Centrifuge: Evaluating full-system HLS-generated heterogenous-accelerator SoCs using FPGA-Acceleration

Qijing Huang*, Christopher Yarp*, Sagar Karandikar*, Nathan Pemberton*, Benjamin Brock*,
Liang Ma†, Guohao Dai‡, Robert Quitt*, Krste Asanovic*, John Wawrzynek *
*University of California, Berkeley
{qijing.huang, cyarp, skarandika, nathanp, brock, robertquitt, krste, johnw}@berkeley.edu
†Politecnico di Torino; ‡Tsinghua University
liang-ma@polito.it; dgh14@mails.tsinghua.edu.cn

I. ABSTRACT

To overcome the end of traditional scaling, modern SoC systems consist of general-purpose compute augmented with large numbers of specialized accelerators. However, building and evaluating these systems is extremely expensive and time-consuming, even in early stages of development. While high-level modeling and back-of-the-envelope calculations can provide early insights into a new system, there are key effects that only manifest at the full-system level. However, full-system design has traditionally required writing RTL or developing complex software models for the entire design.

In this paper, we describe a methodology and implement an open-source flow ("Centrifuge") that can rapidly generate and evaluate heterogeneous SoCs by combining an HLS toolchain with the open-source FireSim FPGA-accelerated simulation platform. Our system can quickly produce complete SoC systems with many integrated HLS-generated accelerators as specified by the user, simulate them quickly and cycle-accurately on FPGAs, and run complete software stacks on top, including booting Linux and running full application frameworks. Our system allows users to easily explore a variety of accelerator integration techniques, by automatically integrating accelerators in several ways—as tightly coupled RoCC accelerators, as accelerators that communicate over the standard on-chip network, and lastly as "disaggregated" accelerators that are directly attached to an Ethernet network between SoCs. By integrating these tools, our methodology allows users to rapidly generate an entire hardware/software stack for a customized SoC that can be fabricated as an ASIC and evaluate its end-to-end performance using cycle-exact FPGA simulation, allowing for agile design-space exploration of novel accelerator-based systems.

II. INTRODUCTION

Due to the end of Moore's law and classical scaling, architects today must resort to building heterogeneous and specialized systems to continue to satisfy the ever-growing appetite for compute of today's applications. Today's systems-on-chips (SoCs) contain a multitude of accelerators to optimize for common workloads. However, this heterogeneity comes at the expense of increased hardware development time, not only including the design of individual accelerators, but also the selection and evaluation of the set of accelerators that should be included in a particular system. Early in the process of selecting a set of accelerators to include in a system, architects frequently rely on high-level/abstract software modeling. While this kind of modeling is sufficient for early design space exploration and saves the time of traditional cycle-accurate modeling or RTL design-entry, it generally does not account for effects that only manifest when an accelerator is integrated into a complex system.

To achieve a greater level of detail in evaluation, architects frequently use cycle-accurate full-system simulation platforms. These simulators span a wide range of design points, making tradeoffs in simulation accuracy, simulation performance, and ease-of-use. Broadly speaking, these simulators can be broken into software-based simulators and hardware-accelerated simulators. In comparison with hardware-accelerated simulation, software-based simulation is simpler to use, but requires significant modeling expertise and validation. Furthermore, due to low simulation performance, software-based cycle-accurate simulation is unable to run long-running workloads, which makes it difficult to determine if an accelerator is actually beneficial when deployed in a system. In comparison, FPGA-accelerated simulators are able to simulate systems at much higher simulation rates, but require specifying an accelerator design by writing RTL, which drastically slows down early design-space exploration. With either of these simulation techniques, a key hurdle is the fact that a design must be developed and converted into RTL or a software model, which requires a significant time investment.

To bypass this issue, High-Level Synthesis (HLS) tools have been developed, which allow users to specify designs in a more software-centric manner but produce an RTL design that can later be used in hardware-accelerated simulation. While HLS tools have traditionally been restricted to producing accelerators that run on FPGAs, recently there has been an explosion in the use of HLS tools to generate and refine accelerator designs that are ultimately integrated into a complex system and taped-out, including those at Google, NVIDIA, Bosch, Qualcomm, etc [1]. A key advantage of using HLS to generate accelerators is that the accelerators can be verified at the C-code-level and the HLS tool can be trusted to produce correct output. Verification in this form is considerably faster and more productive than traditional hardware verification [2].

In this paper, we describe a methodology and open-source toolchain that we developed to rapidly generate, and evaluate heterogeneous SoCs by combining an HLS toolchain with the open-source FireSim FPGA-accelerated simulation platform:

1) We provide a flow that generates full SoC systems containing user-defined accelerators written in HLS. This flow integrates the Rocket Chip SoC generator with custom accelerators generated with Vivado HLS. Accelerators in the generated system can be attached to the system in three ways: ① coprocessor-style RoCC accelerators, ② accelerators that connect to the SoC's on-chip network, and ③ disaggregated accelerators that attach directly to Ethernet.

2) We provide a flow that automatically generates software infrastructure to interact with the accelerators on the generated SoC systems from within accelerators.

3) We add a Verilog FAME-1 [3] pass to the open-source FireSim

simulator to support simulating designs that contain Chisel blackboxes of Verilog designs, in our case, the accelerator designs produced by Vivado HLS.

4) We generate SoCs with several integrated accelerators and evaluate accelerators with different coupling and software stack. In addition, we conduct three case-studies to demonstrate the capability of the toolchain.

