ESP4ML: Platform-Based Design of Systems-on-Chip for Embedded Machine Learning

Davide Giri, Kuan-Lin Chiu, Giuseppe Di Guglielmo, Paolo Mantovani and Luca P. Carloni
Department of Computer Science · Columbia University, New York
[davide_giri, chiu, giuseppe, paolo, luca]@cs.columbia.edu

***Abstract*—We present ESP4ML, an open-source system-level design flow to build and program SoC architectures for embedded applications that require the hardware acceleration of machine learning and signal processing algorithms. We realized ESP4ML by combining two established open-source projects (ESP and HLS4ML) into a new, fully-automated design flow. For the SoC integration of accelerators generated by HLS4ML, we designed a set of new parameterized interface circuits synthesizable with high-level synthesis. For accelerator configuration and management, we developed an embedded software runtime system on top of Linux. With this HW/SW layer, we addressed the challenge of dynamically shaping the data traffic on a network-on-chip to activate and support the reconfigurable pipelines of accelerators that are needed by the application workloads currently running on the SoC. We demonstrate our vertically-integrated contributions with the FPGA-based implementations of complete SoC instances booting Linux and executing computer-vision applications that process images taken from the Google Street View database.**

I. Introduction

Since 2012, when the use of deep neural networks for classifying million of images from the web gave spectacular results [1], [2], the design of specialized accelerators for machine learning (ML) has become the main trend across all types of computing systems [3]. While the initial focus was mostly on systems *in the cloud*, the demand for enabling machine learning into embedded devices *at the edge* keeps growing [4]. To date, most research efforts have focused on the accelerator design in isolation, rather than on their integration into a complete system-on-chip (SoC). However, to realize innovative embedded systems for such domains as robotics, autonomous driving, and personal assistance, ML accelerators must be coupled with accelerators for other types of algorithms such as signal processing or feedback control. Furthermore, as the complexity of ML applications keeps growing, the challenges of integrating many different accelerators into an SoC at design time and managing the shared resources of the SoC at runtime become much harder.

In this paper we present ESP4ML, a system-level design flow that enables the rapid realization of SoC architectures for embedded machine learning. With ESP4ML, SoC designers can integrate at design time many heterogeneous accelerators that can be easily connected at run-time form various tightly-coupled pipelines (Fig. 1). These accelerator pipelines are reconfigured dynamically (and transparently to the application programmer) to support the particular embedded application that is currently running on top of Linux on the SoC processor.

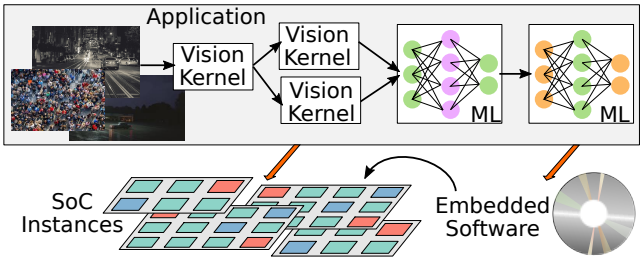

Fig. 1. The proposed design flow maps full embedded applications into a complete SoC instance, which hosts reconfigurable pipelines of ML accelerators and other accelerators (e.g. for Computer Vision) connected via a NoC.

To realize ESP4ML, we embraced the concept of *open-source hardware (OSH)* [5], in multiple ways. First, our main goal is to simplify the process of designing complete SoCs that can be rapidly prototyped on FPGA boards. The ESP4ML users can focus on the design of specific accelerators, which is simplified with high-level synthesis (HLS), while reusing available OSH designs for the main SoC components (e.g. the Ariane RISC-V processor core [6]). Second, ESP4ML is the result of combining two existing OSH projects that have been independently developed: ESP and HLS4ML.

- ESP is a platform for developing heterogeneous SoCs that promotes the ideas of platform-based design [7], [8].
- HLS4ML is a compiler that translates ML models developed with commonly used open-source packages such as Keras and PyTorch into accelerator specifications that can be synthesized with HLS for FPGAs [9], [10]. While originally developed for research in particle physics, HLS4ML has broad applicability.

