Xilinx Versal ACAP/SoC for (Quasi) Real-Time Data Processing

Ben Rosser Tianjia Du Timothy Hoffman David Miller

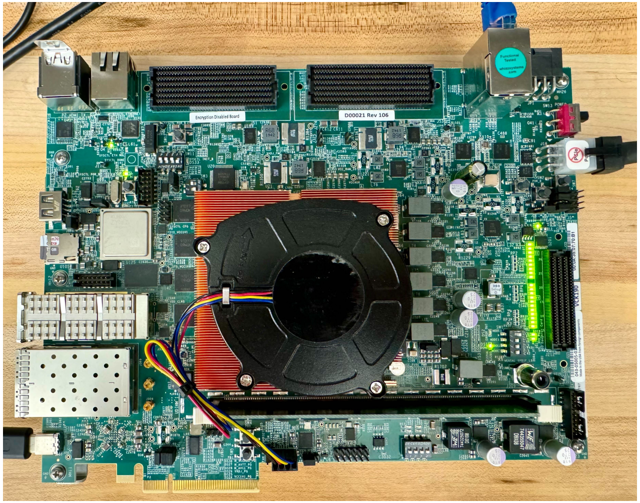
University of Chicago

October 5, 2023

Introduction

- Two disclaimers:
  - I'm **not an FPGA/SoC expert**; I'm a physics postdoc (with some digital logic experience).
  - This work is **still ongoing**: hope to show more concrete results/conclusions in the future.
- Interested in hardware acceleration of algorithms for e.g. collider **trigger systems**:
  - Want to run **high-quality** particle reconstruction algorithms, make decisions in real time.
  - Increasing demand for hardware-based machine learning; lots of community interest.
  - Approaches using **high level synthesis** (e.g. hls4ml): very powerful, but still require significant firmware engineering expertise for final implementation– which is in short supply.
- This talk: how do the new Xilinx Versal FPGAs fit into this picture?
  - Can "data scientists" successfully deploy neural networks to them with minimal FPGA experience? If so, what are the limitations of this approach?
  - We looked at **reproducing** examples using the Xilinx Vitis AI framework.
  - We started attempting to deploy a custom neural network using these tools (PELICAN); and explored some of the potential problems and limitations.

Xilinx Versal ACAP and AI Engines



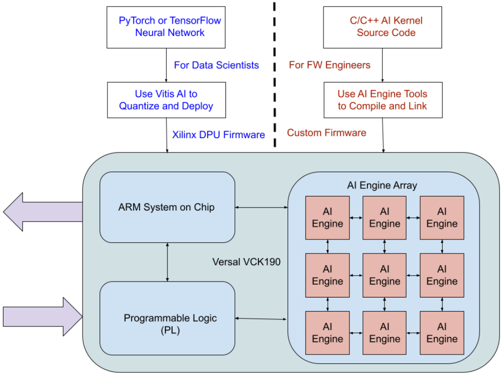
vck190 Evaluation Kit

- We purchased a Xilinx Versal ACAP evaluation kit for R&D:
    - Adaptive Compute Acceleration Platform has FPGAs and ARM SoC.
- Bought a VCK190 (AI Core series).
- This board also contains an array of 400 **AI Engine** tiles with:
    - Scalar and vector processors.
    - $32\,\mathrm{kB}$ local memory.
    - Fast interconnects, AXI interface.
- AI Engines can run up to $1.3\,\mathrm{GHz}$; intended for real-time processing and ML applications.

Programming AI Engines with Vitis AI

- AI engines **cannot** be programmed directly through (high-level) synthesis or RTL.
- Xilinx provide two different "flows" for programming AI engines:
  - Writing C/C++ code, compiling and creating **AI Engine Kernel** using `aiecompiler`.
  - Using **Vitis AI** to deploy pre-trained neural networks directly on the board.

- Vitis AI intended "for data scientists":
  - Requires Xilinx DPU (Deep Learning Processor Unit) firmware.
  - DPU uses AI Engines on the Versal; programmable logic on other boards.
- Integrates with Python ML environments:
  - Supports PyTorch and TensorFlow.
  - Vitis AI provides **quantizer**, **model inspector**, and **optimizer** (requires license)
  - (Mostly) open source, **does not require Vitis**.



