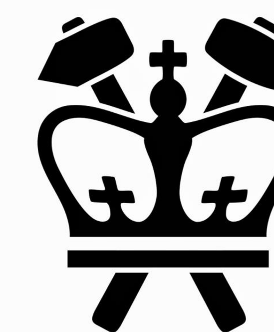
ESP for Machine Learning

Davide Giri, Kuan-Lin Chiu, Giuseppe Di Guglielmo, Paolo Mantovani, and Luca P. Carloni
Department of Computer Science · Columbia University, New York



Why ESP?

Heterogeneous systems are pervasive
Integrating accelerators into a SoC is hard
Doing so in a scalable way is very hard
Keeping the system simple to program is even harder

ESP makes it easy

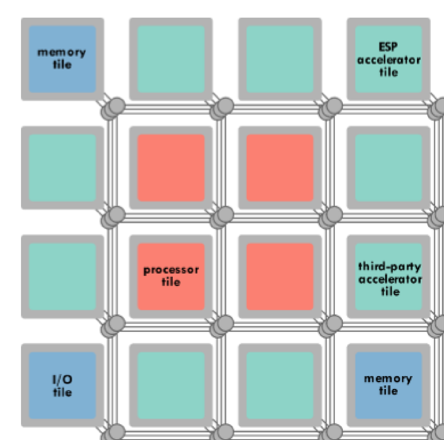
ESP combines a scalable architecture with a flexible methodology
ESP enables several accelerator design flows and takes care of the SW/HW integration

We hope that ESP will serve the OSH community as a Platform to develop software for RISC-V and accelerators for any application domain

ESP Architecture

The ESP architecture implements a distributed system, which is scalable, modular and heterogeneous, giving processors and accelerators similar weight in the SoC

- RISC-V Processors
- Many-Accelerator
- Distributed Memory
- Multi-Plane NoC



Processor Tile

- Processor off-the-shelf
  - RISC-V Ariane (64 bit)
    SPARC V8 Leon3 (32 bit)
  - L1 private cache
- L2 private cache
- IO/IRQ channel
  - Accelerator config. registers, interrupts, flush, UART, ...

Accelerator Tile

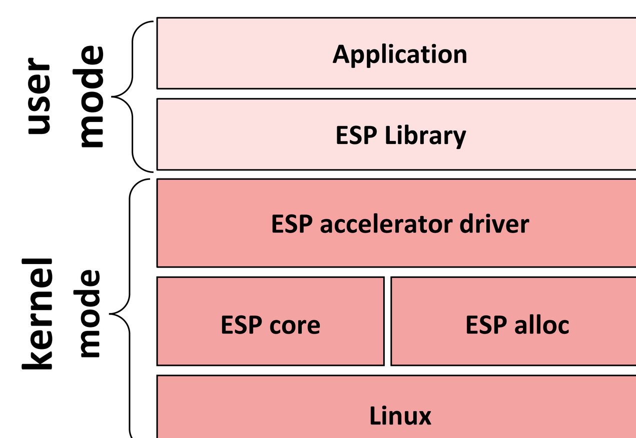
- Accelerator Socket w/ Platform Services
  - Direct-memory-access
  - Run-time selection of coherence model:
    - Fully coherent
    - LLC coherent
    - Non coherent
  - User-defined registers
  - Distributed interrupt

Memory Tile

- External Memory Channel
- LLC and directory partition
  - Supports coherent-DMA for accelerators
- DMA channels
- IO/IRQ channel

