This article has been accepted for publication in IEEE Micro. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MM.2021.3073893

Accelerator Integration for Open-Source SoC Design

Davide Giri
Kuan-Lin Chiu
Guy Eichler
Paolo Mantovani
Luca P. Carloni

Columbia University

***Abstract*—The open-source hardware community contributes a variety of processors and accelerators, but combining them effectively into a complete SoC remains a difficult task. We present a design flow for the seamless hardware and software integration of accelerators into a complete SoC and for its evaluation through rapid FPGA-based prototyping. By leveraging ESP, an open-source platform for agile heterogeneous SoC design, we demonstrate FPGA prototypes of various SoC designs, featuring the NVIDIA Deep Learning Accelerator and the Ariane RISC-V 64-bit processor core.**

INTRODUCTION

Heterogeneous system-on-chip (SoC) architectures are pervasive across computing domains, from supercomputers to smartphones [1]. To meet energy efficiency and performance goals, they rely on an increasing variety of specialized hardware accelerators [2]. High degrees of component heterogeneity, however, complicate SoC design and evaluation. As SoC complexity grows with each generation, the addition of new capabilities is increasingly limited by the engineering effort and team sizes [3].

Open-source hardware (OSH) holds the promise of boosting the SoC design and evaluation process by enabling the reuse of pre-designed and pre-validated components across different SoC projects [4]. Most OSH contributions consist of individual components. While these are obviously important, the ultimate goal is to integrate and evaluate them as part of a complete SoC. Hardware accelerators, in particular, are typically designed with limited consideration of the implications of their integration in an SoC.

The hardware and software integration of accelerators in an SoC is a complex task that is critical to the performance of the overall system. More generally, evaluating SoC architectures is becoming harder because it requires accounting for complex interactions among many heterogeneous components. Unlike slow RTL simulations, FPGA prototyping enables the execution of real workloads on top of the operating system and the full software stack. Unlike fast simulation models, FPGA prototypes can realistically reproduce the complex interactions among SoC components and with external memory. By simplifying the SoC design effort, agile methodologies for accelerator integration and evaluation with FPGA-based prototypes support the OSH community and promote architecture innovation.

We present an integration flow for third-party accelerators that enables the seamless design and FPGA-prototyping of complete SoCs from multiple OSH components. The integration flow is now part of ESP[1], our open-source platform for SoC

[1] www.esp.cs.columbia.edu/

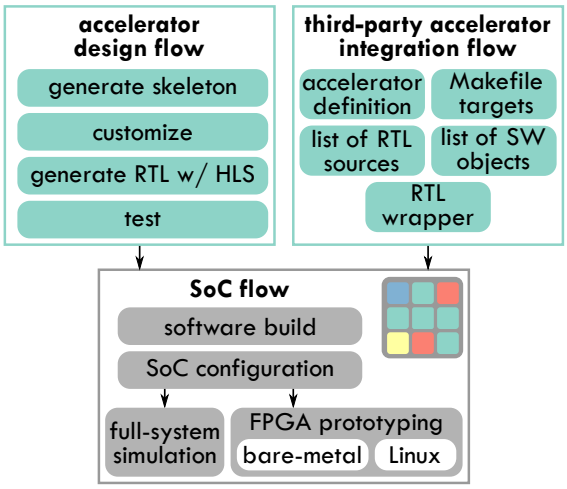


Figure (1) Steps of the ESP design methodology.

design [5], [6]. Figure 1 shows the main steps of the enhanced ESP methodology that we used to integrate many third-party intellectual property (IP) blocks, including accelerators for video encoding, deep learning and natural-language processing. We demonstrate the new capabilities of ESP with two exemplary OSH contributions: the NVIDIA Deep Learning Accelerator (NVDLA)[2] and the Ariane RISC-V 64-bit processor [7]. With the new flow, we enable ESP users to integrate these and other components into complete SoC architectures, which we prototype with FPGAs and evaluate by accelerating the computation of various neural networks.

The Architecture

The ESP platform combines a scalable architecture and a flexible methodology [5], [6].

The ESP architecture is structured as a tile grid. For a given application domain, the architect decides the structure of the SoC by determining the number and mix of tiles with the help of the ESP graphical user interface. For example, Figure 2 shows an SoC instance with 16 tiles organized in a 4×4 grid with a set of processor, accelerator and memory tiles. There are four main types of tiles: processor tile, accelerator tile, memory tile for the communication with main memory, and auxiliary tile for peripherals (e.g. UART or Ethernet) or system utilities (e.g. the interrupt controller).

