Towards Scalable and Efficient Parallel RTL and Network Simulation Systems

by

Jie Tong

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2025

Date of final oral examination: 08/22/2025

The dissertation is approved by the following members of the Final Oral Committee:
Tsung-Wei Huang, Associate Professor, Electrical and Computer Engineering, Chair
Umit Y. Ogras, Professor, Electrical and Computer Engineering, Co-chair
Mikko Lipasti, Professor, Electrical and Computer Engineering
Dan Negrut, Professor, Mechanical Engineering

Dedicated to my family, friends, and all those I've met along the way who have shaped and supported my journey.

Acknowledgements

Completing a Ph.D. is a challenging yet deeply rewarding journey—one that would not have been possible without the support, guidance, and encouragement of many people to whom I am sincerely grateful.

First and foremost, I would like to express my deepest gratitude to my Ph.D. advisors, Dr. Tsung-Wei Huang and Dr. Umit Y. Ogras. Their unwavering support, insightful guidance, and constant encouragement have shaped not only this dissertation but also my development as a researcher. I am especially thankful for their mentorship, intellectual rigor, and the freedom they gave me to explore my ideas. Their patience and belief in my work have been invaluable throughout my doctoral journey.

I am also thankful to my doctoral committee members, Dr. Mikko Lipasti and Dr. Dan Negrut. Their constructive feedback and thoughtful suggestions were instrumental in refining this dissertation. I am grateful to the faculty and staff at the University of Wisconsin–Madison for their dedication to teaching, research mentorship, and academic service. Their commitment created a stimulating intellectual environment, a collaborative spirit, and a supportive community that made my

time in Madison both productive and memorable.

I am especially grateful to my lab mates and friends at university who shared thoughtful discussions, technical insights, and moments of friendship that made this journey more meaningful: Shui Jiang, Cheng-Hsiang Chiu, Che Chang, Wan-Luan Lee, Chih-Chun Chang, Boyang Zhang, Yi-Hua Chung, Aditya Das Sarma, Bangya Liu, and many others. I would also like to thank my previous roommates, Wei Ye, Jiawei Zhou, and Ruohui Wang, for their companionship during my years in Madison.

I would like to express my sincere appreciation to my mentors and collaborators in industry. I am grateful for the opportunity to intern at NVIDIA, where I worked under the mentorship of Paulo de Moura, Bryan Walsh, Chuck Davenport, and many others. Their guidance provided me with invaluable industrial experience and helped shape my career trajectory. I also thank Emily Shriver from Intel Labs for her support during my early Ph.D. journey.

Finally, I offer my deepest gratitude to my parents and grandparents, whose unwavering love and support have been the foundation of everything I have accomplished. I am also profoundly thankful to my girlfriend for her love, patience, and encouragement. Words cannot fully express how much their presence and support have meant to me.

Contents

List of Tables

List of Figures

Abstract

As the complexity of deep learning accelerators and data center networks continues to increase, traditional simulation tools struggle to meet the demands for scalability, performance, and analytical fidelity. This thesis addresses these challenges through two complementary approaches: a unified, compiler-driven framework for accelerating RTL simulation, and an ML-assisted analytical methodology for scalable network performance modeling.

The thesis first presents *BatchSim*, a parallel RTL simulator that leverages inter-cycle batching and task graph parallelism to amortize simulation overhead and improve runtime efficiency across large-scale designs. Building on this, *ScaleRTL* introduces a compiler-based code generation flow that detects structurally repeated components and reuses evaluation logic to eliminate redundant code generation. By targeting both CPU and GPU backends using MLIR, *ScaleRTL* achieves dramatic compilation speedups and scalable simulation performance. To support heterogeneous simulation platforms, *HeteroRTL* introduces architecture-aware partitioning to map complex, control-heavy modules to CPUs and parallel processing elements to GPUs. This hybrid execution model maximizes compute resource utilization

and further accelerates simulation workloads.

Complementing the RTL contributions, *MQL* offers a machine-learning-assisted analytical modeling technique for large-scale data center networks. By combining queuing theory with the Maximum Entropy method and regression tree learning, MQL achieves less than 3% modeling error while providing 100×–9000× speedups over packet-level simulations like ns-3. It delivers detailed latency estimation at scale, supporting rapid, accurate performance evaluation of distributed network architectures.

In summary, these contributions advance the state of the art in simulation and modeling across both RTL and network domains. Through compiler optimization, structural reuse, parallelism, and lightweight learning, this dissertation provides scalable solutions for high-performance simulation in modern hardware systems.

Chapter 1

Introduction

As hardware systems continue to grow in complexity, ranging from deep learning accelerators composed of thousands of replicated compute units to data center networks with tens of thousands of interconnected nodes, simulation becomes a critical bottleneck in both design and verification workflows. Register Transfer Level (RTL) simulation remains the gold standard for functional validation, yet existing tools are increasingly unable to keep pace with modern design scales due to excessive runtime and compilation overhead. Meanwhile, packet-level simulation of large-scale networks offers high fidelity but is prohibitively slow, making rapid design space exploration impractical. This thesis tackles both challenges with two distinct but complementary solutions: a unified, compiler-driven simulation framework for RTL, and an ML-assisted analytical modeling methodology for scalable network performance estimation.

RTL Simulation: Unified Compiler-Based Framework. RTL simulation is foun-

dational in the verification of hardware designs, but conventional approaches suffer from inefficiencies when scaling to large, hierarchical, and structurally repetitive systems. Simulators like Verilator rely on cycle-accurate evaluation using C++ and multithreading, which leads to high synchronization overhead and redundant code generation. Deep learning accelerators, for example, frequently contain systolic arrays and replicated cores that are evaluated independently, despite their structural similarity.

To overcome these limitations, this thesis introduces a unified RTL simulation framework built on the MLIR compiler infrastructure. The first component, *BatchSim*, introduces inter-cycle batching to amortize simulation overhead and applies task graph parallelism to achieve efficient execution across cycles. Next, *ScaleRTL* extends the framework by identifying structurally repeated logic and reusing evaluation code to reduce both compilation time and binary size, supporting both CPU and GPU targets. Finally, *HeteroRTL* introduces architecture-aware partitioning to map control-heavy modules to CPU and parallel compute blocks (e.g., PEs) to GPU, enabling hybrid simulation that maximizes hardware utilization. Collectively, these techniques form a compiler-based pipeline that achieves scalable, parallel RTL simulation across heterogeneous platforms.