With this methodology, we can rapidly generate an entire hardware/software stack for a customized SoC that can be fabricated as an ASIC and evaluate its end-to-end performance using FPGA simulation, allowing for rapid design-space exploration of novel accelerator-based systems, while providing cycle-exact performance measurements with little user effort. We call our implementation of this approach Centrifuge. Once a user is satisfied with the baseline accelerated system produced by Centrifuge, they can then continue to hand-optimize the design, as if they had written RTL from scratch.

III. RELATED WORK

With the increasing complexity of workload and systems in the datacenter, hardware-software co-design is becoming critical to truly optimize full-systems. Several projects have explored high-level modeling for accelerator design. Aladdin [4] is a software simulator that takes C code as input and estimates performance, power, and area of a target accelerator design. The program behavior is modeled with dynamic data dependence graphs (DDDG), which can be generated from C code directly. This model assumes that all data can be preloaded into the local scratchpads, which falls short for real designs with limited on-chip memory budget and complex memory access patterns. [5] addresses this issue by extending Aladdin with the gem5 full-system simulator [6] to provide support for simulating complex accelerator-system interactions. This work shows that the pareto-optimal Energy-Delay Product (EDP) points for accelerators evaluated in isolation differ from ones explored through full-system co-design. While this approach is fast and easy to deploy, detailed accelerator design insights are difficult to gain as no true hardware is generated. Besides, its simulator speed ($\sim$50KIPS) limits the deployment of full-stack software, whereas in our system that runs at tens of MIPS [7], [8], the real impact of accelerators can be manifested at the application level. PARADE [9] is another extension to gem5 that leverages HLS to generate accurate accelerator models for accelerator-rich architecture(ARA) on complex network-on-chips (NoCs). It provides a global accelerator manager to manage the accelerator runtime. In all of these cases however, the prior work does not move the designer towards obtaining an actual implementation—once these tools generate an accelerator design and a designer selects a particular set of accelerators, the designer must then write RTL or HLS for the accelerators or the glue logic. Thus, our methodology differs critically from these past approaches because it directly evaluates a rapidly-generated RTL-ready design which requires less additional architectural work to be realized in silicon. Embedded Scalable Platforms (ESP) [10] proposed a similar approach to design accelerator SoCs using HLS. Cosmos [11] has leveraged both HLS and memory optimization tools to improve design space exploration (DSE) for accelerators. Differing from ESP and Cosmos, we aim to provide a fast simulation environment to evaluate an accelerator in a full-stack setting. Our framework quickly provides a baseline set of interfaces and an easy-to-use simulation environment that software developers can program against and use
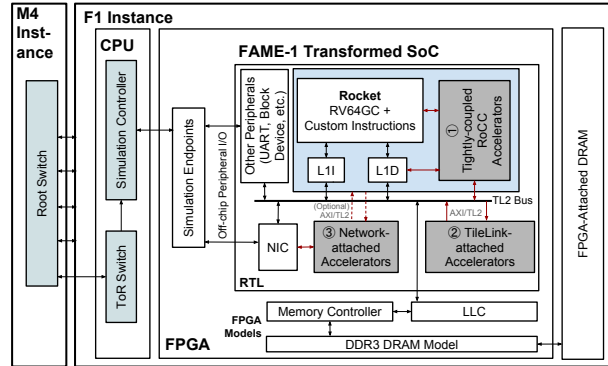

Figure 1: Block Diagram of FireSim Simulating Centrifuge-generated SoC with Accelerators

for performance optimization of the software stack, even before real silicon is available.

IV. CENTRIFUGE

In this section, we detail the key components of our flow for agile generation and evaluation of multi-accelerator SoCs, named *Centrifuge*[1]. We first describe how we build an SoC with integrated HLS-generated accelerators, then outline our extensions to the FireSim FPGA-accelerated simulation platform [12] to enable fast cycle-exact simulation of our generated SoCs.

A. Generating a Base SoC with Rocket Chip

As the basis for our SoC system, we use the Rocket Chip generator [13], an open-source SoC generator written in Chisel that provides standard SoC components including the RISC-V Rocket Core (replaceable with the BOOM Out-of-Order core) and uncore components. FireSim provides several standard peripherals including a UART, Block Device, and NIC [12]. Altogether, this produces a Linux-capable RISC-V SoC that can interface with a standard Ethernet network. The base SoC components are shown in the "RTL" box in Figure 1, excluding the gray accelerator boxes. We configure the system to have 16 KB L1 I/D Caches, a 4 MiB LLC, 16GB of DDR memory, and a 200 Gbit/s Ethernet NIC. The gray boxes in Figure 1 show three methods for integrating accelerators into the SoC. We detail these in the following section.

B. Integrating Accelerators into the SoC

To enable exploration of various accelerator designs, we supply shim infrastructure to incorporate HLS-generated accelerators into the aforementioned SoC in three distinct ways:

① RoCC accelerators (coprocessor sharing L1 and LLC with the processor, invoked by RoCC instruction)

② TileLink accelerators (closely-coupled accelerator sharing LLC with the processor, invoked by either RoCC instruction or memory-mapped I/Os(MMIO))

③ Network-attached accelerators (TileLink accelerators with direct connection to the Ethernet)

These three models are representative of recent academic and commercial designs [14]–[16]. Below we outline these three configurations in detail.