ESP Software API

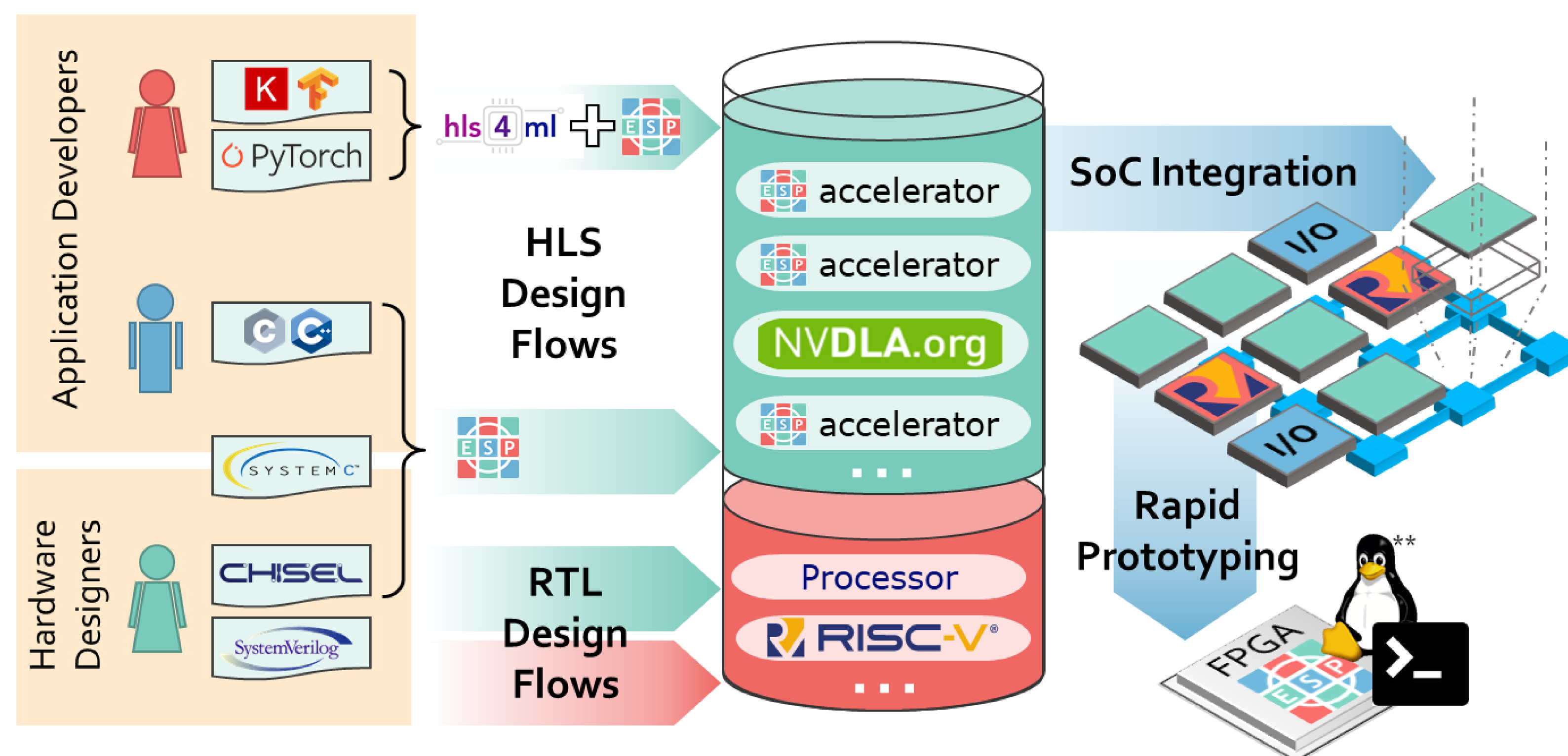
- Generation of device driver and unit-test application
- Seamless shared memory




```
// Example of existing C application
// with ESP accelerators that replace
// software kernels 2, 3 and 5
{
  int *buffer = esp_alloc(size);
  for (...) {
    kernel_1(buffer,...);    /* existing software */
    esp_run(cfg_k2);         /* run accelerator(s) */
    esp_run(cfg_k3);         /* run accelerator(s) */
    kernel_4(buffer,...);    /* existing software */
    esp_run(cfg_k5);         /* run accelerator(s) */
  }
  validate(buffer);          /* existing checks */
  esp_cleanup();             /* free memory */
}
```

