Embedded AI on FPGA using High Level Synthesis

Final year project for the specialization program in Embedded Systems Engineering

The goal of the project is the full development of a Convolutional Neural Network (CNN) accelerated on a CPU / FPGA system using High Level Synthesis (HLS). HLS, also referred to as C synthesis, electronic system-level (ESL) synthesis, algorithmic synthesis or behavioral synthesis, is an automated design process that inputs a C / C++ like piece of code and outputs a hardware (RTL) accelerator in VHDL, Verilog or SystemC. The CNN we want to accelerate on FPGA using HLS is based on LeNet (figure 1) for handwritten and machine-printed character recognition [1][2] and the FPGA target is a ZedBoard platform featuring a Zynq-7000 SoC (ARM dual core Cortex-A9 / Xilinx FPGA). General descriptions of CNNs are given for example in [3][4][5], you may have to complete these references with your own research.

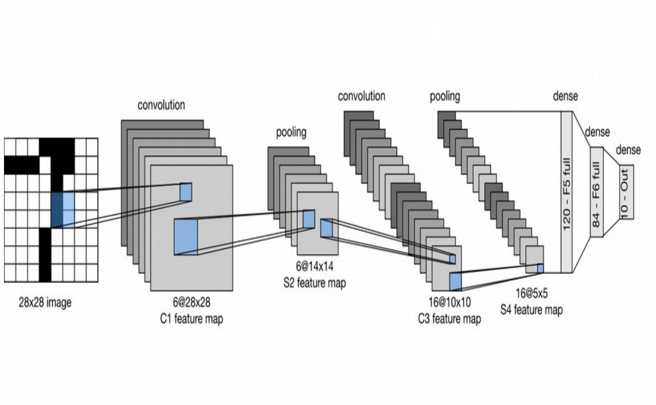

Figure 1. Data flow in LeNet 5. The input is a handwritten digit, the output a probability over 10 possible outcomes.

LeNet architecture is described in [1][2]. An easy starting point to learn LeNet and CNNs in general is to begin with a concrete example. You will first have to install Keras / TensorFlow [6]. Globally speaking for this project, it is recommended to work under a Linux environment, more especially Ubuntu 18.04 (GPU acceleration is not mandatory for this project). You can begin with the tutorial described in [2] to play with a concrete classification example and get familiar with basic CNN concepts.

For our project, we will use a slightly different LeNet model (more suited to hardware acceleration) described in `lenet_keras.py` and located in the template project archive (`HLS_IA.tar.gz`) provided to you at the beginning. A C project is present in directory `FLOAT` of the archive to let you apply LeNet inference on the MNIST dataset using the (floating-point) weights learned from `lenet_keras.py` script execution.
The code present in the archive is operational (`make && ./lenet_cnn_float`), but not fully complete. Only the object files (`.o`) of layer functions (`conv1`, `pool1`, `conv2`, `pool2`, `fc1`, `fc2`) are available to help, test, compare and debug your own code. Your work is precisely to develop the source code for these functions and this code must be supported by a HLS tool. You can check function prototypes in `lenet_cnn_float.h`. A main program `lenet_cnn_float.c` is provided to let you test LeNet correct inference results.

A primary objective is to develop your own working source code for functions `conv1`, `pool1`, `conv2`, `pool2`, `fc1`, `fc2`. This code must be designed to be HLS compliant, more especially with Xilinx Vivado HLS tool (to run Vivado HLS, enter `xilinx_env_vivado18.2` and `vivado_hls` in Ubuntu Terminal).