Network Modeling: ML-Assisted Analytical Framework. While RTL simulation targets low-level hardware verification, performance modeling of large-scale data center networks (DCNs) presents a different challenge. Packet-level network simulators, though accurate, are infeasible for evaluating networks with thousands of nodes and complex routing schemes. Analytical modeling offers a lightweight

alternative but struggles to maintain accuracy under realistic traffic conditions and topologies.

To address this, the thesis introduces *MQL*, a machine learning-assisted queuing latency modeling framework. MQL first constructs analytical latency estimates using the Maximum Entropy method, and then applies regression tree learning to correct systematic errors observed when compared with packet-level simulation. This approach achieves less than 3% modeling error on average and up to 9000× speedup over ns-3 simulations, while providing queue- and tier-level visibility for network designers. Unlike the RTL framework, MQL does not rely on compiler infrastructure, but instead combines analytical insight with lightweight supervised learning for scalable and accurate performance prediction.

This dissertation contributes two distinct frameworks for accelerating simulation and modeling at different layers of the hardware/software stack:

1. *BatchSim*: A parallel RTL simulation technique that employs inter-cycle batching and task graph execution to reduce runtime overhead.

2. *ScaleRTL*: A compiler-based code generation flow that identifies structural redundancy and emits reusable simulation code for CPUs and GPUs.

3. *HeteroRTL*: A hybrid simulation framework that partitions RTL designs between CPU and GPU execution based on architecture-aware analysis.

4. *MQL*: An ML-augmented analytical modeling methodology for data center networks that balances scalability and accuracy without relying on packet-level simulation.

Collectively, these contributions push the frontier of high-performance simulation by enabling faster design iteration, improved scalability, and practical performance modeling for both compute-intensive accelerators and large-scale communication networks.

The remainder of this dissertation is organized as follows: Chapters 2–5 provide detailed descriptions of the four core contributions outlined above.

Chapter 2

BatchSim: Parallel RTL Simulation using Inter-cycle Batching and Task Graph Parallelism

As the design complexity continues to increase, parallelizing Register Transfer Level (RTL) simulation has become crucial for verifying the design functionality with reasonable performance and turnaround time. State-of-the-art simulators focus on exploring parallelism within a single simulation cycle. However, intra-cycle parallelism does not scale well because its instruction volumes cannot offset the overhead of multithreading. To overcome this challenge, we introduce *BatchSim*, a parallel RTL simulator leveraging inter-cycle batching and task graph parallelism. Unlike existing RTL simulators, BatchSim combines multiple cycles into a single simulation workload, ensuring sufficient instruction volumes for effective parallelization.

Since RTL simulation consists of many irregular patterns, BatchSim partitions the simulation workload into a set of dependent subgraphs and parallelizes their executions using task graph parallelism. Compared with state-of-the-art RTL simulators, BatchSim can achieve 11%–98% speed-up on large industrial RTL designs.

2.1 Overview

Register Transfer Level (RTL) simulation plays a crucial role in the overall design flow because it verifies the functionality of a hardware design at the early stage [127]. Hence, RTL simulation is the cornerstone for various verification tasks, such as functional testing, debugging, and design space exploration. As the system-on-chip (SoC) complexity continues to grow, achieving industry-quality verification sign-off demands a substantial and growing amount of compute resources to simulate RTL for dozens of different units within an SoC across many thousands of stimuli. Therefore, RTL simulation can be very time-consuming in the verification process. For instance, researchers have reported that RTL simulations can take over 70% of the entire runtime when achieving coverage closure for a custom deep learning accelerator [97, 99]. Speeding up RTL simulation runtime is thus crucial for completing functional verification tasks with reasonable turnaround time and performance.

Many new algorithms have recently been proposed to accelerate RTL simulation. To give a few popular examples, Verilator[127], the leading open-source RTL simulator, transpiles (source-to-source compiles) an input RTL source (Ver-

ilog) into optimized C++ simulation code through abstract syntax tree (AST) traversals. ESSENT[8] enhances the simulation performance by partitioning an input RTL graph into several subgraphs with similar activities for load balancing. RTLflow[97] simulates multiple stimuli at one time by transpiling an input RTL source into optimized C++ and CUDA code. By harnessing the power of GPU task graph computing [95, 99, 22], RTLflow significantly improves the simulation throughput performance. RepCut[134] improves simulation efficiency by replicating specific nodes within an RTL graph to reduce synchronization overhead among threads. Through this replication-aided partitioning, RepCut can divide an input RTL graph into independent subgraphs that can completely run in parallel (i.e., embarrassing parallelism). Khronos[146] optimizes the memory access patterns during the simulation by proposing a queue-connected operation graph that captures temporal data dependencies, reschedules operations, and merges state accesses across cycles.

Despite improved simulation performance, existing simulators are largely limited to *single-cycle* simulation (see Figure 2.1), where the instruction volumes (e.g., simulation instruction, arithmetic operations) are typically not enough to parallelize most of the computing tasks. Specifically, running parallel RTL simulation can incur certain threading overhead at each cycle, such as scheduling tasks, synchronization, and dynamic load balancing [93]. For a simulation workload with N cycles, the overhead will accumulate N times. However, if we could simulate a batch of B cycles simultaneously, we could reduce the overhead to N/B times while allowing each thread to remain actively engaged in processing more instructions. This type

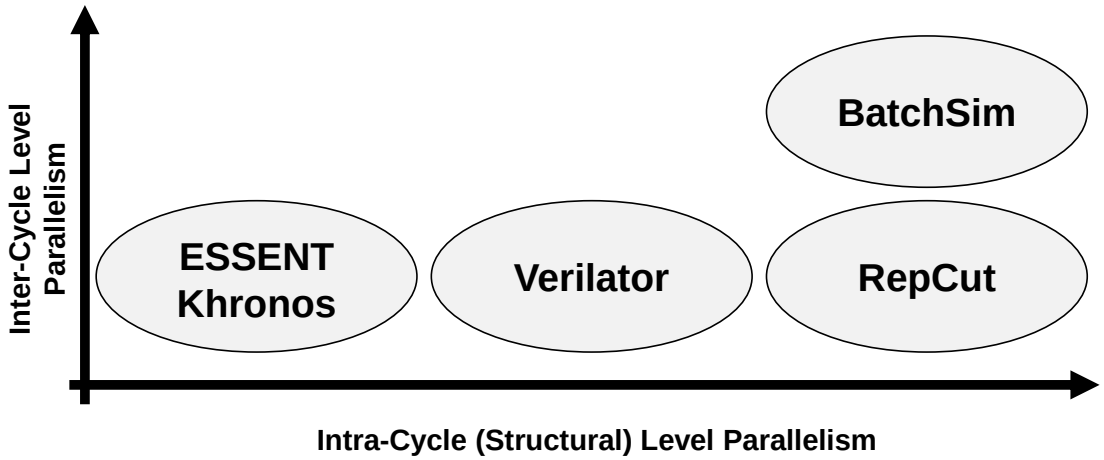


Figure 2.1: BatchSim explores both intra- and inter-cycle parallelism to significantly improve the performance of parallel RTL simulation.

of *batch* or *inter-cycle* simulation can bring significant yet untapped performance advantages to parallel RTL simulation.