To combine these two projects and reach our main goal [1]:

1) We enhanced the ESP architecture to support the reconfigurable activation of pipelines of accelerators, by implementing *point-to-point (p2p) communication* channels among them. This is done by reusing only the preexisting interconnection infrastructure without any overhead, i.e. without any addition of channel queues, routers, or links in the network-on-chip (NoC).
2) We augmented the ESP methodology with an *application programming interface (API)* that for a given embedded application and a target SoC architecture allows the specification of the software part to be accelerated as a simple dataflow of computational kernels.

[1]We released the contributions of this paper as part of the ESP project on Github [8].

3) We developed a *runtime system* on top of Linux that takes this dataflow and translates it into a pipeline of accelerators that are dynamically configured, managed, and kept synchronized as they access shared data. This is done in a way that is fully transparent to the application programmer.
4) We enhanced the SoC integration flow of ESP by designing new parameterized interface circuits (synthesizable with HLS) that encapsulate accelerators designed for Vivado HLS [11], without requiring any modification to their designs. This provides an adapter layer to bridge the *ap_fifo* protocol from Vivado HLS to the ESP accelerator interface so that ESP4ML users are only responsible for setting the appropriate parameters for DMA transactions (i.e., transaction length and offset within the virtual address space of the accelerator).
5) We encapsulated HLS4ML into a fully automated design flow that takes an ML application developed with KERAS TensorFlow and the *reuse factor* parameter to control parallelization specified within HLS4ML and returns an accelerator that can be integrated within a complete SoC. This required no modification to the code generated with the HLS4ML compiler.

We demonstrate the successful vertical integration of these contributions by presenting a set of experimental results that we obtained with ESP4ML. Specifically, we designed two complete SoC architectures, implemented them on FPGA boards, and used them to run embedded applications, which invoke various pipelines of accelerators for ML and computer vision. Compared to an Intel processor, an ARM processor, and an NVIDIA embedded GPU, energy-efficiency speedups (measured in terms of frames/Joule) are above $100\times$ in some cases. Furthermore, thanks to the efficient p2p-communication mechanisms of ESP4ML, the execution of these applications presents a major reduction of the off-chip memory access compared to the corresponding versions that use off-chip memory for inter-accelerator communication, which is normally the most efficient accelerator cache-coherence model for non-trivial workloads with regular memory access pattern [12].

II. BACKGROUND

We give a quick overview of the ESP and HLS4ML projects to provide basic information to read the subsequent sections.

Embedded Scalable Platforms. ESP is an open-source research platform for the design of heterogeneous SoCs [7]. The platform combines an architecture and a methodology. The flexible tile-based architecture simplifies the integration of heterogeneous components through a combination of hardware and software sockets. The companion methodology raises the level of abstraction to system-level design by decoupling the system integration from the design and optimization of the various SoC components (accelerator, processors, etc.) [13].

The ESP tile-based architecture relies on a multi-plane packet-switched network-on-chip (NoC) as the communication medium for the entire SoC. The interface between a tile and the NoC consists of a wrapper (the hardware part of a socket) that implements the communication mechanisms together with other *platform services*. For example, the socket of an accelerator tile typically implements: a configurable direct-memory access (DMA) engine, interrupt-request logic, memory-mapped registers, and the register-configuration logic.

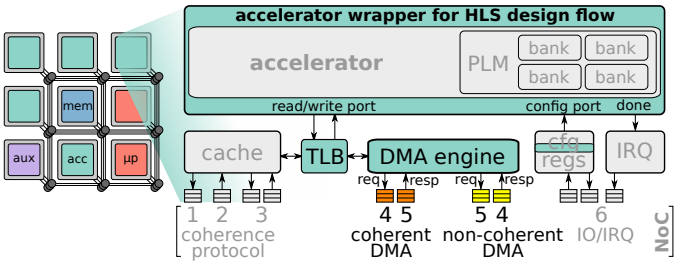

Fig. 2. Example of a 3×3 instance of ESP with zoom into the accelerator tile, taken from Giri et al. [14]. Components in gray with shaded text are integrated from ESP without modifications, while we modified the components in color with black text to support the ESP4ML design flow.

