Enabling Scalable Heterogeneous Hardware Integration Co-simulation with Socket IPC

Ruohan Yan
UC Berkeley
Berkeley, California
Email: yrh@berkeley.edu

Zekai Lin
UC Berkeley
Berkeley, California
Email: zekailin00@berkeley.edu

Hansung Kim
UC Berkeley
Berkeley, California
Email: hansung_kim@berkeley.edu

John Kubiatowicz
UC Berkeley
Berkeley, California
Email: kubitron@cs.berkeley.edu

Yakun Sophia Shao
UC Berkeley
Berkeley, California
Email: ysshao@berkeley.edu

***Abstract*—Amidst the challenges of an increasingly heterogeneous hardware landscape, the integration and evaluation of new hardware intellectual properties (IPs) remain a significant problem, constrained by complexity and inefficiency. In the first half of this paper, we introduce a simple and intuitive abstraction of procedure calling using socket inter-process communication (IPC) across hardware blocks, designed to address these challenges. We present a lightweight implementation in C++ and evaluate various design choices.**

In the second half of this paper, we perform two in-depth case studies of hardware integration using the proposed library. The first case study showcases a CPU-GPU co-simulation. We demonstrate the modularity of the communication scheme by showing mix-and-match capabilities through combinations of functional and Register-Transfer Level (RTL) simulations, and show up to 3.1× faster simulation, with only 5% cycle inaccuracy in large kernels. In the second case study, we examine a many-accelerator co-simulation executing a transformer encoder workload, showcasing communications on fine granularity, parallelization capabilities, and simulation scalability. We show parallel simulations enabled around 15-35% faster simulation time over an approximated monolithic SoC integration baseline with 8 workers, and true cycle numbers can be predicted with 94.9% accuracy.

Our findings indicate that our socket-based hardware communication library significantly eases the integration process for simulation. We believe it facilitates faster design iteration through accurate true cycle time prediction, single-component testing, and efficient simulation.

I. Introduction

The contemporary computer architecture landscape has entered a renewed phase of rapid development. Traditional architectures like CPUs and GPUs are seeing more significant development efforts as new products enter the market, with an emphasis on more parallelism and increased heterogeneity. On the CPU side, AMD has fitted the datacenter-oriented 4th generation EPYC with up to 128 cores [1], while the newly announced Ventana Veyron V2 boasts up to 192 cores at 16 cores per cluster [12]. Growing heterogeneity in the newer System-on-Chips (SoCs) has been a general trend, driven by the diverse requirements of modern computing, such as artificial intelligence (AI). For instance, SambaNova Systems has designed a novel dataflow architecture to more efficiently inference very large models [20]. Tenstorrent has been pursuing a similar goal as well [21]. Apple's recently released M3 Max contains a ray tracing accelerator, a 16-core neural engine, and a number of media encode and decode engines [3]. Examples like Cerbras' WSE-2, which is a neural network accelerator 56× larger than the largest GPU [9] show scaling and heterogeneity in one package.

As such, there is a growing need to simulate different novel architectures from various vendors, and in a scalable way. Traditionally, this requires modifying each hardware design extensively to ensure compatibility in terms of memory system, clocking, and instruction sets, among others. In the end, all components are merged into a single large *monolithic* design for simulation, iteration, and performance modeling.

This process not only incurs significant engineering costs, but the end result is slow to simulate due to the large size of the design, impeding verification [14]. Especially, this monolithic simulation is undesirable if only a single component in the design, such as an accelerator, is being iterated on during development while others are kept unchanged, as the increased simulation time might slow down the iteration speed and make it harder to efficiently explore the design space. This is also true for integrating an external IP, where the designer might be concerned more with correctly interfacing the IP with the rest of the system, rather than the fidelity of its simulation.

Previous work has attempted to tackle this problem by enabling co-simulation, where the individual components of a larger design are simulated in discrete simulation instances, and the framework implements communication methods between the instances to correctly interface the designs and construct a larger system. However, the proposed frameworks either have limited support for simulation backends [19], necessitate the use of a specific implementation environment for the target design [18], or use point-to-point communication methods that are hard to scale [5], requiring significant engineering effort to apply to a diverse set of target IPs.

In this paper, we propose a novel co-simulation framework with a specific focus on scalability and low effort of integration. Our framework features a simulator-agnostic socket-based inter-process communication scheme and a corresponding software library that can be easily adapted to different hardware IPs, and a server-client architecture for better scalability to larger designs. Specifically, our key contribution can be summarized as follows:

- We propose a co-simulation framework that enables scalable, coherent integration of discrete hardware simulations into a larger design, without requiring major modification in the design components.
- We showcase that our co-simulation framework supports the mix-and-match of different simulation backends, enabling useful tradeoffs between simulation speed and fidelity.
- We propose a server-client architecture for message passing that enables scaling out to larger designs with multiple simulation endpoints.
- We demonstrate in two separate case studies that our co-simulation framework can be easily integrated into software only, can achieve speedup over corresponding monolithic simulations, and make segregated design iteration possible while maintaining correctness.

The rest of the paper is structured as follows. In Section II, we provide background and motivation for the framework and the case studies. In Sections III, IV, and V, we outline the design and implementation of the library, followed by benchmarks. In Sections VI and VII, we present two detailed case studies: a CPU-GPU co-simulation, and a many-accelerator integration. In Sections VIII and IX, we discuss further research directions and conclude our work.

II. Background & Motivation

A. Related Work on Co-simulation Frameworks

Previous work has explored the design space of co-simulation frameworks that enable the modeling of a larger hardware design by integrating multiple discrete simulations of individual hardware models into a single coherent co-simulation. These frameworks mainly differ in terms of simulation backends they support, modifications required to accommodate existing IPs into the framework, and choice of communication medium between the simulations.

Muñoz-Quijada et al. [19] propose a framework that enables the co-simulation of a software model and an FPGA-accelerated RTL simulation through the use of UNIX-named pipes. While it allows co-simulation without major modification in the design, the framework only supports FPGA environments for the high-fidelity RTL simulation, whereas our approach allows for mix-and-match of different simulation backends. Similarly, CFC [18] enables coherent co-simulation of full SoC designs through inter-process communication, while supporting multiple simulation backends not limited to FPGAs. However, CFC relies on the Chisel HDL and ChiselTest testing environment, requiring the user to package every non-Chisel IP into a Chisel black box module and set up its own ChiselTest environment. In contrast, our framework is agnostic to any specific HDL environment, allowing the integration of external IPs without additional packaging efforts through the software workload or simulation runtime. This is showcased in our CPU-GPU co-simulation case study in Section VI.