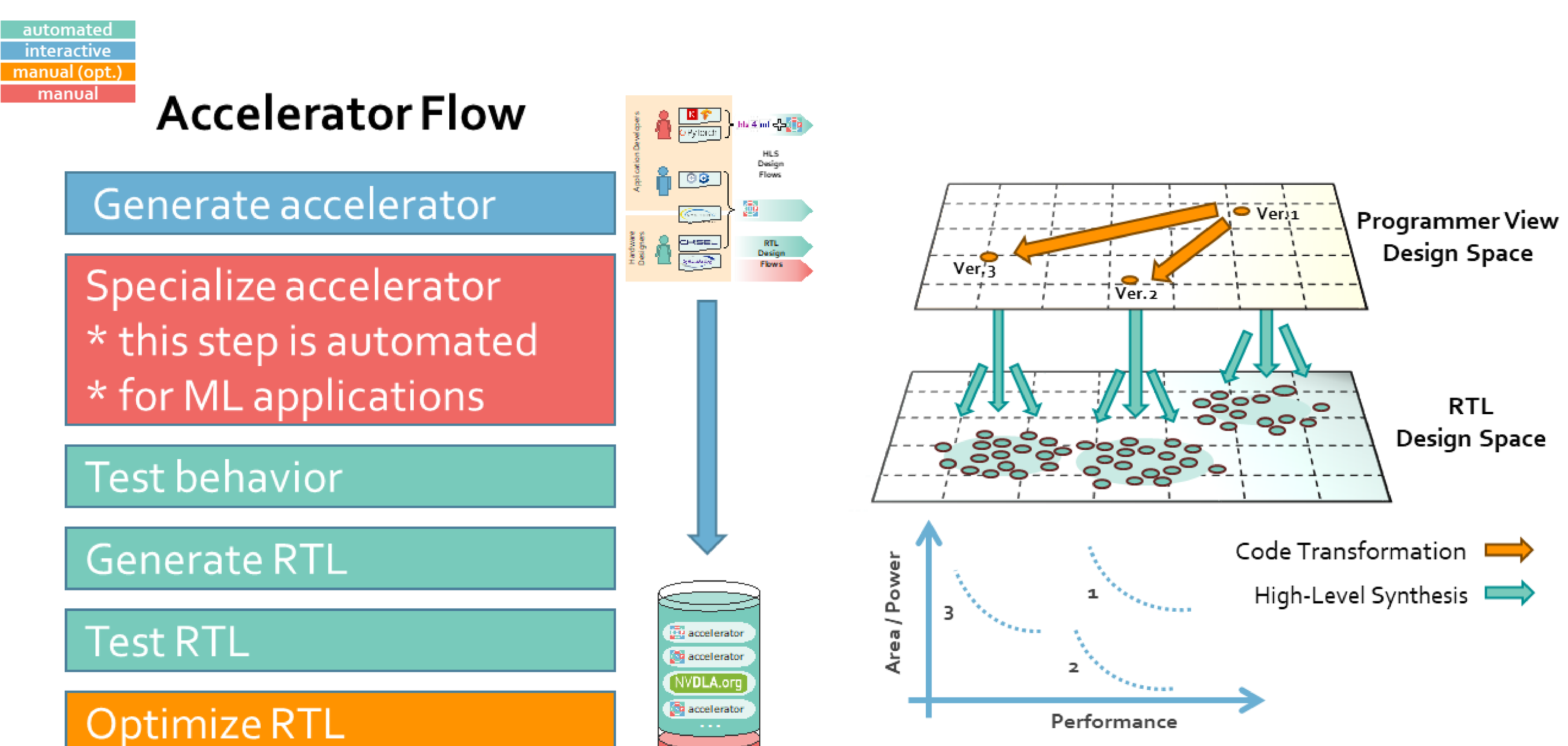
ESP Methodology

ESP Vision: Domain Experts Can Design SoCs



ESP Accelerator Flow

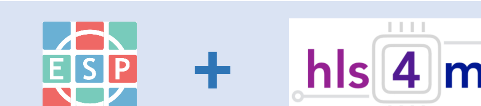
- Developers focus on the high-level specification, decoupled from memory access, system communication, hardware/software interface
- A graphical user interface application comes along with the ESP platform



