Project
CSEE 4868: SoC Platforms

November 18th, 2024

IMPORTANT: To get full credit:

- do not modify the directory structure and the file names in the GIT repository,
- make sure the code runs on the `socp0X.cs.columbia.edu` servers,
- 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<target_layer>.sh` .

You will carry out the project individually. A repository is assigned to each student and can be cloned by executing:

Part A: `git clone git@soc.cs.columbia.edu:socp2024/prj-UNI.git`
Part B: `git clone git@soc.cs.columbia.edu:socp2024/prjB-UNI.git`

1 Convolutional Layers

Recall the DWARF-7 convolutional neural network (CNN) from the homework assignments (Figure 1). This project focuses on the convolutional layer from Homework #2 and Homework #4, 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.
The project will be divided into two parts with the following dates. Part A repositories are available now. Additional material will be made available at the start date for Part B. Details on deadlines and grading can be found in Section 5.

1. Part A: November 18th - December 7th
2. Part B: December 10th - December 20th

For part A, every student will complete a *synthesizeable* design and implementation of an accelerator of one of the four convolutional layers. In doing so, each students will compete with the other students in the class who have been assigned the same layer.
At the end of part A, your grade will depend on the distance between your implementations and those in the Pareto curve derived by considering the implementations of the students who accelerated the same layer. Additionally, you will have many opportunities for collecting extra-credit points throughout the course of the project. We suggest to carefully study the `convolution_compute()` function of the *Mojo* application and see how the convolution is implemented. Adding prints in the programmer's view can help you find out the values of the parameters passed to each layer.
Furthermore, understanding the general idea a convolutional layer is important. In addition to the lecture slides, the following are some recommended references on CNNs:

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

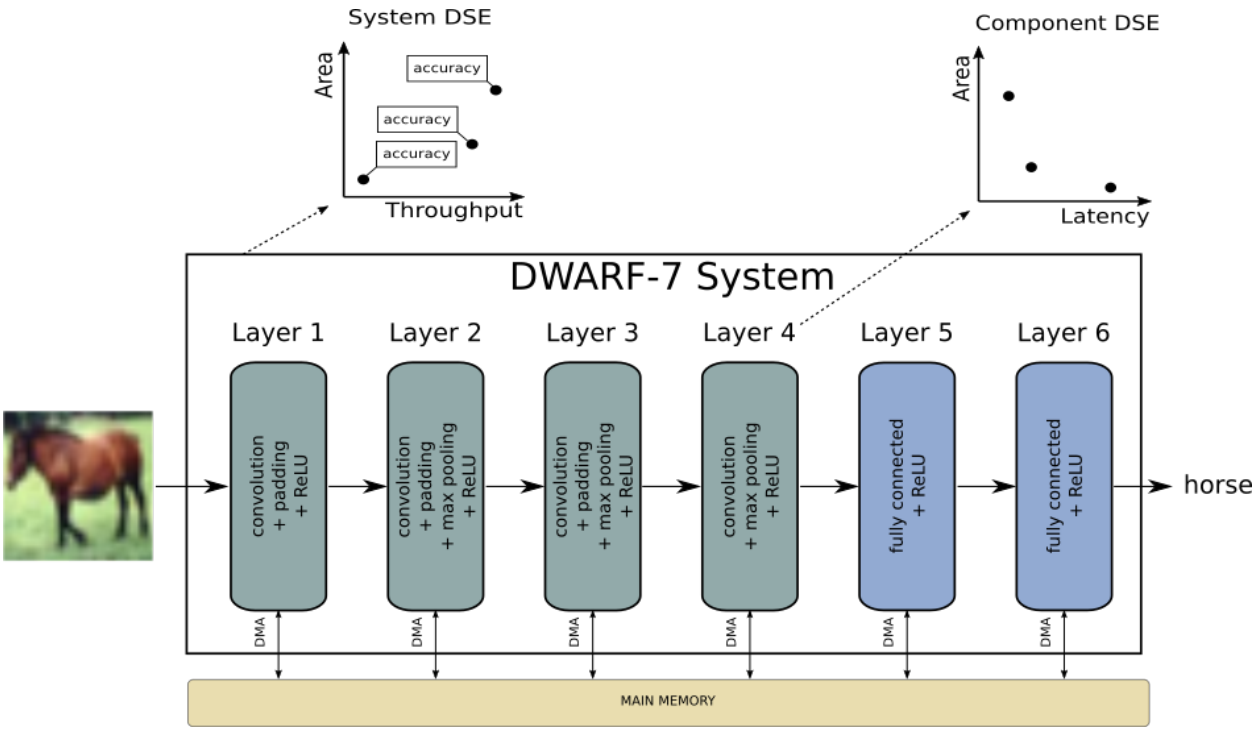

Figure 1: *DWARF-7 system for the project with the layers. The design-space exploration will be performed both at component and system level. The performance metrics for the exploration are shown in the two charts.*