Switchboard [5] is an open-source framework that similarly allows communication between distinct hardware models, which may be implemented in FPGA, RTL simulation or software, in order to simulate a bigger design. Its choice of simple shared-memory queues as the communication medium allows for fast inter-process communication and relatively low effort of integration of existing IPs. However, it only supports point-to-point connections between the models, making it hard to scale the model organization to multiple endpoints. Furthermore, Switchboard requires hardware changes to connect to a Verilog model, whereas our approach can optionally be implemented in the software stack only.

B. Co-simulation of CPU and GPU

To demonstrate how an efficient co-simulation framework can aid in the development and simulation of a large-scale hardware design, we include a case study of modeling a System-on-chip that integrates CPU and GPU cores in Section VI.

We integrate Vortex [22] as the target GPU design into the SoC. Vortex provides an open-source Verilog implementation of a GPGPU design, as well as a complete OpenCL software stack based on PoCL [13]. The project mainly focuses on FPGA as the hardware environment whereas implementation on an ASIC platform is left as future work. Vortex supports two simulation backends: SimX, a C++-based cycle-approximate architectural simulator, and cycle-accurate RTL simulation using software Verilog simulators such as VCS or Verilator.

For the CPU design, we leverage Rocket [4], a Chisel-based open-source in-order core generator. Rocket supports multiple simulation backends, ranging from the ISA-level functional model, Spike [7], to VCS or Verilator-backed RTL simulations, to FPGA-accelerated FireSim [15] simulations. Rocket's wide range of support for different backends makes it an ideal target for demonstrating our co-simulation framework's capabilities of mix-and-matching different backends across the design components.

Chipyard is an agile framework for designing and evaluating full-system hardware developed at Berkeley. It is composed of a collection of tools and libraries designed to provide an integration between open-source and commercial tools for the development of systems-on-chip [2]. A full SoC integration of the external Vortex GPU and the Rocket CPU in the Chipyard framework is very challenging. However, the use of a socket-based IPC for co-simulation of a Vortex GPU and a Rocket CPU greatly simplifies the integration, as major modifications of the Vortex GPU are not required.

Co-simulation based on inter-process communication also enables mixing functional and RTL simulations. Mixing functional and RTL simulations greatly reduces the time of a single design iteration of a particular hardware module in the system-on-chip. In the case of GPU and CPU integration, if we only want to iterate the design of GPU, running RTL simulation on GPU and functional simulation on CPU not only reduces the time cost of CPU simulation, but also accurately simulates the GPU design.

In Section VI, we attempt to model the SoC integration of a Rocket CPU and a Vortex GPU by running two simulation processes and using socket-based IPC to transfer data between the two hardware units. Simulations of CPU and GPU can be either RTL or functional. On the CPU side, RISC-V proxy kernel [6] loads the host binary that is statically linked with a PoCL runtime and a modified Vortex GPU driver. The GPU driver transfers data and starts GPU execution using the socket library interface, which makes socket syscalls to the proxy kernel. The proxy kernel captures the syscalls and forwards them to the Front-End SerVeR (*FESVR*) that does the actual inter-process communication on the host system. Once the data and commands are received from the process running CPU simulation through socket IPC, the process running GPU simulation starts the execution. It transfers the data back to the CPU process after execution completes.

C. Co-simulation of Heterogeneous Accelerators

Efficient co-simulation also becomes important in the context of multiple accelerators integrated into a single SoC design. A prominent application use case for such configuration is large language models (LLMs). With growing model sizes of LLMs [24], it has become desirable to parallelize the inference of a large model across multiple hardware accelerators, making modeling such a workload in simulation a compelling objective.

In Section VII, we attempt to model this use case by simulating ML workloads running on multiple instances of Berkeley's machine learning accelerator generator, Gemmini [11]. Gemmini is a full-stack, full-system Deep Neural Network (DNN) accelerator, written in Chisel [8] and is part of the Chipyard ecosystem [2]. Its main execution block consists of a systolic array, making matrix multiplication highly performant. It interacts with a CPU through the RoCC interface [4], comprised of a command interface for custom instructions from the CPU, as well as a Tilelink [10] memory system interface. A typical matrix multiplication lifecycle starts with a memory load into its internal scratchpad, followed by multiple preload and compute instructions, and finally a memory store writes compute results from the internal scratchpad to the external memory system. The source code of Gemmini, in Chisel, compiles down to Verilog files, which can then be simulated in RTL simulators. Furthermore, the Spike RISC-V ISA simulator [7] has been extended with a functional model of Gemmini.

At the present time, we are aware of two efforts to integrate multiple Gemmini's into one design. **MoCA** [16] is able to support multiple accelerators in one SoC, but is limited to having each accelerator run a different workload. **AuRORA** [17] uses dedicated manager and client nodes in the hardware design to facilitate a virtualized acquire/release system for the accelerators. We believe for workload simulation and performance modeling, our approach is much simpler than AuRORA, although AuRORA has the benefit of being synthesizable.

The specific workload we aim to run is a Transformer encoder. Transformers are a revolutionary sequence architecture in the field of deep learning due to their effectiveness in various tasks such as natural language processing (NLP) [23]. The encoder in a transformer is responsible for processing the input data into a higher, more abstract representation. It does this through a series of layers, each comprising two key components: a self-attention mechanism and a feed-forward neural network. In our evaluation, we focus on only one of such layers due to simulation time constraints.

III. System design

A. Goals

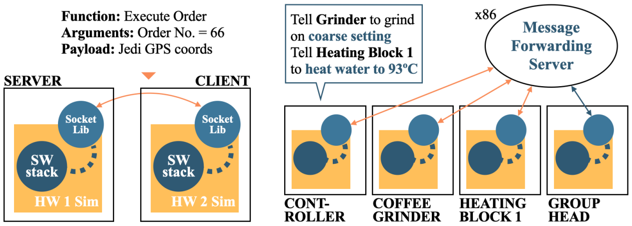
We will first outline some guiding objectives for our design.

1) *Lightweight*. The interface should be easy to adopt in existing systems with a minimal footprint, with an emphasis on portability and few dependencies.
2) *Intuitive*. The interface should require minimal prerequisite knowledge of other libraries and/or protocols, and should be immediately understandable.
3) *Flexible*. The interface should adapt to different communication scenarios, including design hierarchies, hardware interactions, and modeling requirements. This lends well to the heterogeneous hardware landscape.
4) *Performant*. The interface should be efficient enough to not be prohibitively slow to obtain useful performance insights. It should ensure scalability, i.e. simulation performance should scale well with design sizes.

