Michael Masenheimer- Vitis AI + ResNet-50 Workflow Notes

*Note that this work was done on a Windows 11, 64 bit x64 based system, so different machines may demand different steps-I did my best to document the separation between windows and unix/linux steps. If there is any confusion here, I recommend checking out the quick start guide here: https://xilinx.github.io/Vitis-AI/3.5/html/docs/quickstart/vek280.html

Remoting into the EEPP Arizona physics server for Vitis through the terminal (I'm using a GitBash terminal on Windows 11)

1. Boot up Cisco Secure client, login with UA netID and activate UA SSL VPN

2. Initialize master server for exporting the vitis display to. (This allows you to set your gitbash $DISPLAY variable to whatever the master server is, in my case it's "Michaels-Laptop:0.0"), check the system tray to see if it's running–it should have 0 clients before remoting into the eepp. On client startup for the xserver, do "Start no client", and uncheck "native opengl" and check "disable access control", then click finish.

3. Once the VPN is instantiated, open a GitBash terminal and run export DISPLAY={NAME OF X SERVER}, I recommend running an echo $DISPLAY and cross checking what the current display is set to to the xserver name.

4. When you remote in, run ssh -Y {YOUR USERNSMAE}@eepp-t400.physics.arizona.edu and enter the password, it'll give you an auth warning, but that doesn't apply to running vitis or xclock (a display test). I recommend running xclock after this to see if the display is working. A little clock should show up on the screen, and to get out of it, click the x in the top right. If the clock isn't showing up, try running echo $DISPLAY again to see if your display variable is set correctly

5. Now we source vitis using source /tools/Xilinx/Vitis/2024.1/settings64.sh after that, simply run vitis and it'll initialize the Vitis IDE. I like to have a terminal open in vitis, so there should be a button on the left-hand side to open one, and you can select the kind of terminal, (given you have the correct software installed), but GitBash works great or the default macOS or linux terminals are great too).

Compiling the RESNET_50 model through docker (I'm on windows, so some extra steps are needed), estimated 12-14 GB of storage needed

* I noticed that for this section, I couldn't pull anything from the Vitis AI repository when I had the U of A VPN on, so make sure to turn it off before continuing

1. FOR WINDOWS USERS ONLY– STEPS 1-3 We need to install the Windows Subsystem for Linux (WSL) in order to evaluate Vitis AI. Open a powershell and run [Powershell] > wsl --install -d Ubuntu-20.04

2. Depending on your system requirements, you might need to install a different distribution of WSL, so see https://xilinx.github.io/Vitis-AI/3.5/html/docs/reference/system_requirements.html, you can also see the distribution options by running [Powershell] > wsl --list --online

3. Now we start a distribution. I use [Powershell] > wsl -d Ubuntu-20.04 on my machine

4. Now It's time to clone the VItis AI Repository (Approx. 250 MB). Once you have done this step, you don't need to do it again. If you're on windows, open up WSL, and if on unix/linux run [Host] $ git clone https://github.com/Xilinx/Vitis-AI **Make sure you have Git installed on your machine prior to this step.** https://git-scm.com/downloads

5. Next, we must install docker desktop. I believe it is the same process if you're on unix/linux-based systems or WSL. I followed https://www.docker.com/get-started/ and used [Host] $ sudo apt update && sudo apt install docker.io to install docker and then sudo usermod -aG docker mmasenheimer to add myself (mmasenehimer) as a user to the docker group

6. Check to see if you can run a docker container by running [Host] $ docker run hello-world This will automatically download a test image from docker and run it in a container, it should spit out "hello world" if run correctly. You should have the latest version of docker installed but if you're unsure, run [Host] $ docker --version to see what current version you're running