This chapter presents BatchSim, a parallel RTL simulator using inter-cycle batching and task graph parallelism. Unlike existing simulators that evaluate one cycle per iteration, BatchSim simulates multiple cycles simultaneously by merging consecutive RTL graphs and leverages task graph parallelism to parallelize the simulation workload. We evaluate the performance of BatchSim on large industrial RTL designs. Compared with state-of-the-art RTL simulators, BatchSim can achieve 11%–98% speedup. We believe this late-breaking result will inspire new simulation research by exploring inter-cycle batch parallelism.

2.2 Background and Motivation

2.2.1 Full-Cycle RTL Simulation

RTL simulation transpiles RTL design code (such as Verilog or FIRRTL) into software code (such as C++ or LLVM IR), allowing compilers to optimize the simulation code for improved performance and efficiency. The simulation evaluates the design on a one-cycle per iteration basis, beginning each cycle by setting the clock and input, as shown in Listing 2.1. The RTL design is structured as a directed acyclic graph, known as an RTL computation graph. In each cycle, the simulator processes inputs and traverses this graph to generate output values. The code within a full-cycle simulator is relatively straightforward, simulating the entire design in every cycle. This approach ensures remarkably consistent execution times for each cycle. For smaller designs, this method typically achieves reasonably high instruction throughputs. However, as the design and the RTL computation graph grow in size, the demands on the host processor and memory can become overwhelming, potentially leading to performance bottlenecks.

```
Design dut;
size_t cycle = 0;
while (cycle < max_cycle)
{
  dut.set_clock();
  dut.load_input();
  dut.eval();
  dut.dump(cycle);
  ++cycle;
```

```
}
```

Listing 2.1: A C++ code snippet for full-cycle RTL simulation.

2.2.2 Motivation

State-of-the-art full-cycle RTL simulators have implemented various optimization techniques to enhance performance at the intra-cycle level, as illustrated in Figure 2.1. Notably, ESSENT[8] and Khronos[146] operate on a single-threaded model, whereas Verilator[127] and RepCut[134] employ multi-threaded simulations by partitioning the RTL computation graph and managing intra-cycle communications. Generally, larger computation graphs yield more significant benefits from parallel simulation because the relative costs of multithreading and synchronization overhead decrease as the scale increases. However, due to the fixed size of the com-

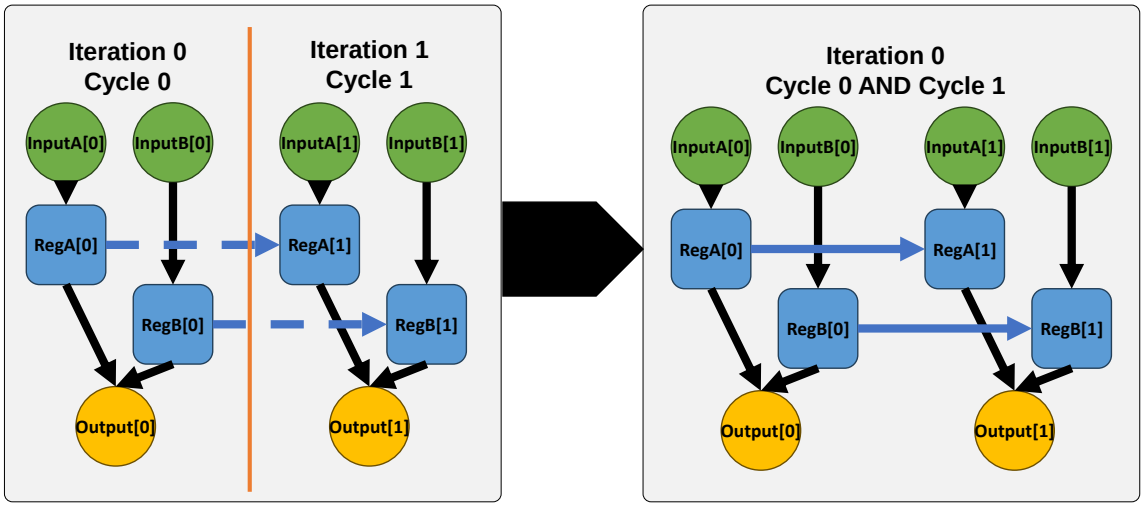


Figure 2.2: BatchSim batches consecutive cycle graphs and merges them into a multi-cycle computation graph.

putation graph inherent to the RTL design, small and medium-sized designs do not benefit as much from parallel simulation. Recognizing this limitation, we propose a novel approach as shown in Figure 2.2: batching consecutive cycle graphs and merging them into a multi-cycle computation graph. This strategy allows multiple cycles to be evaluated in a single iteration, potentially enhancing parallel performance. By partitioning and scheduling multi-threaded operations more effectively, this method reduces the relative multithreading overhead compared to the overall end-to-end simulation process, offering a promising direction for improving simulation efficiency across various design sizes.

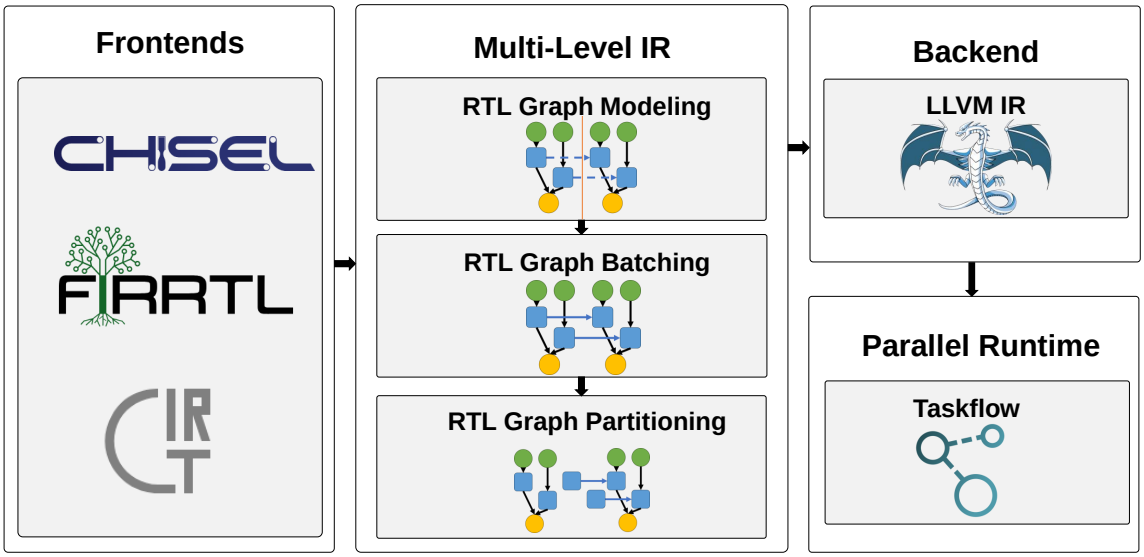


Figure 2.3: Overview of BatchSim.