B. Design

The communication scheme consists of two functions only: `send` and `receive`. `send` takes the destination of the request, the function name to call on the destination remote simulation, a set of agreed-upon arguments, and an optional data payload. `receive` takes the function name to receive, a buffer to store received data, and importantly whether the `receive` call is *blocking* or *non-blocking*. In a *blocking* `receive` call, the call doesn't return until a request of the supplied function name arrives at the caller's socket, and therefore each blocking call guarantees one request to process. In a *non-blocking* call, the call should return with a request if there is one readily available with the correct function name, but should not wait for one to become available. Since a socket connection is bidirectional, any component dialed into the socket communication channels may initiate a `send` or a `receive`.

Figure 1(a) showcases an example: the Emperor hardware block may transmit an "Execute Order 66" to the clone trooper, something the Emperor knows the clone trooper understands, with destination, function name, and arguments correctly set.



(a) Peer-to-peer (b) Server-client

Fig. 1. Communication schemes & examples

On the clone trooper's side, its software runtime could be periodically checking if there are any new "Execute Order"'s in a non-blocking way and act accordingly. [1]

We believe this send/receive interface is simple yet versatile enough to handle a lot of situations. For example, when a large amount of data is to be transferred unidirectionally, each individual transmission can be done asynchronously, which means the sender does not wait for any confirmation from the receiver (`send`s only). However, if data hazards are present, or transaction level synchronization is required, the sender may choose to call `receive` itself with blocking enabled after sending the request. The receiver, after the request has been received, may choose to `send` a response to indicate the request completion. In essence, a reverse direction request may be used in place as a response, enabling synchronous remote procedure calls.

To create a connection, hardware blocks can connect in either a peer-to-peer fashion as shown in Figure 1(a), where there is a notion of a *server* and a *client*, or if, many hardware blocks are to be simulated, connect as clients to a dedicated central server as shown in Figure 1(b). In both cases, the server should be started first for the clients to connect to. In a peer-to-peer setup, the destination in `send` calls is not relevant, but they carry more weight in a server-client setup, as is the case in the example.

IV. Implementation

We implemented the interface in the form of a C++ library, supporting IPC through either TCP or Unix domain sockets (UDS). A dedicated socket message forwarding server is also written in C++, which receives messages from one client and forwards them to the intended destinations.

To establish a connection, a client would call `init_client` with a TCP port or a UDS path, along with an intended *endpoint ID*, which is used to identify the destination of a message. In a peer-to-peer connection, this ID is ignored by the server, but in a central server setup, the dedicated server will assign the next available endpoint ID if the intended ID is not available.

A. Sending

In our implementation, function names are integers, therefore in a `send` call, both the destination (endpoint ID) and the function name (function ID) are specified as integers. Every request is divided into packets of 1024 bytes. The first packet of a request contains a header, which contains the size of the entire request in bytes, the source endpoint ID, the destination endpoint ID, and the function ID. `send` requires the arguments and payload to be stored in `std::vector<char>`'s; however it is less intuitive to manually marshal and unmarshal arguments to and from a vector, therefore we provide two template functions that do the marshaling automatically. The advantages of using a vector lie in safer memory management, as well as the implicit size argument supplied alongside the vector itself.

B. Receiving

Receiving in our implementation is two-phased. An internal `fetch` first downloads all complete outstanding messages, regardless of function ID, from the socket and stores it in a `std::dequeue`; then, the library looks through received messages to find the desired function name to process. As an added optimization, the search is first done once before fetching, and the first found message is popped and directly returned to avoid expensive socket accesses. The nature of this decoupled fetch and search procedure enables message receiving to be *coalesced*, beneficial when the sending side has a large amount of data or commands to push through. In a blocking `receive`, the fetch and search procedures alternate until one desired message is found; in a non-blocking `receive`, only one iteration is performed (search-fetch-search).

C. Message forwarding server

We intend the dedicated message forwarding server to run locally on the Linux (x86) host, where the simulation processes are in, for best performance. When a connection request is received from a client, a new socket is created for the connection, and a new thread is spawned for each client. Each thread listens to and reads from its client, storing socket writes to a local buffer; when the buffer is full, or no more messages are arriving, the buffer is flushed to the destination buffer by writing to the corresponding socket. A mutex lock is acquired to ensure no race conditions exist if a socket has multiple writers. No partial messages may be in the buffer when flushing, therefore messages arrive without corruption.

Errors are handled as gracefully as possible in the forwarding server, including disconnections. As a result, the server can persistently stay in the background and cater to requests as clients come and go. In fact, during the evaluation tests for the second case study, the server stayed on for the entire duration.

V. Microbenchmarks

For the microbenchmarks, the RTL simulations uses the Rocket chip [4] simulated using Synopsys VCS, and the functional counterpart uses Spike [7]. The software stack for RISC-V CPU cores mainly consists of a Front-End SerVeR (*FESVR*), and an optional *proxy kernel* [6]. FESVR can

[1] For context, in Star Wars, execute order 66 is what Emperor Palpatine (the main antagonist) issued to the clone troopers (imperial army) to turn against and eliminate the Jedis (the good guys).

be considered as a simulation runtime, whereby it manages simulation lifecycle events like binary loading and termination, as well as provides utilities such as file IO syscall handling (on the Linux host) and printing. FESVR code runs mostly in the Linux host system. The proxy kernel is an optional lightweight kernel that provides virtual memory, user mode execution, and basic syscalls to a single application binary, which runs single-threaded. In particular, socket syscalls are delegated through FESVR to the Linux host system syscall.

In our testing, we evaluate three test cases with the proxy kernel: peer-to-peer UDS, peer-to-peer TCP, and server-client UDS. In every case, we vary the message sizes and message counts to understand performance under different communication patterns. In the blocking test cases, we record the time taken to send data of a certain size back and forth, fully receiving the previous message before sending the next one. In the non-blocking test cases, we record the time for one side to completely send all test messages, wait for the opposite side to fully receive, and repeat for the other direction.