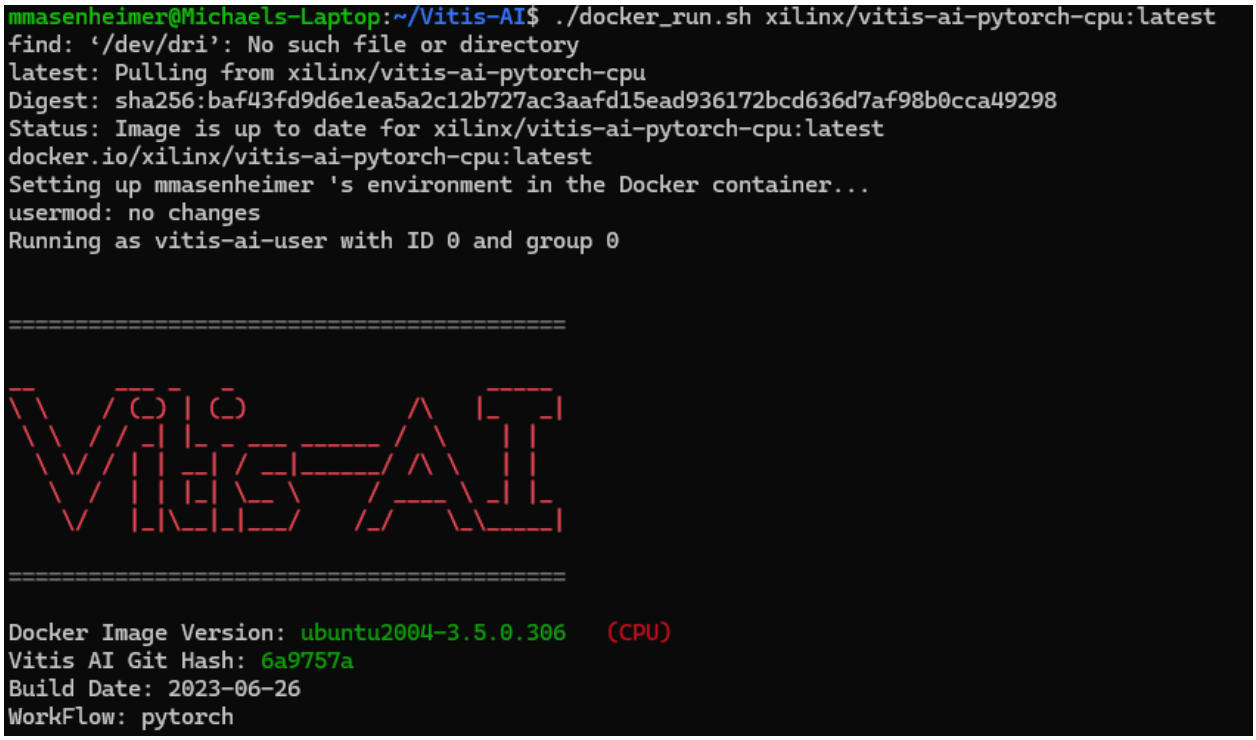
7. Pulling the vitis AI Docker. Now that we have docker set up and have cloned the Vitis repository, we can download the container with pre-trained models. The actual download is around 4-5 GB compressed and 8 GB after extraction. Run [Host] $ docker pull xilinx/vitis-ai-pytorch-cpu:latest

8. After pulling the AI Docker, we need to install the Xilinx cross compiler so we can dock the Resnet_50 model. **If you've already installed the compiler, skip this step.** This requires a lot of disk space, I found to be around 5 GB. By default, the cross compiler will be installed in ~/petalinux_sdk_2023.1. Make sure whatever path you choose has read/write permissions, although I just went with the default which does.
Run [Host] $ cd Vitis-AI/board_setup/vek280
Run [Host] $ sudo chmod u+r+x host_cross_compiler_setup.sh
Run [Host] $ ./host_cross_compiler_setup.sh
(This will take a while)

9. After the installation is complete, follow the prompts on the screen, and then run this command: [Host] $ source ~/petalinux_sdk_2023.1/environment-setup-cortexa72-cortexa53-xilinx-linux

After step 9, we will start a docker container targeting resnet_50, I found that closing the terminal or commanding “exit” stops the container completely and you have to go back and restart

10. Notice that the /workspace directory in Docker corresponds to your /Vitis-AI directory on the host. Thus note that [Docker] /workspace/examples/vai_runtime/resnet50_pt = [Host] Vitis-AI/examples/vaiexamples/vai_runtime/resnet50_pt. This is just a check for correct directory installation. Run [Host] $ cd ../.. To backtrack directories. Now run [Host] $ ./docker_run.sh xilinx/vitis-ai-pytorch-cpu:latest You should see something like the following image:



11. Now we need to activate the conda pytorch environment. There is a way to use TensorFlow instead of Pytorch but for the sake of simplicity and remaining in the realm of defined documentation through Xilinx, we’ll stick with torch. Run [Docker] $ conda activate vitis-ai-pytorch

