SonicSim: Socket-based Hardware Co-Simulation With Inter-process Communication

Richard Yan

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-61
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-61.html

May 7, 2024

SonicSim: Socket-based Hardware Co-Simulation With Inter-process Communication

by

Ruohan Richard Yan

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electric Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Yakun Sophia Shao, Chair
Professor Christopher W. Fletcher

Spring 2024

The thesis of Ruohan Richard Yan, titled SonicSim: Socket-based Hardware Co-Simulation With Inter-process Communication, is approved:

Chair  Date 5/6/2024

Chris Fletcher Date 5/3/2024

University of California, Berkeley

SonicSim: Socket-based Hardware Co-Simulation With Inter-process Communication

Abstract

SonicSim: Socket-based Hardware Co-Simulation With Inter-process Communication

by

Ruohan Richard Yan

Master of Science in Electric Engineering and Computer Sciences

University of California, Berkeley

Professor Yakun Sophia Shao, Chair

Modern computer architecture is increasingly large and heterogeneous, and coherent co-simulation of the design is an effective tool to quickly iterate the design of individual sub-components and lower the cost of design and evaluation. However, scalably applying co-simulation methods to a variety of hardware designs is challenging, especially because different hardware blocks often necessitate the use of their own tools and simulation frameworks, requiring a lot of manual work to integrate them into a single coherent simulation.

To address these shortcomings, we propose *SonicSim*, a novel co-simulation framework with a specific focus on scalability and low effort of integration. SonicSim defines a simple inter-process communication protocol across hardware blocks, and supports point-to-point connections and central server-client architecture to better scale to more simulation endpoints. When leveraging a lightweight MMIO-based host-target interface, our framework lifts the integration to the software workload level, requiring minimal modifications in the target hardware design. From the case studies of CPU-GPU co-simulation and many-accelerator co-simulation, we demonstrate that SonicSim enables mix-and-matching of different simulation backends across the design components and achieves significant simulation speedup with accurate cycle time prediction, exposing useful tradeoffs between simulation speed and fidelity to the designer. Finally, we quantify that applying our framework requires minimal lines-of-code changes to the target hardware and software workload.

Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I want to thank Professor Yakun Sophia Shao for her ungoing support of my endeavors in this space, as well as extensive inputs regarding the writing. Professor John Kubiatowicz provided valuable feedback during the course of this project, for which I am grateful. I'd also like to thank Apple for graciously funding my studies for the Masters degree.

This project cannot be completed without the help of my 262A project partner Zekai Lin, who was largely behind the effort of the CPU-GPU integration section. My good friend Hansung Kim was also instrumental in terms of the project conception, the writing, and the preparation for the ISPASS submission.

Finally, I'd like to thank my parents for sending me abroad to study, and cheering me up when I was at my lowest. Without you, I wouldn't have been able to make it this far in life. Thank you for for being the immovable support I can always fall back on with my eyes closed.

Chapter 1

Introduction

The contemporary computer architecture landscape has entered a renewed phase of rapid development, with growing scale and heterogeneity for the newer System-on-Chips (SoCs) in the spotlight. 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 [16]. In the meantime, driven by the diverse requirements of modern computing, such as artificial intelligence (AI), chips are featuring a portfolio accelerators that are diverse and unconventional. SambaNova Systems has designed a novel dataflow architecture to more efficiently inference very large models [28], while 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 [11] show scaling and heterogeneity in one package.

As such, there is a growing need to holistically simulate different novel architectures from various vendors, and in a scalable way. It is insufficient to individually evaluate each component, as the inter-component dynamics cannot be captured this way, and workloads like ML inference may require the entire SoC to correctly execute. Traditionally, this calls for 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 [18]. 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 these challenges by either electing to re-model all components on a microarchitectural level simulation such as gem5 [10], or co-simulate the individual components of a larger design in discrete simulation instances, with a framework

implementing block-specific 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 [24], necessitate the use of a specific implementation environment for the target design [23], 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 report, 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, *SonicSim*, that enables scalable, coherent integration of discrete hardware simulations into a larger design, without requiring major modification in the design components.
- We show our co-simulation framework supports the mix-and-match of different simulation backends, enabling useful tradeoffs between simulation speed and fidelity.
- We show that SonicSim encapsulates higher level integration abstractions, making it possible to avoid hardware changes and instead communicate at the host-target interface level and software workload level.
- We demonstrate in two separate case studies that SonicSim can be easily integrated, can achieve speedup over corresponding monolithic simulations, and make segregated design iteration possible while maintaining correctness.

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

Chapter 2

Background & Motivation

Large, heterogeneous computer architectures benefit from co-simulation as an effective simulation method that allows fine-grained control over the tradeoffs between simulation speed and fidelity. However, designing a scalable and easy-to-use co-simulation infrastructure is challenging. In this section, we introduce the importance and challenges of co-simulation in modern hardware design process, discuss prior work in co-simulation frameworks, and motivate the need for a more scalable and easy-to-use framework design.

2.1 Importance of Co-simulation

As technology scaling comes to an end, modern computer architectures have opted to integrating multiple heterogeneous compute units into a single design to accommodate a diverse set of applications without sacrificing performance [3, 14, 15, 19]. However, simulation of such a hardware design becomes increasingly challenging as more heterogeneous components are integrated. As the design size grows, a *monolithic* simulation of the entire design becomes intractable in terms of both speed and implementation [18].

An effective solution to this problem is *co-simulation*. Co-simulation enables scalable modeling of a larger hardware design by integrating the discrete simulations of individual design components into a single coherent simulation. A major advantage of co-simulation is *segregated* iteration of the individual design components. For example, a designer of a specific hardware IP might wish to quickly iterate the design process by accurately evaluating the behavior of the IP at a high detail, but modeling other IPs at a lesser fidelity in order to speed up the overall simulation. This is also true for integrating external IPs, where the designer would be more concerned with interfacing the IP with the rest of the design correctly rather than simulating its behavior at higher fidelity. In such cases, a co-simulation infrastructure can establish a clear barrier of design and evaluation by modeling the interfaces between the individual design components and thereby decoupling their simulations from each other.

2.2 Challenges of Co-simulation

However, designing an effective co-simulation infrastructure is challenging. An effective co-simulation framework has to be *scalable* in three ways: (1) to multiple different simulation backends, (2) to different target IPs without requiring major modification in their design, and (3) to a large number of simulation end-points.