Xilinx RESNET Tutorial

- Xilinx provide Vitis AI tutorials for both PyTorch and TensorFlow.
- First question: **do these tutorials actually work?** Can we reproduce their results?
- For Vitis AI 3.0+: basic tutorial using ResNet network for image recognition.
  - ResNet-18 trained on CIFAR10 dataset: **we ran this, will walk through today**.
  - ResNet-18 trained on ImageNet dataset: same approach, **just different dataset**.
  - Note: newest release (Vitis AI 3.5) doesn't have pre-built DPU images for the VCK190.
- Must set up board with DPU: can download pre-built image or build IP (requires Vitis).
- Steps (common to both tutorials):
  - Preprocess data (images), set up directory structures appropriately, etc.
  - Train the model (can skip if using pre-trained model).
  - Inspect the model: is it compatible with the target DPU?
  - Quantize and compile the model.

Vitis AI Setup

- Install instructions; see also official Xilinx instructions:
  - Xilinx provide docker containers; code can also be installed manually from source.
  - Docker containers with Nvidia GPU support must be **built manually**.
  - CPU-only containers can be downloaded from dockerhub: xilinx/vitis-ai-tensorflow2-cpu
- Instructions for setting up GPU TensorFlow2 container:

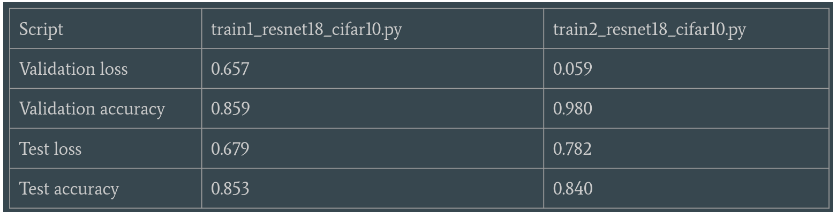
```
$ git clone https://github.com/Xilinx/Vitis-AI.git
$ cd Vitis-AI
$ git clone -b 3.5 https://github.com/Xilinx/Vitis-AI-Tutorials.git tutorials
$ cd docker && ./docker_build.sh -t gpu -f tf2
```

- To activate and run a container (and then start the tutorial):

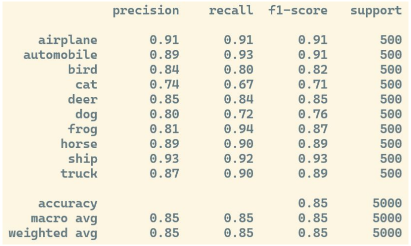
```
$ cd ..
$ ./docker_run.sh xilinx/vitis-ai-tensorflow2-gpu:3.5.0.001-bbccde60d
> conda activate vitis-ai-tensorflow2
> cd /workspace/tutorials/RESNET18
```

Training from Inside Vitis AI

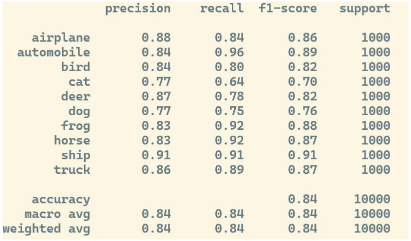
- Training results, pre-quantization, from 50 epochs (batch size 256) and two different runs.
- Train1 does file I/O; Train2 loads images over memory interface; meant to be equivalent.



Script	train1_resnet18_cifar10.py	train2_resnet18_cifar10.py
Validation loss	0.657	0.059
Validation accuracy	0.859	0.980
Test loss	0.679	0.782
Test accuracy	0.853	0.840



```
              precision    recall  f1-score   support

    airplane       0.91      0.91      0.91       500
  automobile       0.89      0.93      0.91       500
        bird       0.84      0.80      0.82       500
         cat       0.74      0.67      0.71       500
        deer       0.85      0.84      0.85       500
         dog       0.80      0.72      0.76       500
        frog       0.81      0.94      0.87       500
       horse       0.89      0.90      0.89       500
        ship       0.93      0.92      0.93       500
       truck       0.87      0.90      0.89       500

    accuracy                           0.85      5000
   macro avg       0.85      0.85      0.85      5000
weighted avg       0.85      0.85      0.85      5000
```