---

[1]Our tool is named Centrifuge because it lets us *rapidly iterate* on novel many-accelerator SoCs and *separates* the good accelerators from the bad.
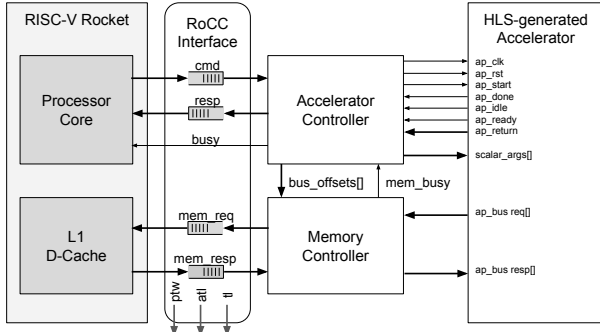
Figure 2: RoCC Accelerator

1) RoCC Accelerators: The RoCC accelerator interface provided by Rocket Chip allows a user to integrate an accelerator closely with the processor in the SoC. The accelerator can receive commands directly from the general-purpose processor through a dedicated command queue. Programs can issue these commands using custom RoCC instructions that fit within the RISC-V ISA. RoCC accelerators also have ports directly into the private L1 data cache of the general-purpose core, as well as a port into the next-level cache in the system. These interfaces are shown in 2. The L1 data cache access that the RoCC interface provides is architected around a request/response model. One key aspect of the RoCC memory interface is that requests are not guaranteed to execute and return in order. A tag field in the request and response packets is used to associate responses to their corresponding requests.

Our HLS-generated RoCC accelerators use the Vivado HLS ap_bus interface for pointer type arguments. The ap_bus interface is a standard interface used by Vivado HLS. One key difference between ap_bus and RoCC is that ap_bus assumes that dependent memory operations occur in order. It is important to note that Vivado HLS generates separate ap_bus ports for each pointer and array argument given when specifying an accelerator. This means that order must be maintained between buses as well as within a single bus. With ap_bus, the behavior of simultaneous reads and writes to the same address in the same cycle is undefined as there is no mechanism to determine the expected ordering between buses in the same cycle. This is not an issue experienced by software C developers who expect sequential access to memory and do not expect multiple memory transactions to occur simultaneously. While Vivado will check for scheduling conflicts within one variable, a conflict could potentially occur if two pointer or array arguments address an overlapping range of memory.

To resolve these issues, we implement a custom memory/accelerator controller bridges that allow Vivado HLS generated accelerators to be attached to the RoCC interface. To maintain the relative order of dependent memory accesses, we add issue logic on the memory request path that is similar to scoreboarding in a single-issue out-of-order processor. To handle responses, we add return logic that resembles a network switch. A detailed block diagram of the memory bridge is presented in 3.

2) TileLink-attached accelerators: Looser coupling of accelerators to a local application core's LLC is achieved by attaching accelerators to the on-chip TileLink interconnect [17] in Rocket Chip. TileLink is an open and free chip-scale cache-coherent interconnect standard used by the Rocket Chip SoC-generator to connect devices on low-latency SoC buses. It supports a MOESI-
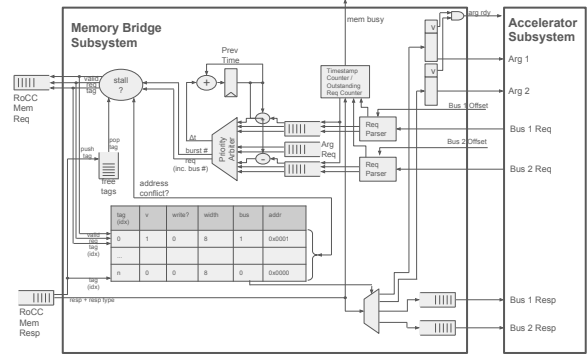

Figure 3: RoCC-ap_bus Memory Bridge Architecture

equivalent protocol to provide coherent access for an arbitrary mix of caching or non-caching masters. There are three levels of conformance protocols and five channels implemented as five physically distinct unidirectional parallel buses with one sender and one receiver on each. The completion of data transactions is out-of-order to improve throughput.

The Rocket Chip generator also takes advantage of a library called Diplomacy [18], which supports automatic parameter negotiation and checking between SoC components. Users only need to specify protocol requirements on the master and slave nodes on a TileLink bus. At elaboration time, diplomacy automatically negotiates protocol implementations, then generates bus and interrupt signal connections. In HLS C programs, pointer-type arguments to a C function are synthesized into AXI4 master ports when the *m_axi* interface pragma is specified. Each memory access in the C code is turned into an AXI4 request in the generated hardware. To attach accelerators with these AXI-4 memory systems generated by Vivado HLS, we use an open TileLink-to-AXI4 bridge adapter [17] to connect accelerators to the Rocket Chip SoC.

3) Network-attached accelerators: The last accelerator-integration option we provide is to allow accelerators to directly interface with an external Ethernet network by directly communicating with the in-SoC NIC to send/receive packets, without the intervention of the general-purpose processor. There are two ways for the accelerator to directly send and receive data to and from the network.

Accelerator MMIO to NIC. The accelerator can directly post send and receive commands to the NIC in the SoC by accessing the NIC's MMIO control registers in the HLS code. The send and receive buffers are pre-allocated as local buffers and set to the *s_axilite* interface in the HLS wrapper, so both buffers appear to be memory-mapped slaves on the TileLink bus. The accelerator can issue a send command by telling the NIC the base address and the length of the data it intends to send and a destination MAC address. If the accelerator is expecting incoming data, it issues a *post_recv* command to the NIC with the address of the receive buffer. Currently, we require that the general-purpose processor not simultaneously access the network interface while the accelerator is using the NIC.

Dedicated Send/Receive Queues. Accelerators can also directly communicate with the NIC and thus the external Ethernet network through dedicated send and receive queue pairs. To implement this functionality, we extend the NIC design with routing/tagging based on the Ethernet Ethertype field. The NIC automatically directs
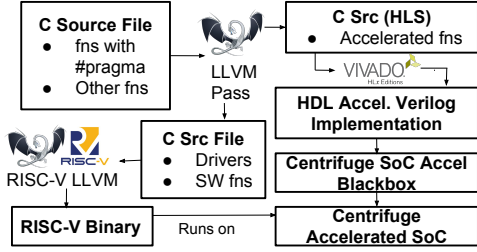
3

Figure 4: Centrifuge HLS Flow, C to Acclerator RTL + Software Driver/Application

network packets to the accelerator's queues if the Ethertype value is "`ACCEL_ONLY`".