In ESP, an NoC is an $M \times N$ 2D-mesh, corresponding to a grid of tiles of configurable size: e.g., Fig. 2 shows a 3×3 instance of an ESP SoC with two processor tiles, one memory tile, one auxiliary tile, and five accelerator tiles. An *NoC plane* is a set of bi-directional links of configurable width (e.g. 32 or 64 bits) that connect pairs of adjacent tiles in the NoC. The ESP architecture allots two full planes of the NoC to the accelerators, which use them to move efficiently long sequences of data between their on-chip local private memories and the off-chip main memory (DRAM). These data exchanges, called either *loads* or *stores* depending on their direction, happen via DMA, i.e. without involving the processor cores, which instead typically transfer data at a finer granularity (i.e. one or few cache lines) [15]. Note that DMA requests and responses are routed through decoupled NoC planes to prevent deadlock when multiple accelerators and multiple memory tiles are present. In Section IV we show how we leverage this DMA queues decoupling to efficiently implement p2p communication for ESP4ML.

The ESP methodology supports a design flow that leverages SystemC and Cadence Stratus HLS [16] for the specification and implementation of an accelerator to be plugged into the accelerator wrapper, as shown in Fig. 2. ESP users are responsible for the core functionality of their accelerators and for adapting the template load/store functions provided in the synthesizable SystemC ESP library.

HLS4ML. The HLS4ML project allows designers to specify ML models and neural-network architectures for a specific task (e.g. image classification) by using a common open-source software such as Keras [17], PyTorch [18], and ONNX [19]. A trained ML model to be used for inference is described with a couple of standard-format files: a `JSON` file for the network topology and a `HDF5` file for the model weights and biases. These are the inputs of the HLS4ML compiler, which automatically derives a hardware implementation of the corresponding ML accelerator that can be synthesized for FPGAs using HLS tools [20]. While HLS4ML currently supports only Vivado HLS [11], its approach can be extended

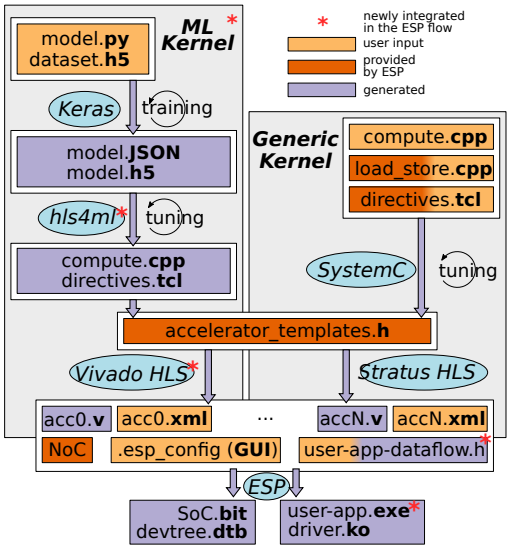

Fig. 3. The proposed design flow for embedded machine learning.

to other HLS tools [9], possibly targeting ASIC as well.

For an ML accelerator, the trade-offs among latency, initiation interval, and FPGA-resource usage depend on the degree of parallelization of its inference logic. In HLS4ML, these can be balanced by setting the *reuse factor*, which is a single configuration parameter that specifies the number of times a multiplier is used in the computation of a layer of neurons.

III. The Proposed Design Flow

Fig. 3 shows the ESP4ML flow to design SoCs for embedded ML applications. From the ESP project, we adopted the flow to design and integrate accelerators for generic computational kernels (right) and we implemented a new flow to design accelerators for ML applications, which leverages HLS4ML (left). Furthermore, we enabled the runtime reconfiguration of the communication among accelerators through a software application (generated from a user-specified dataflow) and a new platform service for reconfigurable p2p communication (implemented in the wrapper of the accelerator tile).

In order to integrate accelerators compiled by HLS4ML, we extended the SoC generation flow of ESP to host RTL components synthesized with Vivado HLS. We designed a new template wrapper that is split into a source file for Vivado HLS synthesis directives and an RTL adapter for the ESP accelerator tile. These template source files are automatically specialized for a particular instance of ML accelerator depending on input and output size as well as on precision and data type (e.g. 16-bits fixed-point).