12. Cross-compiling the resnet_50 example in docker.
Run [Docker] $ cd examples/vai_runtime/resnet50_pt (Pointing the directory towards resnet)
Run [Docker] $ sudo chmod u+r+x build.sh (Changing file mode permissions)
Run [Docker] $ bash -x build.sh (Actually compiling the model, I found that the line in the quick start guide for this has a *Unicode en dash* (U+2013) instead of a regular hyphen, causing a directory error in bash. The command above corrected that). You should see something like the image below

```
+ result=0
+ pkg-config --list-all
+ grep opencv4
opencv4                 OpenCV - Open Source Computer Vision Library
+ result=1
+ '[' 1 -eq 1 ']'
++ pkg-config --cflags --libs-only-L opencv4
+ OPENCV_FLAGS='-I/usr/include/opencv4/opencv -I/usr/include/opencv4'
++ basename /workspace/examples/vai_runtime/resnet50_pt
+ name=resnet50_pt
+ sample=resnet50_pt
+ [[ g++ == *\s\y\s\r\o\o\t* ]]
+ g++ -O2 -fno-inline -I. -I/home/vitis-ai-user/.local/Ubuntu.20.04.x86_64.Debug/include -I/home/vitis-ai-user/.local/Ubuntu.20.04.x86_64.Release/include -L/home/vitis-ai-user/.local/Ubuntu.20.04.x86_64.Debug/lib -L/home/vitis-ai-user/.local/Ubuntu.20.04.x86_64.Release/lib -Wl,-rpath=/home/vitis-ai-user/.local/Ubuntu.20.04.x86_64.Debug/lib -Wl,-rpath=/home/vitis-ai-user/.local/Ubuntu.20.04.x86_64.Release/lib -o resnet50_pt -std=c++17 /workspace/examples/vai_runtime/resnet50_pt/resnet50_pt.cpp -lvart-runner -lvart-dpu-controller -I/usr/include/opencv4/opencv -I/usr/include/opencv4 -lopencv_videoio -lopencv_imgcodecs -lopencv_highgui -lopencv_imgproc -lopencv_core -lglog -lxir -lunilog -lpthread
(vitis-ai-pytorch) vitis-ai-user@Michaels-Laptop:/workspace/examples/vai_runtime/resnet50_pt$
```

If the compilation process does not report an error and the executable file resnet50_pt is generated, then the host environment is installed correctly. If an error is reported, double-check that you executed the source ~/petalinux.... Command.

13. If you are going to run the model in vitis (next section), keep the terminal window open. The model is now compiled under (vitis-ai-pytorch) vitis-ai-user@Michaels-Laptop:/workspace/examples/vai_runtime/resnet50_pt$

Running AIE ML simulation on our compiled resnet_50 model through Vitis (from the previous section)

1. Working on running tests on the compiled model this week (It is taking a while to figure out). There are a few types of tests we could undergo.

    - VART test (Validation with Vitis AI Runtime) which validates that the model runs correctly, completely in the vitis ai docker. It runs inference on a set of images. (Girrafe.jpg and sample.jpg). Shows FPS, accuracy and DPU usage. I'm still in testing here, I don't have concrete instructions yet.

    - Quantization validation test (Will run in the future, I'm in the process of looking for a good dataset of 100-1000 images to test). This tests the possibility of reducing the precision of resnet weights and activation for faster inference and lower memory usage. This test verifies that quantization hasn't degraded the model's accuracy too much. In progress

    - AI Engine Graph test Running custom signal processing using a dataflow model. According to xilinx, this should be run before deploying the model to the VEK. This test examines data flow between PLIOs and kernels, kernel execution, and other graph behavior. Looking to test this week.

- Actual board deployment and execution I think this should be the final test for a model, given the others validate the deployment, and behavior. Can't do yet until the board in configured correctly, which we can figure out how to do in https://xilinx.github.io/Vitis-AI/3.5/html/docs/quickstart/vek280.html under setting up the target.

- Performance and Profiling test This test will display layer-level latency and hardware utilization. Good to know when writing custom models as we can set our own benchmarks. Can be done so by running:

  [Host} $ cd /workspace/examples/vai_runtime/resnet50_pt/
  [Host] $ env DPU_PROFILE=1 ./resnet50 ./resnet50.xmodel ./images

  Note that this test can be done on the actual VEK board or in the docker container

2. These tests can all be done on already compiled models, I haven't gotten around to any of them except for performance and profiling. Once I run the graph test in vitis, I'll document the process.