2.3 BatchSim

Figure 2.3 illustrates the architecture of BatchSim, which comprises four main components: frontends, multi-level IR, backend, and parallel runtime. BatchSim utilizes the frontends and Intermediate Representations (IRs) in CIRCT[2] to accommodate various RTL designs and generates MLIR[85] dialects to leverage existing code generation and optimization passes. BatchSim incorporates the IR compilation infrastructure and RTL graph modeling from Khronos[146], integrates our inter-cycle graph batching pass, and adopts the graph partitioning method from RepCut[134]. It also employs the code emitting capabilities of the LLVM backend. Additionally, BatchSim utilizes the advanced parallel runtime, Taskflow[61, 55, 60, 23, 91, 93, 92], to facilitate multithreading task scheduling and synchronization, enhancing its efficiency and scalability.

2.3.1 Inter-Cycle Graph Batching

We utilize the internal data structure of the multi-level IR to handle the RTL design evaluation, which in MLIR[85] is represented as a graph. This graph comprises all operations and operands with their dependencies, forming a data dependency computation graph. In this RTL computation graph, traditional control flows such as if-else statements are absent. The computation graph is primarily focused on updating the values of signals, which are allocated as global variables in memory prior to launching the simulation. All input signals serve as graph ingress points, while intermediate and output signals act as egress points. The computation graph

is traversed and the signals are updated in each cycle. To implement the inter-cycle batching method, we developed a pass that clones input and output signals, appending suffixes like "_t0", "_t1" to them. Similarly, functions are cloned with suffixes "_t0", "_t1" added. These cloned functions are then sequentially placed within the main function according to their time order. An example of the output from the inter-cycle graph batching is shown in Listing 2.2. Subsequently, we utilize MLIR's built-in inline pass to inline all the sub-functions into the main function. Thanks to MLIR's Single Static Assignment (SSA) properties, all registers are automatically renamed, avoiding any naming conflicts.

```
def_queue @io_input_t0 depth 1 : i8 delay [0]
def_queue @io_output_t0 depth 1 : i1 delay [0]
def_queue @io_input_t1 depth 1 : i8 delay [0]
def_queue @io_output_t1 depth 1 : i1 delay [0]
func.func @Design_t0(){
  // evaluate design
}
func.func @Design_t1(){
  // evaluate design
}
func.func @Design(){
  call @Design_t0() : () -> ()
  call @Design_t1() : () -> ()
  return
}
```

Listing 2.2: An RTL evaluation IR after the inter-cycle batching pass.

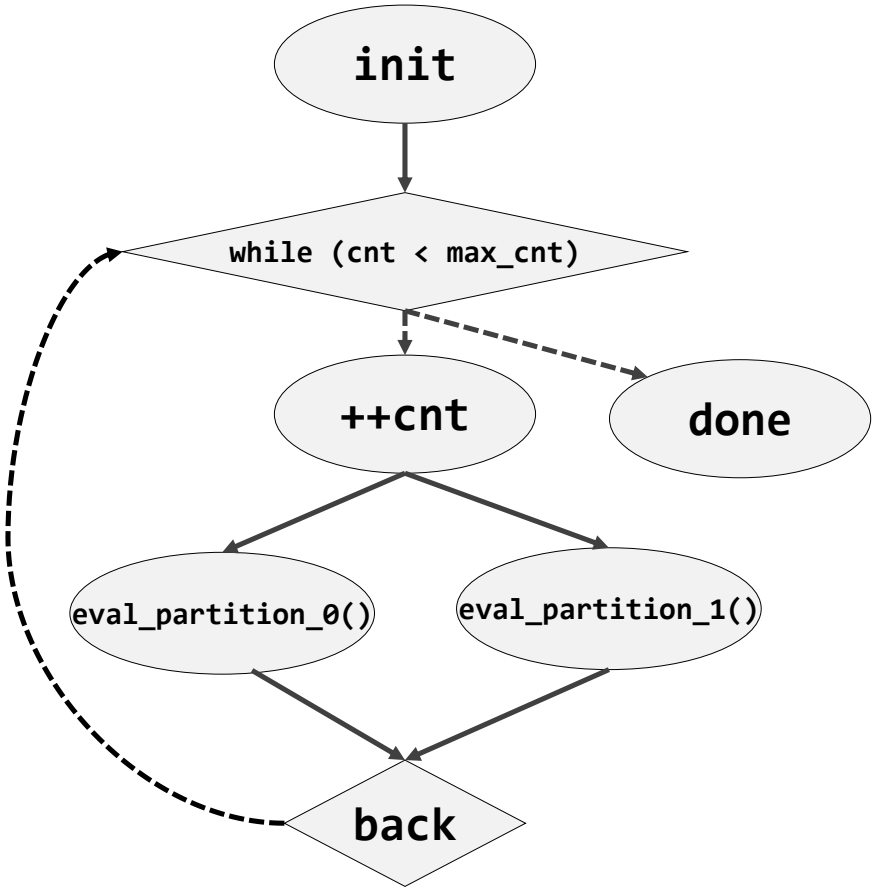


Figure 2.4: A task graph for the RTL simulation, which is parallelized through Taskflow.

2.3.2 Parallel Runtime

After completing the inter-cycle batching and graph partition passes, we build a *task graph* to describe the simulation workload. Figure 2.4 shows a simulation task graph example. Based on this task graph, we can initiate the multi-threaded simulation. In BatchSim, we utilize Taskflow[61, 55], a general-purpose task-parallel programming system, to describe our simulation task graph. Taskflow is comprised

solely of C++ header files, making it straightforward to integrate with the RTL simulator's C++ wrapper. Given the task dependency graph, we employ Taskflow's conditional tasking method, as depicted in Listing 2.3. In the provided code snippet, the *partition_* functions, generated by BatchSim, are organized into independent functions. These are then compiled to LLVM IR and subsequently to binary object code. Taskflow's runtime efficiently manages the scheduling and synchronization of these partitions, launching them in parallel and minimizing runtime overhead.

```
init.precede(cond);
cond.precede(body, done);
body.precede(task_eval_0, task_eval_1);
task_sync.succeed(task_eval_0, task_eval_1);
task_sync.precede(task_update_0, task_update_1);
task_print.succeed(task_update_0, task_update_1);
task_print.precede(increment);
increment.precede(back);
back.precede(cond);
executor.run(taskflow).wait();
```

Listing 2.3: Taskflow code for Figure 2.4.