Scaling to diverse simulation backends is essential for providing fine-grained control over the simulation speed and fidelity tradeoffs. For example, Vortex GPGPU [29] supports two simulation backends: a cycle-approximate microarchitectural simulation similar to GPGPU-sim [9], and Verilator-based RTL simulation [27], where the latter simulates the design at a much higher granularity but at a drastically reduced speed. An SoC designer that integrates Vortex cores would therefore want to support both simulation backends in order to flexibly control the simulation speed when accurate modeling of the GPU component is not necessary. However, since the two simulators have different toolchains without a commonly defined API, interfacing them into a co-simulation requires a lot of manual work.

Similarly, scaling to different target IPs is equally important as supporting different simulation backends. The Vortex GPGPU core is written in Verilog, whereas the Rocket CPU core [4] is written in the Chisel HDL [8]. The co-simulation framework should not be tied to a specific HDL or implementation environment to be applicable to a wide range of target IPs. Moreover, scaling to a high number of design components is difficult if the framework does not provide an adequate communication medium, and forces the designer to establish ad-hoc communication channels between IPs manually.

2.3 Limitations of Prior Efforts

Prior work has explored the design space of co-simulation frameworks with differing sets of the simulation backends they support, modifications required to accommodate existing IPs into the framework, and choice of communication medium between the simulations. However, to our knowledge, none of the work achieves scalability to all of 1) simulation backends, 2) target IPs and 3) many endpoints to be truly useful across a wide range of target designs.

Muñoz-Quijada et al. [24] 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 for high-fidelity RTL simulations, whereas our approach allows for mix-and-match of different simulation backends. Similarly, CFC [23] 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 exter-

nal 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 5.

More recently, Switchboard [5] is an open-source co-simulation framework that supports multiple simulation backends including FPGA, RTL simulations and software functional models. Its use of simple shared-memory queues and lightweight packet format enables relatively low effort of integration for existing IPs. However, it only supports point-to-point connections between the IPs, 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.

Finally, prior work on heterogeneous SoC simulation makes it easy to integrate heterogeneous hardware blocks into a single gem5 simulation [25, 12, 26]. In addition to heterogeneous integration, our work focuses on the co-simulation perspective, where the execution of simulation instances are decoupled from each other and mix-and-match of different simulation backends across the blocks is made possible.

Chapter 3

System Design & Implementation

As we motivated in the previous section, there is a need for a co-simulation framework that focuses on scalability and ease of use against a diverse set of hardware designs. We will now discuss the design of SonicSim that accomplishes these goals, as well as implementation details.

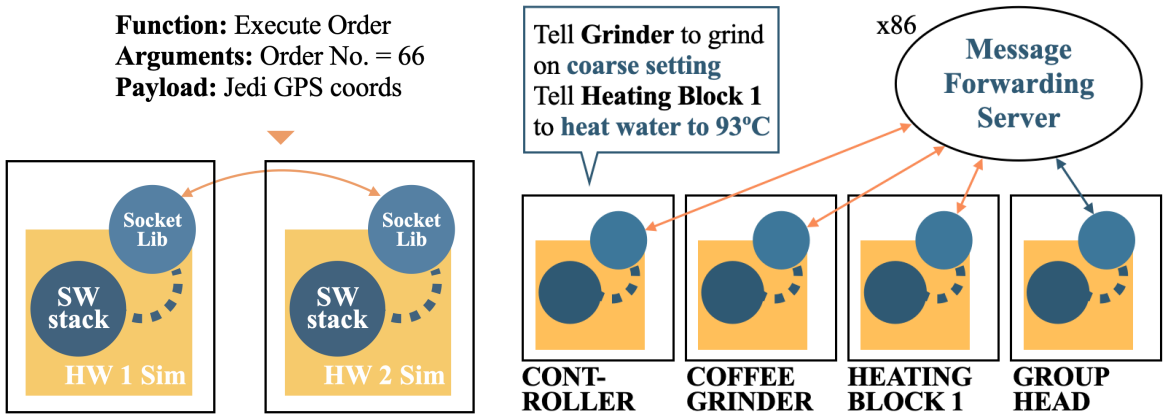
3.1 Communication Protocol

The communication protocol 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 communication channels may initiate a `send` or a `receive`.

Figure 3.1(a) showcases an example: the Emperor hardware block may transmit an "Execute Order 66"[1] to the clone trooper, something the Emperor knows the clone trooper understands, with destination, function name, and arguments correctly set. 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.

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

[1] In Star Wars, execute order 66 is what Emperor Palpatine (antagonist) issued to the clone troopers (army) to turn against the Jedis (good guys).



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

Figure 3.1: Communication protocols and examples

transaction level synchronization is required, the sender may choose to call `receive` itself with blocking enabled after sending the request. The receiver will `send` a response after receiving to indicate request completion. In essence, a reverse direction request may be used in place as a response, enabling synchronous remote procedure calls.

Hardware blocks can connect in either a peer-to-peer fashion as shown in Figure 3.1(a), or, if many hardware blocks are to be simulated, connect as clients to a dedicated central server as shown in Figure 3.1(b).

3.2 Implementation

We implemented the interface in the form of a C++ library, supporting IPC through either TCP or Unix domain sockets (UDS), the two types of socket-based IPC available on UNIX. 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.

Sending

In our SonicSim 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.

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).

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.

Chapter 4

Microbenchmarks

The design of SonicSim allows for a certain design space with regards to the blocking/non-blocking nature of the messages, the communication architecture of the simulation endpoints, and the actual implementation of the inter-process communication. In this section, we carry out microbenchmarks to quantify impact the performance of the different design choices and better navigate the design space.

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 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.