The portion of the wrapper processed by Vivado HLS implements the control logic to make DMA transaction requests and handles the synchronization between DMA transactions and the computational kernel. Fig. 4 shows the gist of the top-level function: the `LOAD` function gets and unpacks data from the data read port into local memories; the `COMPUTE` function calls the computational kernel (e.g. generated from HLS4ML); the `STORE` function packs the data from local memory and

Code snippet 1:

```
void TOP (word *out, word *in1, unsigned conf_size,
      dma_info_t *load_ctrl, dma_info_t *store_ctrl)
{
  word _inbuff[IN_BUF_SIZE];
  word _outbuff[OUT_BUF_SIZE];

go:
  for (unsigned i = 0; i < n_chunks; i++) {
    LOAD(_inbuff, in1, i, load_ctrl, 0);
    COMPUTE(_inbuff, _outbuff);
    STORE( _outbuff, out, i, store_ctrl, conf_size);
  }
}
```

Fig. 4. Example of top-level function of the ESP wrapper for Vivado HLS.

pushes them to the data write port. In addition, both `LOAD` and `STORE` functions set the appropriate virtual address and length for the current transaction. This information is computed based on the current iteration index of the main loop, the size of the dataset and the size of the local buffers. Some of the parameters needed are set at runtime through configuration registers (e.g. `conf_size`).

The RTL portion of the wrapper includes a set of shallow FIFO queues that decouple the control requirements of the FIFO interface in Vivado HLS from the protocol of the accelerator tile in ESP. In addition to FIFO queues, the wrapper binds the ESP configuration registers to the corresponding signals of the accelerator, such as `conf_size` in Fig. 4. The list of registers is specified into an `XML` file for each accelerator following the default ESP integration flow.

IV. Point-to-Point Communication Services

Section III explains how ESP4ML users can specify the accelerators for their target embedded applications. Once these are implemented as RTL intellectual property (IP) blocks, the ESP graphic configuration interface can be used to pick the location of each accelerator in the SoC and generate the appropriate hardware wrappers, including routing tables, and Linux device drivers. The ESP infrastructure then generates a bitstream for Xilinx FPGAs and a bootable image of Linux that can run on the embedded RISC-V processor in the ESP SoC [21].

The ESP design flow, however, used to lack the ability to map the application dataflow onto the user-level software and to dynamically reconfigure the NoC routers to remap DMA transactions onto p2p data transfers among accelerators. Hence, we developed a new p2p platform service for ESP architectures that is compatible with the generic accelerator tile wrapper.

First, we defined two additional registers common to all accelerators. The `LOCATION_REG` is a read-only register that exposes the x-y coordinates of an accelerator on the NoC to the operating system. The `P2P_REG` is the p2p configuration register, which holds the following information: p2p store is enabled, p2p load is enabled, number of source tiles (1 to 4) for the load transactions, x-y coordinates of the source tiles. We also modified the ESP device driver such that any registered accelerator, (discovered when `probe` is executed) is added to a global linked list protected by a `spinlock`. This

Code snippet 2:

```
#include "libesp.h"
#include "dflow1.h"

int main(int argc, char **argv)
{
    int errors = 0;
    contig_handle_t contig;
    uint8_t *buf;

    // Allocate memory
    buf = (uint8_t*) esp_alloc(&contig, DATASET_SIZE);

    // Initialize buffer
    init_buffer(buf);

    // Execute accelerator(s) dataflow.
    // The configuration specifies the communication
    // for each accelerator invocation: DMA or P2P.
    esp_run(dflow1_cfg, NACC);

    // Validation
    errors += validate_buffer(buf);

    // Free memory
    esp_cleanup();

    return errors;
}
```

Fig. 5. Generated ESP4ML code to spawn multiple HW-accelerated threads.

list allows any thread executing the code of an accelerator device-driver in kernel mode to access information related to other accelerators. Since this information includes the base address of the configuration registers, a device name, already known in user space, can be mapped to the corresponding x-y coordinates. These coordinates are not exposed to user space and the application dataflow can be specified by simply using the accelerator names. Hence, the application is completely independent from the particular SoC floorplan.