Pushing the compiled resnet50.xmodel and other important stuff into a github for universal sourcing

I wanted to go through the xilinx ai repository and trim off any unnecessary files and unrelated branches, so I ended up copying what I thought was important to a local file on my computer from the docker and then pushing them onto my personal repository: https://github.com/mmasenheimer/resnet50-model

1. I made a file serving as a medium between the docker container and my github, since apparently you can't push stuff straight from the container (I think?). While you're in the container, copy the file directory to your medium folder (I just threw it on my desktop) and run [Host] $ mkdir -p /mnt/c/Users/mmase/Desktop/resnet50model My directory will be different from yours, of course!

2. Next you're going to want to get your docker container id, which is needed for copying stuff inside the container to the folder. Do so by opening up a new WSL gitbash terminal and type [Host] $ docker ps copy the id, and then run [Host] $ docker cp dd835a07a693:/workspace/examples/vai_runtime/resnet50_pt/resnet50/resnet50.xmodel /mnt/c/Users/mmase/Desktop/resnet50model/resnet50.xmodel where the items in pink correspond to the container id and the path to that medium folder, respectively.

3. I copied a few things into this folder, two test images, the actual compiled resnet50.xmodel, a README from xilinx specifically about the VEK model, resnet50_pt which is the runtime executable for the model, and finally a list of ImageNet class labels. I then headed over to my gitHub and created a new repository, called resnet50-model, then executed the following lines on a windows powershell terminal:

   [ps] > cd /mnt/c/Users/mmase/Desktop/resnet50model
   [ps] > git init
   [ps] > git add <all of the file names separated by a space>
   [ps] > git commit -m “some really cool awesome message about the commit to the repo”
   [ps] > git push origin main

Now any machine can clone this repository (after I do some housekeeping), and have the compiled model, the executable, and a few test images available without having to download the entire 500 MB xilinx repo. This week I might try to get a few variable size models in this github for further testing. https://github.com/mmasenheimer/resnet50-model

Michael Masenehimer FPGAs and ML inference with hls4ml workflow notes

(Notes from Systems Group at ETH Zurich)

- The large collision frequency on the LHC produces 100TB of data per second, “triggering” means filtering the data to reduce it to manageable levels

Why use FPGAs?

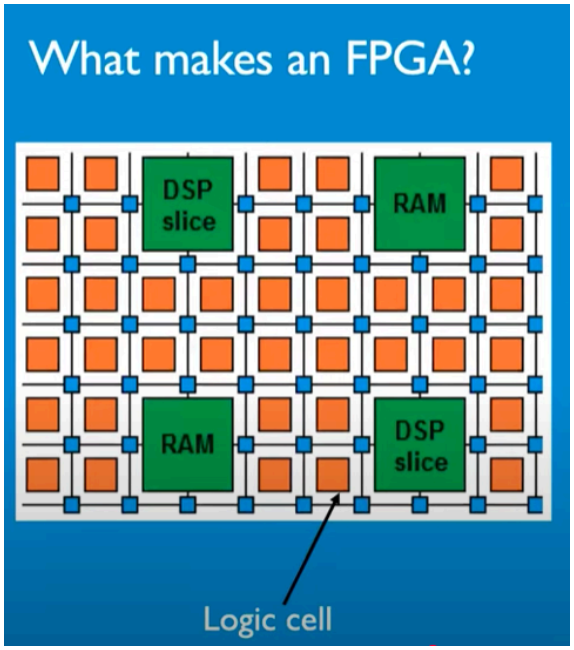
- Custom hardware acceleration, precisions and memory management (full control over the hardware)
- Data-flow architecture with no scheduling or control overheads
- FPGAs enable space and time multiplexing; compared to GPUs which usually only allow time multiplexing (Combining multiple signals into one signal)
- Each layer in the neural network gets a dedicated custom piece of hardware for itself
- Data is streamed between layers enabling deeply pipelined designed designs

FPGAs vs GPUs

- Layers are processed sequentially with some scheduling overhead (the kernel gets access to the whole GPU by default)
- There is less support for custom precisions (single-bit operands aren't very good) and memory hierarchy and data movement