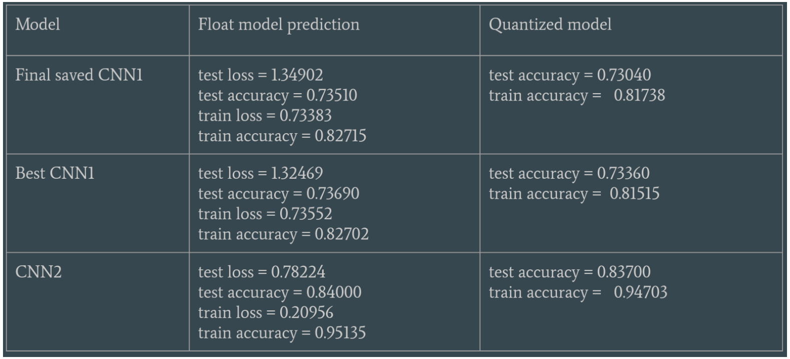


```
              precision    recall  f1-score   support

    airplane       0.88      0.84      0.86      1000
  automobile       0.84      0.96      0.89      1000
        bird       0.84      0.80      0.82      1000
         cat       0.77      0.64      0.70      1000
        deer       0.87      0.78      0.82      1000
         dog       0.77      0.75      0.76      1000
        frog       0.83      0.92      0.88      1000
       horse       0.83      0.92      0.87      1000
        ship       0.91      0.91      0.91      1000
       truck       0.86      0.89      0.87      1000

    accuracy                           0.84     10000
   macro avg       0.84      0.84      0.84     10000
weighted avg       0.84      0.84      0.84     10000
```

```
source run_all.sh cifar10_dataset && source run_all.sh run_cifar10_training
```

Model Inspection and Quantization

- **Inspection**: is the floating point model compatible with the target board DPU?
    - How does the model get partitioned for different DPU architectures?
    - What layers might not get mapped onto a DPU, and why?
- **Quantization**: quantizes frozen model from 32 bit floats to 8 bit fixed point.
    - Quantization differs for different trainings (not immediately obvious why!)

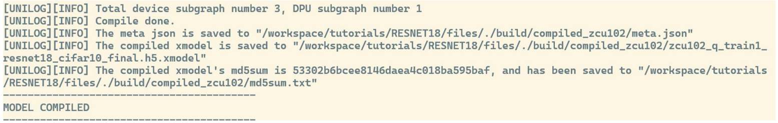


Model	Float model prediction	Quantized model
Final saved CNN1	test loss = 1.34902 test accuracy = 0.73510 train loss = 0.73383 train accuracy = 0.82715	test accuracy = 0.73040 train accuracy = 0.81738
Best CNN1	test loss = 1.32469 test accuracy = 0.73690 train loss = 0.73552 train accuracy = 0.82702	test accuracy = 0.73360 train accuracy = 0.81515
CNN2	test loss = 0.78224 test accuracy = 0.84000 train loss = 0.20956 train accuracy = 0.95135	test accuracy = 0.83700 train accuracy = 0.94703

```
source run_all.sh quantize_resnet18_cifar10
```

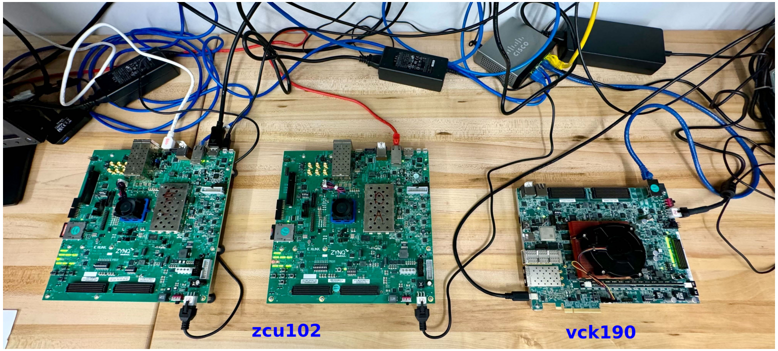
Compilation and Deployment

- Compiles model into target **board-specific** `*.xmodel` format:
  - ZCU102, VCK190
  - VEK280, VCK5000
  - Alveo V70
- We have access to both **VCK190**, **ZCU102**.
- On each board, installed:
  - Xilinx DPU firmware.
  - Test dataset.
  - Example SoC control software (using VART API to access DPU).
  - Compiled model files.

