IMPORTANT:

- do not modify the directory structure and the file names in the GIT repository,

- to build and run your code **source "/opt/cad/scripts/tools_env.sh"**;
- DO NOT commit binaries, output files or log files;
- DO NOT commit modifications to any `Makefile` and `test.sh` .

1 Convolutional Layers

Recall the DWARF-7 convolutional neural network (CNN) This project
focuses on the convolutional layer for which we provide a baseline implementation of a hardware accelerator with an ESP-compatible interface. Your task is to perform a design-space exploration (DSE) on the provided accelerator, integrate the accelerator in ESP and develop an application to execute the CNN inference pass on FPGA.

2. Part B. December 12th - December 22nd

-

- Material from Stanford's Spring 2017 class [1] *CS231n: Convolutional Neural Networks for Visual Recognition.*
  - Notes (highly recommended): `http://cs231n.github.io/convolutional-networks/`

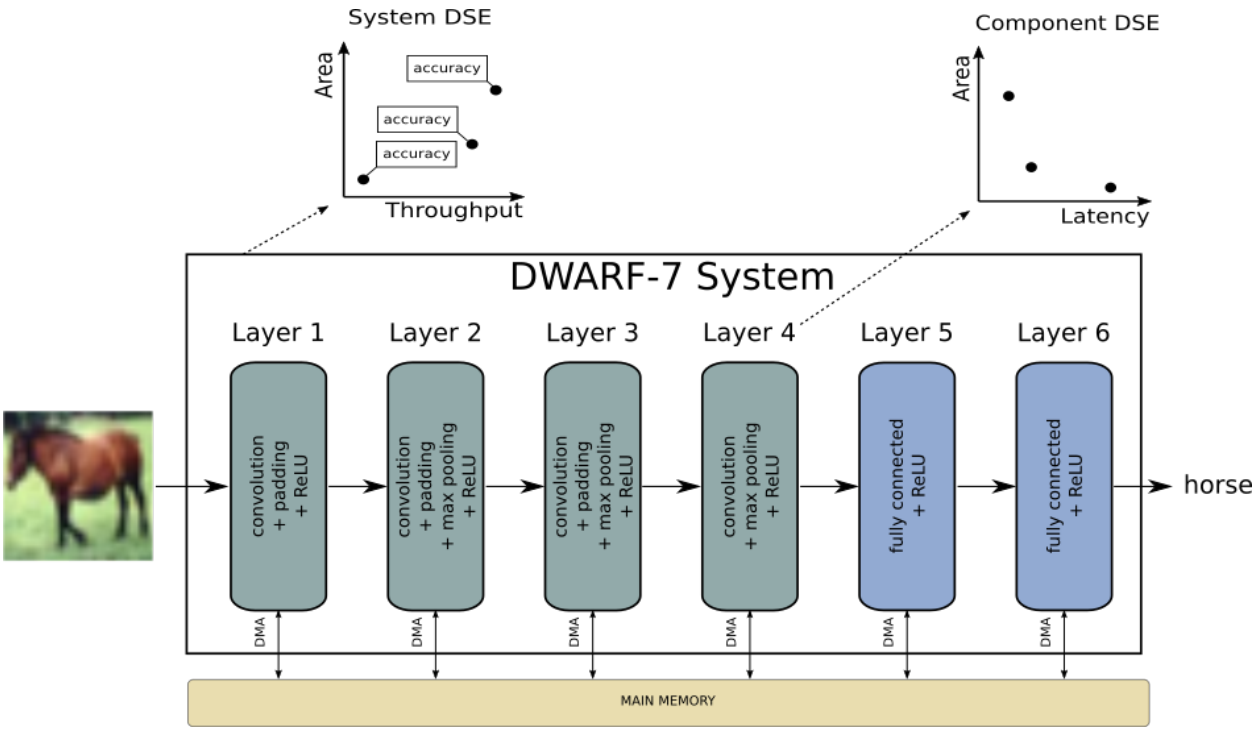

Figure 1: *DWARF-7 system for the project with the layers. The design-space exploration will be performed both at component and system level. The performance metrics for the exploration are shown in the two charts.*

  - Slides: `http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture5.pdf`
  - Lecture video: `https://www.youtube.com/watch?v=bNb2fEVKeEo&feature=youtu.be`
- Beginner's overview: part 1 [2] and part 2 [3].