In this mode, rather than issuing commands and polling the NIC, the accelerator can directly send and receive the Ethernet packets through decoupled FIFO queues. We split each queue into two sub-queues, one for passing the Ethernet header and one for passing the payload. We can then treat the payload queue as a data stream in a dataflow programming model with blocking read and non-blocking write. This allows us to take a design written in the Vivado HLS dataflow model and split it into multiple network-attached accelerators. Currently, the flow requires that you have sufficient buffering and that the send and receive rate are matched. In future work, we plan to add a flow-control mechanism to designs.

C. Generating Accelerators with Vivado HLS

As alluded to earlier in this section, we generate accelerator RTL by taking advantage of Vivado High-Level Synthesis (HLS) rather than requiring users to manually write RTL for accelerator designs. While HLS tools have previously been relegated to FPGA-based deployment [19], [20], ASIC CAD tool vendors have begun to ship HLS tools geared towards ASIC designers [1], [21], [22].

The process of converting a high-level (C) description of an application to a hardware accelerator in Centrifuge is detailed in 4.

At a high-level, the flow to integrate HLS-generated accelerators into the SoC is as follows:

1. The programmer develops a standard C program and identifies a function to accelerate (HLS-compatible).

2. Our LLVM pass replaces all calls to the accelerated function with calls to a new wrapper function. This wrapper function calls the accelerator.

3. LLVM writes a RISC-V assembly file which is assembled and linked by the standard RISC-V GCC toolchain.

4. The function name to be accelerated is placed into a tcl script that is used to drive HLS.

5. Vivado HLS produces a Verilog implementation of the accelerated function.

6. Our custom FAME-1 transformation is applied to the generated Verilog.

7. A pair of controllers are attached to the accelerator and act as bridges between the accelerator and the RoCC/TileLink2/Network interfaces. They handle the command/response messages between the processor and the accelerator and memory request/response messages.

8. The accelerator is added to the SoC and the design is elaborated by Chisel.

The following sections outline the key differences between generating local (RoCC or Tilelink) accelerators and disaggregated/distributed accelerators from high-level specifications.

1) Generating RoCC or Tilelink Accelerators: We developed an LLVM pass to collect information about the function to accelerate and to replace the accelerated function with custom instructions or wrapper functions that invoke the accelerator. Once LLVM receives a function name as command line input, our LLVM Module Pass iterates through the function list in the program module and finds the function with the input function name. Since C does not support overloading, each function should have a unique name and function names can be used as function identifiers.

Upon finding the function, our LLVM pass can emit the type information of the function arguments and the return value, which is used in our automation flow. We then create a new wrapper function that shares the same prototype as the accelerated function and construct the new function body using the LLVM IRBuilder.

For a RoCC-attached accelerator, the main part of the function body is formed by three lines of RISC-V inline assembly: one RISC-V custom instruction that calls the accelerator and two fence instructions before and after the custom instruction to ensure memory consistency. The custom inline assembly takes the function arguments as its register inputs and returns the output to the return register. Once the accelerator finishes running, the function will return the value of the return register.

For TileLink-attached accelerators, the flow generates code to supply the accelerator's arguments by writing the input values to corresponding MMIO addresses that map to registers inside the generated accelerator. The MMIO addresses are parsed from generated Verilog code for the accelerator. The control code to start and poll for the accelerator done signal is also emitted.

Finally, the pass loops through the function call site and replaces all uses of the accelerated function with calls to the accelerator wrapper.

2) Generating Network-attached Accelerators: For this accelerator integration, the user must write the application in a dataflow model that conforms to the Kahn Process Network model [23]. The user must first verify that the implementation does not deadlock in software. Then, the user can partition their dataflow implementation into many standalone accelerators and directly connect the data streams to the network interface wrapper generated by Centrifuge, then expose the network interface arguments to the top level of each HLS design. This flow allows users to build large disaggregated accelerator systems at a much higher-level of abstraction than manually writing RTL and manually interacting with a NIC.

D. Generating the software stack for a complete SoC

To complete the generation of our SoC system, we need to produce software shims that provide access to the generated accelerators from various levels of the software stack. In the previous section, we discussed how our LLVM pass will generate workload binaries with calls to the accelerator, either as bare-metal programs or programs that expect to run on Linux. Below, we outline the system-level software shims to provide access to accelerators.

1) Running Bare-metal: In a bare-metal environment, the interaction between software and accelerators is straightforward. In order to invoke a RoCC accelerator, the custom instruction assigned to the target accelerator needs to be called with arguments stored in

4

processor registers. For TileLink accelerators, we need to perform store operations to the memory-mapped registers to pass function arguments and control commands. In both cases, similar to a software function call, we pass in scalar arguments directly and pointer arguments as physical memory addresses. The accelerator can directly access memory through the caches (L1 for RoCC, L2 for Tilelink). All requests are serviced by the memory controller without directly involving the processor.

2) Running on Linux: In Linux, RoCC-based accelerators are invoked using custom instructions and can use the processor TLB to perform translations; they do not require special operating system drivers. Tilelink accelerators, however, become more complex with an operating system due to virtual memory handling. Specifically, the drivers and software wrappers for an accelerator/application must support the following three features:

Accessing MMIO from user space. TileLink accelerators are controlled through memory-mapped physical addresses. In order for the application to access these addresses, they must first be mapped into the application's virtual address space. On Linux, physical memory can be accessed through the "/dev/mem/" special file. By calling `mmap` with the offset set to the desired physical address, we can map any physical address into our virtual memory. This procedure is handled automatically in our generated wrapper code.