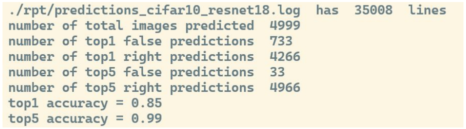


```
[UNILOG][INFO] Total device subgraph number 3, DPU subgraph number 1
[UNILOG][INFO] Compile done.
[UNILOG][INFO] The meta json is saved to "/workspace/tutorials/RESNET18/files/./build/compiled_zcu102/meta.json"
[UNILOG][INFO] The compiled xmodel is saved to "/workspace/tutorials/RESNET18/files/./build/compiled_zcu102/zcu102_q_train1_
resnet18_cifar10_final.h5.xmodel"
[UNILOG][INFO] The compiled xmodel's md5sum is 53302b6bcee8146daea4c018ba595baf, and has been saved to "/workspace/tutorials
/RESNET18/files/./build/compiled_zcu102/md5sum.txt"
-----------------------------------------
MODEL COMPILED
-----------------------------------------
```

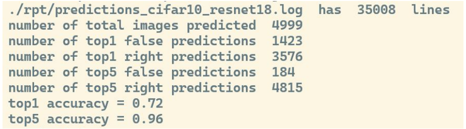
```
source run_all.sh compile_resnet18_cifar10
```



Versal vs Zynq Performance



```
./rpt/predictions_cifar10_resnet18.log  has  35008  lines
number of total images predicted  4999
number of top1 false predictions  733
number of top1 right predictions  4266
number of top5 false predictions  33
number of top5 right predictions  4966
top1 accuracy = 0.85
top5 accuracy = 0.99
```



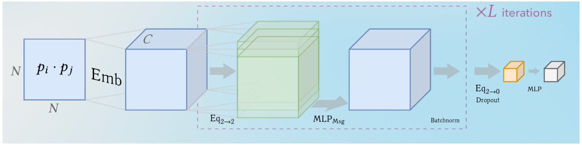
```
./rpt/predictions_cifar10_resnet18.log  has  35008  lines
number of total images predicted  4999
number of top1 false predictions  1423
number of top1 right predictions  3576
number of top5 false predictions  184
number of top5 right predictions  4815
top1 accuracy = 0.72
top5 accuracy = 0.96
```

- Models really do run on the boards; accuracy comparable to post-quantization!
  - The **same model** was trained and quantized, so accuracy is the same on both boards; different training runs continue to give slightly different results.
- Performance statistics can be gathered using vaitrace:
  - `vaitrace [--txt] ./control_app ./vck190_resnet18_cifar10.xmodel ...`
  - Generates either text output or CSV files which can be loaded into Vitis Analyzer.
- Versal performs **faster** and **more efficiently**, but AI engines **use more bandwidth**.
- Full vaitrace results available in backup, includes information about CPU function calls.

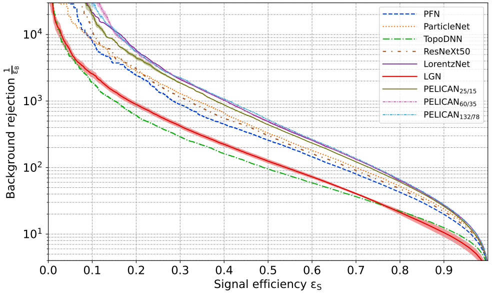
Board	SW Runtime (ms)	HW Runtime (ms)	Efficiency (%)	Avg Bandwidth (MB/s)
zcu102	1.583	1.420	4.3	7516.129
vck190	0.499	0.399	1.9	26876.923

PELICAN: Lorentz-Invariant Networks

- Can we extend these examples to something nontrivial?
- Testing applying these methods to a novel network, PELICAN:
  - Developed by our collaborators.
  - Built to be **Lorentz and permutation equivariant**.
  - Use Lorentz-invariant dot products of 4-momenta as basis to build network of Lorentz invariant/covariant functions.
  - Excellent performance as a top quark tagger; promising for other tasks.
  - The type of network we might want to deploy as part of a trigger system.