2 Baseline Micro-Architecture

hown in Figure 1, the DWARF-7 network is divided into 6
layers. The system can process data in a streaming fashion because it can work as a pipeline at the granularity of a layer. For example while Layer 2 is working on the inference of a dog image, Layer 1 could be working on the image of a cat and Layer 3 on one of a automobile.
For Part A, Each student is assigned ***target LAYER 2 - convolutional layer***.
We provide a baseline accelerator implementation that works for all the convolutional layers. The infrastructure is the same with the exception of the accelerator interface. We transitioned the interface of our
hardware accelerator implementation from AXI-compliant to the custom in-house ESP interface, which follows the same principles as AXI but adopts a distinct naming convention and API. This modification is aimed at facilitating the integration of the accelerator into an ESP-based System-on-Chip (SoC), a process that will be the focused of Part B of the project.
The provided accelerator merges together multiple functionalities: accumulation, activation, padding (also called resize) and pooling. Notice that only some convolutional layers in DWARF-7 require all of these functionalities.

4 Part B - SoC Integration and FPGA Deployment

In Part B, you will integrate the accelerator designed in Part A into a complete SoC, alongside open-source processors, memory, and IO, interconnected with a network-on-chip (NoC). You will explore the broad space of the SoC design by prototyping the full SoCs on FPGA and evaluating the runtime of an inference application and the total resources used by the SoC. While this may sound like a daunting task, this whole process is made easy by *ESP* [1]. ESP is a platform for agile SoC design developed by the SLD group. ESP automates the integration of accelerators into a complete SoC, provides a push-button flow for generating FPGA bitstreams of SoCs, and provides software that simplifies the development of applications invoking custom accelerators. In Part B, you will use ESP, assisted by its graphical user interface (GUI), to perform system-level design space exploration and try to produce a Pareto-optimal SoC in terms of performance and area.

4.1 Project Structure

Important.

You are provided with 2 ESP *SoC design folders* in Part B for your 2 SoC implementations. The folders are called `xilinx-vc707-xc7vx485t-small` and `xilinx-vc707-xc7vx485t-fast`. The prefix is the name of the FPGA board for which you will be deploying your synthesized FPGA designs. The following are the files you will find in each design folder.

- `conv_layer.h` - Header file for the inference application.
- `data_ariane` - Dwarf model binary for Ariane. `DO NOT MODIFY`.

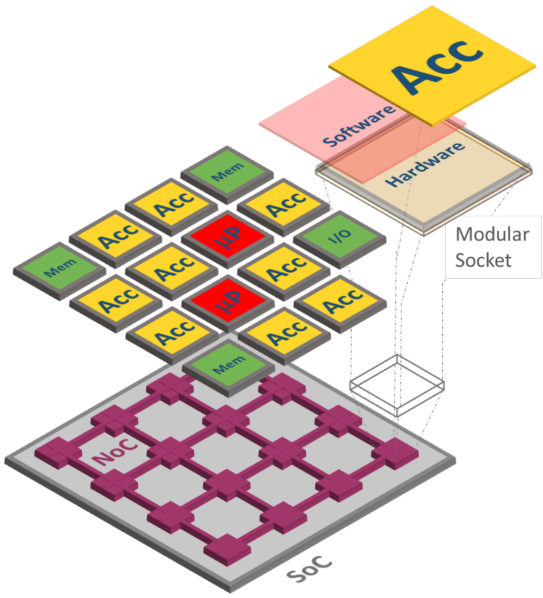

Figure 3: *Many accelerators integrated into a NoC-based SoC with microprocessor (CPU) and memory tiles.*

- `data_leon3` - Dwarf model binary for Leon3. `DO NOT MODIFY`.
- `esp_xilinc-vc707-xc7vx485t_defconfig` - Default ESP SoC configuration. `DO NOT MODIFY`.
- `get_area.sh` - Obtain the area of the SoC from Vivado reports. `DO NOT MODIFY`.
- `load_model_ariane.sh` - Load the model for Ariane into the FPGA's DRAM. `DO NOT MODIFY`.
- `load_model_leon3.sh` - Load the model for Leon3 into the FPGA's DRAM. `DO NOT MODIFY`.
- `Makefile` - `DO NOT MODIFY`.
- `systest.c` - Main C file for the inference application.
- `testbench.vhd` - testbench of the SoC design. `DO NOT MODIFY`.
- `top.vhd` - top level RTL of the SoC design. `DO NOT MODIFY`.