To support accelerator p2p transactions we made minor modifications to translation-lookaside buffer (TLB) and DMA controller in the ESP accelerator tile wrapper [15]. A key aspect of our implementation is that all p2p transactions are on-demand, that is they must be initiated by the receiver. The sender accelerator tile waits for a p2p load request before forwarding data to the NoC. Implementing p2p stores on-demand is necessary to prevent long packets of data being stalled in the NoC links while the accelerator that is downstream in the dataflow is not ready to accept them. For the same reason our solution guarantees the "consumption assumption" [22] for all supported dataflow configurations. An accelerator tile will only request data when it has enough space to store it locally.

This mechanism is completely transparent to the accelerator, which still operates as if regular DMA transactions were to occur, while performance and energy consumption largely benefit from close-distance communication and a drastic reduction in accesses to DRAM or to the last-level cache.

We built this p2p communication service without adding any NoC planes, nor queues at the NoC interface, because we rely on queues that are otherwise unused for regular DMA transactions. Specifically, we carefully reused available queues in the ESP accelerator tile.

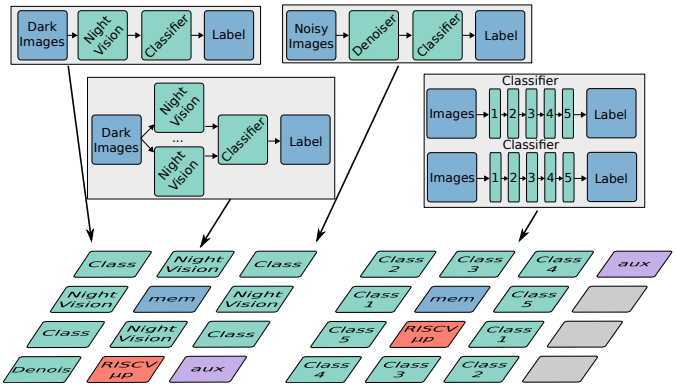

Fig. 6. Dataflow of the four case-study applications discussed in the evaluation and corresponding mapping onto two instances of ESP.

V. Runtime System for Accelerators

After implementing the p2p service, we developed a software API to hide the details of memory allocation, accelerator invocation, and synchronization from user-space software. Dependencies across accelerators are specified through a simple dataflow. By modifying a template that is automatically generated for the given SoC architecture, the ESP4ML users can define a dataflow of accelerator invocations. For each invocation they can specify whether to use DMA or p2p communication and they can set other accelerator-specific communication parameters.

The snippet in Fig. 5 shows an example of automatically generated applications that reads two dataflow configurations from `dflow1.h`. For each configuration the application spawns as many threads as the number of running accelerators to exploit all the available parallelism in the dataflow. Since accelerators that use the p2p service are automatically synchronized in hardware, the software runtime incurs minimal overhead. This is limited to the `ioctl` system calls that are used to start the accelerators [15]. When ESP4ML users set the dataflow parameters to use DMA only, dependencies are enforced with `pthread` primitives. Thanks to our software runtime, ESP4ML users can dynamically reshape the data traffic on the NoC to activate a reconfigurable pipeline of accelerators for the given embedded application In addition, they can tune the throughput of the system by balancing each stage of this pipeline: e.g., if a slow accelerator is feeding a faster one, multiple instances of the slower accelerator can be activated to feed a single accelerator downstream.

VI. Experimental Results

Applications. Street View House Numbers (SVHN) is a real-world image dataset obtained from Google Street View pictures [23]. SVHN is similar to the MNIST dataset, but it is ten times bigger (600,000 images split in training, test, extra-training datasets). For SVHN, the problems get significantly more laborious due to the environmental noise in the pictures (including shadows and distortions). We developed two embedded applications for the SVHN dataset: digit classification and image denoising. For both, we adopted ML solutions and

TABLE I
SUMMARY OF RESULTS USING THE BEST-CASE CONFIGURATION

	NIGHTVISION & CLASSIFIER	DENOISER & CLASSIFIER	MULTI-TILE CLASSIFIER.
LUTS	48%	48%	19%
FFS	24%	24%	11%
BRAMS	57%	57%	21%
POWER (W)	1.70	1.70	0.98
FRAMES/S ESP4ML	35,572	5,220	28,376
FRAMES/S INTEL I7	1,858	30,435	82,476
FRAMES/S JETSON	377	2,798	6,750

trained our models in KERAS. Recalling the ESP4ML flow overview of Fig. 1, the upper part of Fig. 6 shows concrete instances for these two applications.