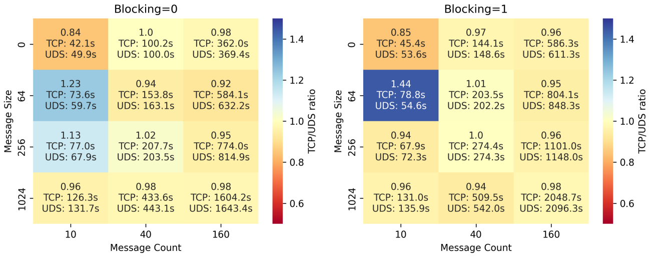


Fig. 2. RTL simulation, TCP vs. UDS (lower=TCP better)

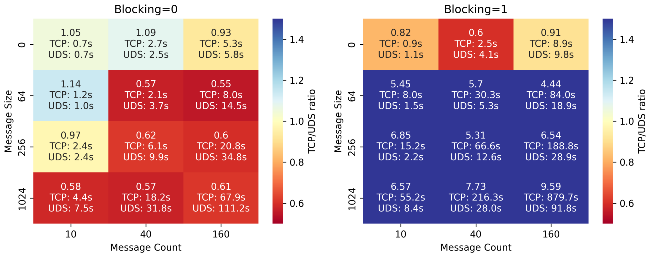


Fig. 3. Functional simulation, TCP vs. UDS (lower=TCP better)

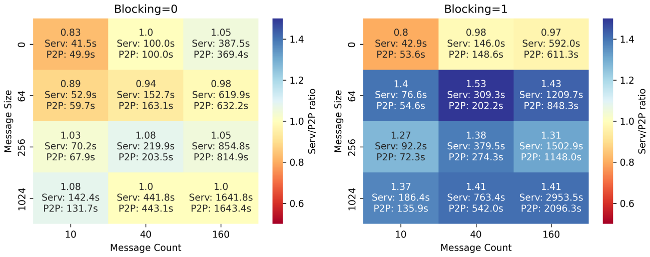


Fig. 4. RTL simulation, P2P vs. Server-client (lower=Server better)

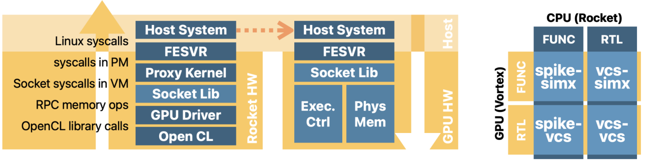
Figures 2, 3, and 4 show our microbenchmark results. Looking at the raw values, the time cost scales directly with more messages and larger message sizes. In particular, when the message size is small, time scales sublinearly with message count, but the relationship approaches linear as data size increases, where payload transmission time appears to dominate and the overhead is amortized. This trend seems to be present for each configuration. Furthermore, it seems like due to the per-message fixed cost, increasing message size does not produce a proportional time penalty, which incentivizes fewer larger messages compared to more smaller messages. For a total payload size of 10240B, transferring 160 messages takes 4.4× the time compared to 10 messages for non-blocking; the number is 6.1× for blocking.

In figure 2, we compare TCP versus UDS as the IPC socket channel. It seems that TCP is slightly more efficient in RTL simulations by about 2%-6%. For functional simulations shown in 3, non-blocking messaging lends well to TCP, but UDS is a lot more efficient if the major pattern of communication is synchronous. Finally, we look at the overhead of using a server-client setup. To our surprise, the server-based communication scheme was slightly more efficient in smaller test cases. In general, for non-blocking test cases, the server had up to 8% overhead, but for blocking test cases the server added about a 30%-50% overhead. The difference may be explained by the request coalescing capabilities built into the server with the buffer fill-then-flush paradigm.

VI. Case Study: CPU-GPU

A. *Overview*

This case study investigates the co-simulation of a Rocket CPU and a Vortex GPU in the Chipyard framework using our socket-based hardware communication library. It shows the integration of the CPU and GPU does not require extensive modifications to the design of the two hardware modules. Co-simulation of the CPU and GPU is also more scalable than the simulation of the monolithic SoC, as only the corresponding module is simulated in each individual process. Finally, modular simulation allows us to iterate the design of a particular module with cycle-accurate simulation, while running a functional simulation for another module.



(a) Software stack (b) Co-simulation

Fig. 5. System design of CPU-GPU co-simulation

B. *System Design & Implementation*

Figure 5 shows that the co-simulation of CPU and GPU uses the peer-to-peer communication setup with two simulation processes. GPU simulation process acts as a server listening for the connection from the CPU simulation process. The CPU simulation runs a 64-bit binary linked with the PoCL runtime, a modified Vortex driver communicating with the GPU process, and the socket library itself. The socket library makes socket syscalls to the proxy kernel, and the syscalls are then translated to host system syscalls by FESVR.

Each simulation runs a single Vortex GPU core or a single Rocket CPU core. The simulations of CPU and GPU can be either RTL or functional. Spike, a RISC-V ISA simulator, and Simx, a simulator developed by the Vortex team, are used for functional simulations of CPU and GPU respectively.

The workload used for the co-simulation of CPU and GPU is softmax. The softmax kernel code is written in OpenCL C Language, which can be compiled to a RISC-V binary by the PoCL runtime. Softmax workload works well for GPU simulation because of its nonlinear operation of exponentiation, which is complex enough to fully explore the general purpose cores of Vortex GPU. For the evaluation, we ran simulations with variable input vector length for the softmax kernel to test the scalability of our socket-based co-simulation setup.

1) CPU stack: The size of the host binary is about 7 MB. All libraries are linked to the host code statically, including the PoCL runtime, GPU driver, and the socket library. However, the PoCL runtime linked to the host code does not contain the LLVM library and cannot compile the OpenCL C code to RISC-V binary on the fly. Instead, an offline PoCL compiler is used to compile the kernel code to PoCL binary; the program during execution only reads this binary and translates the binary to a format that can be executed on Vortex GPU.

In simulation, the Rocket core first loads the proxy kernel, which then loads and executes the host binary. Proxy kernel is necessary because PoCL runtime makes filesystem syscalls for caching PoCL binaries and the socket library makes socket syscalls for inter-process communication with the GPU simulation server. The proxy kernel receives the syscalls and forwards them to FESVR, which delegates the syscalls to the Linux host system.

The Vortex GPU driver used by the PoCL runtime was modified to communicate with the GPU using the socket library. Specifically, when the GPU device is initialized in the host program, the driver as a client connects to the GPU simulation server. For `clEnqueueWriteBuffer` and `clEnqueueReadBuffer` operations specified in OpenCL, the corresponding `upload` and `download` operations transfer the data through socket inter-process communication. The GPU driver can also `start` the GPU execution and `wait` for the execution to finish using the socket library.

2) GPU stack: GPU simulation is launched first as a server. To enable socket-based co-simulation, the Host-Target Interface (HTIF) in FESVR is modified and linked with the socket library. After the host binary running in the CPU simulation initializes the GPU driver and connects to the GPU simulation server, the GPU simulation starts to listen and process the incoming requests with the function calls `receive` and `send`.