SoC Flow

- Generate sockets
- Configure RISC-V SoC
- Compile bare-metal
- Simulate system
- Implement for FPGA
- Configure runtime
- Compile Linux
- Deploy prototype

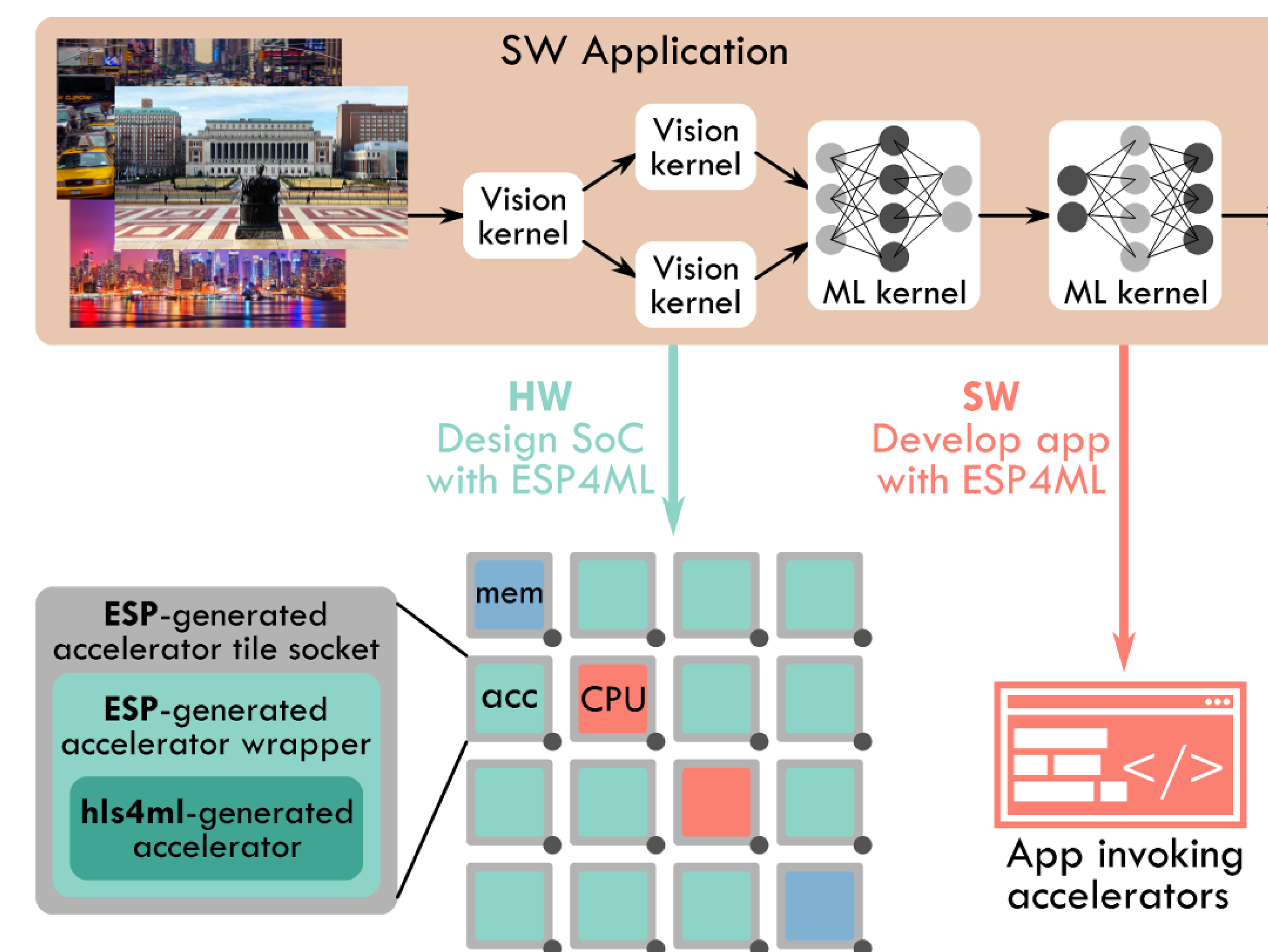
ESP4ML



- **hls4ml**
  - Automated generation of accelerators from ML models (Keras/Pytorch/ONNX)
  - Currently targets FPGA only (Xilinx Vivado HLS)
- **ESP**
  - Automated integration of hls4ml accelerators in ESP accelerator tiles
  - Seamless mapping of complex applications onto many-accelerator SoCs
  - Reconfigurable accelerator-to-accelerator communication