2.4 Evaluation

2.4.1 Evaluation Setup

We evaluate BatchSim's performance on large industrial designs, Gemmini[30], SIGMA[122], RocketChip[6], and BOOM[143], as listed in Table 2.1. These designs

range from deep-learning accelerators and SoC designs to RISC-V cores. The complexity of these designs can be assessed by counting the number of IR nodes and edges in the table. All experiments were conducted on an Ubuntu 22.04 x86_64 machine. The machine was equipped with a 20-core Intel i5-13500 processor running at 4.8 GHz, with 128 GB RAM. We compile all the programs on clang++-17 and llc-17 with optimization flags -O2 enabled.

2.4.2 Baseline

We consider Khronos[146] as our baseline to evaluate the performance of Batch-Sim in terms of inter-cycle batching and task graph parallelism. The simulation's performance with a single thread and a batch size of one serves as the baseline value. We then calculate the relative speedup by varying the thread count from one to eight and the batch size from one to four. Each configuration is run ten times to compute the average performance. The baseline results are depicted under "Batchsize 1" bars in Figure 2.5. Notably, for SIGMA and RocketChip benchmarks, without inter-cycle batching, multithreading performs worse than single-threading because the overhead of multithreading outweighs the advantages of parallelism.

Table 2.1: Evaluated Benchmarks for BatchSim

Benchmark	IR Nodes	IR Edges	Description
Gemmini	78k	135k	Gemmini Matrix Multiplication
SIGMA	17k	29k	Sparse and Irregular GEMM
RocketChip	35k	79k	SoC consisting of Rocket Core
BOOM-Small	118k	214k	1-wide with 32 ROB BOOM Core
BOOM-Medium	170k	315k	2-wide with 64 ROB BOOM Core
BOOM-Large	230k	460k	3-wide with 96 ROB BOOM Core

2.4.3 Performance Comparison

Figure 2.5 compares BatchSim's performance enhancement over the baseline, exploring various thread counts from one to eight and batch sizes from one to four. The results demonstrate significant performance improvements with multithreading. For example, in the SIGMA benchmark, inter-cycle batching increases speedup from $0.57\times$ to $1.36\times$ with six threads, and for the RocketChip benchmark, speedup improves from $0.90\times$ to $1.24\times$ with four threads. This indicates that inter-cycle batching effectively converts multithreading's negative performance impacts into positive gains. Additionally, in the Gemmini and BOOM series (Small, Medium, and Large), using an optimal six threads, the speedup gains increase 11%–98%. These results underscore BatchSim's considerable effectiveness in boosting the efficiency of RTL parallel simulations.

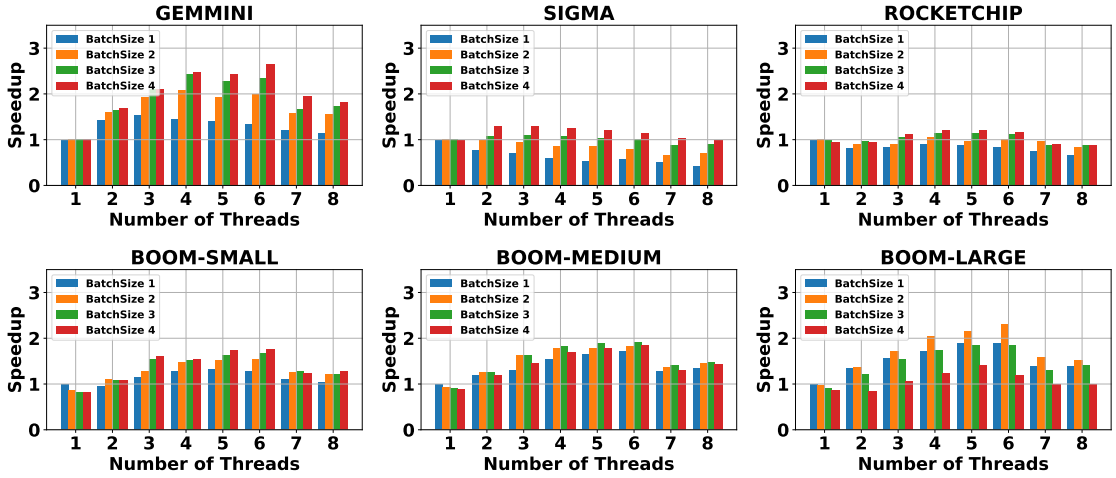


Figure 2.5: Speedup improvement of BatchSim across varying thread counts and batch sizes.

2.5 Conclusion

This chapter introduces BatchSim, a parallel RTL simulator that incorporates inter-cycle batching and task graph parallelism to enhance simulation efficiency. BatchSim enables simultaneous multi-cycle simulation by merging the consecutive RTL graphs and employs task graph parallelism to parallelize the simulation workload. We evaluated the performance of BatchSim on several industrial designs. Compared with state-of-the-art RTL simulators, BatchsSim can achieve a speedup of 11%–98%. To further improve the performance, our future work will focus on optimizing the memory layout to mitigate false sharing. Inspired by the recent success in GPU-accelerated EDA workloads [40, 39, 35, 32, 38, 33, 36, 41, 34, 58, 59, 67, 94, 96], we plan to also leverage the power of GPU to accelerate BatchSim.

In this work, Jie Tong was the primary contributor, leading the majority of the research and development efforts. Liangliang Chang, Umit Y. Ogras, and Tsung-Wei Huang supervised the project, offering guidance and oversight throughout the project. All authors contributed to the preparation and review of the final manuscript.

Chapter 3

ScaleRTL: Scalable Code Generation for RTL Simulation of Deep Learning Accelerators with MLIR

As deep learning accelerators scale in complexity, efficient Register Transfer Level (RTL) simulation becomes crucial for reducing the long runtime of hardware design and verification. However, existing RTL simulators struggle with high compilation overhead and slow simulation performance, particularly for large deep learning accelerator designs, where components are heavily reused and hierarchically structured. This inefficiency arises because existing simulators repeatedly regenerate and recompile redundant code, failing to leverage the structural parallelism inherent in deep learning accelerators. To address this challenge, we propose ScaleRTL, a scalable and unified code generation flow that automatically produces optimized

parallel RTL simulation code for deep learning accelerators. Built on the MLIR infrastructure, ScaleRTL identifies repetitive design patterns, reduces code size and compilation time, and generates efficient simulation executables that exploit both CPU and GPU parallelism. Compared to state-of-the-art RTL simulators, ScaleRTL achieves a compilation speedup of three to five orders of magnitude and up to 15× and 300× simulation speedup on CPU and GPU, respectively.