Figures 4.1, 4.2, and 4.3 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

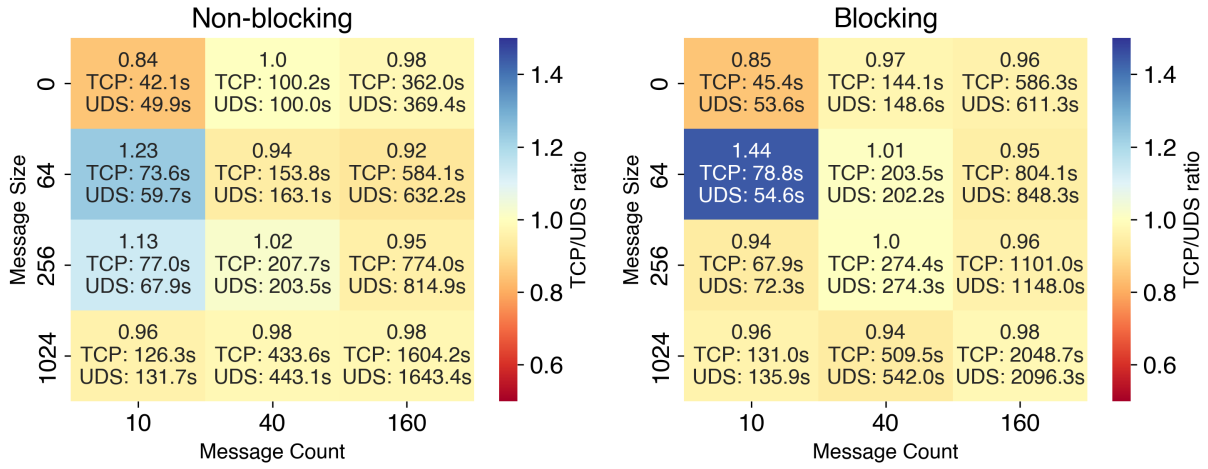


Figure 4.1: RTL simulation, TCP vs. UDS (lower=TCP better)
TCP is slightly faster than UDS, for both blocking and non-blocking.

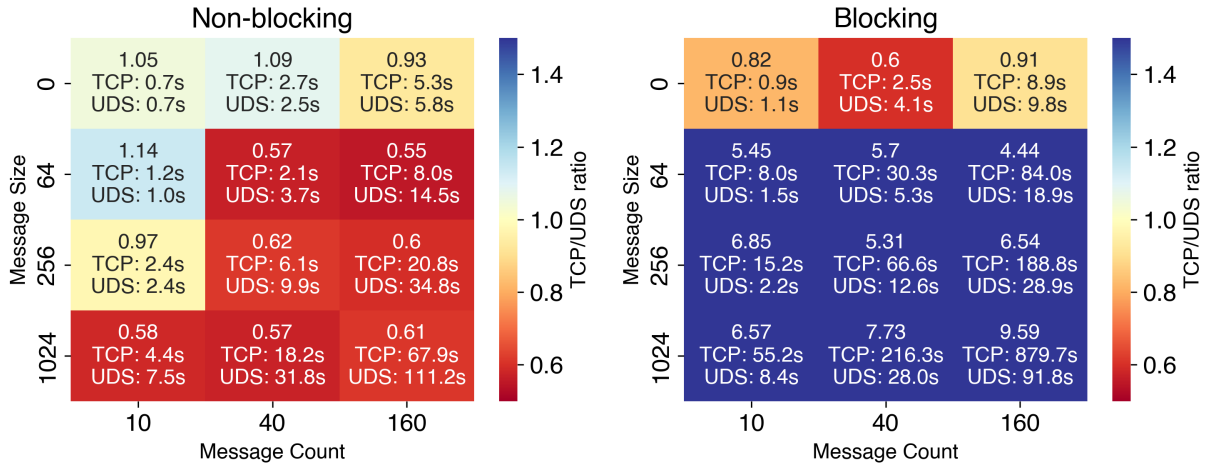


Figure 4.2: Functional simulation, TCP vs. UDS (lower=TCP better)
TCP is faster for non-blocking messages while UDS is faster for blocking.

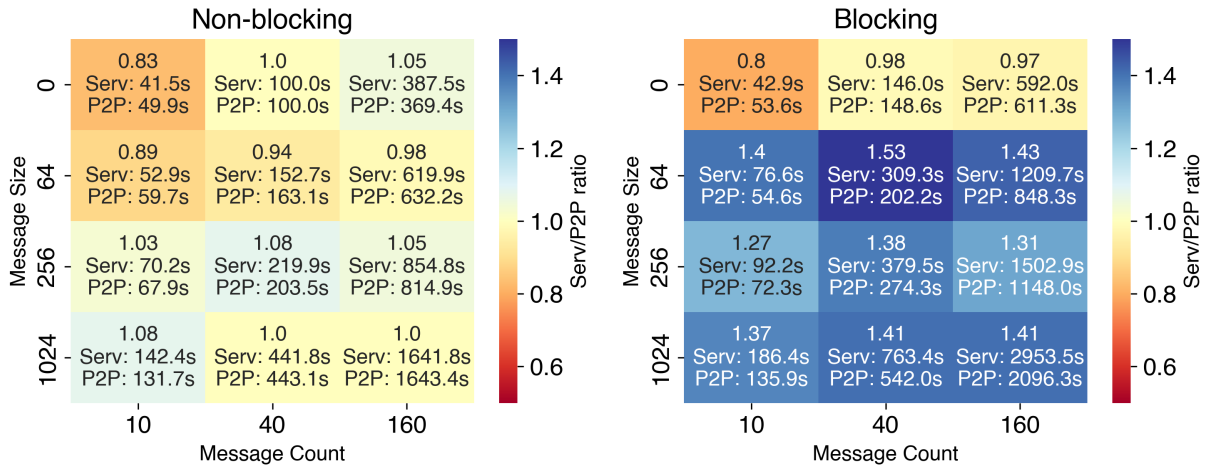


Figure 4.3: RTL simulation, P2P vs. Server-client (lower=Server better)
Central server adds 30%-50% overhead but only when blocking.

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 4.1, 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 4.2, 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.

Chapter 5

Case Study: CPU-GPU

5.1 Overview

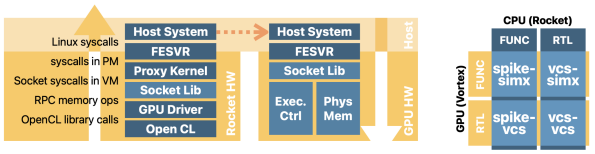
To demonstrate how SonicSim can aid in the development and simulation of a large-scale hardware design, we include a case study of modeling a System-on-chip that integrates CPU and GPU cores. It demonstrates that the co-simulation integration is possible without hardware modifications, is more scalable than its monolithic SoC counterpart, and allows fast single-module design iteration with a mix of functional and cycle-accurate simulation.