2 Baseline Micro-Architecture

As known from previous homework assignments and shown in Figure 1, the DWARF-7 CNN 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 one of the four convolutional layers.
The layer assignments will be communicated by November 19th.
We provide a baseline accelerator implementation that works for all the convolutional layers. The infrastructure is the same as that of Homework #4 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.

3 Part A. Design Space Exploration of the Convolutional Layer Accelerator

The main goal of the project is to perform a **competitive** design-space exploration for the convolutional Layer 2 of the system shown in Fig. 1. You should start from the baseline micro-architecture that we provide and explore a wide variety techniques for design-space exploration with the help of high-level synthesis (HLS). We purposely provide a sub-optimal baseline that has a lot of room for improvement.
You are required to obtain at least 3 Pareto-optimal micro-architectures that are named FAST, MEDIUM, and SMALL, respectively. These have been specified as targets for the HLS in the file `build_prj.tcl`, where you can add synthesis

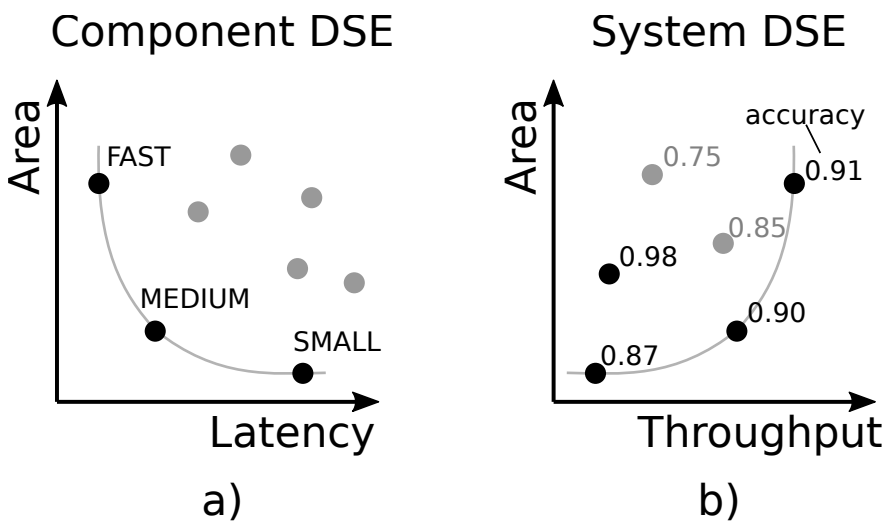

Figure 2: *a) Example of component-level Pareto curve. Here the three darker dots are Pareto optimal. b) Example of system-level Pareto curve. Notice that the micro-architecture labeled with 0.98 accuracy is Pareto optimal because no other design has better accuracy.*

attributes to each *hls_config*. Even though you should design 3 different micro-architectures, you are asked to commit a single SystemC module where C preprocessor directives are used to switch between the different implementations depending on the target:

```
#if defined(SMALL)
// Source code specific to the SMALL target...
#elif defined(MEDIUM)
// Source code specific to the MEDIUM target...
#elif defined(FAST)
// Source code specific to the FAST target...
#endif
```

As part of the optional work described in Section 5.2, you are invited to keep improving your own Pareto frontier, by either advancing your Pareto optimal designs or by adding new ones, throughout the duration of the project. Eventually, you should aim at having your designs on the Pareto curve of the class for the specific convolution layer assigned to you. Improving your designs on a regular basis, daily if possible, helps you to achieve this goal.
Daily, starting from Wednesday, November 20th at 11.59 pm, we are going to automatically test your submitted designs. We will consider the last `git` commit of the day. The performance of the valid designs will be published daily on the *project website*: http://alba.cs.columbia.edu:3838/socp2024/

The website allows you to observe the anonymized results of the other students.

In your repository you will find your ID in the `README.md` file. On the website, we will use your ID to label your designs. Since students will compete for some parts of the project, you are discouraged from sharing your ID with others.