Jan Offermann



2211.00454, 2307.16506

Porting PELICAN using Vitis AI

- How do we integrate Vitis AI with a new model like PELICAN?
  - Built using PyTorch, not Tensorflow.
  - Pre-trained; can start with pre-built `xilinx/vitis-ai-pytorch-cpu` containers.
- Load the model, run the quantizer:
  - Specify mode: calibration or testing.
  - Specify device (CPU or GPU) and output directory for quantization results.
  - Specify the **input shape**: what dimension is your input data?
  - **Pre-processing** may be useful to get input data in hardware-friendly format!
- Must **test** model after quantization.

```
...
from pytorch_nndct.apis import torch_quantizer
...
def main():
    # Instantiate and load PELICAN model..
    model = PELICANClassifier(...)
    model.load_state_dict(pretrained_state)
    # Need the input data shape.
    input_shape = next(iter(dataloader))
    # Run the quantizer!
    quantizer = torch_quantizer(quant_mode,
                                model,
                                (input_shape),
                                output_dir,
                                device="cpu")
    model = quantizer.quant_model
    # Then run PELICAN model test suite.
    ...
```

Issues Quantizing Networks

- Does this work out of the box? **Yes**... with some caveats.
    - Lots of warnings (`VAIQ_WARN`); complaints about tensors that could not be quantized.
    - Quantization only supports **floats** and **doubles**– not **integers and booleans**...
    - Some tensors evaluate to infinity (or NaN): maybe a result of inputs that couldn't be quantized propagating through the model... needs investigation.
- Running **model inspector** needed to better understand what's happening.

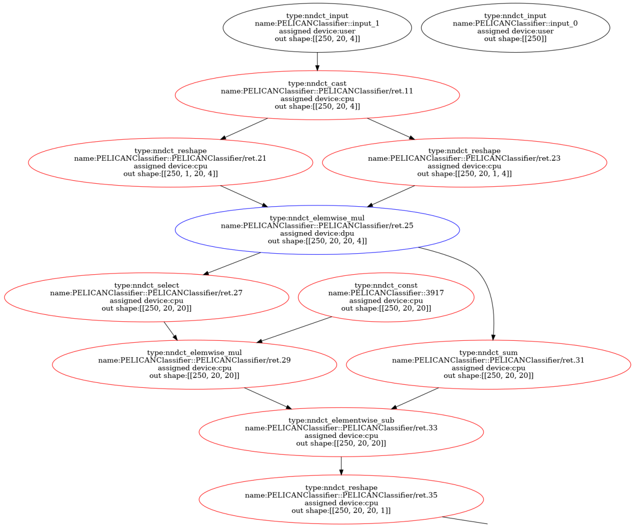
```
[QUANTIZER_TORCH_TENSOR_TYPE_NOT_QUANTIZABLE]: The tensor type of PELICANClassifier::input_0
    is torch.int64. Only support float32/double/float16 quantization.
[QUANTIZER_TORCH_TENSOR_TYPE_NOT_QUANTIZABLE]: The tensor type of PELICANClassifier::input_2
    is torch.bool. Only support float32/double/float16 quantization.
[QUANTIZER_TORCH_TENSOR_VALUE_INVALID]: The tensor type of
    PELICANClassifier::PELICANClassifier/Net2to2[net2to2]/Eq2to2[eq_layers]/ModuleList[0]/ret.
    have "inf" or "nan" value. The quantization for this tensor is ignored. Please check it.
[QUANTIZER_TORCH_TENSOR_VALUE_INVALID]: The tensor type of
    PELICANClassifier::PELICANClassifier/Eq2to0[agg_2to0]/ret.193 have "inf" or "nan" value.
    The quantization for this tensor is ignored. Please check it.
```