5.2 Background

We integrate Vortex [29] 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 [17]. 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 Verilog simulators such as VCS or Verilator.

For the CPU design, we leverage Rocket [4], an 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 [20] simulations. Rocket's support for a wide range of backends makes it an ideal target for demonstrating SonicSim's capabilities of mix-and-matching different backends across components.

The SoC hosting Rocket is built with Chipyard, 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



(a) Software stack (b) Co-simulation

Figure 5.1: System design of CPU-GPU co-simulation

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.

5.3 System Design & Implementation

Figure 5.1 shows that the co-simulation of CPU and GPU uses the peer-to-peer communication setup with two simulation processes. The CPU simulation runs a host binary linked with the PoCL runtime, which calls a driver to execute OpenCL calls. The driver internally invokes the SonicSim library, which 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.

CPU stack

All libraries are linked to the host code statically, including the PoCL runtime, GPU driver, and the SonicSim library. The kernel binary is precompiled. The GPU driver used by the runtime is modified to communicate with the GPU using SonicSim. When the GPU device is initialized in the host program, the driver as a client connects to the GPU simulation server. Data transfers and control signals onward are handled by the driver through socket IPC.

GPU stack

The 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 SonicSim 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`.

Table 5.1: Physical memory map of the GPU

Address	Usage	Address	Usage
`0x7c000000`	Finished? MMIO register	`0x80000000`	Binary code
`0xc0100000`	Heap region for operands	`0xffffffff`	Stack start
`0x7fff0000`	Kernel launch params		

HTIF is modified to non-blockingly `receive` four function calls: `write`, `read`, `run`, and `wait`, at a fixed interval. The lifecycle of a kernel in terms of CPU-GPU communication using these four functions are as follows:

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 5.1.
2. **Upload kernel binary**: The kernel binary is uploaded similarly, which is then written to GPU DRAM.
3. **Start execution**: When HTIF receives the `run` request, it `reset`s 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 completion through an MMIO register. It sends back an acknowledgment after the GPU finishes.
5. **Download results**: A `read` request prompts HTIF to read the destination buffer from the heap and send it to back to the CPU process.

5.4 Evaluation

Baseline

We aim to compare our work to a full Rocket-Vortex SoC integration baseline; however due to the difficulty of such integration, we emulated this by running and adding the results of two independent simulations. In each simulation, only one block is active at a time executing its stack, while the other lays dormant to scale the design size up to the full monolithic CPU-GPU SoC.

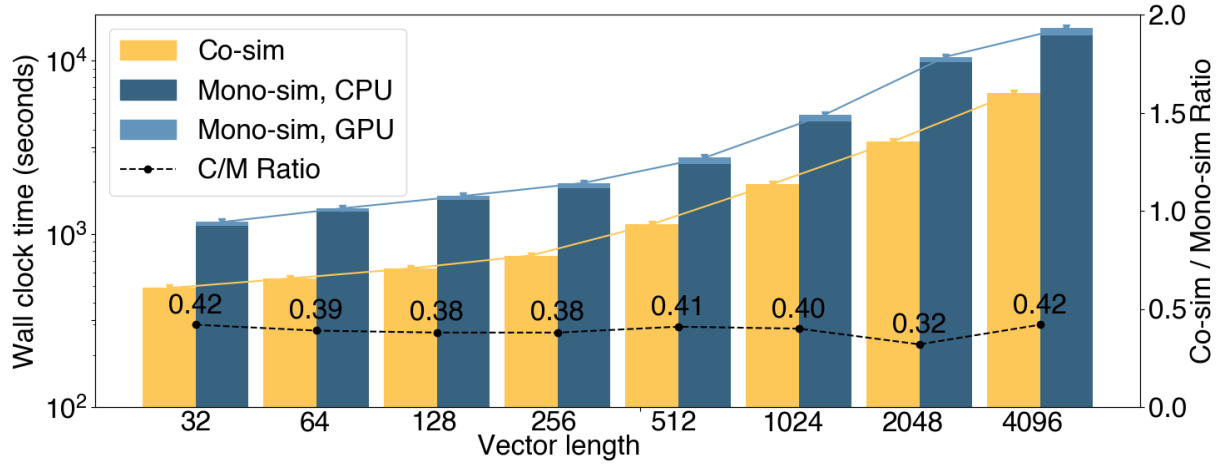


Figure 5.2: Real time comparison: RTL-RTL co-simulation
Our work simulates at 2.38× to 3.13× the speed versus the baseline.

Experiment setup

We test two cases of co-simulation with SonicSim. 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 workload used for the co-simulation of CPU and GPU is Softmax, a typical operation in ML suitable for GPGPUs. The Softmax kernel code is written in the OpenCL C Language, which can be compiled to a RISC-V binary by the PoCL runtime.

We created a host binary that 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 SonicSim library. The real time and the cycle counts from co-simulation are then compared against those of the monolithic simulation.

RTL-RTL co-simulation

This case shows that RTL-RTL co-simulation using SonicSim 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.

Figure 5.2 shows the comparison of wall clock time elapsed during the RTL-only monolithic simulation and the RTL-RTL socket co-simulation. We see 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 roughly linear. The most important