[2]www.nvdla.org

Sockets and Services. Each tile is encapsulated into a *modular socket* that interfaces it to a packet-switched network-on-chip (NoC). In addition, the socket implements a set of *platform services* that provide pre-validated solutions for IP configuration and memory access. These are key to enable rapid integration and prototyping of heterogeneous SoCs. The platform services are one of the keys to rapid prototyping of full SoCs. At design time, it is possible to choose the combination of services for each tile. At runtime, many of these services offer reconfigurability options.

Tiles can access six independent NoC planes through a set of queues that handle requests and responses for each service. The router microarchitecture is based on a simple control flow and lookahead routing scheme, which enables the overlap of routing and port arbitration in a single cycle per hop.

Processor Tile. By integrating the 64-bit RISC-V Ariane core[3], we enhance ESP to allow designers to choose between two open-source processors: Ariane and the 32-bit SPARC-V8 LEON3 core[4]. Both processors can run Linux and come with private L1 caches. The modular socket of the processor tile augments them with a private L2 cache of configurable size. The processor integration into the distributed ESP system is transparent, i.e. no ESP-specific software patches are needed to boot Linux. A MESI directory-based protocol provides support for system-level coherence on top of three dedicated planes in a multiplane packet-switched NoC [8].

Memory Tile. Each memory tile contains a DDR channel to external DRAM and a partition of configurable size of the last-level cache (LLC) and corresponding directory. The maximum number of memory tiles for a given FPGA board is determined by the available DDR channels.

Accelerator Tile. Each accelerator tile contains the specialized hardware for a loosely-coupled accelerator that executes a coarse-grained task. The modular socket for a native ESP accelerator decouples the design of the accelerator from the rest of the SoC by providing components that handle memory mapped registers, interrupt re-

[3]www.github.com/pulp-platform/ariane
[4]www.gaisler.com/index.php/products/processors/leon3

This article has been accepted for publication in IEEE Micro. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MM.2021.3073893

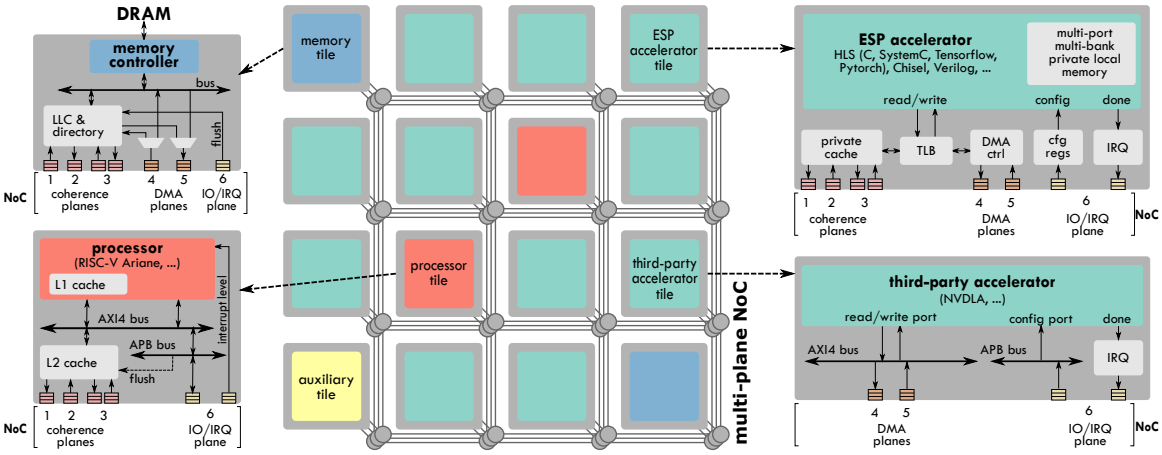


Figure (2) An instance of an ESP SoC architecture with a 4×4 tile grid.

quests, DMA, virtual memory and various levels of hardware-coherent transactions. The concept of socket plays a key role in supporting the flexibility of the ESP methodology because it accommodates accelerators designed with many different design flows. Figure 2 contrasts the socket for an accelerator designed with the ESP accelerator flows with our new socket for the integration of accelerators that are available as third-party IP blocks.

Accelerator Programming. The invocation of native ESP accelerators leverages a software stack and an application programming interface (API) to allocate shared data and configure accelerators both in bare metal and on top of Linux. Conversely, thanks to our new socket, third-party accelerators can be controlled using their own unmodified software stack. The lightweight API, which can be easily targeted from applications or by a compiler, invokes the accelerators through Linux device drivers, which are automatically generated.