There are also 4 types of requests defined in HTIF that match to message types in the GPU driver. They are `write`, `read`, `run`, and `wait`. For every tick of the simulation, HTIF calls socket `receive` for all types of requests in a non-blocking way. The corresponding operations are done if HTIF receives the requests. The communication scheme between the CPU simulation and the GPU simulation is defined as follows:

Memory addresses	Usages
`0x7c000000`	Execution finished MMIO register
`0x7fff0000`	Kernel launch parameters
`0x80000000`	Kernel binary
`0xc0100000`	Heap region for operand data
`0xffffffff`	Stack region (grows downwards)

TABLE I
PHYSICAL MEMORY MAP OF VORTEX GPU

1) **Upload parameters and operands**: The arguments and operands required by the kernel are uploaded to the GPU simulation server through `write` requests. HTIF then writes the data to the appropriate regions in Table I.
2) **Upload kernel binary**: The kernel binary is uploaded similarly, which is then written to the memory address `0x80000000`.
3) **Start execution**: When HTIF receives the `run` request, it `resets` the GPU core to start kernel execution.
4) **Wait for execution to finish**: After receiving a `wait` request from the CPU process, FESVR continuously checks for execution completion through an MMIO register. It sends back an acknowledgment after the GPU finishes.
5) **Download data**: When receiving a `read` request, HTIF reads the destination buffer in the heap and sends it to back to the CPU process.

C. Evaluation

1) Baseline: Because of the difficulty of a full SoC integration of a Rocket CPU and a Vortex GPU, we emulated this setup by running and adding the results of two independent simulations. One simulation contains a Rocket core running the host binary and a dummy Vortex core that does nothing. Another simulation contains a Vortex core running the kernel binary and a dummy Rocket core that does nothing. We ensure that the design sizes of the two independent simulations are as large as the size of a full SoC monolithic simulation of CPU and GPU. Adding the two simulations together would approximate the monolithic simulation as our baseline.

2) Experiment setup: In the experiment, we test two cases of co-simulation with the socket library. The first case runs RTL simulations for both CPU and GPU. The second case runs a CPU functional simulation and a GPU RTL simulation. The length of the input vector to the softmax kernel is used as a variable to validate the scalability and cycle accuracy of the co-simulations.

We created a host binary that repetitively dispatches the same softmax kernel to the GPU with different input vector lengths, ranging from 32 to 4096. We then recorded the real time and the cycle counts of code execution for both CPU and GPU RTL simulations. For the cycle count of CPU simulation, we subtracted the number of cycles spent inside the socket library. The real time and the cycle counts from co-simulation are then compared against those of the monolithic simulation.

3) RTL-RTL co-simulation: This case shows that RTL-RTL co-simulation using the socket library is more scalable than a full SoC monolithic simulation. At the same time, the cycle counts obtained from co-simulation are consistent with those from the monolithic simulation. As co-simulation does not require extensive modifications to the RTL designs of the hardware modules, this approach is more favorable for simulating a diverse set of IPs with fast design iterations.

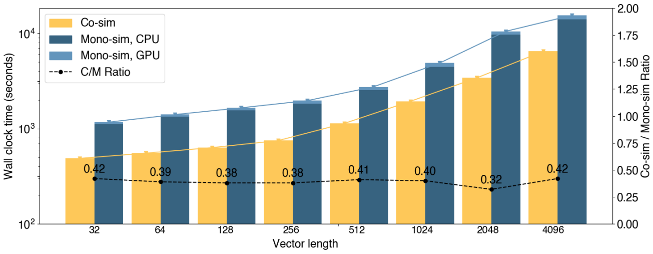


Fig. 6. Real time comparison: RTL-RTL co-simulation

a) Real time: Figure 6 shows the comparison of simulation time between the monolithic simulation and RTL-RTL co-simulation. There is a constant time overhead of initializing the PoCL runtime and the GPU driver, but the relationship between input vector length to the softmax kernel and simulation time is linear. The most important observation in the figure is that the simulation speeds of the socket-based co-simulations are 2.38× to 3.13× that of the monolithic simulations. This suggests that the co-simulation of individual cores with smaller design sizes is much more scalable than a full SoC simulation with a large design size.

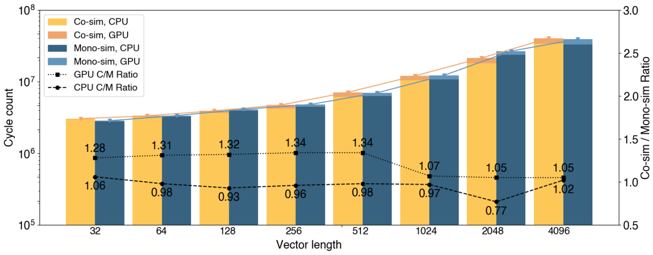


Fig. 7. Cycle count comparison: RTL-RTL co-simulation

b) Cycle time: Figure 7 shows that socket-based co-simulation also approximates the cycle count of the monolithic simulation. The cycle count of GPU in the co-simulation case is about 1.3x larger than in the monolithic simulation case when the vector length is small, but the overestimate reduces to 5% as the vector length increases, amortizing the overhead. For the CPU cycle count, the inaccuracy is largely within 5% with occasional outliers.

4) Functional-RTL co-simulation: This test case shows that the mix-and-match capabilities of the socket library enable useful tradeoffs between fidelity and the speed of simulation. Functional CPU and RTL GPU co-simulation allows fast design iterations on the GPU core without spending a large portion of time RTL simulating the CPU core.

a) Real time: Figure 8 shows the comparison of simulation time between the monolithic simulation and the Functional-RTL co-simulation of CPU and GPU. Because the CPU process is running a functional simulation using Spike, it greatly reduces the time needed for simulating the SoC. Co-simulation is about 30x faster when the input vector length is small. As the vector length increases, this factor goes down and stabilizes at around 10x. This large speedup in simulation time is expected as we are not simulating the Rocket CPU design at all. Functional-RTL co-sim demonstrates one use case of the socket library that when we only want to iterate the RTL design of a single hardware module in the SoC, we can replace other hardware modules with functional simulators, which significantly speeds up the design iteration time.

