This branch is 2 commits ahead, 26 commits behind master. Pull request Compare

ErinTruax Updated for Vitis AI 1.2 ... Sep 21, 2020 3

View code

README.md

XILINX

Vitis AI Tutorials

MNIST Classification using Vitis AI and TensorFlow

Current status

1. Tested with Vitis AI 1.2.1
2. Tested in hardware on ZCU102 and Alveo U50

Introduction

This tutorial introduces the user to the Vitis AI TensorFlow design process and will illustrate how to go from a python description of the network model to running a compiled model on the Xilinx ZCU102 evaluation board.

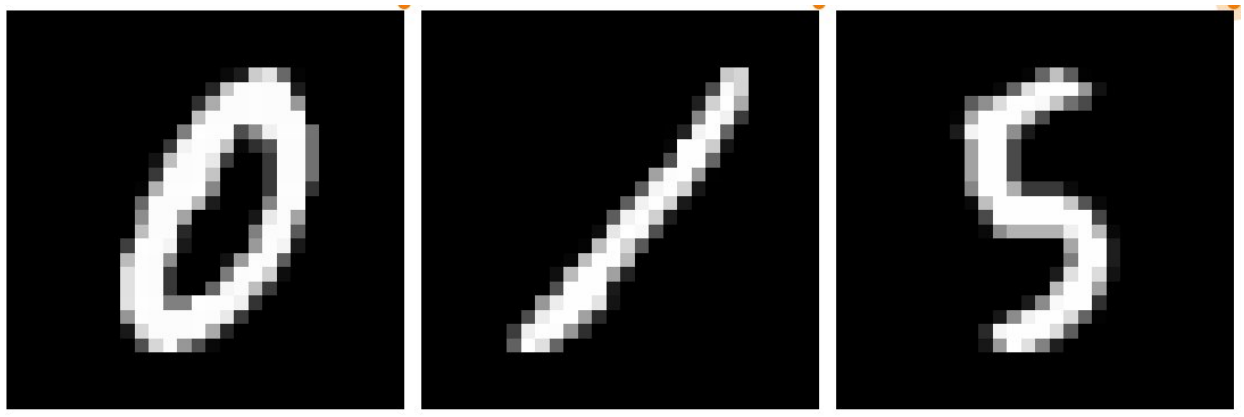
The application code in this example design is written in Python and uses the Unified APIs that were introduced in Vitis AI 1.0.

We will run the following steps:

- Training and evaluation of a small custom convolutional neural network using TensorFlow.
- Removal of the training nodes and conversion of the graph variables to constants (..often referred to as 'freezing the graph').
- Evaluation of the floating-point frozen model using the MNIST test dataset.
- Quantization of the floating-point frozen model.
- Evaluation of the quantized 8bit model using the MNIST test dataset.
- Compilation of the quantized model to create the .elf (for Zynq) and .xmodel (for Alveo) files ready for execution on the DPU accelerator IP.
- Download and run the application on the ZCU102 and/or Alveo U50 evaluation board.

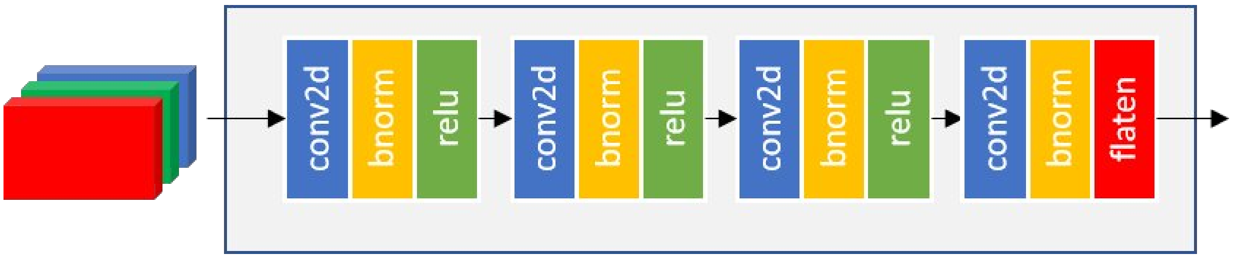
The MNIST dataset

The MNIST handwritten digits dataset is a publicly available dataset that contains a total of 70k 8bit grayscale images each of which are 28pixels x 28pixels. The complete dataset of 70k images is normally divided into 60k images for training and 10k images for validation. The dataset is considered to be the 'hello world' of machine learning and makes a simple introduction to learn the complete Xilinx Vitis-AI flow.



The convolution neural network

The convolutional neural network in this design has deliberately been kept as simple as possible and consists of just four layers of 2D convolution interspersed with batch normalization and ReLU activation. The network is described in the customcnn.py python script.