Accelerator Integration Flow

We present first the ESP flows to design new accelerators, then the integration flow for third-party accelerators.

Accelerator Design Flows

The ESP flexible methodology embraces the use of a variety of languages for component development. Users can choose to specify a new accelerator at different abstraction levels, including cycle-accurate RTL descriptions like SystemVerilog or Chisel, loosely-timed or untimed behavioral descriptions with C/C++/SystemC, and domain-specific languages for deep-learning applications [9].

For accelerators specified with high-level languages, ESP provides a set of accelerator design flows that consist of a mix of automated and manual steps, as shown in Figure 1. In particular, these flows simplify the design of new loosely coupled accelerators and their integration into the architecture. ESP supports all main commercial high-level synthesis (HLS) tools.

Generate Skeleton. Designers can automatically generate a fully-working and HLS-ready accelerator skeleton by providing a small set of parameters. These include: unique name and ID, desired HLS tool, a list of application-specific configuration registers, and some information about the accelerator input and output data. The skeleton comes with a unit testbench, synthesis and simulation scripts, a bare-metal test application, and a Linux device driver with a test application. The skeleton is a basic specification that uses a set of configurable templates provided by ESP.

Customization. Starting from the automatically generated skeleton, designers must customize the accelerator computation part. In addition, they are responsible for customizing the input-generation and output-validation functions in the unit testbench and in the bare-metal and Linux test applications. Finally, in case of complex data access patterns, they may need to extend the communication part of the accelerator. The resulting specification is the HLS-ready code, which is the entry point of the design automa-

tion process for both the SystemC and C/C++ flows. For deep learning accelerators, ESP provides a fully automated flow that does not require any manual customization. This flow leverages HLS4ML[5], an open-source compiler that translates trained machine-learning models created with Keras, ONNX or PyTorch into accelerator specifications in C/C++ that can be synthesized with Xilinx Vivado HLS. After completing the HLS-ready code, all types of ESP accelerator flows follow the same automated steps, regardless of the particular language chosen for the specification.

Generate RTL with HLS. Designers can automatically generate one or multiple RTL implementations of an accelerator with a simple command that runs the selected HLS tool. The HLS-generated RTL code is automatically added to the ESP library of IP blocks for integration.

Validation. The validation step runs the accelerator unit testbench, which models the behavior of the accelerator tile socket.

Third-party Accelerator Integration Flow

When designing an accelerator, the obvious choice is to comply with one or more SoC interface standards. We implemented adapters for the AXI, AXI-Lite, AHB and APB standards from ARM, which are among the most widely adopted. Then, we designed a new tile socket for third-party accelerator integration in a modular way such that at design time the SoC architect may select a different adapter for each specific accelerator tile.

The idea behind our design of the new socket is simple: We modified the accelerator tile socket by relying on its modularity to have a set of bus-standard interfaces between a generic accelerator and the NoC, as illustrated by the block diagrams of the two accelerator tiles in Figure 2.

Third-Party IP Socket. ESP accelerators are normally simpler than a generic third-party IP accelerator because they rely on all the services provided by the accelerator tile socket. Instead, third-party accelerators and their software stack typically embed their own solutions for aspects like accelerator configuration and address translation. For this reason, we stripped the accelerator socket of some components that would not be used by a third-party accelerator. We replaced the DMA engine and the accelerator TLB with an AXI-to-NoC bridge to handle the accelerator memory transactions. We replaced the accelerator configuration registers with a NoC-to-APB bridge to connect to the accelerator configuration port. For flexibility, we added an optional adapter to convert APB transactions to AXI or AXI-Lite. We also enhanced the module responsible for interrupt delivery to support both level-sensitive interrupts and edge-sensitive interrupts.

After implementing the third-party socket, we augmented the ESP infrastructure to implement a *third-party accelerator integration flow (TPF)*. With TPF, users can integrate existing accelerators in a few steps (Figure 1).