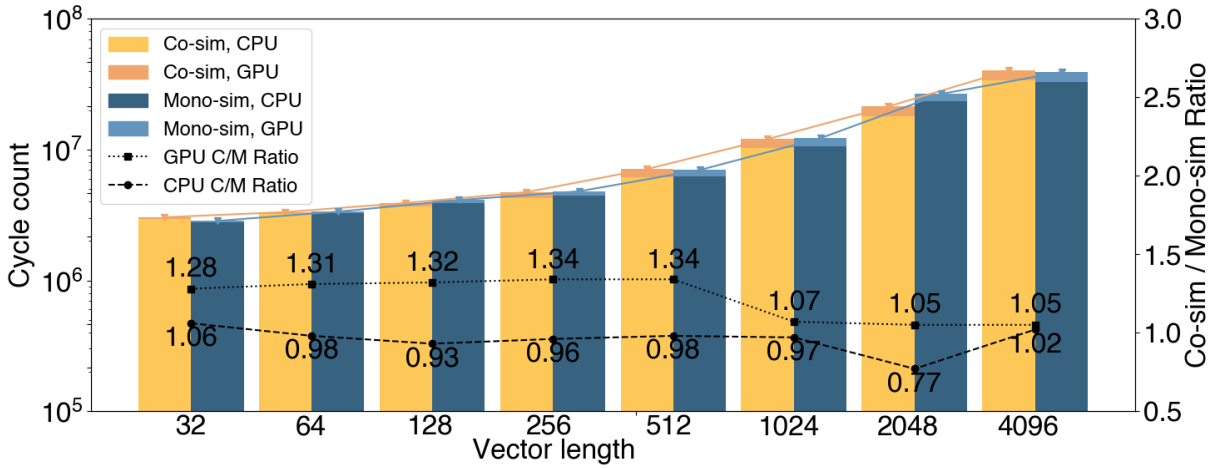


Figure 5.3: Cycle count comparison: RTL-RTL co-simulation
At larger kernel sizes, cycle inaccuracy is down to 5%.

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, suggesting the co-simulation of individual cores with smaller design sizes is much more scalable than a full SoC simulation.

Figure 5.3 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 1.3× 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.

Functional-RTL co-simulation

This test case shows that the mix-and-match capabilities of SonicSim 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.

Figure 5.4 compares simulation time between the baseline monolithic simulation and the functional-RTL co-simulation. Because the CPU process, Spike, is a functional model, the overall end-to-end simulation time is greatly reduced. Our data show co-simulation is about 30× faster when the input vector length is small; as the vector length increases, this factor goes down and stabilizes at around 10×. In cases such as this where components not of interest are significantly impacting performance, replacing them with functional models becomes possible and desirable for fast iteration.

Even with a key component replaced by a functional model, we were able to obtain useful cycle metrics from the co-simulation, as shown in Figure 5.5. In this plot, CPU and total

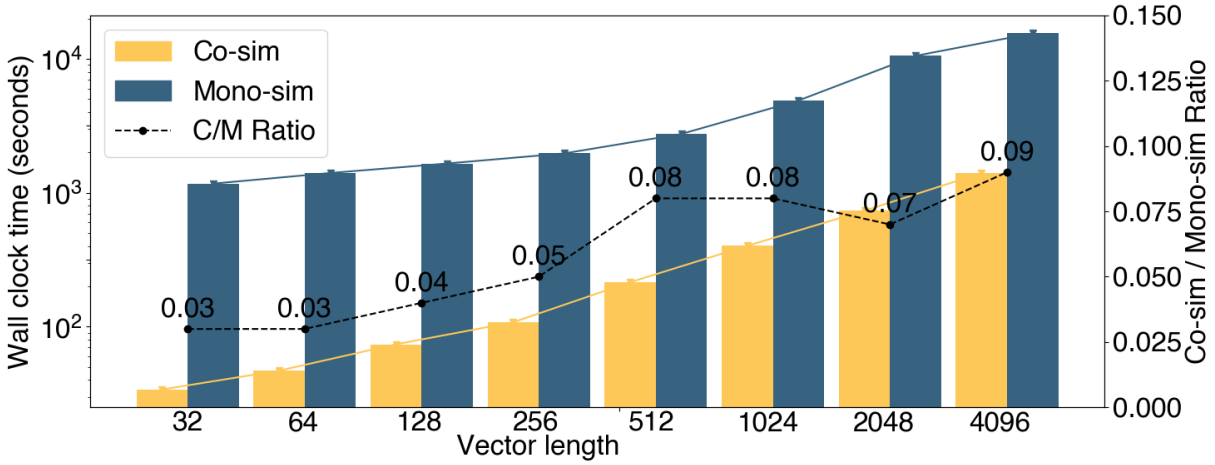


Figure 5.4: Real time comparison: Funct-RTL co-simulation
Functional-RTL simulation is 10×-30× faster than RTL-RTL.

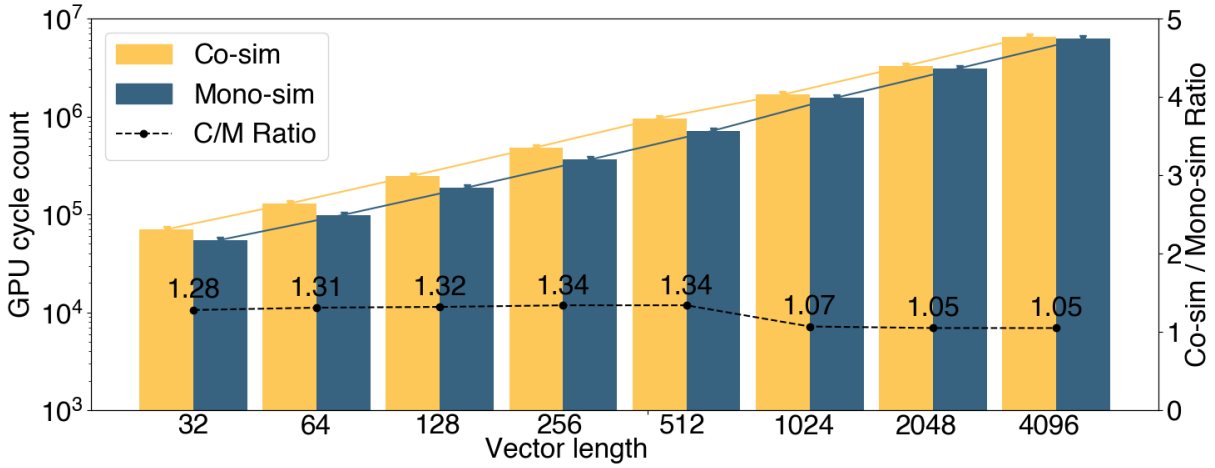


Figure 5.5: Cycle count comparison: Funct-RTL co-simulation
Even with a functionally simulated CPU, the GPU cycles are only 5% off.

Table 5.2: Lines of code for the CPU-GPU case study

Component	Lines	Component	Lines
PoCL runtime changes	110	GPU driver changes	43
Proxy kernel changes	191	FESVR changes	147
Softmax kernel code	22	Softmax host side library	263

cycle counts are excluded because Spike is not cycle-accurate; however, the GPU cycle 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.