3.1 Overview

ASIC accelerators play a critical role in boosting the performance of deep learning backbone applications, such as GEMM, DNNs, and transformers in the modern AI industry [30]. To validate the functionality of a hardware design before physical implementation, *Register Transfer Level* (RTL) simulation plays a key role in regression testing, debugging, and design space exploration. However, with the rapidly increasing size and complexity of deep learning accelerators, RTL simulation has become significantly more time-consuming. For instance, recent research has reported that RTL simulation can take several hours to days to achieve coverage closure for validating a deep learning accelerator [97]. Thus, accelerating RTL simulation is critical for managing increasing design complexity and meeting short time-to-market demands in the accelerator market.

To mitigate the runtime challenge of RTL simulation, researchers have proposed various parallel RTL simulation algorithms. For example, Verilator [127], a widely used open-source RTL simulator, transpiles Hardware Description Language (HDL)

into C++ based on RTL abstract syntax trees (ASTs) and uses disjoint-set-based partitioning to enable multithreading. RepCut [134] converts RTL source code to FIRRTL [64] and introduces a replication-aided partitioning algorithm to reduce synchronization overhead in parallel simulation. Khronos [146] and BatchSim [130] parse RTL designs using MLIR and generate evaluation functions through LLVM IR. RTLflow [97], built atop Verilator, transpiles RTL code into CUDA for GPU execution but requires thousands of input stimuli to outperform CPU-based simulation. Despite improved performance, innovations of parallel RTL simulators have evolved largely in *isolation*, and many shareable components have been largely ignored. Consequently, designing new RTL simulation algorithms is extremely time-consuming and error-prone due to numerous software fragmentations, duplicated engineering efforts, and re-innovations of code optimizations.

On the other hand, prior research on parallel RTL simulation has primarily focused on generic RTL designs, such as digital circuits written in SystemVerilog or High-Level-Synthesis (HLS) languages. For a given RTL source, existing simulators flatten the entire design into an *RTL graph* [127], where nodes represent logic elements containing a set of instructions, and edges represent data dependency between nodes. Then, these simulators partition the RTL graph into dependent subgraphs for parallelism and generate evaluation functions. An *evaluation function* simulates the graph for a cycle by consuming inputs and propagating them through the graph. However, these approaches do not exploit structural information. Even when partitions consist of homogeneous logic elements, they still regenerate the same evaluation code for those elements. As shown in Figure 3.1(a) and (b), when

a systolic array contains explicitly duplicated processing elements (PEs), existing RTL simulators continue to regenerate evaluation functions for structurally identical partitions. This results in inefficiencies, as these simulators repeatedly generate and recompile redundant code instead of leveraging the structural parallelism inherent in deep learning accelerators. Prior works such as Verilator [127] and Dedup [135] offer limited support for deduplication in RTL simulation code generation. Verilator [127] focuses on small SystemVerilog statements and does not handle full structural components, while Dedup [135] targets multi-core SoC-style designs that emphasize heterogeneity and connectivity, rather than scalability of deep learning accelerators.

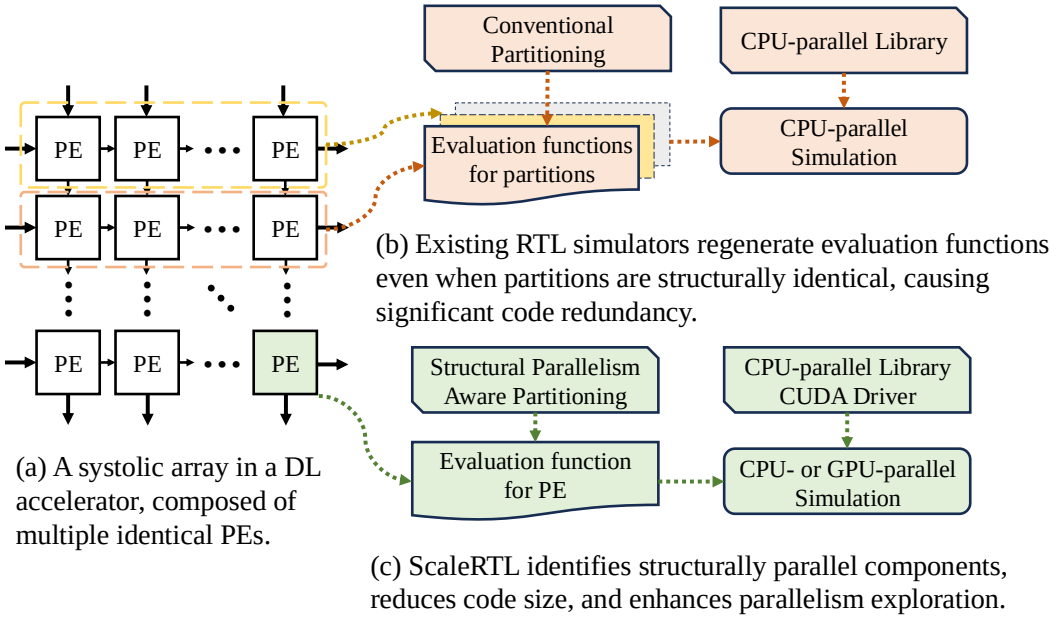


Figure 3.1: Comparison of existing RTL simulation approaches and ScaleRTL.

To tackle these challenges, we introduce *ScaleRTL*, a scalable code generation flow that automatically generates optimized parallel RTL simulators for deep learn-

ing accelerators. Figure 3.1(c) illustrates the ScaleRTL flow. Unlike prior works, ScaleRTL introduces a structural-parallelism-aware partitioning method that identifies structurally parallel components in a deep learning accelerator design and generates evaluation functions for these components. As a result, the generated and compiled evaluation functions can be reused during simulation, avoiding redundant code generation that traditional compilers and simulators fail to eliminate. To unify the code generation flow for both CPU- and GPU-parallel simulation, ScaleRTL builds atop the *multi-level intermediate representation* (MLIR) [85], which supports versatile and customizable dialects and IR transformations. For CPU-parallel simulation, ScaleRTL emits evaluation functions in LLVM IR, compiles them into object files, and links them with a simulation wrapper containing the CPU-parallel library. For GPU-parallel simulation, ScaleRTL emits evaluation functions in PTX format, loads the kernel using the CUDA driver, and executes it using CUDA Graph to reduce repetitive launch overhead. We summarize our technical contributions as follows:

- We introduce a scalable code generation flow that exploits structurally parallel components and eliminates redundant code in deep learning accelerator RTL simulation.
- We develop a unified code generation flow that automatically generates CPU- and GPU-parallel RTL simulators using MLIR, which enables simulation across different architectures.
- We integrate CUDA Graph to reduce kernel launch overhead, further acceler-

ating GPU-parallel RTL simulation.