Enabling the Model Inspector

- What is needed to run the inspector?
  - What **DPU architecture** do you intend to deploy on?
  - DPUCVDX8G: Versal, 8-bit quantization.
  - Otherwise, same inputs as quantizer.
  - Runs on CPU/GPU, emulates DPU.
- Needed to **adapt input data format** to run inspector successfully:
  - PELICAN assumed its input would be a dictionary; Inspector assumed tuple/list.
  - Feature described as "experimental".
- Generates very detailed report, with visualization of quantized graph.

```
...
from pytorch_nndct.apis import torch_quantizer
from pytorch_nndct.apis import Inspector
...
def main():
    # Instantiate and load PELICAN model
    ...
    # Need the input data shape.
    input_shape = next(iter(dataloader))
    # Run the inspector.
    inspector=Inspector("DPUCVDX8G_ISA3_C32B6")
    inspector.inspect(model, (input_shape),
                      device="cpu",
                      image_format='png')
    # Run the quantizer post-inspection
    ...
```

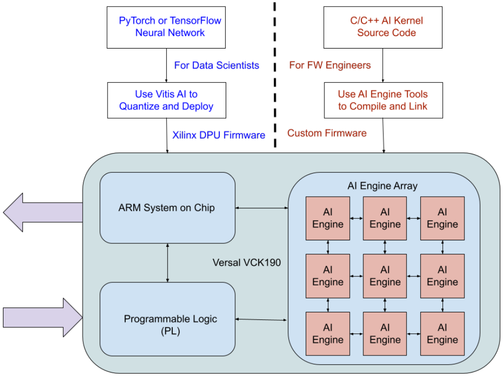
Results from the Model Inspector


type:nndct_input name:PELICANClassifier::input_1 assigned device:user out shape:[[250, 20, 4]]
type:nndct_input name:PELICANClassifier::input_0 assigned device:user out shape:[[250]]
type:nndct_cast name:PELICANClassifier::PELICANClassifier/ret.11 assigned device:cpu out shape:[[250, 20, 4]]
type:nndct_reshape name:PELICANClassifier::PELICANClassifier/ret.21 assigned device:cpu out shape:[[250, 1, 20, 4]]
type:nndct_reshape name:PELICANClassifier::PELICANClassifier/ret.23 assigned device:cpu out shape:[[250, 20, 1, 4]]
type:nndct_elemwise_mul name:PELICANClassifier::PELICANClassifier/ret.25 assigned device:dpu out shape:[[250, 20, 20, 4]]
type:nndct_select name:PELICANClassifier::PELICANClassifier/ret.27 assigned device:cpu out shape:[[250, 20, 20]]
type:nndct_const name:PELICANClassifier::3917 assigned device:cpu out shape:[[250, 20, 20]]
type:nndct_elemwise_mul name:PELICANClassifier::PELICANClassifier/ret.29 assigned device:cpu out shape:[[250, 20, 20]]
type:nndct_sum name:PELICANClassifier::PELICANClassifier/ret.31 assigned device:cpu out shape:[[250, 20, 20]]
type:nndct_elementwise_sub name:PELICANClassifier::PELICANClassifier/ret.33 assigned device:cpu out shape:[[250, 20, 20]]
type:nndct_reshape name:PELICANClassifier::PELICANClassifier/ret.35 assigned device:cpu out shape:[[250, 20, 20, 1]]

- Model summary: mostly CPU...
  - **DPU nodes**: 14 total
  - **CPU nodes**: 116 total
  - Some **inputs** appear unused.
- Why does DPU implementation fail?
  - "All the children nodes are assigned to CPU."
  - "Input of reshape is not on DPU."
  - "`nndct_cast` can't be converted to XIR."
  - "xir::Op{name = . . . , type = . . .} has been assigned to CPU: [DPU does not support eltwise SUB]."
- Investigation required; changes to model likely needed!

Programming AI Engines Directly

- Vitis AI is quite powerful, but doesn't work out of the box on complex networks.
  - Some tools available: WeGo, integration with ONNX; support for custom NN layers.
  - But for complex networks (or for **non-ML applications!**) may need to program AIE directly.
- **AI Engine tools** approach:
  - Write C/C++ code; build with `aiecompiler`
  - Emulate AI engines, run simulation with `aiesimulator`.
  - Create, link AI engine kernel using Vitis.
  - Integrate with PL kernels or SoC code.
  - I wrote a simple "hello world" example.
- More sophisticated tutorials from Xilinx:
  - Including implementing neural network (LeNet) by hand, without using Vitis AI.
  - We are presently evaluating this workflow!