Translating from virtual to physical addresses. To handle address translation, the software wrappers for TileLink accelerators call an automatically-generated RoCC accelerator that interacts with the hardware page-table walker to translate from virtual to physical addresses.

Ensuring physically contiguous data-layout. For pointer arguments that fit within a single page, translation alone is sufficient. However, if the argument spans multiple pages, the allocated memory may not be physically contiguous. In lieu of maintaining a TLB in each accelerator, Centrifuge requires that all arguments be made physically contiguous before invoking an accelerator. To allow this, we provide a Linux driver that allocates a large, physically-contiguous, region of memory at boot time and exposes it to users through a modified mmap system call. Users can allocate space for their arguments using this system call (minimizing overheads). If the user would prefer to not modify their source, the generated function wrappers can copy arguments into contiguous memory automatically when the accelerator is invoked.

V. Case Studies

In this section, we evaluate several microbenchmarks and perform three case studies to demonstrate the capability of our methodology. We ran our experiments on FireSim on Amazon F1 instances and use Vivado HLS to synthesize application C code into hardware as it is stable and free to the community. Our microbenchmarks are adapted from CHStone [24] and HLSpolito on GitHub [25]. With the microbenchmarks, we demonstrate that Centrifuge can be used to evaluate the following design tradeoffs:

Acceleration Region. We first conducted a sweep to extract functions from the microbenchmarks and compiled them to the RoCC accelerators with Centrifuge. Results in Figure 5 showing the accelerator speedup can be used to direct decision on which code region to accelerate. For example, the *adpcm* example should be accelerated at the *encode* level instead of at the basic operation level (e.g. *logsch*, *quantl*, *logscl*) .
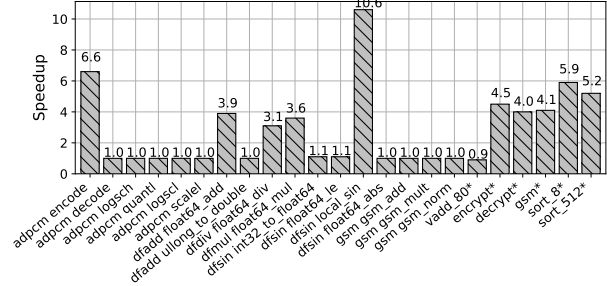

Figure 5: RoCC Accelerators SpeedUp Compared to Software(* indicates accelerator with pointer type inputs)
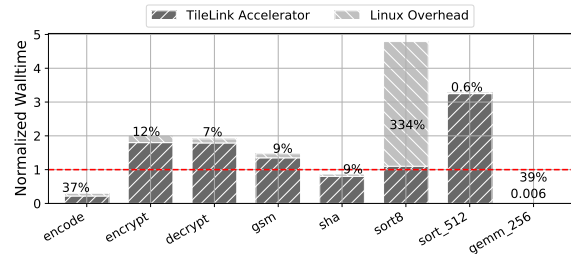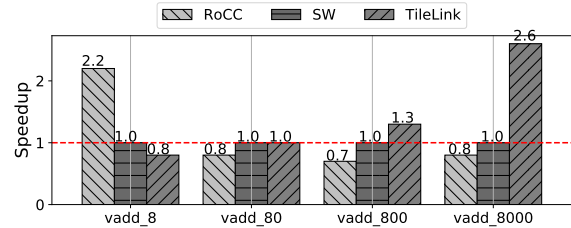

Figure 6: Tilelink Accelerators with Linux Driver

Figure 7: Different Coupling for *vadd* Accelerator

Software Stack. We then generated five TileLink accelerators and ran them under Linux. Figure 6 shows the runtime breakdown of the accelerators normalized to the software performance. The slowdown of Tilelink accelerators on Linux is mostly due to performing address translation on each argument. Note that RoCC accelerators do not experience any slowdown on Linux because they are virtually-addressed. With Centrifuge, we can evaluate and optimize the physically-addressed TileLink accelerator and its Linux driver together.

Accelerator Coupling. Lastly, we show how different coupling affects the accelerator performance with Centrifuge by accelerating a communication-bound kernel *vadd* in different sizes. From Figure 7, we see that the RoCC accelerator outperforms software when the vector size is small. As we increase the vector size, the TileLink accelerator gets a higher speedup compared to software. There are three main factors that affect the accelerator speedup: the interface bandwidth, the cache hit latency and the cache size. The TileLink system bus is 512-bit wide, whereas the RoCC memory interface is 64-bit wide. The L2 hit latency is $20\times$ longer than L1 hit latency. For the RoCC accelerator, once it starts to miss in L1 cache, it will suffer from a similar cache access latency as the TileLink accelerator. However, since TileLink accelerator has wider memory accesses, it performs better in a more bandwidth-bound scenario.
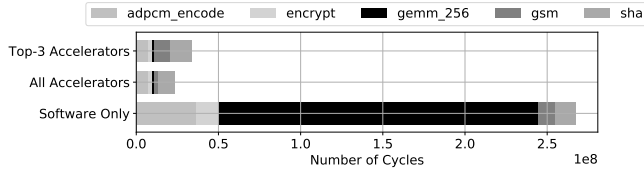
Figure 8: Breakdown of key computational kernels in a hypothetical smart-house assistant SoC. The top-3 accelerators for end-to-end performance are *adpcm_encode*, *gemm_256*, and *encrypt*

A. Putting it all together: Smart-House Hub

In this case study, we demonstrate Centrifuge using a hypothetical SoC intended for a smart-house assistant (e.g. Alexa, Google Home, etc...). Our device will need to listen for user audio commands, encode them into an appropriate format, perform machine-learning inference to detect commands, and finally encrypt the command for transfer to the cloud over a wireless network.