Lines of code

Lines of code required to conduct this case study is shown in Table 5.2.

Chapter 6

Case Study: Many accelerators

6.1 Overview

This case study investigates the practical application and efficacy of SonicSim 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.

6.2 Background

The immense scale of large language models (LLMs) [31] has made it desirable to parallelize inference across multiple hardware accelerators, making modeling such a workload in simulation a compelling objective. In this case study, we attempt to explore this idea by simulating ML workloads running on multiple instances of Berkeley's machine learning accelerator generator Gemmini [14]. 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 [13] 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 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.

There are two previous efforts to integrate multiple Gemmini's into one design. *MoCA* [22] is able to support multiple accelerators in one SoC, but is limited to having each accelerator run a different workload. *AuRORA* [21] uses dedicated manager and client nodes in the hardware design to facilitate a virtualized acquire/release system for the accelerators. 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) [30]. 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.

6.3 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 SonicSim. 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.

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 are 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.

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 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.

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.

Gemmini workers

Each worker, as previously stated, is its own simulation process, thereby having independent states. The worker hardware simulated consists of a Rocket core and a Gemmini attached to it. The Rocket core runs a "headless" binary that communicates with SonicSim, 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. This setup eliminates the need to change the Gemmini hardware design, as only the software stack is modified.

Bare-metal SonicSim

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.

6.4 Evaluation

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 correctness, 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.

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 6.1;
- 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.

Table 6.1: Encoder layer parameters for different sizes tested

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

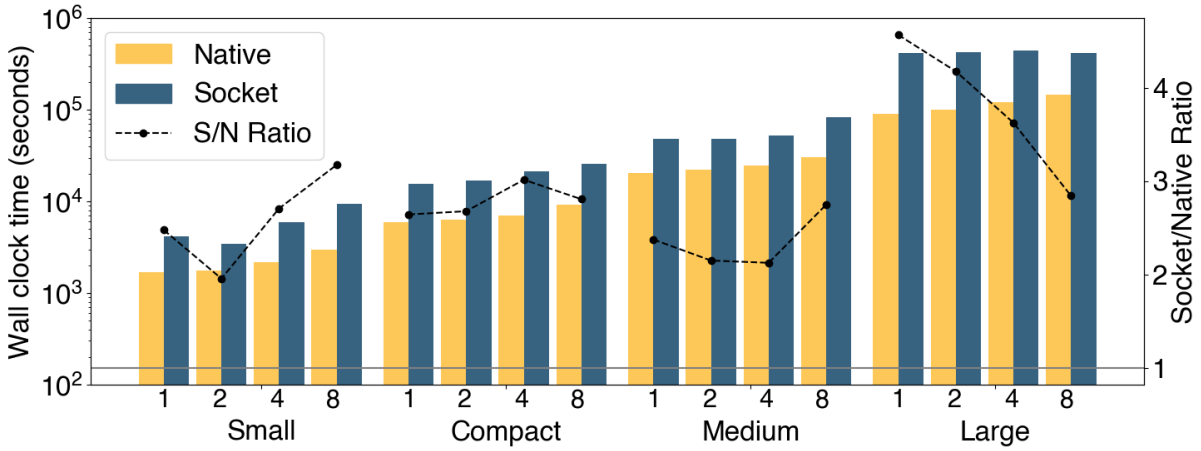


Figure 6.1: RTL simulation results running serial execution
SonicSim has about 2× to 4.5× overhead.

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 SonicSim library by timing entrances to and exits from library function calls.

Results and analysis

Figure 6.1 shows the real time comparisons of serial execution using SonicSim (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 SonicSim. The overhead likely stems from latencies in MMIO, socket communication, and fencing.

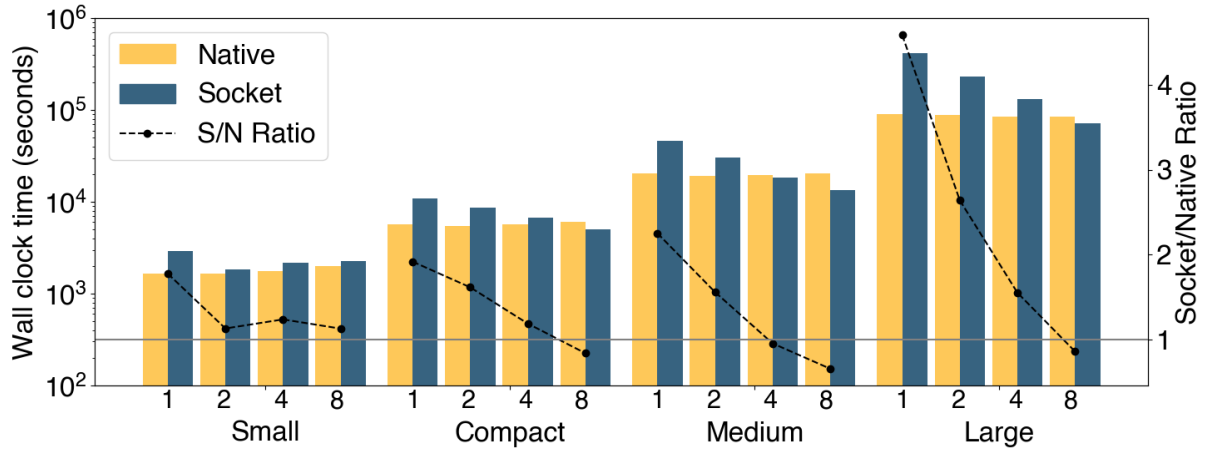


Figure 6.2: RTL simulation results running parallel execution
With enough workers, parallel SonicSim simulation is 15% to 35% faster.

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.

Figure 6.2 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 SonicSim simulation approaches the 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 SonicSim 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.

We can more visibly see the cause of the efficiency in Figure 6.3. 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.

Figure 6.4 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 SonicSim socket syscalls 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 SonicSim.