We evaluate ScaleRTL on a set of deep learning accelerator RTL designs. Compared to state-of-the-art RTL simulators, ScaleRTL achieves a compilation speedup of three to five orders of magnitude and up to 15× and 300× simulation speedup on CPU and GPU, respectively. To the best of our knowledge, ScaleRTL is one of the earliest research efforts to explore the application of MLIR and GPUs in deep learning accelerator RTL simulation. We open-sourced [1] ScaleRTL to support hardware design and EDA-inspired compiler research.

3.2 Background and Motivation

3.2.1 RTL Simulation and Development Challenge

RTL design source code is typically written in hardware description languages (HDLs) like SystemVerilog or Chisel. To enable simulation, these designs are translated into C++ or LLVM IR, wrapped in a simulation framework, and compiled into an executable. Full-cycle simulators, such as Verilator [127], Khronos [146], and BatchSim [130], are widely used to capture cycle-accurate outputs and exploit parallelism. In these simulators, the RTL design is transformed into a directed graph, known as the *RTL graph*, where nodes represent logic elements and edges denote data dependencies. Simulating each cycle corresponds to evaluating this graph, where input values propagate through logic elements to produce outputs.

[1] `https://github.com/TongJieGitHub/ScaleRTL`

This evaluation process is repeated thousands to millions of times to validate the design's functionality [97].

The typical approach to building an RTL simulator involves representing the RTL graph in an intermediate representation, applying optimizations, and generating efficient simulation code. For example, RTLflow [97] leverages an AST-based IR to capture high-level RTL information, partitions the IR into macro tasks, and schedules them across threads for parallel execution. Similar strategies have been adopted by existing simulators [127, 134, 146, 99, 130, 50, 61]. However, innovations in simulation IRs and parallel algorithms have evolved in isolation, leading to software fragmentation, duplicated engineering efforts, and redundant code optimization. This lack of modularity makes developing new RTL simulation algorithms highly time-consuming and error-prone.

3.2.2 MLIR

MLIR [85] is a novel infrastructure designed to simplify the building of new compiler components atop the LLVM project. Specifically, MLIR provides a rich set of composable abstractions, including operations, types, attributes, and regions, that empower developers to represent programs at multiple levels of abstraction. Developers can also define custom dialects and transformation methods to achieve unified code optimizations across diverse sources. To preserve designers' intent and capture high-level information, we build ScaleRTL on top of the popular FIRRTL [64] and CIRCT IRs, which directly models the RTL source. The primary benefit of using MLIR is its capability to offer deeper insights at the IR level compared to the

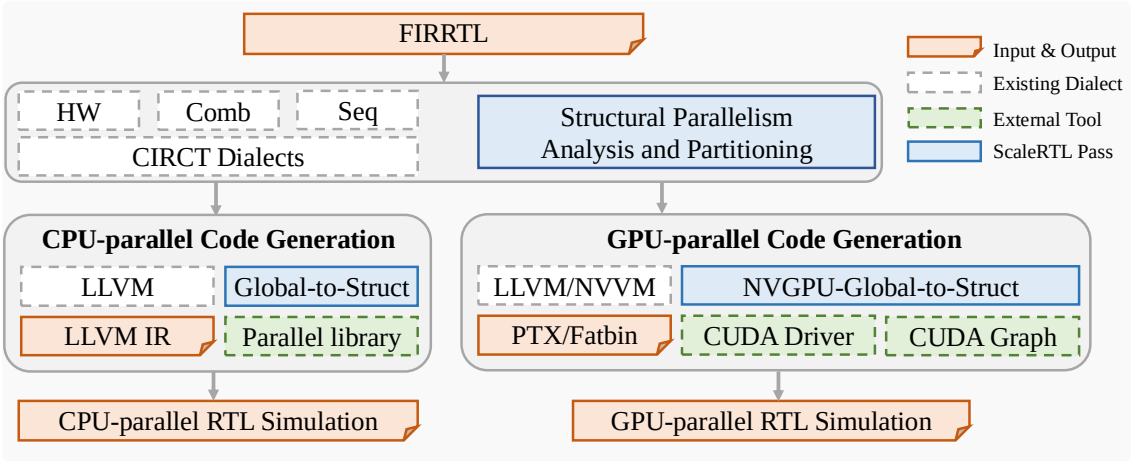


Figure 3.2: Overview of ScaleRTL.

source code, allowing for greater opportunities to exploit data parallelism.

3.3 ScaleRTL

Figure 3.2 illustrates the proposed ScaleRTL framework. At a high level, ScaleRTL compiles RTL source code (FIRRTL) into RTL simulation executables for both CPU and GPU targets. It is built atop MLIR [85] and CIRCT IR, which provide off-the-shelf dialects and compilation passes for general-purpose compilation and hardware modeling. ScaleRTL consists of three main components: Structural parallelism analysis and partitioning, CPU-parallel code generation, and GPU-parallel code generation. Additionally, we integrate CUDA Graph [95] to further enhance the performance of GPU-based simulation.

3.3.1 Structural Parallelism Analysis and Partitioning

The first step in RTL simulation code generation is to use CIRCT tools to convert the FIRRTL source design into CIRCT dialects, such as Comb, Seq, and HW. Listing 3.1 provides an example of a GEMM design written in the HW dialect.

```
module {
  hw.module @GEMM(%arg0: i32, %arg1: i32, ...) -> i32 {
    ...
    %PE.io_data_2_out_bits, ... = hw.instance "PE" @PE(clock: %clock
   : i1, reset: %reset: i1, ...) -> (io_data_2_out_bits: i16, ...)
    %PE_1.io_data_2_out_bits, ... = hw.instance "PE_1" @PE(clock: %
   clock: i1, reset: %reset: i1, ...) -> (io_data_2_out_bits: i16,
   ...)
    ...
  }
}
```

Listing 3.1: Example RTL Design in HW Dialect.