1) Evaluating the Baseline Application: We begin by measuring runtimes for each of these kernels without accelerators. Figure 8 shows how the runtimes of these steps might compare in a typical deployment ("Software Only"). Notice that the lion's share of time is spent in audio preprocessing (*adpcm_encode*) and the matrix-multiply underlying command classification (*gemm_256*). The remaining time is split roughly evenly between hashing (*sha*), encryption (*encrypt*), and wireless encoding (*gsm*).

2) Generating Accelerators: Having identified the key kernels in our application, we begin by adding HLS annotations to each function. This mostly involves identifying inputs and outputs, and ensuring the function prototype has the correct number of arguments. By modifying the source code and annotations, the design can be further specialized for hardware deployment if appropriate. We then run Centrifuge on our annotated application, specifying each kernel. The result is RTL for a new SoC with the specified accelerators and a new application binary with each accelerated function replaced with a call to an accelerator.

3) Evaluating Accelerators: The next step is to evaluate our accelerators in an end-to-end system using FireSim. With FireSim, we can run our code as if it were on a real machine, including any timing measurements. Figure 8 shows the runtime breakdown using our new accelerators ("All Accelerators"). We notice that *gemm_256* shows the greatest improvement, both local (250×) and end-to-end (11.5×), and should likely be included. The adpcm encoder shows a more modest local improvement (about 5×) but has the second largest impact on total runtime (6% end-to-end improvement). Both encryption and gsm-encoding show 4-5x improvements in local runtime, but have a modest 1% impact on end-to-end performance; we may choose to include these if power and area permit. Finally, the sha accelerator sees little improvement locally, and has a very small impact on end-to-end runtime; we would likely choose to not accelerate that function.

4) Continue Hardware and Software Development: Putting it all together, we decide to include the *gemm_256*, *adpcm_encode*, and *encrypt* accelerators and leave the remaining kernels to the CPU. This results in an 8x improvement in end-to-end runtime (including all the accelerators would result in an 11.5% improvement). In addition, we now have a consistent hardware/software interface and a high-performance simulator for use by our software team, while hardware engineers can continue to optimize the kernels, using either HLS or hand-written RTL.
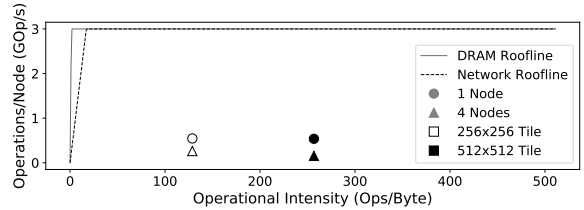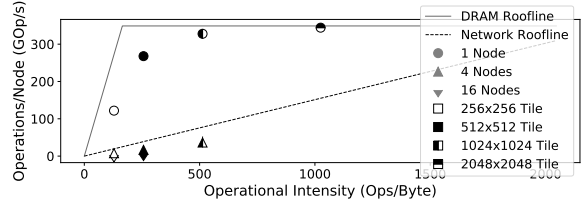

Figure 9: CPU Roofline Model

Figure 10: Accelerator Roofline Model

B. Distributed Matrix Multiplication Accelerator

We applied Centrifuge to a MPI-based distributed matrix multiplication implementation [26]. This algorithm employs MPI's one-sided communication protocol, along with extensive tiling and prefetching. We used two separate implementations for the core matrix-multiplication algorithm; an HLS-optimized kernel (adapted from [27]), and a CPU-optimized tiling algorithm. The accelerator is integrated as a TileLink accelerator. It runs 441 × faster than the CPU for performing 8-bit integer matrix multiplication.

We first validated the HLS design by running the C simulation and comparing the output against a golden reference. This takes less than a minute for each debugging iteration. We then used Centrifuge to generate the SoC and simulated the accelerator on FireSim. It took ~3 hours to generate the FPGA images and seconds to return the test results for multiplying matrices of size 256×256. In comparison, it would take around half a day to run the bare-metal program on a commercial software-based RTL simulator. Using a software-based RTL simulator, it would be infeasible to validate the design in a more realistic setting, for example, using a larger input size, running under Linux, and in a network environment.

We can further evaluate the complex interaction between the accelerator, CPU, and the network by employing Centrifuge. To evaluate the design, we first calculate peak performance using a roofline model [28]. Because we are running a full-system cycle-level simulator, we were able to use standard evaluation tools like STREAM [29] and iperf [30] to find the actual bounds for the roofline model. The distributed dgemm framework described above was taken from an existing high-performance library and run unmodified (except for calls to the accelerator and special memory allocation of arguments as described in section IV-D1). We ran the experiments on 1, 4, and 16-node configurations with 2.0 GB/s measured DRAM bandwidth and 1.2 Gbit/s measured network bandwidth.