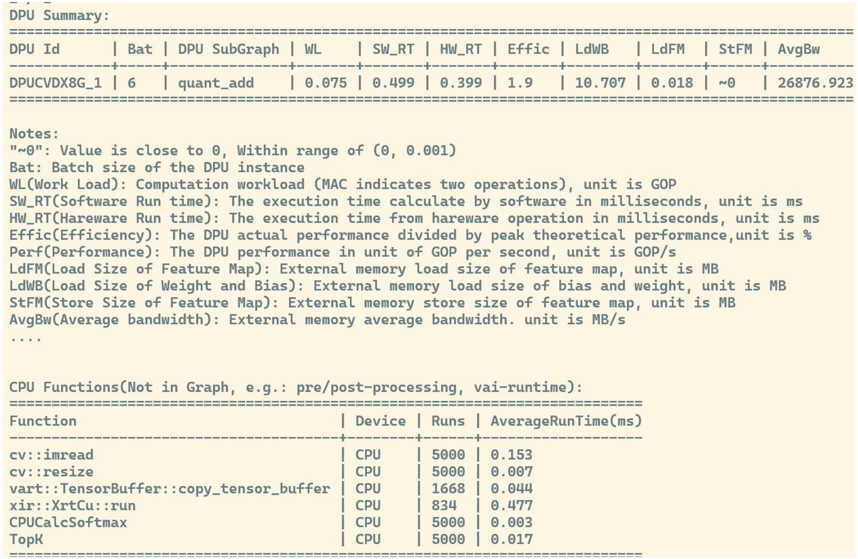


Conclusion

- The Xilinx Versal AI series boards have powerful features intended for machine learning:
  - Vitis AI provides a framework for easily deploying neural networks on these boards.
  - Easy to run a network on different board types; comparison between Versal and Zynq shows clear performance differences, even for relatively simple examples.
  - Tools don't necessarily work out of the box with all custom network architectures.
  - And to build a **real** system, still need to integrate Xilinx DPU with other firmware.
- We'd like to understand better what (if anything) we can use these boards for:
  - Combination of SoC with programmable logic and AI engine functionality quite exciting.
  - Evaluating different workflows and use cases in e.g. a trigger context.
  - Hope to have more concrete results to present in the future!
- Thanks for your attention!