Preparing the host machine and target boards

The host machine has several requirements that need to be met before we begin. You will need:

- An x86 host machine with that meets the sytem requirements and internet access to download files.
- A GPU card suitable for training is recommended, but the training in this tutorial is quite simple and a CPU can be used.
- You should follow the host and target setup instructions provided in Quick Start for Edge and in Alveo card setup. Ignore the DPUCAHX8H Overlay Usage section as we will run that as part of this tutorial.

For more details, refer to the latest version of the *Vitis AI User Guide* (UG1414).

This tutorial assumes the user is familiar with Python3, TensorFlow and has some knowledge of machine learning principles.

Shell scripts in this tutorial

- `0_setenv.sh` : Sets all necessary environment variables used by the other scripts. Most variables can be edited by the user to configure the environment for their own requirements. Note that the image_input_fn.py Python script contains a reference to the list of calibration images as an absolute path:

```
calib_image_list = './build/quantize/images/calib_list.txt'
```

..so if either of the `BUILD` or `QUANT` variables are modified in `0_setenv.sh` then the absolute path in image_input_fn.py will also need to be modified.

It is highly recommended to leave the `CALIB_IMAGES` variable set to 1000 as this is the minimum recommended number of images for calibration of the quantization.

- `1_train.sh` : Runs training and evaluation of the network. Will save the trained model as an inference graph and floating-point checkpoint.
- `2_freeze.sh` : Converts the inference graph and checkpoint into a single binary protobuf file (.pb). The output .pb file is generally known as a 'frozen graph' since all variables are converted into constants and graph nodes associated with training such as the optimizer and loss functions are stripped out.
- `3_eval_frozen_graph.sh` : This is an optional step which tests the accuracy of the frozen graph. The accuracy results should be very similar to the results obtained after training.
- `4_quant.sh` : This script first creates a set of image files to be used in the calibration phase of quantization and then launches the `vai_q_tensorflow quantize` command to convert the floating-point frozen graph to a fixed-point integer model.
- `5_eval_quant_graph.sh` : This step is optional but highly recommended - it will run the same evaluation function that was used to evaluate the frozen graph on the quantized model. Users should confirm that the accuracy reported by the evaluation of the quantized model is sufficient for their requirements and similar to the results for the floating-point models.
- `6_compile_u50.sh` : Launches the `vai_c_tensorflow` command to compile the quantized model into an .xmodel file for the Alveo U50 accelerator card.
- `6_compile_zcu102.sh`: Launches the `vai_c_tensorflow` command to compile the quantized model into an .elf file for the ZCU102 evaluation board.
- `7_make target_u50.sh` : Copies the .xmodel and images to the `./build/target_u50` folder ready for use with the Alveo U50 accelerator card.
- `7_make target_zcu102.sh` : Copies the .elf and images to the `./build/target_zcu102` folder ready to be copied to the ZCU102 evaluation board's SD card.

Image pre-processing

All images are undergo simple pre-processing before being used for training, evaluation and quantization calibration. The images are normalized to bring all pixel values into the range 0 to 1 by dividing them by 255.

Implementation

Clone or download this GitHub repository to your local machine where you have installed the necessary tools. Open a linux terminal, cd into the repository folder then into the 'files' folder. Start the Vitis AI docker - if you have a GPU in the host system, it is recommended that you use the GPU version of the docker container, however if a GPU is not available then the CPU docker container will be sufficient:

```
# navigate to densenet tutorial folder
cd <path_to_MNIST_design>/files

# to start GPU docker
source ./start_gpu_docker.sh

# ..or to start CPU docker
source ./start_cpu_docker.sh
```

The docker container will start and you should see something like this in the terminal:

```
==========================================

__      ___ _   _                   _____
\ \    / (_) | (_)            /\   |_   _|
 \ \  / / _| |_ _ ___ ______ /  \    | |
  \ \/ / | | __| / __|______/ /\ \   | |
   \  /  | | |_| \__ \     / ____ \ _| |_
    \/   |_|\__|_|___/    /_/    \_\_____|

==========================================

Docker Image Version: latest
Build Date: Wed Apr 15 11:01:32 CEST 2020
VAI_ROOT=/opt/vitis_ai
For TensorFlow Workflows do:
  conda activate vitis-ai-tensorflow
For Caffe Workflows do:
  conda activate vitis-ai-caffe
For Neptune Workflows do:
  conda activate vitis-ai-neptune
mharvey@XITMHARVEY33:/workspace$
```

> 💡 If you get a "Permission Denied" error message when starting the Docker container, it is almost certainly because the docker_run.sh script is not set as being executable. You can fix this by running the command:
>
> ```
> $ chmod +x ./docker_run.sh
> ```

The complete tools flow can be run just by executing the `source ./run_all.sh` command, or by running each of the steps in order (from 0_xx to 7_xx):

```
$ source ./0_setenv.sh
$ source ./1_train.sh
.
.
$ source ./7_make_target_u50.sh
```

If you have run the flow for the ZCU102, ensure that you have a .elf file in the `./build/target_zcu102` folder after all steps have been completed. For the Alveo U50 flow, check that a customcnn.xmodel file exists in the `./build/target_u50` folder.