3.1 Performance metrics

Part A is based on two metrics: area and effective latency. They are evaluated on a single instance of your accelerator. Each of your valid micro-architectures will appear as a point in a *Component DSE* chart on the website as shown in Figure 2. A micro-architecture is valid if it meets the accuracy constraint explained below.

Accuracy and correctness. There are three requirements for an implementation to be valid:

- *Functional correctness.* The results of the behavioral SystemC simulation with `float` data types must match exactly the results of the programmer's view.
- *HLS-generated RTL correctness.* The results of the behavioral SystemC simulation with fixed-point data types must match the results of the RTL-SystemC co-simulation of the HLS-generated Verilog.

- *Accuracy.* The component accuracy must be at least 50% for the design to be valid. The accuracy test executes inference on 6 images and at least 3 have to be classified correctly. The test executes the behavioral simulation with your chosen fixed-point precision for the layer and the original floating-point precision for the other layers.

Area. The area of your accelerator is calculated in terms of FPGA-resource utilization on the target *Xilinx VC707* FPGA board. Specifically, the area is calculated based on the average of the percentage utilization of the main FPGA resources: BRAM (RAM18, RAM36), LUTs, FFs, Latches and DSPs (Mults).

$$A\_layer = 100 * (\ \#\text{RAM18} \cdot 0.00023148 + \#\text{RAM36} \cdot 0.00046296 + \#\text{LUTs} \cdot 0.00000085 + \\ + \#\text{DSPs} \cdot 0.0001462 + \#\text{FFs} \cdot 0.00000042 + \#\text{LATCHES} \cdot 0.00000042\ )/6$$

Be aware that by increasing the number of ports and access patterns of the private local memory (PLM) of the accelerator the number of BRAMs is going to grow significantly. Note that you cannot exceed the resources available on the FPGA. You can calculate the area for each microarchitecture by running the script `hls/syn/get_area.sh` and checking the result in `hls/syn/area_<cfg>.log`.

Effective Latency. The effective latency L_{Eff} (*ns*) of a layer corresponds to the amount of time needed to process completely a given input image across the layer. The effective latency of a layer is reported at the end of the execution of the RTL simulation. You can calculate the effective latency for each microarchitecture by running the script `hls/syn/get_latency.sh` and checking the result in `hls/synz/latency_<cfg>.log`. **Note** : Please make sure you have successfully run make HLS i.e. `make hls-<cfg>` and RTL Simulation i.e. `make <image>-<cfg>-sim` before running this script.

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 Columbia 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

4.1.1 ESP SoC Design Folders

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`.
- `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`.

[1] esp.cs.columbia.edu

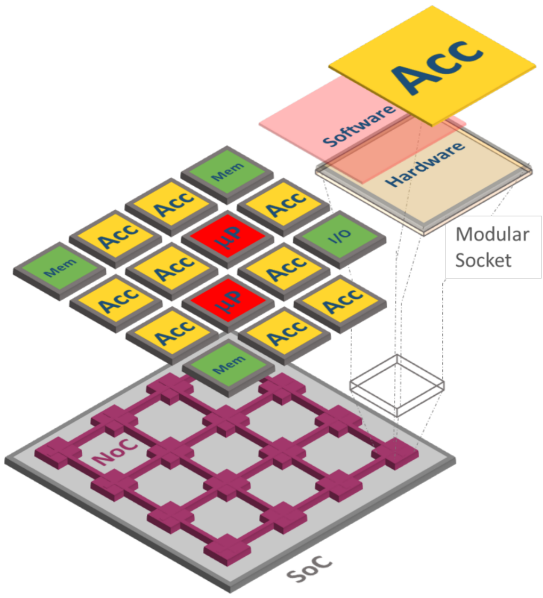

Figure 3: *Many accelerators integrated into a NoC-based SoC with microprocessor (CPU) and memory tiles.*

- `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`.

4.1.2 System Accuracy Simulator

In the `accuracy` directory, we have provided a simple SystemC simulator that allows you to check the accuracy of your target system before generating an FPGA bitstream and deploying software on it. We encourage you to make use of this simulator to 1) check that your chosen accelerators give an overall accuracy that is acceptable to you, 2) know the system's expected accuracy in order to aid in verifying and debugging your baremetal application.