For the digit classification problem, we defined a Multilayer Perceptron (MLP) with four hidden layers. The size of the fully connected network is 1024x256x128x64x32x10. We used dropout layers with a 0.2 rate to prevent overfitting during training. The trained model accuracy is 92%. For the denoising problem, we designed an autoencoder model. The network size is 1024x256x128x1024, and the compression factor in the bottleneck is 8. We added Gaussian noise to the SVHN dataset and trained the model with a 3.1% reconstruction error.

We also developed one application outside the ML domain, which is a night computer vision application consisting of three kernels: noise filtering, histogram, and histogram equalization. For the purpose of this evaluation, we darkened the SVHN dataset and we used this Night-Vision application as a pre-processing step of the MLP classifier described above.

Accelerators and SoCs. We designed two SoCs that we synthesized for FPGA with the ESP4ML flow. As shown in Fig. 6, these SoCs contain many (up to ten) accelerators for the target applications and one Ariane RISC-V core. Table I shows the FPGA resources usage and the dynamic power dissipation as reported by Xilinx Vivado. We designed the Classifier and the Denoiser with KERAS and we compiled them with HLS4ML within the ESP4ML flow. We then designed a partitioned version of the Classifier, by distributing the computation across five accelerators. Finally, we designed the accelerator for the Night-Vision kernels by leveraging another HLS-based design flow within ESP: i.e., we designed them in SystemC and synthesized them with *Cadence Stratus HLS*.

Experimental Setup. We implemented the two ESP4ML SoCs of Fig. 6 on a Xilinx Ultrascale+ FPGA board with a clock frequency of 78MHz. We ran all the experiments by using this board and executing the test embedded applications on top of Linux running on the Ariane core. We compared the execution of these applications on the ESP4ML SoC with the hardware accelerators versus the execution of the same applications in software on the following two platforms: (a) an Intel i7 8700K processor and (b) an NVIDIA Jetson TX1 model, which is an embedded system that combines a 256-core NVIDIA Maxwell GPU with a Quad-Core ARM Cortex-A57 MPCore. Based on the available datasheet, we considered values of power consumption equal to $1.5W$ and $10W$ for the ARM core and the GPU, respectively. For the Intel core, we estimated a TDP of $78.6W$ (the nominal value is $95W$).

Results. The three bottom lines of Table I report the performance of the three platforms measured in terms of processed frames per second. The FPGA implementations of the SoC designed with ESP4ML offer better performance compared to a commercial embedded platform like the Jetson TX1. The Intel i7 cores predictably provides the best performance, aside for the case of the Night-Vision application, which is a single-threaded program.

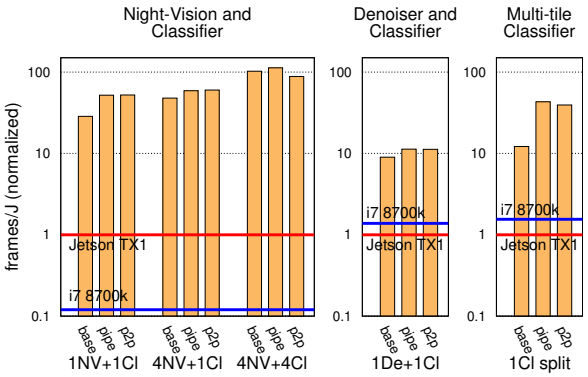

Fig. 7. Energy efficiency in terms of $frames/Joules$ for the ESP4ML compared to an NVIDIA embedded GPU and an Intel i7 core. The three bars show the performance of two ESP4ML techniques: reconfigurable pipelines of accelerators w/ (*p2p*) and w/o (*p2p*) p2p communication.