A complete guide for the development of HLS compliant code (a subset of C language that can be managed by HLS tools) is given for example in [7]. But these rules are rather intuitive and can be summarized and simplified as follow:
- HLS operates at function level

    - One function leads to one accelerator.
    - Explicit function arguments must be expressed (see I/O interface)
    - Several functions can be cascaded to generate bigger accelerators.
    - `main()` function cannot be synthesized
- Memory model
    - Simple variable mapping (scalar → register, array → memory blocks)
    - No dynamic memory allocation, limited support of pointers (use arrays instead)
    - Use arrays of static size (generally mapped to local memory)
- I/O interface
    - Functions arguments define the accelerator interface.
    - The accelerator interface is generally a memory type interface (generic)
    - Accelerator calls implies data transfers to/from local memories (and must be optimized)
- Loops
    - Loops are where most optimization time is spent
    - Use deterministic loops (loops with constant and fixed number of iterations) to promote loop level optimizations (unrolling, pipelining).
- Data types
    - Limited support of structures
    - Use fixed point arithmetic: this means replacing all floating-point operations (double, float) in accelerated functions by fixed-point operations (16, 32 or 64-bit integer)

Key issues for proper development of HLS compliant C code are to understand:
- How C/C++ constructs map to hardware
- How the HLS tool infers the accelerator interface (I/Os): Crucial issue for acceleration efficiency
- How the HLS tool handles array/memory accesses: Critical point for (loop) parallelism exploitation
- How the HLS tool handles loops: Automatic and efficient loop unrolling, pipelining, merging

Any C construct that cannot be turned intuitively into a hardware implementation (e.g. variable → register, array → memory block, int addition → 32-bit adder, etc.) must be discarded. This is generally the case for complex or non-trivial C constructs (pointers, structures, etc.) or operations (e.g. floating-point addition). In general, HLS tools don't support floating point operations because they require the generation of Floating-Point Units (FPU) which are too complex for embedded implementations.

To produce HLS compliant code for LeNet CNN functions to be accelerated, it is advised to start with fully connected layers `fc1`, `fc2` (simplest layers), then pooling `pool1`, `pool2` and convolution layers `conv1`, `conv2` (more complex). For convenience reasons, we start from a floating-point version because the original weights are floating-point values. Then, we can derive a fixed-point version by replacing all float and double computations with integer operations (16, 32 or 64-bit). To derive a fixed-point version (call it `lenet_cnn_fixed_point.c`) from previous floating-point reference code, it will be necessary to analyze different possibilities for integer and decimal parts using different number of bits (e.g. "int 18.13" or "short 9.6", where the latter is a 16-bit fixed-point format made out of 1 bit for the sign, 9 bits for the integer part and 6 bits for the decimal part).

From there, the real RTL design process takes place. You can start first with the distinct synthesis of each function `conv1`, `pool1`, `conv2`, `pool2`, `fc1`, `fc2` separately. Then, RTL refinement and Design Space Exploration (DSE) are carried out to increase the use of FPGA resources (slices, LUTs, BRAMs, DSP blocks) and therefore improve significantly each accelerator performances. This process mainly takes place by exploring various possibilities of data mapping (BRAM, LUTRAM, array partitioning) and loop parallelism (pipelining, unrolling) using HLS pragmas [8]. Here you will check if your code is synthesizable with Vivado HLS and then improve iteratively the results introducing appropriate pragmas progressively in the source code. You will probably have to process each function separately before you are able to synthesize the top level `lenet_cnn` function successfully, while respecting the resource limits of the FPGA device.

Finally, you will use another Xilinx tool called SDSoC to fully and automatically implement a fully executable LeNet C code with its accelerators on a ZedBoard / Linux system, including all necessary drivers and software calls (to run SDSoC, enter `xilinx_env_sdx18.2` and `sdx` in Ubuntu Terminal). The use of SDSoC is rather intuitive and well documented on the web, so we leave it up to you to setup a working project under this design environment.

In the end, you should be able to run a fully operational demo of LeNet automatic character recognition, with hardware acceleration showing minimum performance speedup of 4 and energy efficiency improvement of 100 (against full software execution on ARM dual core Cortex-A9). The final pdf report must include a detailed quantitative analysis of performance and energy efficiency improvements. You have to expose explicitly in the report the measurement and estimation methods used to provide these numbers (use energy delay product for energy efficiency).