To use the simulator, you should navigate into the `accuracy` folder and modify the file `precision_test_fpdata.hpp` file. You must first set the `LAYER_X_FIXED_POINT` define to 1 for any accelerators you intend to accelerate; other layers will be run with a floating-point implementation, as is the case in the baremetal application. Then, for any accelerated layers, you should define the widths of the fixed-point data types (e.g. `LAYER_FPDATA_WL_X`) to match those of your chosen accelerators. You can obtain this information from the project website. Finally, to run the simulator, you can execute `make accuracy`, which will report the accuracy from inference on 10 images.

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 asccelerator that you wish to instantiate (e.g. small, medium, or fast).

   NOTE: To accelerate any layer, you may use the default accelerator implementation that is labeled with the team ID `z` for each layer (i.e. `1z`, `2z`, `3z`, `4z`). To accelerate the layer that was assigned to you in Part A, besides the default accelerator implementation, you can only use one of your three designs (e.g. small, medium, or fast) that correspond to your last commit in your repository. For each of the other

three convolutional layers (i.e. the layers that were not assigned to you in Part A). you are allowed to use any of the designs implemented by the other teams. Your choice of these designs should be based on:

- **An analysis of their performance and characteristics as reported on the *project website*: http://alba.cs.columbia.edu:3838/socp2024/**
- **Considering the constraint that the DMA width of each accelerator imposes on the NoC parallelism (the DMA widths must match, as discussed also below);**
- **Considering the impact that the fixed-point precision of a given accelerator implementation might have on the accuracy of the accuracy of the overall design.**
- **To aid in the integration of other teams' accelerators, we have made available each design's corresponding testbench (`system.cpp`) in the folder `/opt/tb` with the name `system_<team_id>.cpp`. In particular, you should pay attention to how the testbench prepares the accelerator's input data and processes the output data. If the accelerator designer has changed the layout of data in memory to use a different datatype or DMA width, then this will require you to change the baremetal application accordingly.**

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. Hence, the smallest possible SoC layout is a 2x2 grid with these three tiles and one empty tile.

   DSE Tip: the interactions between the CPU and the memory and between the accelerator and the memory are an important factor for performance.

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. The private cache can only be enabled in accelerator tiles if the DMA width matches the CPU architecture size (32 for Leon3/Ibex, 64 for Ariane) **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 designs 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.

8. **Unusable GUI Features.** In the GUI, you cannot modify the Debug Link and the Data Allocation Strategy.

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

4.3 Design and Prototyping Flow

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

1. **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`. **Each team should only run one make vivado-syn process at a time.**

2. **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. You should pay careful attention to the data types used by the accelerators and to the fact that the application will need to be properly modified when a mix of accelerators with different data types is used. 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.

3. **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.

4. **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).

5. **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`.

6. **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".

7. **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 in which you have reserved the FPGA on the Google Sheet. Violations of this rule may result in point deductions at the instructor's discretion.

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. For two-student teams, only one student should commit and push the designs – we will grade from the repo you push to.

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 Required Work (600 Points)

5.1.1 Part A (390 points)

By the deadline of December 7th at 11.59pm, you are required to submit the design of three micro-architectures of the accelerator for the assigned layer, as explained in Section 3. Your designs will be evaluated based on the results of the whole class as follows: 130 points for each micro-architecture (SMALL, MEDIUM, FAST) for which you submitted a Pareto optimal design at component level. Students with no Pareto optimal design will get a score proportional to the distance of their design that is closest to the Pareto curve.

5.1.2 Part B (210 points)

By the deadline of December 20th 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 of the whole class as follows: 105 points for each SoC design (SMALL, FAST) for which you submitted a Pareto optimal design at the full-system SoC level. Students with no Pareto optimal design will get a score proportional to the distance of their design that is closest to the Pareto curve.
For each instance of your accelerator design that is used by another team you will get 5 points.

5.2 Optional Work for Extra Credit

During both part of the project there is the opportunity to complete optional work for extra credit. This work is based solely on improving your own design.

5.2.1 Optional Work for Part A (Up to 152 Points)

Every day from November 20th to December 7th, a script will automatically evaluate all the designs that have been submitted on or before 11.59pm. The new valid designs will be automatically plotted on the charts on the website. This offers the opportunity for optional work that is highly encouraged and will be evaluated as follows:

- Daily self-progress: 90 points. 5 points per day if you improve your own Pareto frontier at the component level (Nov. 20th to Dec. 7th included, that is 18 days) and submit your design by 11.59 pm. Your own Pareto frontier is considered *improved* if at least one among your new designs is better than an older design by at least 5% either in terms of area or effective latency.
- Pareto optimal at component level on Nov. 25th and Dec 2nd: 31 points per date. On each of these two dates the points are assigned if at least one of your design is Pareto optimal with respect to those of the other students. For each date, these points are assigned based on all the designs submitted on or before 11.59pm.