GPUs	We use FPGAs when:
Much higher frequency compared to FPGAs (8x-10x)	Extremely low latency requirements (~100s of ns)
More memory and bandwidth	Low power requirements and constrained environments (like satellites or robots)
Faster for highly parallel, floating-point operations	In-network processing
	Custom precisions and highly irregular operations such as embeddings

FPGA Architecture



- Logic cells (or look-up tables) perform arbitrary functions on small bit width inputs
- These can be used for boolean operations, arithmetic, small memories
- Flip-Flops (or registers) data in time with the clock pulse, so data can be stored at every clock cycle and read at every clock cycle

---

- DSPs (Digital Signal Processors) are specialized units for multiplication and arithmetic
- Faster and more efficient than using LUTs for these types of operations
- For neural networks, DSPs are often the most scarce resource

---

- BRAMs are small, fast memories which can access data in one clock cycle
- A “bigger” FPGA has nearly 40MB of BRAM, chained together as needed, which is also suitable for larger models such as ResNet
- Recent accelerator cards also come equipped with off-chip HBM memory, up to 800 GBps

- In addition, there are specialised blocks for I/O, making FPGAs popular in larger computer systems
- High speed transceivers with Tb/s total bandwidth
- Low power per Op (relative to CPU/GPU)

How FPGAs are Programmed

- Hardware description Languages (HDL): are programming languages which describe electronic circuits

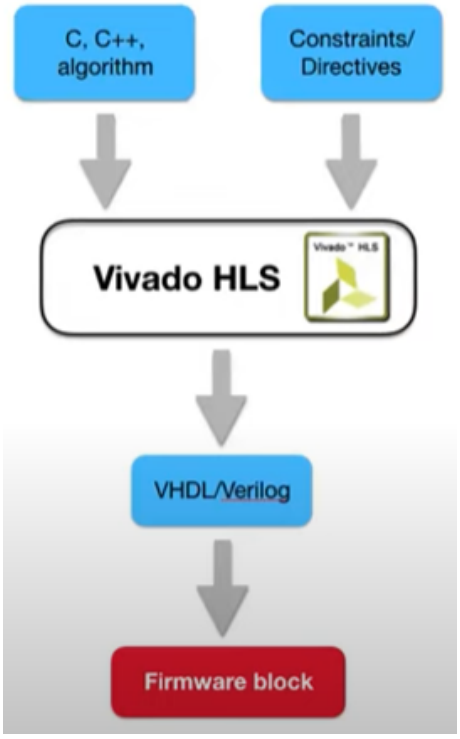
High level Synthesis:

-Compile from C/C++ to VHDL

-Pre-processor directives and constraints used to optimize the design

-Drastic decrease in firmware development time

-We use AMD/Xilinx Vitis HLs



HLS examples

```
for(int i = 0; i < 16; i++) {
    #pragma HLS UNROLL 4
    c[i] = a[i] + b[i];
}
```

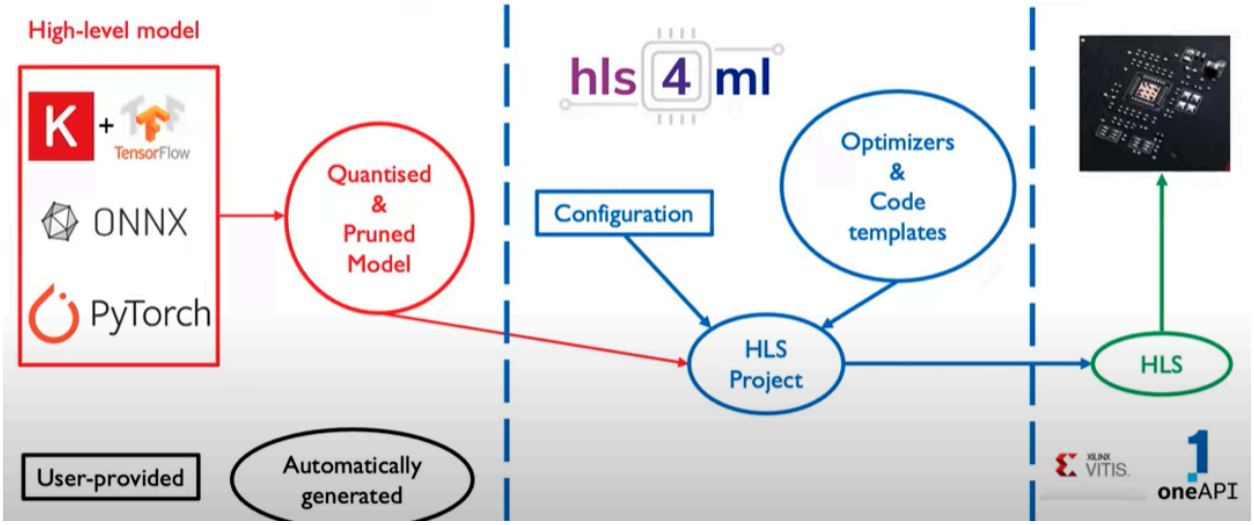
- Note the loop parallelised 4 times in each clock cycle
- Total block execution: 6 / 4 = 4cc

```
float weights[16];
#pragma HLS ARRAY_RESHAPE variable=weights block factor=block_factor
#pragma HLS RESOURCE variable=weights core=ROM_2P_BRAM
```