Figure 9 shows that the performance of the workload on CPU is dominated by a lower bound than its peak compute bound. As shown in Figure 10, by running the accelerator, the workload becomes communication bound, and our accelerator performance matches the measured roofline. The accelerator achieves a peak throughput of 344 GOp/s for 16-bit integer multiply-accumulates
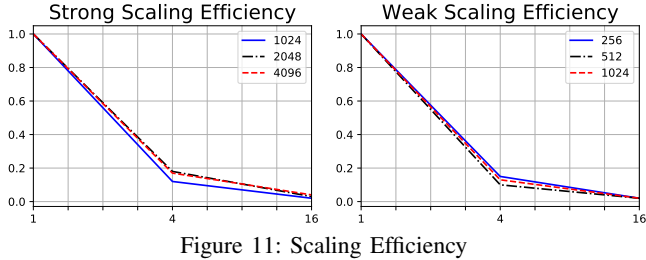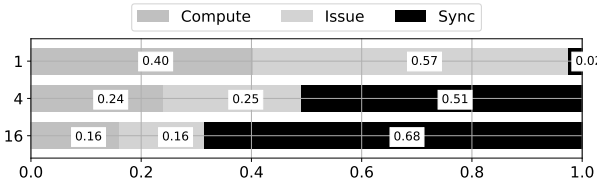
Figure 11: Scaling Efficiency

Figure 12: DGEMM Runtime Breakdown for 1024×1024 Tiles

(a ∼ 600× improvement compared to our processor). On one node, the performance of accelerator follows the DRAM roofline (memory communication bound), while on four or sixteen nodes, its performance tracks the network roofline instead (network communication bound). Therefore, for the distributed workload in this example, major improvement should be made to the network bandwidth instead of the accelerator itself. Figure 11 shows the strong and weak scaling efficiency of the distributed workload running on the accelerators. A detailed runtime breakdown for the distributed workload with tile size 1024×1024 is shown Figure 12.

C. Deep Learning Accelerators

Our flow, with fast design feedback, is particularly suited for developing accelerators for rapidly changing deep learning algorithms. In this section, we will describe several deep learning accelerators developed with Centrifuge. Note that all the accelerators in this section took less than one month to implement.

D. Design for New Algorithms

Figure 13 shows the basic building block for a new efficient network design called DiracDeltaNet [31]. In this design, all 3x3 convolutions are replaced with a 1x1 convolution and shift operation, while the addition-skip connection is replaced with concatenation and shuffle operations. Figure 14 shows our hardware dataflow design for the building block. In the design, all layers are spatially mapped to corresponding hardware units. There are three 8×8 Multiply-Accumulate(MAC) units to support the three 1x1 convolution layers. The weights are pre-fetched into the on-chip buffers. The input activations are loaded to the FIFOs from DRAM. Each hardware unit starts its execution based on the arrival of data. Since we preload all the weights, each input activation can be reused output_channel_size times after it is fetched from DRAM. Table I shows the Ops/cycle for different DiracDeltaNet subgraphs on our TileLink accelerator. As the compute-to-communication ratio varies among different subgraphs, the empirical performance of the accelerator varies drastically (∼4×). The algorithm designer can then leverage the information from current hardware architecture to optimize the DNN design to include more hardware-efficient layers.

E. Distributed Accelerators

The dataflow architecture with large weight buffers mentioned in the previous example is also known to have low latency for
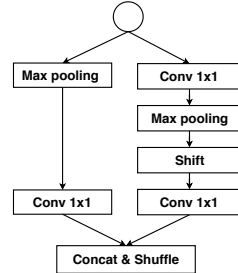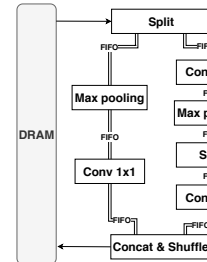

Figure 13: DiracDeltaNet

Figure 14: Hardware Design

Table I: Accelerator Performance(The workload size is represented as image_width × channel_depth)

Workload Size	Total Ops	Ops/cycle
32×16	196608	4.55
32×32	786432	15.12
32×64	3145728	20.59
16×128	3145728	21.35
8×64	196608	17.09

inference with a small batch size. However, it might not be economical to have such a large design with all the layers hard wired together. Instead, we can have many composable deep neural network accelerators with different dataflow modules, and have them directly communicate with each other through a high-performance network as shown in Figure 15. We implement this design based on VGG16 [32]. We first tested the idea with a small 2-layer neural network with two 16×16 Conv3×3. By replacing the data stream with the Ethernet connection, we reduced the total latency by 1.5%. This indicates that the overhead from the direct network connection is tolerable. We then prototyped two accelerators with our framework: one with only the convolution clusters, and one with both convolution and fully connected layers for reducing the results. Both designs can directly send and receive Ethernet packets to the network through the NIC. The weights are 2-bits and the activations are 4-bits in the hardware. Assuming the chip is running at 3.2 GHz, for a 64 × 64 large image, it takes 13191136 cycles (4.1ms) to classify 1 frame on a single node accelerator, and 11151953 cycles (3.5ms) to finish the same task on a two-node system that has direct accelerator-to-accelerator network connections. While both designs have the same number of compute units, the two node design benefits from increased aggregate memory bandwidth. In this case, the benefits of increased memory bandwidth outweigh any overheads from the network.