1) *Accelerator definition.* Fill in a short XML file with some key information. This includes a unique accelerator name and ID, and the names of the reset, clock, and accelerator-interrupt signals.
2) *Makefile targets.* Create a Makefile with all targets that apply among RTL generation, and device-driver compilation, user-space runtime, and bare-metal driver. The software compilation targets cross-compile the original accelerator software for the target processor among the ones available in ESP. This allows running the original accelerator software *as is*.
3) *RTL sources.* List all RTL source files by hardware-description language.
4) *Software objects.* Create a list file for driver modules, software executables, libraries, and any other binary required by the accelerator original software. List the processors available in ESP that can run the accelerator software.
5) *RTL wrapper.* Write a Verilog top-level wrapper to expose the standard bus interfaces for the new ESP socket. This step consists in connecting wires without implementing any logic.

These simple manual steps make the new integration flow flexible and general. As shown in Figure 1, both the TPF and the flow for new accelerator designs converge into the same agile SoC design flow.

[5] fastmachinelearning.org/hls4ml

SoC Design Flow

All the accelerators designed and integrated with the accelerator flows can be automatically instantiated in an SoC and their software can be automatically compiled.

Software Build. ESP automates building both the bare-metal binaries and the Linux image for testing. The accelerator RTL is discovered by the graphical user interface (GUI) for SoC configuration. A set of Make targets is generated to compile the bare-metal test applications. These can leverage some ESP utility functions to discover, configure and invoke accelerators and control the socket services. A single Make target compiles Linux, the accelerator device drivers and the test applications. After the Linux boot completes, an initialization script loads all appropriate drivers, so that accelerators are registered and ready to use.

SoC Configuration. The ESP GUI helps designers configure an SoC design by selecting number, mix and position of the tiles, as well as many other design options, such us processor type or caches size. Based on the configuration, ESP generates the full RTL implementation of the SoC, including the tile socket required by each accelerator.

FPGA Prototyping Flow

The last steps of the SoC flow in Figure 1 concern the SoC evaluation. Full-system RTL simulation is accurate, but it is practical only for the simulation of short programs. Instead, FPGA prototyping enables the execution of real applications on top of an operating system, while reproducing the complex interactions among all SoC components.

Full-System Simulation. For each supported target FPGA board, ESP provides a simple command to simulate the complete execution of the accelerators bare-metal test programs, including bootloader and interaction with peripherals.

FPGA Prototyping. When targeting one of the supported many boards, ESP users can prototype their SoC without prior FPGA experience. The generation of the bitstream file, the programming script and the deployment of software are fully automated. The SoC is controlled through an Ethernet interface that allows quick loading of programs into main memory, updating the bootloader and resetting the processors. The SoC can run bare-metal programs or boot Linux, which allows logging into the system with `SSH` via Ethernet.

Table (1) Neural-networks characteristics.

Model	Dataset	Layers	Input	Model Size
LeNet	MNIST	9	1x28x28	1.7 MB
Convnet	CIFAR-10	13	3x32x32	572 KB
SimpleNet	MNIST	44	1x28x28	21 MB
AlexNet	ILSVRC2012	150	3x224x224	50 MB
ResNet-50	ILSVRC2012	229	3x224x224	98 MB

FPGA-based Evaluation

We demonstrate the proposed SoC design flow with a set of FPGA-based prototypes that integrate NVDLA and Ariane.

NVDLA Integration. We seamlessly integrated the NVDLA with the five simple steps of our integration flow.

NVDLA is an open-source fixed-function, but highly configurable, accelerator. Composed of multiple engines, it can perform deep learning inference. At every invocation, the engines can be configured to execute inference on one neural network layer. To test and evaluate the integration of NVDLA in ESP, we used "NVDLA *small*", featuring 8-bit integer precision, 64 multiply-and-accumulate units, a 128KB local memory, and a 64-bit data AXI4 interface.

The NVDLA Compiler takes as inputs the network topology and a trained Caffe model and generates an NVDLA *Loadable*, which contains the layer-by-layer information to configure the accelerator. After loading the input image and the Loadable through a user-space driver, the NVDLA runtime system submits a series of inference jobs to its Linux device driver. During the configuration phase, a processor reads and writes the NVDLA registers via an APB slave interface. Then, NVDLA exchanges data with the memory hierarchy via its AXI4 master interface. Upon completion of a task, NVDLA uses a level-triggered interrupt line to notify the processor.

Table 1 reports the five neural networks for image classification used in these experiments together with their characteristics.