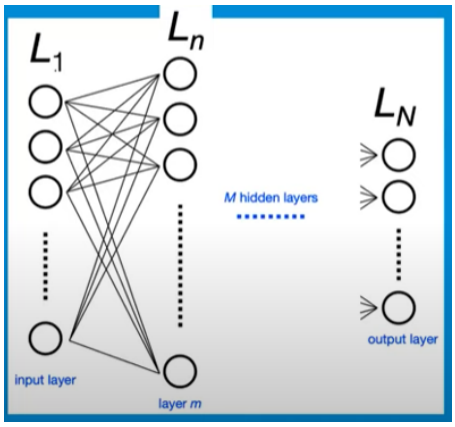
- Reshapes the weight matrix according to a specific block factor
- Stores the resulting array in on-chip ROM

Enter hls4ml: an open-source framework for neural network acceleration on FPGAs

- Front-end agnostics: Keras, PyTorch, or (Q)ONNX
- Back-end agnostic: vivado HLS, Vitis HLS, intel HLS, oneAPI, Catapult HLS, etc
- Many supported layers, Dense, Convolutional, Recurrent, Graph, etc
- High configurability: Tune precision, reuse factor, custom layers etc



Neural network inference



X_n = g_n (W_(n, n-1) x_(n-1) + b_n)

Logic cells or precomputed and stored in BRAMs

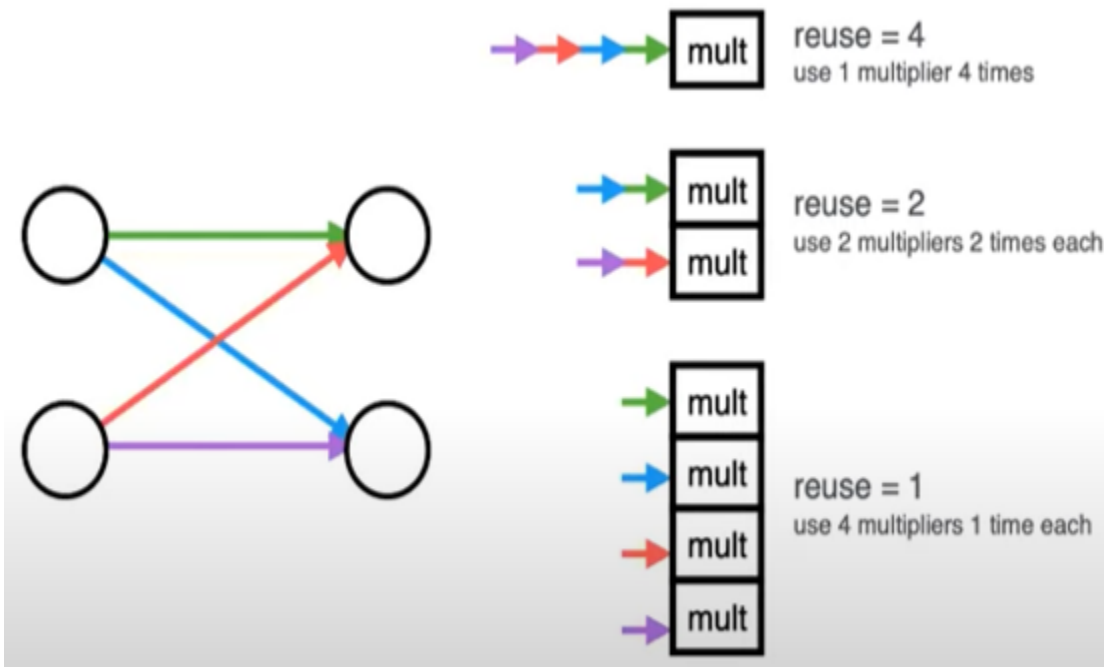
DSPs

Logic cells

-Balance between resources, does the model fit in the latency requirements?