Fig. 7 compares the execution of the applications on the three platforms in terms of energy efficiency, measured as $frames/Joule$ (in logarithmic scale). Notice that all the accelerator execution-time measurements include the overhead of the ESP4ML runtime system managing the accelerators invocations as well as the overhead of the accelerators Linux device drivers. The horizontal blue and red lines show the efficiency of the CPU and GPU, respectively. For the purpose of this comparison, we report the average dynamic power consumption for the two ESP4ML SoCs as estimated by Xilinx Vivado for the whole SoC (i.e. not just for the accelerators active in a specific test). This is a conservative assumption, particularly if one considers that the power consumption depends on the choice of the FPGA and that a Xilinx Ultrascale+ is a particularly large FPGA. Still, the ESP4ML SoCs outperforms both the GPU and the CPU across all three applications, yielding in some cases an energy-efficiency gain of over $100\times$.

Each cluster of bars in Fig. 7 represents an execution based on a different pipeline of accelerators, with the number of accelerators varying from two to eight. The left bar of each cluster shows results for the case where the accelerators are invoked serially in a single-thread application. The middle bars (label *pipe*) correspond to concurrent executions in a reconfigurable pipeline, as the accelerators are invoked with a multi-threaded application (one thread per accelerator). The right bar adds the ESP4ML p2p communication to this pipeline execution. The results for the Night-Vision and Classifier show that the performance increases significantly when the accelerators work concurrently in pipeline. While p2p communication does not provide a major gain in performance in this case, its main benefit is the reduction of off-chip memory accesses, which translates into a major energy saving: as shown in

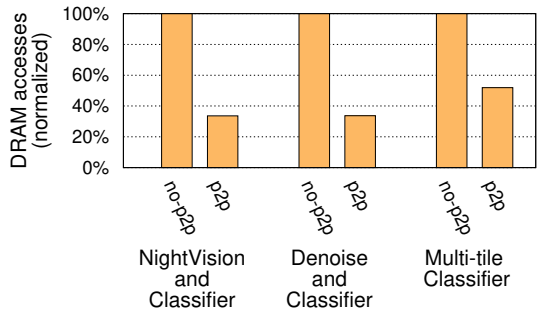

Fig. 8. Relative number of DRAM accesses w/ and w/o point-to-point communication among accelerators for three test applications. The energy savings due to a reduced access to memory are the main benefit of the point-to-point communication among accelerators.

Fig. 8, this reduction varies between $2\times$ and $3\times$ for the target applications.

VII. Related Work

As efforts in accelerators for ML continue to grow, HLS is recognized as a critical technology to build efficient optimization flows [24]. For instance, Hao et al. recently proposed a PYNQ-ZI based approach to design deep neural network accelerators [25]. Meanwhile, various optimization techniques to deploy deep neural networks on FPGA have been proposed [24], [26]–[28]. In this context, HLS4ML [9] is being increasingly adopted by research organizations and is raising interest in the industry [29], [30]. To date, however, most open-source projects focus on the design of accelerators in isolation. Instead, we propose the first automated open-source design flow that leverages ESP and HLS4ML to integrate multi-accelerator pipelines into SoCs. The ESP project initially focused on the integration of generic accelerators specified in SystemC that could operate in pipeline through shared memory [13]. The ESP4ML flow augments ESP with the support of accelerators designed also with common ML API and enable runtime reconfiguration of pipelines with efficient p2p communication.

VIII. Conclusions

ESP4ML is a complete system-level design flow to implement SoCs for embedded applications that leverage tightly-coupled pipelines of many heterogeneous accelerators. We realized ESP4ML by building on the prior efforts of two distinct open-source projects: ESP and HLS4ML. In particular, we augmented ESP with a HW/SW layer that enables the reconfigurable activation of accelerators pipelines through efficient point-to-point communication mechanisms. In addition, we built a library of interface circuits that allow for the first time to integrate HLS4ML accelerators for machine learning into a complete SoC using only open-source hardware components. We demonstrated our work with the FPGA implementations of various SoC instances running computer-vision applications.

Acknowledgments. This work was supported in part by DARPA (C#: FA8650-18-2-7862) and in part by the National Science Foundation (A#: 1764000). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied,of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

We thank the developer team of *hls4ml*. We acknowledge the Fast Machine Learning collective as an open community of multi-domain experts and collaborators. This community was important for the development of this project.