You will also be provided with an empty `tech` folder. This is where you will install your generated accelerator instances to make them visible to the ESP GUI. Your should not modify the contents of this folder and only use the 'make install*' targets (to be described later) to install your accelerator implementations in this folder.

4.2 The ESP GUI and DSE Tips

Figure 4 shows the ESP GUI and labels various options you can change to conduct the design space exploration.

1. **Accelerator Implementation.** By selecting it from the dropdown menu, you can instantiate your accelerator in the SoC. The `Impl` dropdown menu allows you to select the specific implementation of your accelerator that you wish to instantiate (e.g. small, medium, or fast). **DSE Tip:** complex system interactions will now affect the performance of your accelerator, and you may not see the same exact trends as Part A. For example, a heavily optimized accelerator may not perform substantially better than a less optimized version if the memory access latency in the SoC is very high.

2. **SoC Layout.** You may change the layout of the SoC as you wish, so long that it contains 1 CPU tile, 1 memory tile, 1 IO tile, and 1 or more accelerator tile. **DSE Tip:** the interactions between the CPU and the memory and between the accelerator and the memory are an important factor for performance.

Figure 4: *The ESP graphical user interface with various configuration options highlighted.*

3. **CPU Core.** You may select from 2 CPU cores available in ESP: the 64-bit RISC-V Ariane core and the 32-bit RISC-V Leon3 core. There is a tradeoff between the performance of the core and the area it occupies. **DSE Tip:** much of the inference is executed in software, so the performance of the CPU is relevant, but the areas also vary significantly.

4. **Configure Cache Hierarchy.** Here you can enable or disable the ESP cache hierarchy, i.e. the level-2 (L2) cache in the CPU tile and the last-level cache (LLC) in the Memory tile, and configure its size. **DSE Tip:** using caches can improve performance by exploiting locality and enabling on-chip data sharing between CPUs and accelerators. However, using the caches will increase the amount of memory resources of the SoC (to a degree depending on the configured size).

5. **Enable Accelerator Private Cache.** You may equip your accelerator with a private ESP L2 cache to allow it to execute fully-coherently with the CPU core. The ESP cache hierarchy must be enabled to unlock this option. **DSE Tip:** using the fully-coherent mode typically provides benefits for small workloads that fit within the private cache. Adding an L2 cache will increase the SoC's area.

6. **Coherence NoC Parallelism.** Here you may change the bandwidth of the 3 ESP NoC planes that are used for cohernce messages. This must be less than or equal to the cache line size and at least as large as the CPU size (32 for Leon3/Ibex, 64 for Ariane). **DSE Tip:** the coherence planes are used for communication between the CPU and Memory and from the Accelerator to Memory only when the fully-coherent mode is selected.

7. **Direct-Memory Access (DMA) NoC Parallelism.** Here you may change the bandwidth of the 2 ESP NoC planes that are used for DMA depending on other attributes of your SoC. For Leon3, this should remain as 32 bits. For Ariane, it must be at least 64 bits and can be further increased by powers of 2, but must be less than or equal to the size of the cache line. **The DMA width of your accelerator design must match this value. DSE Tip:** the DMA planes are used for communication between the accelerator and the memory for the 3 DMA-based coherence modes. Increasing the bandwidth will move data into the accelerator faster at the price of increased NoC logic.

4.3 Design and Prototyping Flow

The steps to generate and deploy an ESP SoC on FPGA are as follows.

1. **Generate and Install Accelerators.** You can generate your accelerator implementations as you did in Part A by running HLS. Once this is done, you can run `make install-small`, `make install-medium`, or `make install-fast` to copy the corresponding generated implementations to your `tech` folder, which is accessible by ESP. You may run `make install` subsequently if you wish to change the accelerator implementations available to ESP.

2. **Configure SoC and Generate Bitstream.** Launch the ESP GUI with the command `make esp-xconfig`. Once you have configured the SoC to your desire, click the *Generate SoC config* button, which will generate all of the RTL needed for the SoC. Generated files will go in a directory called `socgen`, which you should not modify. To generate the FPGA bitstream, run `make vivado-syn`.