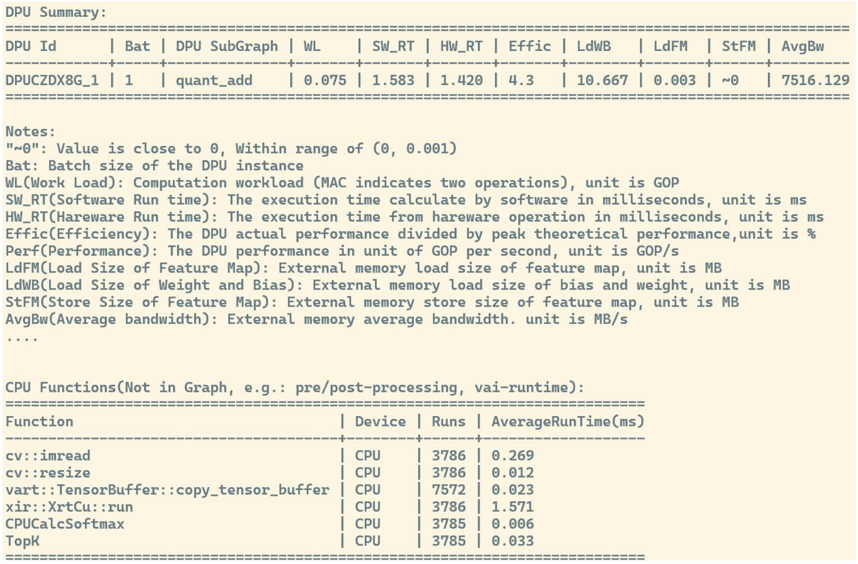
Backup

Vaitrace Results: Versal



```
DPU Summary:
==============================================================================================
DPU Id       | Bat | DPU SubGraph | WL    | SW_RT | HW_RT | Effic | LdWB   | LdFM  | StFM | AvgBw
-------------+-----+--------------+-------+-------+-------+-------+--------+-------+------+----------
DPUCVDX8G_1 | 6   | quant_add    | 0.075 | 0.499 | 0.399 | 1.9   | 10.707 | 0.018 | ~0   | 26876.923
==============================================================================================

Notes:
"~0": Value is close to 0, Within range of (0, 0.001)
Bat: Batch size of the DPU instance
WL(Work Load): Computation workload (MAC indicates two operations), unit is GOP
SW_RT(Software Run time): The execution time calculate by software in milliseconds, unit is ms
HW_RT(Hareware Run time): The execution time from hareware operation in milliseconds, unit is ms
Effic(Efficiency): The DPU actual performance divided by peak theoretical performance,unit is %
Perf(Performance): The DPU performance in unit of GOP per second, unit is GOP/s
LdFM(Load Size of Feature Map): External memory load size of feature map, unit is MB
LdWB(Load Size of Weight and Bias): External memory load size of bias and weight, unit is MB
StFM(Store Size of Feature Map): External memory store size of feature map, unit is MB
AvgBw(Average bandwidth): External memory average bandwidth. unit is MB/s
....


CPU Functions(Not in Graph, e.g.: pre/post-processing, vai-runtime):
============================================================================
Function                                | Device | Runs | AverageRunTime(ms)
----------------------------------------+--------+------+-------------------
cv::imread                              | CPU    | 5000 | 0.153
cv::resize                              | CPU    | 5000 | 0.007
vart::TensorBuffer::copy_tensor_buffer | CPU    | 1668 | 0.044
xir::XrtCu::run                         | CPU    | 834  | 0.477
CPUCalcSoftmax                          | CPU    | 5000 | 0.003
TopK                                    | CPU    | 5000 | 0.017
============================================================================
```

Vaitrace Results: Zynq



```
DPU Summary:
===================================================================================================
DPU Id       | Bat | DPU SubGraph | WL    | SW_RT | HW_RT | Effic | LdWB   | LdFM  | StFM | AvgBw
-------------+-----+--------------+-------+-------+-------+-------+--------+-------+------+---------
DPUCZDX8G_1 | 1   | quant_add    | 0.075 | 1.583 | 1.420 | 4.3   | 10.667 | 0.003 | ~0   | 7516.129
===================================================================================================

Notes:
"~0": Value is close to 0, Within range of (0, 0.001)
Bat: Batch size of the DPU instance
WL(Work Load): Computation workload (MAC indicates two operations), unit is GOP
SW_RT(Software Run time): The execution time calculate by software in milliseconds, unit is ms
HW_RT(Hareware Run time): The execution time from hareware operation in milliseconds, unit is ms
Effic(Efficiency): The DPU actual performance divided by peak theoretical performance,unit is %
Perf(Performance): The DPU performance in unit of GOP per second, unit is GOP/s
LdFM(Load Size of Feature Map): External memory load size of feature map, unit is MB
LdWB(Load Size of Weight and Bias): External memory load size of bias and weight, unit is MB
StFM(Store Size of Feature Map): External memory store size of feature map, unit is MB
AvgBw(Average bandwidth): External memory average bandwidth. unit is MB/s
....


CPU Functions(Not in Graph, e.g.: pre/post-processing, vai-runtime):
===========================================================================
Function                              | Device | Runs | AverageRunTime(ms)
--------------------------------------+--------+------+-------------------
cv::imread                            | CPU    | 3786 | 0.269
cv::resize                            | CPU    | 3786 | 0.012
vart::TensorBuffer::copy_tensor_buffer | CPU    | 7572 | 0.023
xir::XrtCu::run                       | CPU    | 3786 | 1.571
CPUCalcSoftmax                        | CPU    | 3785 | 0.006
TopK                                  | CPU    | 3785 | 0.033
===========================================================================
```