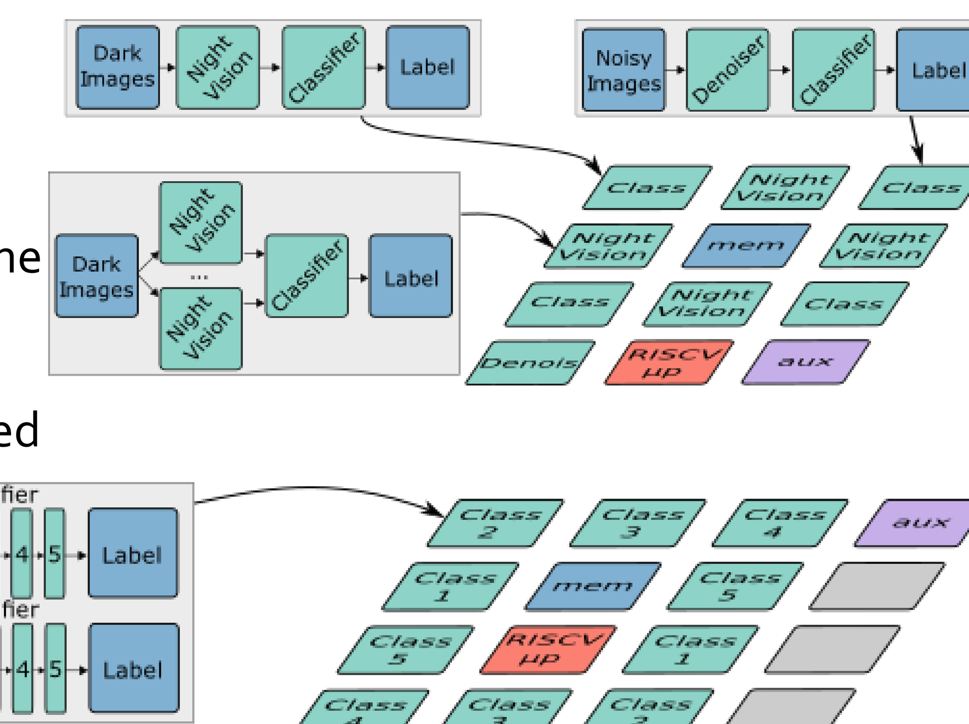
Automated Integration Steps

- Generate a generic Vivado HLS wrapper for hls4ml accelerators
- Place the hls4ml output in the accelerator wrapper folder
- Now ESP sees the wrapper as any other accelerator



Case Study

- Deploy two multi-accelerator SoCs on FPGA (Xilinx VCU118)
- Night-vision (Stratus HLS)
- Image classifier (hls4ml) for the Street View House Numbers (SVHN) dataset from Google
- Denoiser (hls4ml) implemented as an autoencoder



ESP: https://esp.cs.columbia.edu/
hls4ml: https://github.com/hls-fpga-machine-learning/hls4ml