Efficient inference: parallelisation

- Trade-off between latency and FPGA resources determined by the parallelization of the calculations in each layer
- Configure the “reuse factor” = number of times a multiplier is re-used to do a computation

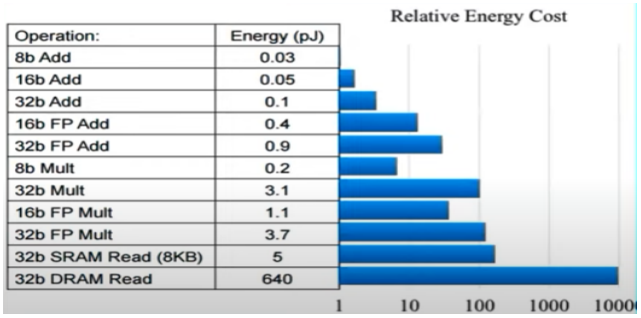


Efficient inference: quantisation

- Floating point operations are expensive

- On FPGAs, we can use fixed-point precision:
- Implemented using integer logic therefore very fast
- Acts like “limited-precision” floating-point, so need to ensure sufficient bits

Operation:	Energy (pJ)
8b Add	0.03
16b Add	0.05
32b Add	0.1
16b FP Add	0.4
32b FP Add	0.9
8b Mult	0.2
32b Mult	3.1
16b FP Mult	1.1
32b FP Mult	3.7
32b SRAM Read (8KB)	5
32b DRAM Read	640



Efficient inference: quantisation-aware training

- Quantisation-aware training enables training models with very low precisions:

- Out-performs post-training quantisation significantly

- At a high level, it performs the forward pass with reduced precision and the backward pass in floating point precision

- Possible to achieve very low precisions (for binary and ternary models)

Building a convolutional neural network in C++ workflow notes

Theory behind CNNs

- CNNs are designed to process data with grid-like topology (images in our case)

- CNNs do not treat inputs as flat vectors like fully connected networks, they use the spatial structure of the data

- Designed to automatically and adaptively learn spacial hierarchies of feature from input images

Training a CNN

- **Forward pass**- each layer applies its transformations -> activations -> pooling -> flatten -> dense -> output

- **Loss calculation**- The predicted outputs are compared to the true labels using a loss function (Research more on this later)

- **Backpropagation**- Gradients of the loss with respect to the weights are computed with the chain rule, gradients flow backwards through the network, adjusting filters and weights

Extra

- In early layers, filters learn edges, textures, and colors
- In deeper layers, they learn object parts, shapes, and entire objects
- CNNs compose features from low-level to high-level

CNN Architecture

Input Layer	Convolutional layer	Activation Function	Pooling layer (max or Avg pooling)	Repeat Conv + ReLu + Pooling	Flatten Layer	Fully connected layer (Dense Layer)	Output layer
-Accepts input image: e.g a 28x28 grayscale image -> 28x28x1 tensor	-Applies a set of learnable filters (kernels)	-Applies a non-linear function: f(x) = max(0, x)	-Reduces the spacial dimension s	-Can stack multiple conv layers to learn more abstract patterns (edges, shapes, objects)	-Converts 3D feature maps into a 1D vector before feeding directly into fully connected layers	-Classic neural network layer where every input is connected to every output node	-Turns final layer outputs into probabilities
-For RGB: 32x32x3 (3 channels)	-Filters slide across the image and computes dot products	-Ensures the network can model complex functions	E.G., a 2x2 max pooling layer picks the max value in each 2x2 window			-Used for final classification	Output: vector of size #classes

	-Each filter generates a feature map, output is filters x new height x new width		-Helps with translation invariance and reduces computation			E.G. a 128-unit dense layer followed by 10-unit output layer for digit classification	

Additional items this week:

- Pushed a non-compiled resnet 18 model into the resnet 50 repository for testing (uses torch)
- Began to create my first network in C++, using eigen library for linear algebra

CNN and FFNN Testing in Vitis workflow notes

Generating a public key for the U of Arizona physics server (to clone my personal github repo)