VI. CONCLUSION

In this paper, we described a methodology and flow, *Centrifuge*, that can rapidly generate and evaluate heterogeneous SoCs by combining an HLS toolchain with the open-source FireSim FPGA-accelerated simulation platform. Our system can quickly produce complete SoC systems with many integrated HLS-generated accelerators as specified by the user, simulate them quickly and cycle-accurately on FPGAs, and run complete software stacks on top, including booting Linux and running full application frameworks. Our system allows users to easily explore a variety of accelerator integration techniques, by automatically integrating accelerators in several ways—as tightly coupled RoCC accelerators, as accelerators that communicate over the standard on-chip network, and lastly as "disaggregated" accelerators that are directly attached to an Ethernet
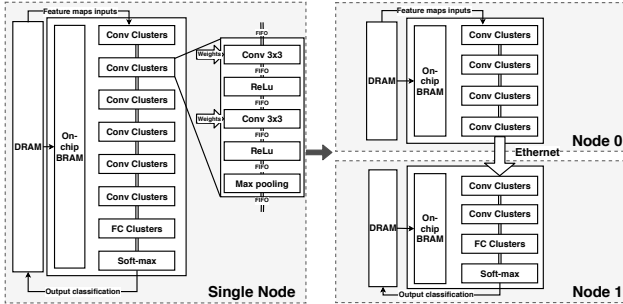
Figure 15: Multi-node accelerators, connected via Ethernet

network between SoCs. We extended the FireSim simulation platform with a new FAME-1 transformation that operates on the Verilog designs emitted by Vivado HLS rather than Chisel RTL. By integrating these tools, our methodology allows users to rapidly generate an entire hardware/software stack for a customized SoC that can be fabricated as an ASIC and evaluate its end-to-end performance using cycle-exact FPGA simulation, allowing for agile design-space exploration of novel accelerator-based systems. We plan to open-source our flow in the near future, allowing users to prototype their own accelerators using the Centrifuge methodology.

REFERENCES

[1] Mentor, "Catapult high-level synthesis." [Online]. Available: https://www.mentor.com/hls-lp/catapult-high-level-synthesis

[2] B. Khailany *et al.*, "A modular digital vlsi flow for high-productivity soc design," in *2018 Design Automation Conference (DAC)*, 2018.

[3] Z. Tan *et al.*, "A case for FAME: FPGA architecture model execution," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, 2010.

[4] Y. S. Shao *et al.*, "Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures," in *International Symposium on Computer Architecture (ISCA)*, 2014.

[5] Y. S. Shao *et al.*, "Co-Designing Accelerators and SoC Interfaces using gem5-Aladdin," in *International Symposium on Microarchitecture (MICRO)*, 2016.

[6] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, 2011.

[7] D. Kim *et al.*, "Evaluation of risc-v rtl with fpga-accelerated simulation," in *First Workshop on Computer Architecture Research with RISC-V*, 2017.

[8] D. Biancolin *et al.*, "FASED: Fpga-accelerated simulation and evaluation of dram," in *Proceedings of the 2019 International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019.

[9] J. Cong *et al.*, "PARADE: A cycle-accurate full-system simulation platform for accelerator-rich architectural design and exploration," in *International Conference on Computer-Aided Design(ICCAD)*, 2015.

[10] L. Piccolboni *et al.*, "Broadening the exploration of the accelerator design space in embedded scalable platforms," in *High Performance Extreme Computing Conference (HPEC)*, 2017.

[11] L. Piccolboni *et al.*, "Cosmos: Coordination of high-level synthesis and memory optimization for hardware accelerators," *ACM Transactions on Embedded Computing Systems (TECS)*, 2017.

[12] S. Karandikar *et al.*, "Firesim: Pfpga-accelerated cycle-exact scale-out system simulation in the public cloud," in *2018 Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[13] K. Asanović *et al.*, "The Rocket Chip Generator," Tech. Rep. UCB/EECS-2016-17, Apr 2016.

[14] W. Qadeer *et al.*, "Convolution engine: balancing efficiency & flexibility in specialized computing," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, 2013.

[15] Xilinx, "Zynq-7000 SoC - Xilinx." [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[16] B. Blaner *et al.*, "IBM POWER7+ processor on-chip accelerators for cryptography and active memory expansion," *IBM Journal of Research and Development*, vol. 57, 2013.

[17] SiFive, "Sifive tilelink specification," August 2017. [Online]. Available: https://static.dev.sifive.com/docs/tilelink/tilelink-spec-1.7-draft.pdf

[18] LowRISCV, "Diplomacy and TileLink from the Rocket Chip." [Online]. Available: https://www.lowrisc.org/docs/diplomacy/

[19] Intel, "Intel FPGA SDK for OpenCL ." [Online]. Available: https://www.intel.com/content/www/us/en/programmable/products/design-software/embedded-software-developers/opencl/developer-zone.html

[20] Xilinx, "Vivado design suite user guide - high-level synthesis," June 2015. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[21] Synopsys, "Next generation synthesis for advanced node designs." [Online]. Available: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test.html

[22] Cadence, "Stratus high-level synthesis." [Online]. Available: https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/digital-design-signoff/stratus-ds.pdf

[23] K. Gilles, "The semantics of a simple language for parallel programming," *Information processing*, vol. 74, 1974.

[24] Y. Hara *et al.*, "Chstone: A benchmark program suite for practical c-based high-level synthesis," in *2008 IEEE International Symposium on Circuits and Systems*.

[25] P. di Torino, "High level synthesis of bitonic sorting algorithm." [Online]. Available: https://github.com/HLSpolito/Bitonic-Sorting

[26] B. Brock *et al.*, "BCL: A cross-platform distributed container library," 2018.

[27] Xilinx, "General Matrix Operation." [Online]. Available: https://github.com/Xilinx/gemx

[28] S. Williams *et al.*, "Roofline: An insightful visual performance model for floating-point programs and multicore architectures," *Communications of the Association for Computing Machinery*, 2009.

[29] J. D. McCalpin, "STREAM: Sustainable Memory Bandwidth in High Performance Computers," https://www.cs.virginia.edu/stream/.

[30] "iPerf - The ultimate speed test tool for TCP, UDP and SCTP." [Online]. Available: https://iperf.fr/iperf-doc.php

[31] Y. Yang *et al.*, "Synetgy: Algorithm-hardware co-design for convnet accelerators on embedded FPGAs," in *Proceedings of the 2019 International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[32] K. Simonyan *et al.*, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.