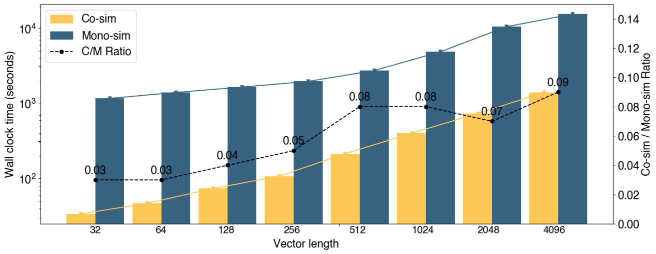


Fig. 8. Real time comparison: Funct-RTL co-simulation

Component	Lines of Code
PoCL runtime modification	110
GPU driver modification	43
Proxy kernel modification	191
FESVR modification	342
Softmax kernel code	22
Softmax host side library	263

TABLE II
LINES OF CODE FOR THE CPU-GPU CASE STUDY

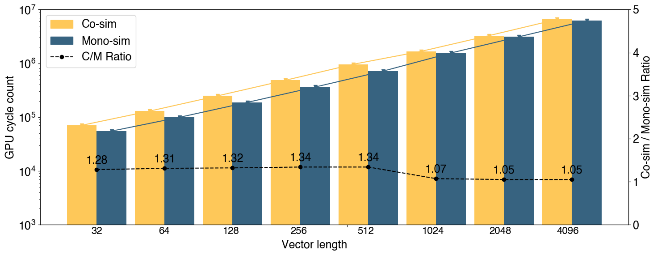


Fig. 9. Cycle count comparison: Funct-RTL co-simulation

b) Cycle time: Figure 9 shows the comparison of GPU cycle counts between the monolithic simulation and functional CPU & RTL GPU co-simulation. CPU and total cycle counts are excluded because the functional simulator Spike does not simulate the RTL design of the Rocket core and is not cycle-accurate. The ratio trendline shows an amortization pattern similar to the functional-RTL co-simulation, with inaccuracy decreasing to within 5% as vector length increases. This shows the effectiveness of cycle modeling, even when one side of the simulation is functional.

5) Lines of code: Lines of code required to conduct this case study is shown in Table II.

VII. Case Study: Many accelerators

A. Overview

This case study investigates the practical application and efficacy of our socket-based hardware communication library in a scenario involving multiple instances of an ML accelerator. The primary focus is on executing a transformer encoder layer workload, which is representative of a realistic use case for such accelerators. In particular, we attempt to offload the workload in a parallel way to simulate a potential accelerator load balancing use case for larger transformer models. Compared to the CPU-GPU integration, this case study focuses on finer granularity communication, enabling parallelization, and ensuring scalability in a simulation environment.

B. System Design & Implementation

In this case study, we use the server-client communication setup, as described previously in Section II. The simulation consists of a *dispatcher* running in the Linux host (x86), and many independent VCS simulation processes each simulating a Gemmini instance, or *worker*. The x86 dispatcher is responsible for offloading matrix multiplication in the self-attention and feed-forward networks, and each worker, running at its own pace, executes the matrix multiplication received through the socket communication layer. Both sides of the task are connected as clients to the central message forwarding server.

Due to our focus on the accelerators only, we have removed the unsupported non-linear operations, namely Softmax and LayerNorm, from the computation. If we include their CPU implementations, we are able to verify our computation results in all simulations with the results obtained from the Pytorch `TransformerEncoderLayer` implementation.

1) x86 dispatcher: The dispatcher is responsible for dividing up the end-to-end transformer workload into evenly-sized chunks for each accelerator. We use a naive splitting scheme, which divides the resultant matrix (the C matrix) in a matrix multiplication along the longer axis. The rationale behind this scheme is to simplify the reassembly process of the whole output matrix, as each worker does not write to overlapping memory regions in the DRAM.

The dispatcher is compiled with the Gemmini library routines; however, instead of initiating RoCC instructions, which is undefined on x86, the instructions are translated into socket procedure calls on remote workers. To summarize the possible procedure calls initiated by the dispatcher:

1) `mvin`, intended for Gemmini to load in operands from DRAM into its scratchpad. For the dispatcher, the relevant operands in DRAM are copied into the socket send payload buffer along with `mvin` parameters as arguments. Together, they are sent over the IPC channel to the worker processes, where the actual RoCC instruction for `mvin` is issued. This call can be non-blocking, since the dispatcher does not require a response before issuing the next command.
2) `mvout`, intended for Gemmini to store matrix multiplication results from its scratchpad to DRAM. The dispatcher sends a request to a worker for it to `mvout` into its local memory, which is then retrieved over IPC. The received data is stored into the dispatcher's local memory for reassembly, to be used in the next operation. This call is blocking, since the resultant matrix must be received and written to DRAM to avoid data hazards.
3) `fence`, intended for Gemmini to wait for memory operations to finish. The dispatcher requests all workers to `fence` locally, and blocks to wait for all responses to arrive back before proceeding.
4) `rdcycle`, intended for Gemmini to read the hardware cycle number for performance statistics. The dispatcher relays the cycle number from the worker.
5) Other RoCC instructions. The entire instruction is sent over IPC as-is. This category includes `config` commands and execution commands like `preload` and `compute`. They do not lead to data hazards, and hence to optimize for performance, these calls are all non-blocking.

A key observation is that the communication happens on an instruction level, which calls for frequent transactions on the IPC channel. Hence, we believe it is a good example of a finer granularity integration.

The dispatcher is also able to perform the inference under *serial* or *parallel* execution. Serial execution indicates workers receive workloads one by one, with one worker active at a time. This serves as a baseline for parallel execution, where for one matrix multiplication, all workers receive their chunks at the same time and are able to process in parallel. By starting a separate thread for each worker, this scheme simulates a multi-tenant SoC environment using thread-parallelism in the x86 host. The threads are joined by the end of one matrix multiplication, after which the process starts again.

2) Gemmini workers: Each worker, as previously stated, is its own simulation process, thereby having independent architectural and micro-architectural states. The worker hardware simulated consists of a Rocket core and a Gemmini attached to it. The Rocket core runs a custom binary that communicates with the socket library, receiving requests to process on the local Gemmini instance and sending results through the IPC channel as needed. Importantly, each worker does not operand and result matrices in DRAM persistently; instead, the dispatcher is the one true source of "DRAM", as if they were integrated into one memory system. As an optimization, before transmission, strided data is packed contiguously to reduce communication overhead.

3) Bare-metal socket library: The proxied syscall overhead generated by the high frequency of commands during computation led to us using a bare-metal version of the socket library, instead of the proxy kernel. The bare-metal library uses MMIO to talk with FESVR RISC-V simulation runtime, which acts as a bridge to the Linux host system. Specifically, Gemmini writes `send` and `recv` calls, including its arguments and payloads, into a predetermined physical memory location. These memory regions are monitored by FESVR, which delegates the `send` and `recv` calls to the Linux host system. This has the added benefit of enabling the binary to run in a physical address space in a bare-metal simulation, as opposed to a virtual memory space using the proxy kernel, avoiding unnecessary address translation overhead.