References

[1] A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. of NIPS*, May 2012, pp. 1097–1105.
[2] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015.
[3] V. Sze *et al.*, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. of the IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.
[4] Y. Deng, "Deep learning on mobile devices: a review," in *Proc. of SPIE*, May 2019, pp. 52 – 66.
[5] G. Gupta *et al.*, "Kickstarting semiconductor innovation with open source hardware," *IEEE Computer*, Jun. 2017.
[6] Ariane, https://github.com/pulp-platform/ariane.
[7] L. P. Carloni, "The case for Embedded Scalable Platforms," in *Proc. of DAC*, Jun. 2016, pp. 17:1–17:6.
[8] Columbia SLD Group, "ESP," www.esp.cs.columbia.edu.
[9] J. Duarte *et al.*, "Fast inference of deep neural networks in FPGAs for particle physics," *Journal of Instrumentation*, vol. 13, no. 07, Jul. 2018.
[10] "HLS4ML," https://fastmachinelearning.org/hls4ml/.
[11] Xilinx, "The Xilinx Vivado design suite."
[12] D. Giri, P. Mantovani, and L. P. Carloni, "Accelerators & Coherence: An SoC Perspective," *IEEE Micro*, vol. 38, no. 6, pp. 36–45, Nov. 2018.
[13] P. Mantovani, G. Di Guglielmo, and L. P. Carloni, "High-level synthesis of accelerators in embedded scalable platforms," in *Proc. of ASPDAC*, Jan. 2016, pp. 204–211.
[14] D. Giri, P. Mantovani, and L. P. Carloni, "NoC-based support of heterogeneous cache-coherence models for accelerators," in *Proc. of NOCS*, Oct. 2018, pp. 1:1–1:8.
[15] P. Mantovani *et al.*, "Handling large data sets for high-performance embedded applications in heterogeneous systems-on-chip," in *Proc. of CASES*, Oct. 2016, pp. 3:1–3:10.
[16] D. Pursley and T. Yeh, "High-level low-power system design optimization," in *VLSI-DAT*, Apr. 2017, pp. 1–4.
[17] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2017.
[18] A. Paszke *et al.*, "Automatic differentiation in PyTorch," 2017.
[19] "Open Neural Network Exchange," https://github.com/onnx/onnx, 2018.
[20] R. Nane *et al.*, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Trans. on CAD*, vol. 35, no. 10, pp. 1591–1604, Oct. 2015.
[21] P. Mantovani *et al.*, "An FPGA-based Infrastructure for Fine-Grained DVFS Analysis in High-Performance Embedded Systems," in *Proc. of DAC*, 2016, pp. 157:1–157:6.
[22] Y. Ho Song and T. M. Pinkston, "A progressive approach to handling message-dependent deadlock in parallel computer systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 3, pp. 259–275, Mar. 2003.
[23] N. Yuval *et al.*, "The Street View House Numbers (SVHN) Dataset," http://ufldl.stanford.edu/housenumbers/, 2011.
[24] X. Zhang *et al.*, "Machine learning on FPGAs to face the IoT revolution," in *Proc. of ICCAD*, Nov. 2017, pp. 894–901.
[25] C. Hao *et al.*, "FPGA/DNN co-design: An efficient design methodology for IoT intelligence on the edge," in *Proc. of DAC*, Jun. 2019, pp. 1–6.
[26] Y. Wang *et al.*, "DeepBurning: Automatic generation of FPGA-based learning accelerators for the neural network family," in *Proc. of DAC*, Jun. 2016, pp. 1–6.
[27] C. Zhang *et al.*, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *Proc. of ICCAD*, Nov. 2016, pp. 1–8.
[28] C. Hao and D. Chen, "Deep neural network model and FPGA accelerator co-design: Opportunities and challenges," in *Proc. of ICSICT*, Oct. 2018.
[29] Xilinx, Inc., "Artificial intelligence accelerates dark matter search," https://www.xilinx.com/publications/powered-by-xilinx/cerncasestudy-final.pdf.
[30] Fast Machine Learning Lab, https://fastmachinelearning.org/.