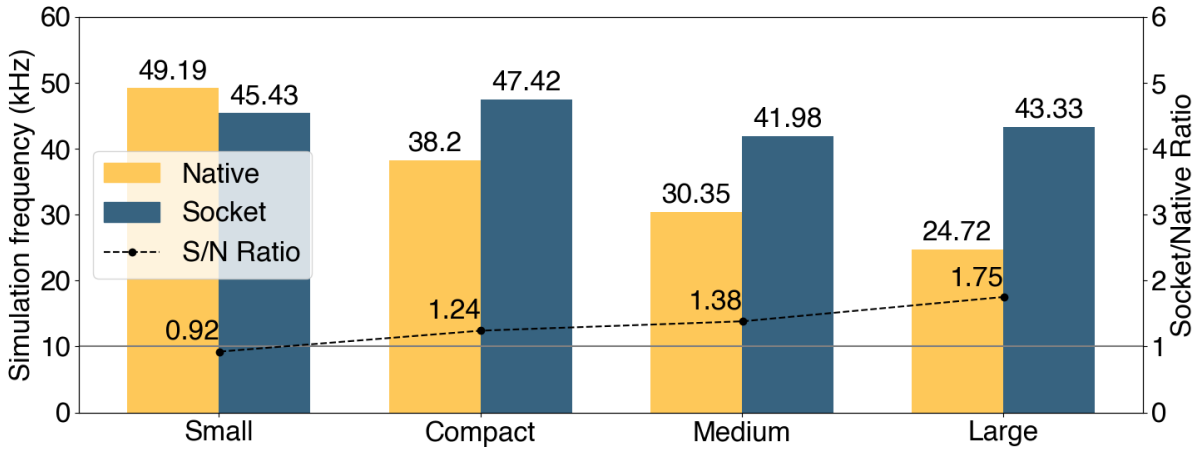


Figure 6.3: Simulation clock speed
Monolithic simulation times scale poorly with larger designs.

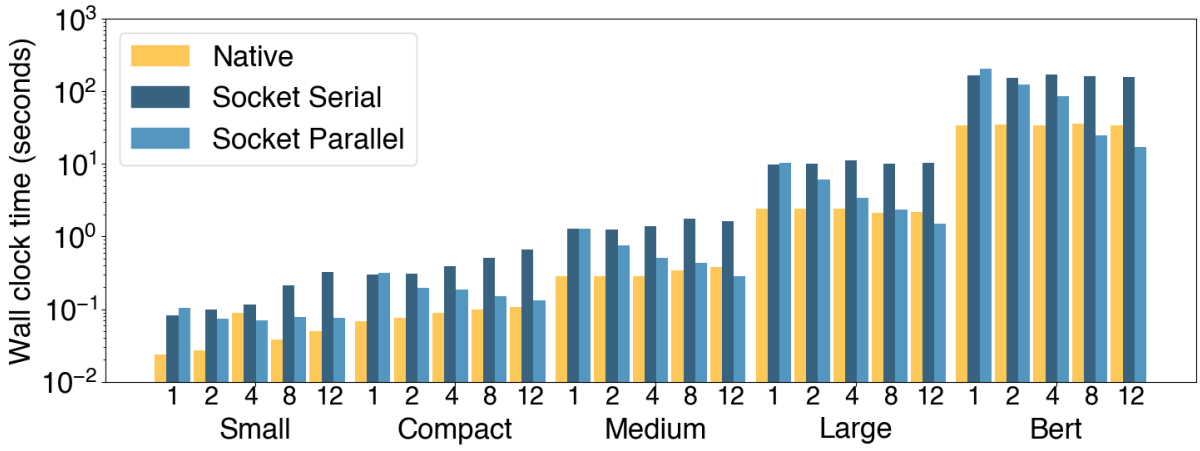


Figure 6.4: Functional simulation results
Serial cases are slower; parallel cases are faster with more workers

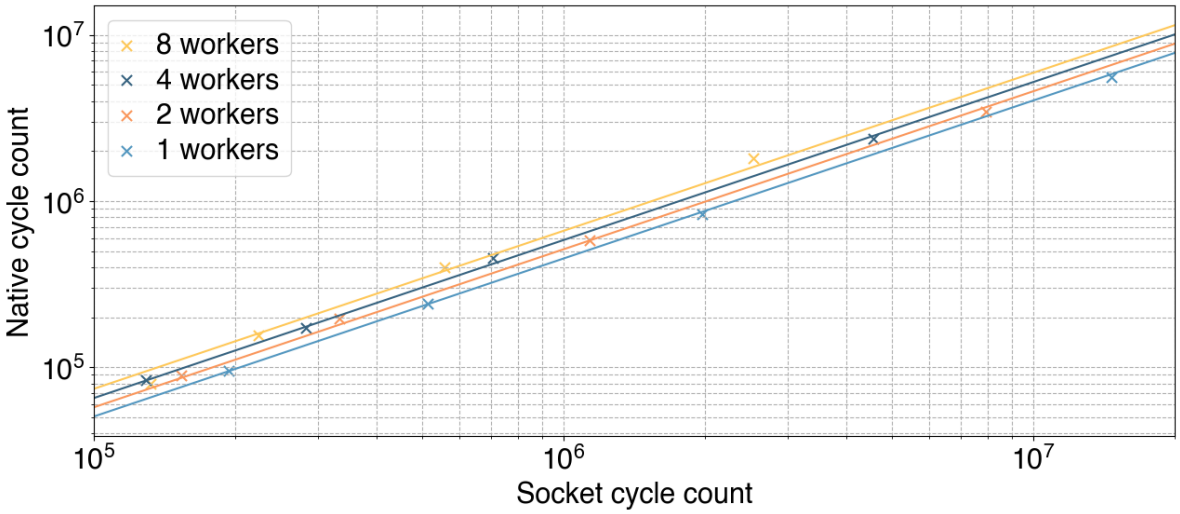


Figure 6.5: Linear regression predicting true cycle numbers
True cycle numbers can be predicted within 5% inaccuracy.

Table 6.2: Lines of code for the many-accelerators case study

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

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.

The recorded cycle numbers proved to be an overestimate when co-simulating. However, we observed a strong correlation between the baseline figures and the SonicSim figures. Using the 16 data points we obtained from the parallel test cases, we fitted a simple linear regression model. The independent variables are log(number of workers) and log(socket cycles), and the predicted dependent variable is log(native cycles). Shown in Figure 6.5, our model predicts the true cycle numbers (non-log) with an accuracy of 94.9%.

Finally, we show the lines of code required to implement each of the components in this case study in Table 6.2.

Chapter 7

Conclusion

7.1 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.