C. Evaluation

1) Baseline: Due to the difficulty of integrating multiple Gemmini's to parallelize an ML workload, we have emulated a baseline for comparison. The baseline hardware is a single design with one Rocket and one Gemmini minimum. For test cases with more than one worker, we modify the generated SoC Verilog code to include more instances of Gemmini; however, to ensure forward simulation progress, the outputs of the extra "dummy" Gemminis are cut off, meaning they do not cause external microarchitectural and architectural state changes. To ensure they are not optimized away by the simulator, each dummy instance receives the same instructions as the real Gemmini, but with different memory inputs.

In the *serial* execution case, the single working Gemmini runs each divided chunk of matrix multiplication in sequence. In the *parallel* execution case, only one worker's worth of workload is being run on this instance, as if it is part of multiple working workers. This is with the expectation that the other instances would have finished in a similar timeframe. Due to the lack of synchronization and parallelization overheads, the baseline is a slight underestimate.

Model Sizes	Small	Compact	Medium	Large	Bert (func only)
No. fp32 parameters	28,032	111,456	444,096	1,772,928	7,084,800
Hidden dimension	48	96	192	384	768
Sequence length	32	48	64	128	512
Expansion dimension	192	384	768	1536	3072
Number of heads	2	4	4	8	12
Runtime memory (MiB)	0.183	0.742	2.725	13.138	69.026

TABLE III
ENCODER LAYER PARAMETERS FOR DIFFERENT SIZES TESTED

2) Experiment setup: In our experiments, we test a combination of different variables:

- Functional or RTL simulation;
- Size of the transformer encoder layer, with possible configurations shown in Table III;
- The number of workers, which can be 1, 2, 4, 8 or 12. The 12-worker case is reserved for functional simulations only, due to time constraints;
- Serial or parallel execution.

In terms of metrics collected, we recorded the real time, which is the wall clock time it takes to run the encoder layer computation workload simulation from start to finish, as well as the cycle time, which is the number of cycles required to execute the computation. We subtracted the number of cycles spent inside the socket library by timing entrances to and exits from library function calls.

3) Results and analysis: Figure 10 shows the real time comparisons of serial execution using our socket-based IPC integration (socket) versus the baseline (native). The reason for a non-parallel test case is to demonstrate the raw overhead added by using an intermediate layer of IPC. Using sockets, the simulation time is around 2× to 4.5× that of the native version. The smallest test case clocks in at around 28 minutes for native, and the largest test case reaches just over 122.5 hours with sockets. The overhead likely stems from latencies in MMIO, socket communication, and fencing. The results show further opportunities for optimization; however, such an overhead is a reasonable tradeoff for a working integrated simulation that produces the correct results.

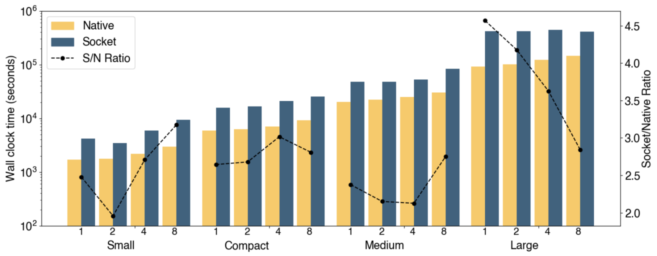


Fig. 10. RTL simulation results running serial execution

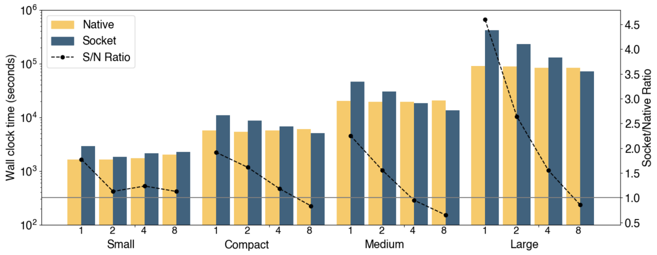


Fig. 11. RTL simulation results running parallel execution

Figure 11 shows the real time comparisons of parallel executed test cases. With only 1 worker, the test cases degrade to the 1 worker serial test cases, albeit with added threading overhead. As the number of workers increases, the socket-linked integration scheme approaches the simulation time of the monolithic baseline, and in some cases overtakes; the downward trend is present in all model sizes we tested, showcasing the scalability of the library. For the compact model size, the 8-worker socket version required 83.7% of the simulation time compared to the native counterpart. For the large model size, the 8-worker socket to native time ratio was 85.8%. Finally, for the medium model size, the 4-worker number was 95.3%, and for 8-worker it was 65.4%, indicating a 34.6% speedup.

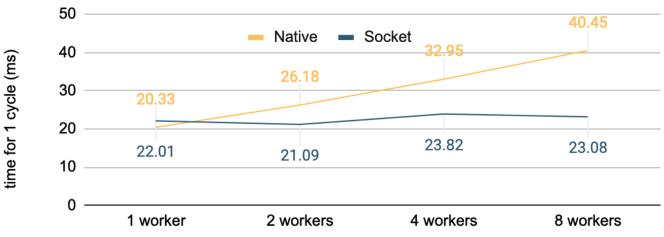


Fig. 12. Simulation time per cycle

We can more visibly see the root of the efficiency in Figure 12. As evident in the graph, the unit cycle simulation time for the native integration simulation time scaled up as the design size increased with more workers, compared to a much more constant scaling with the independent simulation processes linked together with socket IPC. At 8 workers, the average native simulation speed equates to 24.7 KHz, whereas our simulation speed is 43.3 KHz.

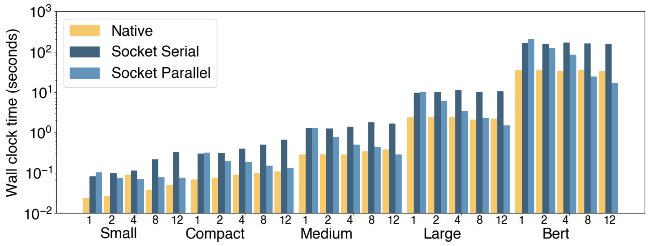


Fig. 13. Functional simulation results

Figure 13 shows the real time comparisons of both serial and parallel test cases when simulated in a functional simulator. We observe similar trends to the RTL simulations. For serial test cases, the simulation time is roughly around 4.5× that of native, ranging from 3.4× up to one case at 6.5×. We suspect the larger ratios are due to the socket communication libraries taking a larger portion due to the faster computation speeds a functional simulator is able to sustain. There is no baseline per se for parallel test cases, since it is not possible to simulate more workers in a single functional environment. However, we can still observe the downward trend of real time required when the number of workers increases in a socket-based integration.