Ariane Integration. We added the Ariane core as a second processor option in ESP. For the hardware integration, we enhanced two adapters

This article has been accepted for publication in IEEE Micro. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MM.2021.3073893

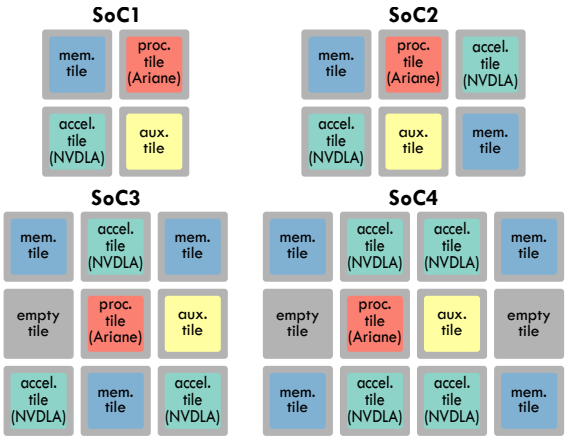


Figure (3) Four SoCs designed with ESP.

that are part of the processor tile socket for AHB-to-cache and AHB-to-NoC communication. They have a latency of at most one clock cycle and support the full throughput of the bus and the NoC router that they connect. We added support for the AXI protocol needed by Ariane, in addition to the AHB protocol used by LEON3. Everything else in the system is decoupled from the processor. The only exception is the interrupt controller, a processor-specific component. We added Ariane's interrupt controller to the auxiliary tile and we modified the interrupt-to-NoC adapter accordingly.

For the software integration, we added the RISC-V compilation option to the automated software-build step of the SoC design flow. When selecting Ariane during the configuration, the resulting SoC can execute any RISC-V program *as is*, with no ESP-specific patches.

The integration of a new processor cannot be as automated as the integration of an accelerator. However, the integration of Ariane shows that the ESP platform highly simplifies this task and can easily support multiple processor options.

SoC Design. Once integrated in ESP, an accelerator can be selected with the GUI and instantiated in multiple tiles during the configuration step. By leveraging the integration capabilities of ESP, we designed and implemented various SoC architectures that include one processor tile with the Ariane core, and different numbers of memory tiles and third-party accelerator tiles containing NVDLA (Figure 3).

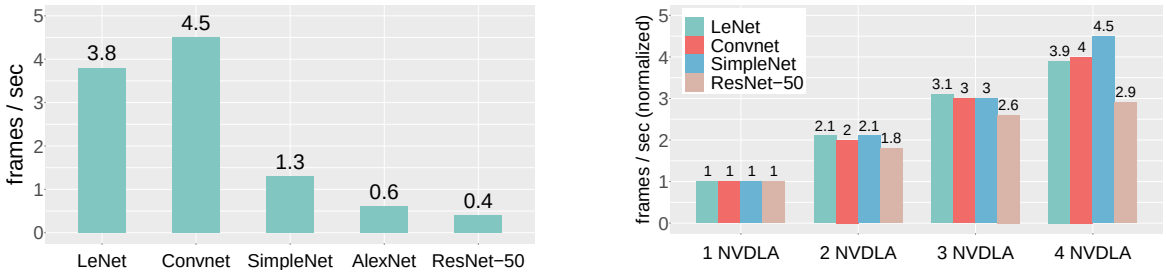
The NVDLA runtime and device driver run on Ariane, which offloads the inference jobs to the accelerator instances as needed. When instantiating multiple memory tiles, ESP automatically partitions the memory hierarchy to leverage the increased off-chip communication bandwidth. Each memory tile is responsible for a portion of the memory address space. ESP updates the device tree and routing tables in the processor and accelerator sockets to map each physical address range to its corresponding memory tile and LLC partition. Hence, the accelerators can benefit from a balanced load distribution across memory tiles.

FPGA Prototyping. We used the ESP push-button FPGA prototyping flow to deploy each SoC on a proFPGA Virtex Ultrascale XCVU440. On this board, the ESP SoCs run at 50MHz.

First, we ran inference jobs on a single NVDLA instance for the networks in Table 1, Figure 4a reports the average number of frames per second (fps) processed by SoC_1 of Figure 3, which has one NVDLA and one memory tile. The performance depends on the network size, varying between 0.4 fps for ResNet50 and 4.5 fps for Convnet. As a reference, a performance of 7.3 fps is reported for the ResNet50 with an ASIC implementation of NVDLA *small* running at a clock frequency of 1GHz, which is twenty times faster than ours.