7.2 Summary

In this report, we propose SonicSim, a socket-based hardware co-simulation framework that focuses on scalability and ease-of-use. Through a CPU-GPU co-simulation and a many-accelerators integration case study, we show that SonicSim can be applied to diverse hardware blocks with minimal engineering effort. We show significant simulation time reduction while retaining close cycle number approximation, enabling accurate performance modeling and fast design iteration.

Bibliography

[1] AMD. *AMD EPYC™ 9004 SERIES PROCESSORS*. 2023. URL: https://www.amd.com/content/dam/amd/en/documents/products/epyc/epyc-9004-series-processors-data-sheet.pdf.

[2] Alon Amid et al. "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs". In: *IEEE Micro* 40.4 (2020), pp. 10–21. DOI: 10.1109/MM.2020.2996616.

[3] Apple. *Apple unveils M3, M3 Pro, and M3 Max, the most advanced chips for a personal computer*. 2023. URL: https://www.apple.com/newsroom/2023/10/apple-unveils-m3-m3-pro-and-m3-max-the-most-advanced-chips-for-a-personal-computer.

[4] Krste Asanovic et al. "The rocket chip generator". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* 4 (2016), pp. 6–2.

[5] Zero ASIC. "Switchboard: An Open Source High-Performance Communication Platform". In: (2023). URL: https://www.zeroasic.com/blog/switchboard-release.

[6] RISC-V International Association. *RISC-V Proxy Kernel and Boot Loader*. 2023. URL: https://github.com/riscv-software-src/riscv-pk.

[7] RISC-V International Association. *Spike RISC-V ISA Simulator*. 2023. URL: https://github.com/riscv-software-src/riscv-isa-sim.

[8] Jonathan Bachrach et al. "Chisel: Constructing hardware in a Scala embedded language". In: *DAC Design Automation Conference 2012*. 2012, pp. 1212–1221. DOI: 10.1145/2228360.2228584.

[9] Ali Bakhoda et al. "Analyzing CUDA workloads using a detailed GPU simulator". In: *2009 IEEE international symposium on performance analysis of systems and software*. IEEE. 2009, pp. 163–174.

[10] Nathan Binkert et al. "The gem5 simulator". In: *ACM SIGARCH computer architecture news* 39.2 (2011), pp. 1–7.

[11] Inc Cerebras Systems. *Wafer-Scale Engine: The Largest Chip Ever Built*. 2021. URL: https://f.hubspotusercontent30.net/hubfs/8968533/WSE-2%20Datasheet.pdf.

[12] Jason Cong et al. "Parade: A cycle-accurate full-system simulation platform for accelerator-rich architectural design and exploration". In: *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2015, pp. 380–387.

[13] Henry Cook. "Productive Design of Extensible On-Chip Memory Hierarchies". PhD thesis. EECS Department, University of California, Berkeley, May 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-89.html.

[14] Hasan Genc et al. "Gemmini: An Agile Systolic Array Generator Enabling Systematic Evaluations of Deep-Learning Architectures". In: *CoRR* abs/1911.09925 (2019). arXiv: 1911.09925. URL: http://arxiv.org/abs/1911.09925.

[15] Rehan Hameed et al. "Understanding sources of inefficiency in general-purpose chips". In: *Proceedings of the 37th annual international symposium on Computer architecture*. 2010, pp. 37–47.

[16] Ventana Micro Systems Inc. "Ventana Introduces Veyron V2 — World's Highest Performance Data Center-Class RISC-V Processor and Platform". In: (2023). URL: https://www.prnewswire.com/news-releases/ventana-introduces-veyron-v2--worlds-highest-performance-data-center-class-risc-v-processor-and-platform-301980591.html.

[17] Pekka Jääskeläinen et al. "pocl: A performance-portable OpenCL implementation". In: *International Journal of Parallel Programming* 43 (2015), pp. 752–785.

[18] 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)*. 2023, pp. 1–8. DOI: 10.1109/CONIT59222.2023.10205723.

[19] Sagar Karandikar et al. "CDPU: Co-designing Compression and Decompression Processing Units for Hyperscale Systems". In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 2023, pp. 1–17.

[20] Sagar Karandikar et al. "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. Los Angeles, California: IEEE Press, 2018, pp. 29–42. ISBN: 978-1-5386-5984-7. DOI: 10.1109/ISCA.2018.00014. URL: https://doi.org/10.1109/ISCA.2018.00014.

[21] Seah Kim et al. "Aurora: Virtualized accelerator orchestration for multi-tenant workloads". In: *56th Annual IEEE/ACM International Symposium on Microarchitecture* (2023). DOI: 10.1145/3613424.3614280.

[22] Seah Kim et al. "Moca: Memory-centric, adaptive execution for multi-tenant Deep Neural Networks". In: *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (2023). DOI: 10.1109/hpca56546.2023.10071035.

[23] Ryan Lund. "Design and Application of a Co-Simulation Framework for Chisel". In: (2021).

[24] Maria Muñoz-Quijada, Luis Sanz, and Hipolito Guzman-Miranda. "SW-VHDL Co-Verification Environment Using Open Source Tools". In: *Electronics* 9.12 (2020), p. 2104.

[25] Jason Power et al. "gem5-gpu: A heterogeneous cpu-gpu simulator". In: *IEEE Computer Architecture Letters* 14.1 (2014), pp. 34–36.

[26] Yakun Sophia Shao et al. "Co-designing accelerators and SoC interfaces using gem5-Aladdin". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2016, pp. 1–12.

[27] Wilson Snyder. "Verilator and systemperl". In: *North American SystemC Users' Group, Design Automation Conference*. 2004.

[28] SambaNova Systems. "SambaNova Announces Next Generation DataScale System, Setting a World Record for Time-to-Train Performance". In: (2022). URL: https://sambanova.ai/press/SambaNova-Announces-Next-Generation-DataScale-System-Setting-a-World-Record-for-Time-to-Train-Performance.

[29] Blaise Tine et al. "Vortex: Extending the RISC-V ISA for GPGPU and 3D-graphics". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 2021, pp. 754–766.

[30] Ashish Vaswani et al. "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[31] Wayne Xin Zhao et al. *A Survey of Large Language Models*. 2023. arXiv: 2303.18223 [cs.CL].