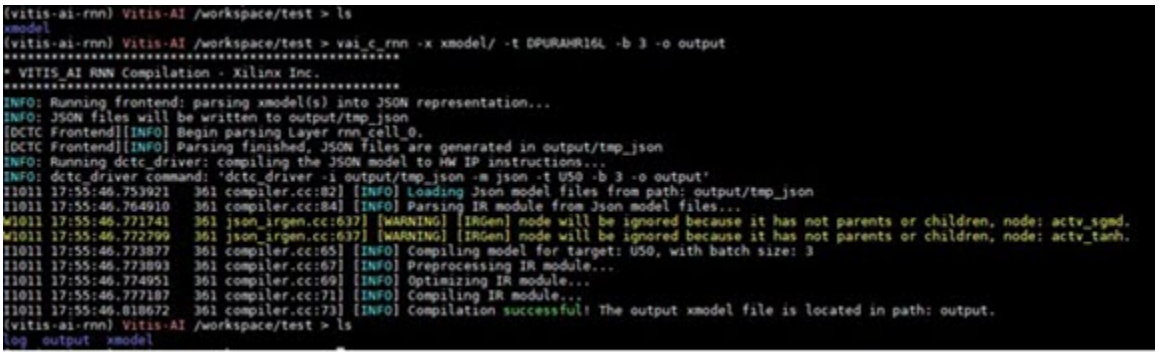
1. Activate the vai_c_rnn conda environment in the docker image:

   ```
   conda activate vitis-ai-rnn
   ```

2. Compile

   Assuming the quantized models for sentiment are prepared in the XMODEL directory, run the compilation command and get results. The output XMODEL is present in the output directory.

Figure 10: **Compile XMODEL**



Deploying and Running Models

XRNN Runtime for DPURADR16L (U25) and DPURAHR16L (U50LV)

XRNN Runtime provides a unified interface API for both DPURADR16L and DPURAHR16L. It inherits from the interface used by other DPUs in Vitis™ AI. This includes both C++ and Python API.

The API documentation can be found here. Even though the interface API is the same as that of other DPUs, some crucial differences are mentioned in the following sections.

Input and Output Tensor Shape

XRNN Runtime expects tensors with the shape = BxNxW, where:

- B is the batch size
- N is the number of frames
- W is the number of 16-bit words in each frame aligned to the tile array of the IP

You can access this information by querying `runner->get_input_tensors()` and `runner->get_output_tensors()`.

However, in RNN applications, the number of frames in each input can vary. So, only the B and W obtained from the tensor shape are useful to the user while N should be based on the actual input or output. You must create input and output tensors with aligned dimensions.

Input and Output Tensor Alignment

If the frame size does not match the aligned size (W), you need to zero-pad each frame. Similarly, the output dimension also has an aligned frame size. You can extract the required data if the actual frame size is less than the aligned size.

Input and Output Tensor Datatype

Input and output datatype should be INT16. FLOAT32 data is not supported currently. So, you need to quantize or dequantize the data accordingly. The shift factors can be accessed from the XMODEL using XIR API. The following is a Python example:

```
graph = xir.Graph.deserialize("rnn.xmodel")
in_pos = graph.get_root_subgraph().get_attr('input_float2fix')
out_pos = graph.get_root_subgraph().get_attr('output_fix2float')
```

Creating Runners in DPURAHR16L

DPURAHR16L IP on the Alveo™ U50 card has two CUs. One CU can process a batch-3 input, while the next CU can process a batch-4 input. These two CUs can work in parallel.

The XRNN compiler generates two different XMODELS for them. Runners created with batch-3 XMODEL are assigned to batch-3 CU only, and runners created with batch-4 XMODEL are assigned to batch-4 CU only. So, to utilize both the CUs, you need to create a runner with each XMODEL.

While passing the input, the batch size should match the batch size supported by the corresponding runner. The batch size of a runner can be accessed from the shape of the tensor returned by `runner->get_input_tensors()`.

This section describes the important parts of the customer satisfaction application in Python running on DPURAHR16L. The complete code can be accessed from `Vitis-AI/demo/rnn_u25_u50lv/apps/customer_satisfaction/run_dpu_e2e.py`.

1. Import required modules using the following command:

   ```
   import vart
   import xir
   ```

2. Load the model on CUs.

   There are two available CUs. The first CU processes batch-3 input while the second CU processes batch-4 input. To utilize both CUs, create two runners, each one with a corresponding XMODEL.

   ```
   runners = []
   models = ["compiled_batch_3.xmodel", "compiled_batch_4.xmodel"]
   for i in range(len(models)):
   graph = xir.Graph.deserialize(models[i])
       runners.append(vart.Runner.create_runner(
                   graph.get_root_subgraph(), "run"))
   ```

3. Quantize the input data using the following command:

```
in_pos = graph.get_root_subgraph().get_attr('input_float2fix')
quantized_lstm_input = quanti_convert_float_to_int16(
        lstm_input.reshape(num_records * 25*32), in_pos)
    .reshape((num_records, 25*32))
```

4. Start the execution. The input data is fed into two runners in an alternating manner. The dimensions for input and output can be accessed from a runner, like batch size and aligned dimensions for input or output. Allocate the output array for `execute_async()` beforehand.

```
lstm_output = np.zeros((num_records, 25*100), dtype=np.int16)
i = 0
num_cores = 2
while count < len(quantized_input):
    inputTensors = runners[i].get_input_tensors()
    outputTensors = runners[i].get_output_tensors()
batch_size, num_frames, runner_in_seq_len = tuple(inputTensors[0].dims)
_, _, runner_out_seq_len = tuple(outputTensors[0].dims)

    input_data = quantized_input[count:count+batch_size]
    batch_size = input_data.shape[0]
    input_data = input_data.reshape(batch_size,
num_sequences,                                runner_in_seq_len)
    output_data = np.empty((batch_size, num_sequences,
runner_out_seq_len),                  dtype=np.int16)
    job_id = runners[i].execute_async([input_data],
[output_data],                                 True)
    runners[i].wait(job_id)
    out_np[count:count+batch_size, ...] =
output_data[..., :output_seq_dim]
            .reshape(batch_size, num_sequences*output_seq_dim)
    count += batch_size
    i = (i + 1) % num_cores
```

   To run both the CUs in parallel, invoke the `execute_async()` call in two different threads. Refer `Vitis-AI/demo/ rnn_u25_u50lv/apps/customer_satisfaction/run_dpu_e2e_mt.py` for example.

5. Dequantize the output using the following command:

```
out_pos = graph.get_root_subgraph().get_attr('output_fix2float')
lstm_output = quanti_convert_int16_to_float(lstm_output, out_pos)
```

XILINX

Appendix A

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado® IDE, select **Help → Documentation and Tutorials**.
- On Windows, select **Start → All Programs → Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

Note: For more information on DocNav, see the Documentation Navigator page on the Xilinx website.

References

These documents provide supplemental material useful with this guide:

1. *Vitis AI User Guide* (UG1414)
2. *Vitis AI Optimizer User Guide* (UG1333)

Please Read: Important Legal Notices

Copyright