I tried to clone the github repository with my xfile model on it for testing on the remote server, but github.com refused the connection for security. Prior to this, the server had never authenticated successfully to GitHub over SSH, so I generated a public and private key to be able to connect to the repo (push, pull code). The key fingerprint is:

SHA256:lBteV3vc/4ywAUCs9F01X8YjA6iXZ17GiSEbwHi0/Ho mmasenheimer21@gmail.com

The randomart image looks something like this…
(https://github.com/mmasenheimer/resnet50-model was cloned onto the server)

```
|     +=+ ....+.o|
|    .ooo* ..+ B+|
|    ..==.B.= = *|
|     .+oB.* = ..|
|       S.+ =   .|
|        . . + o.|
|       . E  . . o|
|        .       |
|                |
+----[SHA256]-----+
```

The next step was to get the xilinx common image file (downloaded from amd site here) onto the physics server in order to run a QEMU-simulated Linux environment. I used an

scp command to transfer the file, and set up a connection between my computer and the server via the ip address, then transferred the file (about 1.2 gb). Tried breaking up the file into 100mb bits but lost connection each time. Tried resync but I need sudo commands on the server. Tried google drive/dropbox but it only downloaded the HTML, not the actual xilinx file.

AIE-ML Performance Analysis

I found an article off the AMD technical information portal about how to conduct performance analysis through vitis, targeting the VEK 280 board.

https://docs.amd.com/r/2024.1-English/Vitis-Tutorials-AI-Engine-Development/AIE-ML-Performance-Analysis

Once I can get the common image installed on the server, run:

/**Common Images Dir**/xilinx-versal-common-v2024.1/environment-setup-cortexa72-cortexa53-xilinx-linux

which will set up the sdktargetsysroot and cxx variables. If we cannot source into the script, run:

/**Common Images Dir**/xilinx-versal-common-v2024.1/sdk.sh.

Now we set environment variables:

```
Set up your ROOTFS and IMAGE to point to the rootfs.ext4 and Image files located in the /**Common Images Dir**/xilinx-versal-common-v2024.1 directory.
```

Set up PLATFORM_REPO_PATHS environment variable to:

$XILINX_VITIS/base_platforms.

Types of simulations

Tool	Description	Used for .xmodel
x86simulator	Simulation for AI engine graphs only	No
v++ -t sw_emu	Software emulation of custom kernels in PL	No
V++ -t hw_emu	Full system hardware emulation (AI engine + pl + ps)	Not really practical for .xmodel
vitis_analyzer	Timeline + profiling tool after emulation or real hardware	No
Vitis AI Runtime (VART)	High level simulation of .xmodel on x86 CPU	Yes

Michael Masenheimer- Testing Resnet architecture + My own model

My model uses EfficientNet B0 CNN architecture

Resnet 50

(Vitis AI analyzer stats)

Thread	Event	Start (ms)	End (ms)
host:pre_proc	preprocess frame	0.0	5.0
host:DPU_enqueue	enqueue model to DPU	5.0	5.1
device:DPU0	DPU inference (layers)	5.1	13.0
host:pre_proc	prepare next frame	6.0	11.0
device:DPU0	DPU inference (next frame)	13.0	21.5

Metric	Value	Notes / Source
Inference Time	~3 milliseconds per image	Need to do more tests to narrow the inference times down + test different resnet models
Throughput	~357 images/sec	Could differ on the Engine
Memory Usage	~98-100 MB	Model size + activations
Number of Parameters	~25.6 million	Standard ResNet-50 architecture

---

EfficientNet B0

(Vitis AI analyzer stats)

DPU Layer-wise Profiling

Layer Name Kernel Type Exec Time (ms) Thread Usage (%)

---

conv2d_0 CONV 1.2 75

batch_norm_0 BATCHNORM 0.3 20

mbconv_3 DEPTHWISE 3.4 90

fc_1 FC 0.8 60

Total Exec Time: 7.7 ms

DPU Utilization: 85%

Inference Time: 1.02 milliseconds per image

Throughput: 978 inferences per second

Memory Usage: 27MB

Number of Parameters: 5.27 million.

This is a header (2 ##)

This is a subheader (3 ###)

And another! (4 ####)

Test

Week of 7/14

- Rough draft of a FFNN in c++

https://github.com/mmasenheimer/neural-networks-for-LHC under “FFNN”