NVDLA *small* does relatively well for smaller networks like LeNet, whereas for larger networks such as ResNet50, the larger NVDLA *full* would achieve better performance, if provided with enough memory bandwidth. The data-processing throughput can be raised by executing in parallel large batches of images across multiple instances of NVDLA *small*. With ESP, it is easy to explore the design space of possible SoC architectures by tuning the number of NVDLA instances and memory channels utilized in parallel. Since the runtime and device driver provided for NVDLA currently work with a single instance, we patched them to enable the simultaneous invocations of multiple instances. Then, during the SoC configuration stage, we can seamlessly select the number of NVDLA instances as well as the number of memory channels so that computation and off-chip communication are well balanced.

Figure 4b shows the results for the four SoC architectures of Figure 3 when running inference for different neural networks. Each SoC presents an increasing number of NVDLA instances and



(a) Performance of one NVDLA instance

(b) Scaling NVDLA instances and DDR channels.

Figure (4) Evaluating the ESP SoC prototypes on an FPGA board (each SoC runs at 50MHZ.)

memory channels utilized in parallel, compared to the previous one. The task parallelization delivers an approximately linear increase in performance. For instance, four NVDLA instances with four memory channels bring approximately a $4\times$ speedup. The parallelization benefit for ResNet-50 is partially limited by the non-negligible data preparation part of the application running serially on a single Ariane core.

Related Work

Among other open-source SoC design platforms, Chipyard supports the integration of Chisel accelerators with RoCC, a custom co-processor interface [10].

Centrifuge integrates HLS-generated accelerators into the bus-based SoC architecture provided by the FireSim FPGA-accelerated full-system hardware simulator [11]. Similarly, another work has integrated NVDLA in FireSim's RISC-V SoC architecture [12].

ESP focuses on SoCs containing many loosely-coupled accelerators designed with a variety of languages and tools. This work augments ESP to provide the automation needed for the seamless integration of third-party accelerators with their software.

Conclusions

We developed a design flow that simplifies the integration of third-party accelerators into complete SoC architectures and their evaluation via FPGA prototyping. We released the contributions of this paper in the public domain with the goal of supporting the progress of the open-source hardware community.

ACKNOWLEDGMENT

This research was supported in part by DARPA (C#:HR001118C0122) and the ARO (G#:W911NF-19-1-0476). The views and conclusions expressed are those of the authors and should not be interpreted as representing the official views or policies of the Army Research Office, the Department of Defense or the U.S. Government.

REFERENCES

1. Y. Shao and D. Brooks, *Research Infrastructures for Hardware Accelerators*. Morgan & Claypool, 2015.
2. W. J. Dally *et al.*, "Domain-Specific Hardware Accelerators," *Communications of the ACM*, 2020.
3. B. Khailany *et al.*, "A Modular Digital VLSI Flow for High-Productivity SoC Design," in *Proc. of DAC*, 2018.
4. G. Gupta *et al.*, "Kickstarting Semiconductor Innovation with Open Source Hardware," *IEEE Computer*, 2017.
5. P. Mantovani *et al.*, "Agile SoC Development with Open ESP," in *Proc. of ICCAD*, 2020.
6. L. P. Carloni, "The case for Embedded Scalable Platforms," in *Proc. of DAC*, 2016.
7. F. Zaruba *et al.*, "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology," *IEEE Trans. on VLSI Systems*, 2019.
8. D. Giri *et al.*, "Accelerators & Coherence: An SoC Perspective," *IEEE Micro*, 2018.
9. ——, "ESP4ML: Platform-Based Design of Systems-on-Chip for Embedded Machine Learning," in *Proc. of DATE*, 2020.
10. A. Amid *et al.*, "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs," *IEEE Micro*, 2020.
11. Q. Huang *et al.*, "Centrifuge: Evaluating Full-System

HLS-Generated Heterogenous-Accelerator SoCs Using FPGA-Acceleration," in *Proc. of ICCAD*, 2019.

12. F. Farshchi *et al.*, "Integrating NVIDIA Deep Learning Accelerator (NVDLA) with RISC-V SoC on FireSim," in *Proc. of EMC2*, 2019.

Davide Giri (davide_giri@cs.columbia.edu) is a Ph.D. student of computer science at Columbia University, New York.

Kuan-Lin Chiu (chiu@cs.columbia.edu) is a Ph.D. student of computer science at Columbia University, New York.

Guy Eichler (guyeichler@cs.columbia.edu) is a Ph.D. student of computer science at Columbia University, New York.

Paolo Mantovani (paolo@cs.columbia.edu) is an associate research scientist at Columbia University, New York.

Luca P. Carloni (luca@cs.columbia.edu) is professor of computer science at Columbia University, New York.