3. **Compile Software.** The application is located in `systest.c`. There are a few TODOs in the application that you should modify based on the configuration of your SoC. Namely, these are the index of the CPU tile, the selected coherence mode, and the fixed-point precision of the accelerator data. You may also modify the application as you wish to run additional or fewer layers on the accelerator. The coherence mode can be selected from the application by changing the value written to the coherence register to one of the following values: `ACC_COH_NONE`, `ACC_COH_LLC`, `ACC_COH_RECALL`, or `ACC_COH_FULL`. The application is compiled with the command `make soft`. This will cross-compile the application for the core that you currently have selected from the ESP GUI.

4. **FPGA Programming.** To upload your generated bitstream to the FPGA, execute `make fpga-program`. From another shell, you can open a UART connection to the FPGA to observe the outputs of your program with the `make UART` command.

5. **Loading the Model.** Next, you should upload the Dwarf model to the FPGA's DRAM with either the `load_model_leon3.sh` or `load_model_ariane.sh` script depending on which core you are using. You only need to load the model once each time you program the FPGA with your biststream.

   The load model scripts load the binary files containing the floating point model to an array at a predefined address. Then, other functions copy this data to other intermediate arrays. Finally, data that needs to be accessed by the accelerator converted to a fixed-point format and copied to the `mem`. The accelerator is only able to access (read or write) data within this array.

   `NB.` If you increase the DMA width of the accelerator design or reduce the precision such that there are multiple accelerator data words per DMA beat, you may need to change the way in which the model is copied from the intermediate storage to the accelerator buffer **mem**. Particularly, you should be careful to arrange the order of data within a beat in the format that your accelerator expects (e.g. `3210` vs. `0123` for a configuration with 4 words per beat).

6. **Running the Inference.** Finally, you can run your application with the command `make fpga-run`. If you wish to change the software before running again, you should execute `make soft` before running again. You may need to execute `make fpga-program` again before subsequent calls to `make fpga-run`.

7. **Closing Your Session.** At the end of your FPGA session, ***you must close your make uart session***.** To do this, press `Ctrl+a` then `z` to open **minicom** options. Then, press `q` to quit and `enter` to select "yes".

8. **Extracting Results.** Your SoC will be evaluated on latency, accuracy, and area. Hence, the Pareto curve for Part B will be a 3-dimensional curve. Latency and accuracy from running on FPGA will be printed from the inference application. The SoC's resource utilization (i.e. area) will be reported in a file generated by Vivado. Similarly to Part A, in order to see the area for your SoC, run the provided `get_area.sh` script, which will put the results in an `area_small.log` or `area_fast.log`.

You should only use Steps 4-7 during a timeslot TO BE DISCUSSED

4.4 Commiting and Your Work

We have provided you with a `commit.sh` script in the top folder of your repository. This script will commit and push the following files within each design folder, which we will use to evaluate your project. You will be prompted to enter a commit message before the commit is pushed.

1. `socgen/esp/.esp_config` - your chosen SoC configuration.
2. `conv_layer.h` and `systest.c` - your software application.
3. `vivado/esp-xilinx-vc707-xc7vx485t.runs/impl_1/top.bit` - your generated FPGA bitstream.

4. `vivado/esp-xilinx-vc707-xc7vx485t.runs/impl_1/top_utilization_placed.rpt` - Vivado utilization reports for your SoC.

5 Evaluation

5.1.2 Part B

By the deadline of December 22nd at 11.59pm, you are required to submit two SoC designs as explained in Section 4. Your designs will be evaluated based on the results Each SoC design
(SMALL, FAST) for which you submitted a Pareto optimal design at the full-system SoC level. No
Pareto optimal design will get a evaluated proportional to the distance of their design that is closest to the Pareto curve.

5.2 Optional Work

-
-

5.2.2 Optional Work for Part B

Intermediate self-progress: Submit each of two SoC design submitted by 11.59 pm on December 17th. The two designs are not evaluated in comparison with the designs of others However, the two designs must have distinct ESP SoC configurations.

6.1 Make Targets and Design Flow

In this section we describe all the useful Makefile targets that we provide. In the targets, `<cfg>` can be either fast, medium or small, and `<image>` can be any of the images in the `data` folder. You should run the targets from the `hls/syn/` directory.