We wish to point out that the task of parallel accelerator utilization inside a monolithic multi-tenant SoC, itself an active area of research, has generally been a very challenging objective in the first place as previously discussed. Our work obtained the simulation time gain based on a much simpler yet more performant alternative.

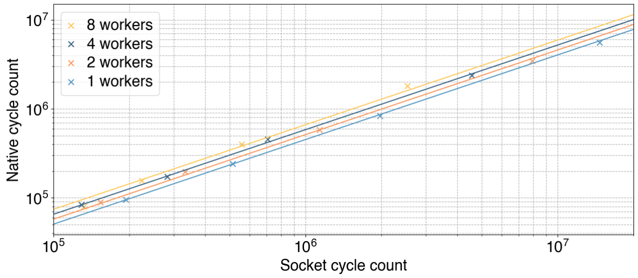


Fig. 14. Linear regression predicting true cycle numbers

For effective performance modeling, we recorded the cycle figures from each socket IPC test case; in each case, even with the cycles spent in the library deducted from the total, the cycle numbers ended up being larger than that of a native implementation. However, the strong correlation we observed between the two sets of figures shows that one still may be used to predict the other. Using the 16 data points we obtained from the parallel test cases, we fitted a simple linear regression model. The independent variables are $\log(\text{number of workers})$ and $\log(\text{socket cycles})$, and the predicted dependent variable is $\log(\text{native cycles})$. Shown in Figure 14, our model predicts the true cycle numbers (non-log) with an accuracy of 94.9%, indicating that the IPC abstraction is a powerful tool for performance estimation during design iteration.

Component	Lines of Code
Native transformer encoder inference	599
Socket-enabled encoder inference (dispatcher)	759 (+160)
Gemmini worker binary	138
Gemmini worker FESVR MMIO interface	154
Bare-metal socket library	312

TABLE IV
LINES OF CODE FOR THE MANY-ACCELERATORS CASE STUDY

Finally, we show the lines of code required to implement each of the components in this case study in Table IV.

VIII. Future Work

We realize that the current socket IPC performance, especially in the finer communication granularity cases, still requires more optimization. In addition, it is evident that some additional functionality would greatly enhance the usefulness of our work as a performance modeling and design iteration tool, and therefore we have compiled a few future directions.

1) Shared memory based IPC. A potential shared memory based IPC implementation of the socket library could greatly outperform the current Unix domain file and TCP based implementations, decreasing the design size threshold to break-even on simulation time.
2) Quantum-based synchronization control. To allow for finer cycle-level synchronization control, instead of a transaction-level synchronization control like our current design, an adjustable simulation quantum could be incorporated into the protocol. This may enhance the cycle count accuracy approximated from the simulation, and provides a tunable knob to trade accuracy with simulation performance.
3) Memory latency and bandwidth modeling. At the current stage, the characteristics of the hardware-to-hardware communication depends almost solely on that of the underlying IPC channel. This may not be sufficient for integrating for example a large memory system, or modeling specifically attaching a core to a particular level of cache. Adding latency and bandwidth constraints between two endpoints may allow for more usage scenarios.

IX. Conclusion

In this paper, we showed a simple yet capable socket-based hardware communication framework. Through a CPU-GPU co-simulation and a many-accelerators integration case study, we demonstrate our design supports simulating heterogeneous architectures in a scalable and performant way. We show significant simulation time reduction while retaining close cycle number approximation, enabling accurate performance modeling and fast design iteration.

References

[1] AMD. Amd epyc™ 9004 series processors, 2023.

[2] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. Chipyard: Integrated design, simulation, and implementation framework for custom socs. *IEEE Micro*, 40(4):10–21, 2020.

[3] Apple. Apple unveils m3, m3 pro, and m3 max, the most advanced chips for a personal computer, 2023.

[4] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 4:6–2, 2016.

[5] Zero ASIC. Switchboard: An open source high-performance communication platform. 2023.

[6] RISC-V International Association. Risc-v proxy kernel and boot loader, 2023.

[7] RISC-V International Association. Spike risc-v isa simulator, 2023.

[8] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221, 2012.

[9] Inc Cerebras Systems. Wafer-scale engine: The largest chip ever built, 2021.

[10] Henry Cook. *Productive Design of Extensible On-Chip Memory Hierarchies*. PhD thesis, EECS Department, University of California, Berkeley, May 2016.

[11] Hasan Genc, Ameer Haj-Ali, Vighnesh Iyer, Alon Amid, Howard Mao, John Charles Wright, Colin Schmidt, Jerry Zhao, Albert J. Ou, Max Banister, Yakun Sophia Shao, Borivoje Nikolic, Ion Stoica, and Krste Asanovic. Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures. *CoRR*, abs/1911.09925, 2019.

[12] Ventana Micro Systems Inc. Ventana introduces veyron v2 — world's highest performance data center-class risc-v processor and platform. 2023.

[13] Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg. pocl: A performance-portable opencl implementation. *International Journal of Parallel Programming*, 43:752–785, 2015.

[14] Shriyanshi Kapoor, Kota Naga Srinivasarao Batta, and Jatin Nagpal. Emulation: Accelerating simulation for rapid verification of modern processor-based subsystems. In *2023 3rd International Conference on Intelligent Technologies (CONIT)*, pages 1–8, 2023.

[15] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 29–42, Piscataway, NJ, USA, 2018. IEEE Press.

[16] Seah Kim, Hasan Genc, Vadim Vadimovich Nikiforov, Krste Asanović, Borivoje Nikolić, and Yakun Sophia Shao. Moca: Memory-centric, adaptive execution for multi-tenant deep neural networks. *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023.

[17] Seah Kim, Jerry Zhao, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. Aurora: Virtualized accelerator orchestration for multi-tenant workloads. *56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023.

[18] Ryan Lund. Design and application of a co-simulation framework for chisel. 2021.

[19] Maria Muñoz-Quijada, Luis Sanz, and Hipolito Guzman-Miranda. Sw-vhdl co-verification environment using open source tools. *Electronics*, 9(12):2104, 2020.

[20] SambaNova Systems. Sambanova announces next generation datascale system, setting a world record for time-to-train performance. 2022.

[21] Tenstorrent. Cards, 2023.

[22] Blaise Tine, Krishna Praveen Yalamarthy, Fares Elsabbagh, and Kim Hyesoon. Vortex: Extending the risc-v isa for gpgpu and 3d-graphics. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 754–766, 2021.

[23] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[24] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models, 2023.