Milestones for validation
Part 1: LeNet CNN / C code
1.1 Documentation: understanding CNNs, convolution, pooling and fully connected layers.
1.2 Learning weights with Keras
1.3 HLS compliant C code for Fully Connected layers (`fc1`, `fc2`)
1.4 HLS compliant C code for Pooling layers (`pool1`, `pool2`)
1.5 HLS compliant C code for Convolution layers (`conv1`, `conv2`)
1.6 Fixed point conversion (`conv1`, `pool1`, `conv2`, `pool2`, `fc1`, `fc2, lenet_cnn`)

Part 2: LeNet CNN / High Level Synthesis (Vivado HLS)
2.1 RTL refinement and DSE: Convolution layers (`conv1`, `conv2`)
2.2 RTL refinement and DSE: Pooling layers (`pool1`, `pool2`)
2.3 RTL refinement and DSE: Fully Connected layers (`fc1`, `fc2`)
2.4 Full LeNet CNN synthesis: top level (`lenet_cnn`)

Part 3: LeNet CNN / FPGA implementation (SDSoC, ZedBoard)
3.1 LeNet CNN / original C code running on ZedBoard / Linux
3.2 LeNet CNN / accelerated C code running on ZedBoard / Linux
3.3 Acceleration measurements / energy efficiency improvements
3.4 Demo & reporting (5 page pdf report + video)

Organization, attendance and evaluation
1) The work is organized in a two-person team, each team works independently. Excessive similarity between two projects will suffer penalties in the final evaluation.
2) Each tandem has access to a ZedBoard platform and a PC workstation with all necessary development software (GCC, Vivado, Vivado HLS, SDSoC).
3) Great work autonomy is an essential aspect of the project: each team must develop its solution by himself, with minimum technical help from a professor. Two hours project meetings will take place every two weeks essentially to supervise progress in the project.
4) Final evaluation is based on the global project advancement (with respect to previous milestones) and quality, final completion, validation and demonstration (December 15), and final report (due to December 15). Final reporting will be limited to 5 pages maximum and structured as follows: Introduction / Specification / System overview, High-Level design / Hardware / Software, Test and results, Conclusion (objective critique, possible improvements), References (papers, links, web), Appendices (code eventually, or link to github).

As a reminder, the overall weight for the full project is **7 (/ 30 ECTS)**.

5) Quality of your code will be considered in the final evaluation. Your code must be accessible on Polytech Github server (https://gitlab.polytech.unice.fr/) and clearly structured into directories and sub directories. Special attention will be paid to the naming of files, functions, variables and to the addition of meaningful comments. A main README.md must be present and explain the structure of your code and how to test the full system. It must also contain a copy of your final pdf report. See for example: https://gitlab.polytech.unice.fr/bilavarn/SLAM_Lidar_Lite_V3_Cartographer
6) Presence to all project sessions is mandatory and will be controlled (an attendance sheet must be filled by each student for each project slot). Absence to non-supervised sessions will suffer penalties in the final evaluation.

Online references

[1] LeNet-5 – A Classic CNN Architecture, https://engmrk.com/lenet-5-a-classic-cnn-architecture/

[2] LeNet – Convolutional Neural Network in Python, https://www.pyimagesearch.com/2016/08/01/lenet-convolutional-neural-network-in-python/

[3] Convolutional Neural Networks (CNNs / ConvNets), https://cs231n.github.io/convolutional-networks/

[4] Convolutional Neural Networks, https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/convolutional_neural_networks.html

[5] Looking inside neural nets, https://ml4a.github.io/ml4a/looking_inside_neural_nets/

[6] Ubuntu 18.04: Install TensorFlow and Keras for Deep Learning, https://www.pyimagesearch.com/2019/01/30/ubuntu-18-04-install-tensorflow-and-keras-for-deep-learning/

[7] Coding Considerations (Xilinx), http://home.mit.bme.hu/~szanto/education/vimima15/heterogen_vivado_hls_6.pdf

[8] HLS pragmas (xilinx), https://www.xilinx.com/html_docs/xilinx2019_1/sdaccel_doc/hls-pragmas-okr1504034364623.html