Running the application on the ZCU102 board

All of the required files for running on the ZCU102 board are copied into the `./build/target_zcu102` folder by the `7_make_target_zcu102.sh` script. The `7_make_target_zcu102.sh` script also copies the test set images to `./build/target_zcu102/images` - the application code will preprocess and classify these images. The entire target_zcu102 folder will need to be copied to the ZCU102 SDcard.

Copying the complete `./build/target_zcu102` folder to the /home/root folder of the flashed SD card can be done in one of several ways:

1. Direct copy to SD Card:

- If your host machine has an SD card slot, insert the flashed SD card and when it is recognised you will see two volumes, BOOT and ROOTFS. Navigate into the ROOTFS and then into the /home folder. Make the ./root folder writeable by issuing the command `sudo chmod -R 777 root` and then copy the entire `./build/target_zcu102` folder from the host machine into the /home/root folder of the SD card.

- Unmount both the BOOT and ROOTFS volumes from the host machine and then eject the SD Card from the host machine.

2. With scp command:

- If the ZCU102 is connected to a network and reachable by the host machine, the target folder can be copied using scp. If you connect directly from your host machine to the ZCU102 using ethernet, you may need to set up static IP addresses.

- The command will be something like `scp -r ./build/target_zcu102 root@192.168.1.228:~/.` assuming that the ZCU102 IP address is 192.168.1.228 - adjust this and the path to the target folder as appropriate for your system.
- If the password is asked for, insert 'root'.

With the target folder copied to the SD Card and the ZCU102 booted, you can issue the command for launching the application - note that this is done on the ZCU102 board, not the host machine, so it requires a connection to the ZCU102 such as a serial connection to the UART or an SSH connection via Ethernet.

The application can be started by navigating into the target_zcu102 folder (`cd target_zcu102`) and then issuing the command `python3 app_mt.py -m model_dir/dpu_customcnn.elf` . The application will start and after a few seconds will show the throughput (in frames/sec) and the accuracy:

```
root@xilinx-zcu102-2020_1:~ cd target_zcu102
root@xilinx-zcu102-2020_1:~/target_zcu102# python3 app_mt.py -m model_dir/dpu_cus
Command line options:
 --image_dir :  images
 --threads   :  1
 --model     :  model_dir/dpu_customcnn.elf
Pre-processing 10000 images...
Starting 1 threads...
FPS=2884.71, total frames = 10000 , time=3.4666 seconds
Correct: 9869 Wrong: 131 Accuracy: 0.9869
```

For better throughput, the number of threads can be increased by using the `-t` option. For example, to execute with 6 threads:

```
root@xilinx-zcu102-2020_1:~/target_zcu102# python3 app_mt.py -m model_dir/dpu_cus
Command line options:
 --image_dir :  images
 --threads   :  6
 --model     :  model_dir/dpu_customcnn.elf
Pre-processing 10000 images...
Starting 6 threads...
FPS=3940.29, total frames = 10000 , time=2.5379 seconds
Correct: 9869 Wrong: 131 Accuracy: 0.9869
```

Running the application on the Alveo U50

This final step will copy all the required files for running on the U50 into the `./build/target_u50` folder. The `7_make_target_u50.sh` script also copies the test set images to `/build/target_u50/images` - the application code will preprocess and classify these images.

The following steps should be run from inside the Vitis-AI Docker container.

Run the `U50_overlay.sh` script (internet connection required) to download and install the correct overlay on the U50 (note that the U50 will need to have been flashed with correct deployment shell - this should have been done in the 'Preparing the host machine and target boards' section above). The complete steps to run on the Alveo U50 are as follows:

```
source ./U50_overlay.sh
cd ./build/target_u50
/usr/bin/python3 app_mt.py -m model_dir/customcnn.xmodel
```

You should see something like this:

```
mharvey@XITMHARVEY33:/workspace/build/target_u50$ /usr/bin/python3 app_mt.py -m m
Command line options:
 --image_dir :  images
 --threads   :  1
 --model     :  model_dir/customcnn.xmodel
Pre-processing 10000 images...
Starting 1 threads...
FPS=8235.12, total frames = 10000 , time=1.2143 seconds
Correct: 9869 Wrong: 131 Accuracy: 0.9869
```

Similar to the ZCU102, the number of threads can be increased for higher throughput:

```
mharvey@XITMHARVEY33:/workspace/build/target_u50$ /usr/bin/python3 app_mt.py -m m
Command line options:
 --image_dir :  images
 --threads   :  6
 --model     :  model_dir/customcnn.xmodel
Pre-processing 10000 images...
Starting 6 threads...
FPS=22256.84, total frames = 10000 , time=0.4493 seconds
Correct: 9869 Wrong: 131 Accuracy: 0.9869
```

Releases

No releases published

Packages

No packages published