NOTE: The website will be available starting from Nov 20th. Each design should pass the validation in order to be considered for credits. Designs that do not pass the validation will not receive credits and will not be plotted on the website.

5.2.2 Optional Work for Part B (Up to 100 Points)

Intermediate self-progress: 50 points for each of two SoC design submitted by 11.59 pm on December 15th. The two designs are not evaluated in comparison with the designs of other students. However, the two designs must have distinct ESP SoC configurations.
For each instance of your accelerator design that is used by another team you will get 5 points.

6 Suggestions for Design-Space Exploration

Here are some suggestions of micro-architectural choices or synthesis options for the design-space exploration.

- **Refactor the source code**: There may be opportunities for optimization that you can explore by modifying the source code with respect to the provided baseline. The functions and the overall structure of the accelerator can be re-arranged or re-implemented in a variety of ways. To explore these different options, it may be helpful to allow these implementations to live side by side within the same shared source file. A recommended approach for doing this is to use C preprocessor directives to switch between the different source code implementations.
- **Modify HLS attributes**:
  - **Target clock period**: Catapult HLS uses the clock period to characterize datapath parts and to schedule parts into clock cycles. The number of operations scheduled in a clock cycle depends on the length of the clock period and the time it takes for the operation to complete. If the clock period is longer, more operations are completed within a single clock period. Conversely, if the clock period is shorter (higher clock frequency), Catapult HLS automatically schedules the operations over more clock cycles. Please, note that for some values of clock period Catapult HLS may not be able to instantiate big arithmetic operators.

    Note: In order to specify a different clock period for each micro-architecture, you are required to modify the target clock period both in the file `build_prj.tcl` to inform the HLS tool, and in the testbench system.hpp file, in order to set the simulation clock for RTL simulation. **We will not accept a submission with a mismatch between the clock periods set in these 2 files.**
- **Modify fixed-point precision**: You can modify the precision of the fixed-point data types by changing the number of bits used for the representation. You can do so by editing the constants in `conv_layer_specs.hpp`. Notice that two different fixed-point types are defined there: `W_FPDATA` is meant to be used for the weights, `FPDATA` for everything else. A data type with less bits requires much smaller and faster operators (addition, multiplication) and less local memory space. Therefore, a reduced precision improves both area and latency, but it might deteriorate the accuracy.

  Reducing the fixed-point precision with respect to the software execution means that your accelerator is discarding some bits of the original 32-bits data words. In this case you could reduce the precision already in the user application in order to store less accelerator's data in memory. In this way the accelerator loads less bytes of data overall. A similar approach applies to the accelerator's output.

  Note: In order to make sure the weights use `W_FPDATA` you would have to modify the files in `/src` to change the type of the place-holders of the weights to make use of `W_FPDATA` as they use `FPDATA`.
- **Apply HLS knobs**:
  - **Loop unroll**: The loop unrolling transformation duplicates the body of the loop multiple times to expose additional parallelism that may be available across loop iterations. You can completely unroll, partially unroll or not unroll at all. Remember that higher parallelism means better performance but more area.
  - **Loop pipeline**: Loop pipelining enables one iteration of a loop to begin before the previous iteration has completed, resulting in a higher throughput design while still enabling resource sharing to minimize the required area. This makes it possible to incrementally trade off improved throughput against potentially increased area.
  - Catapult HLS offers many more "knobs" to interact with. The User Manual, the Reference Guide, and the class slides provide further documentation. We suggest to study them and to keep monitoring the web forum for additional hints.
- **Customize the private local memory (PLM)**: You have complete freedom on the organization and the size of the PLM, as long as it fits on the target FPGA.
- **Customize the DMA channels**: It is possible to both change the bitwidth of the DMA or the number of DMA channels. The biggest effect of this kind of optimization is the accelerator's bandwidth to memory. To apply these modification you should act on the memory model in the SystemC testbench. Notice that the minimum data width on the bus is 32 bits. You are highly encouraged to at least increase the DMA bitwidth by modifying the default values set in `hls/syn/dma.mk`, which is expected to provide great speedups. **The allowed values of DMA_WIDTH for each microarchitecture are any power of 2 which is $\geq$ 32 and $\leq$ 512**.

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<target_layer>.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>-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<target_layer>.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 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<target_layer>.txt`.

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/