- `$ make <image>-<cfg>-exe-syn-fp`
  Behavioral simulation with native floating-point numbers. This simulation is faster than the one with fixed point numbers. The results for this target are stored in: `test.txt`, they have to match those generated by the programmer's view for that specific layer and image.

- `$ make <image>-<cfg>-exe-syn`
  Behavioral simulation with SystemC fixed-point numbers.

- `$ make <image>-<cfg>-exe-syn-tlm`
  Behavioral simulation with SystemC fixed-point numbers leveraging TLM.

- `$ make <image>-accelerated-<cfg>-exe-syn`
  Behavioral simulation with SystemC fixed-point numbers. Since the Verilog simulation is too slow, we provide this *ACCELERATED* target to compare against an accelerated Verilog simulations. It executes only part of the inference and saves the partial output in `accelerated_test_syn.txt`. Similarly to the native simulation, consider renaming and moving the output txt file.

- `$ make hls-<cfg>`
  Perform HLS for one of the micro-architectures (i.e. one `hls_config`).

- `$ make hls-<cfg>-gui`
  Perform HLS for one of the micro-architectures (i.e. one `hls_config`) deploying Catapult HLS GUI.

- `$ make <image>-accelerated-<cfg>-sim`
  RTL simulation of the Verilog accelerator generated by Catapult HLS. The regular Verilog simulation would be too slow, so we provide this *ACCELERATED* target. The partial output is stored in `accelerated_test_syn.txt`. These results must match those of the accelerated fixed point behavioral simulation. The `make hls_<cfg>` needs to be run before running any RTL simulation. Again, you may want to rename and move the txt file.

- `$ make <image>-accelerated-<cfg>-sim-gui`
  This target is the same as above, but it opens the GUI for you, so you can debug more easily by looking at the waveforms.

- `$ make <image>-<cfg>-sim`
  RTL simulation of the Verilog accelerator generated by Catapult HLS. The `make hls_<cfg>` needs to be run before running any RTL simulation.

- `$ make <image>-<cfg>-sim-gui`
  This target is the same as above, but it opens the GUI for you, so you can debug more easily by looking at the waveforms.

- `$ make test-SMALL, make test-MEDIUM, make test-FAST`
  This target executes the `test.sh` script, the regular native simulation, the accelerated behavioral fixed-point simulation and accelerated Verilog simulation. **Note:** Use `<cfg>` in CAPS case for this target only as shown above. All of the simulations run on the cat image. It first runs the native behavioral simulation and validates it against the golden outputs in `/golden_fp_tests`. This also runs the target `make accuracy_comp_<cfg>` that works as described below. Then, it executes HLS and then accelerated fixed point simulations of both behavioral and RTL, and it compares their results. We will use this target to test the correctness of your submitted designs.

- `$ make accuracy_comp_<cfg>`
  Classification test on six the images by using a DWARF-7 where your layer is executed with your choice of fixed point precision in TLM and the other layers are executed in their original floating point precision. This target evaluates the component-level accuracy, which has to be at least 50% for the design to be valid. This test is done by executing the native simulation. We will use this target to test the component-level accuracy of your submitted designs.

- `$ IMAGE=<image> TARGET_LAYER=<layer> make dwarf-run`
  This target is the only one that's executed only under the programmer's view folder (pv) and not under hls/syn. This target executes the programmer's view code and saves the layer's output to `test.txt`.

  Debugging Tips: Note that for most debugging you should be able to run accelerated simulations to help converge at the bugs and validate changes quickly. However in some cases, for changes that you think might have affected code functionality you may refer to the golden fixed point outputs that we have provided in the folder `/hls/syn/golden_accelerated_tests`. This is for the image of a CAT and for FP data widths WL=32, IL=16. These are outputs of functional fixed point simulations from a correct implementation that you can use for verification.

7 Resources

Since Catapult HLS is a powerful industrial CAD tool, you may find it helpful to refer to its documentation in addition to the in-class tutorials we gave on its use. You may find the documentation in `/opt/cad/catapult/shared/pdfdocs/` on the `socp0*` servers. The PDF you are likely to find most helpful is:

- `/opt/cad/catapult/shared/pdfdocs/catapult_useref.pdf`

You may download it to your local machine using `rsync`, `scp` or similar file-transfer methods for easier viewing.

References

[1] https://www.cs.toronto.edu/~frossard/post/vgg16/

[2] https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/

[3] https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/