Unlike generic RTL designs, GEMM exhibits a highly homogeneous layout, where most components, such as PEs and interconnects, are repetitively instantiated. Additionally, from a hardware perspective, these subsequent lines of code are semantically parallel. Thus, we can leverage structural parallelism in deep learning accelerator designs to construct a highly parallel simulator. A key step in this process is to analyze the code, identify and count repetitive components, and extract and partition them from the original top-level design. To achieve this, we design a pass in MLIR that performs these analyses. This pass examines hardware module

hierarchies in MLIR by computing direct and flattened instance counts within a `hw::InstanceGraph`. It identifies the top-level module using a heuristic, computes direct instance counts, and recursively derives flattened counts–estimating the occurrence of each module in a fully flattened design. The pass then returns these counts as a mapping. With this analysis, we can partition the original design into multiple instances and extract repetitive instances as separate modules.

3.3.2 CPU-parallel Simulation Code Generation

After analyzing repetitive components and decomposing the deep learning RTL design into separate modules, we apply a set of IR transformations. This process converts the design from the HW dialect to the LLVM dialect, enabling efficient simulation of each module. An example of this MLIR-based transformation is shown in Listing 3.2.

```
module attributes {llvm.data_layout = ""} {
  ...
  llvm.mlir.global internal @shiftreg()  : i1
  llvm.mlir.global linkonce_odr @clock() : i1
  llvm.mlir.global linkonce_odr @reset() : i1
  ...
  llvm.func @PE() {
    ...
    %25 = llvm.mlir.addressof @shiftreg : !llvm.ptr<i1>
    %25 = llvm.mlir.addressof @reset : !llvm.ptr<i1>
    %26 = llvm.load %25 : !llvm.ptr<i1>
    ...
```

```
    llvm.store %7121, %10412 : !llvm.ptr<i16>
    llvm.return
  }
}
```

Listing 3.2: Example RTL evaluation code in LLVM Dialect.

In the LLVM dialect, signals and internal states are allocated as global variables in the data segment. When lowered to LLVM IR and further to an object file, the evaluation function `@PE` is bound to these global variables. For deep learning accelerators with thousands of PEs, this approach leads to compiling identical code thousands of times, resulting in a large executable with severe code redundancy. To address this, we propose a new simulation paradigm that decouples data from the evaluation function. Instead of binding to global variables, we define a struct that holds all signals and states in a header file and pass a pointer to this struct as an argument to the evaluation function. We refer to this as the `Global-to-Struct` pass.

Listing 3.3 provides an example where the evaluation function takes a struct pointer as an argument, with the struct defined in a header file. To correctly determine memory locations within the struct, we record the byte offsets of all data during the code generation phase. This ensures that the evaluation function can accurately access the converted addresses without error. By separating data from the function, we compile the evaluation function only once, while allocating multiple instances of the struct at runtime. This allows multiple instances of the function to be launched concurrently, reducing data hazards and synchronization

overhead. With the function and header file prepared, we use a CPU-parallel library (OpenMP) to perform parallel simulation for each cycle.

```
// LLVM Dialect
module attributes {llvm.data_layout = ""} {
  llvm.func @PE(%arg0: !llvm.ptr<i8>) {
    %0 = llvm.mlir.constant(0 : i64) : i64
    %1 = llvm.getelementptr %arg0[%0] : (!llvm.ptr<i8>, i64) -> !
   llvm.ptr<i8>
    %2 = llvm.bitcast %1 : !llvm.ptr<i8> to !llvm.ptr<i16>
    ...
    llvm.return
  }
}
// C++ header file
typedef struct EvalContext {
  // Field 0 - Original global: @mem_ext - Byte offset: 0
  char mem_ext[8];
  ...
} EvalContext;
void PE(EvalContext* ctx);
```

Listing 3.3: Example RTL evaluation code in LLVM dialect with a struct pointer as an argument, and the corresponding struct defined in a C++ header file.

3.3.3 GPU-parallel Simulation Code Generation

Figure 3.3 illustrates the GPU code generation process in ScaleRTL. Unlike prior work [132], which uses the GPU dialect to generate GPU-based simulation code,

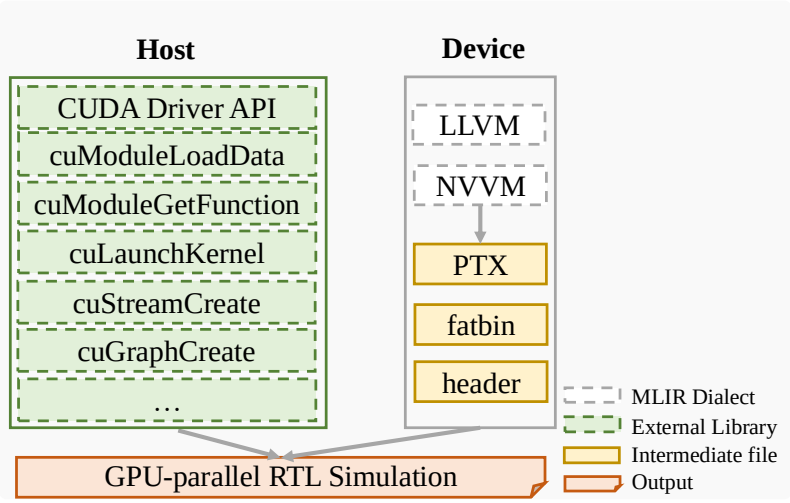


Figure 3.3: GPU code generation flow in ScaleRTL.

we found that relying solely on the provided GPU dialect limits control over kernel management and optimization from the host side. To address this challenge, we design a host-side CUDA code generator that automatically invokes CUDA driver APIs to load modules, manage memory, and launch kernels. On the device side, similar to CPU-parallel code generation, we generate the evaluation function in the LLVM dialect. Since GPU supports launching thousands of threads that execute the same kernel function in a SIMT fashion, we first allocate a chunk of device memory for structs. For each thread, it is essential to compute the correct address and offset to locate the corresponding struct that the thread will evaluate. To achieve this, we precompute and map each data address during code generation by calculating the base address of the struct and the offset of a given data field. Listing 3.4 shows an example evaluation kernel using the NVVM dialect, where thread and block IDs are retrieved and used to compute global memory addresses. Once the LLVM and NVVM dialects are generated, we use the LLVM static compiler `llc` to lower the

code to PTX. To reduce the overhead of just-in-time (JIT) compilation, where PTX is offloaded to the GPU and compiled to SASS for the first execution, we use the PTX assembler `ptxas` to compile PTX into architecture-specific binaries and package them as a fatbin. This approach improves GPU performance while maintaining compatibility across different GPU architectures.

```
module attributes {llvm.data_layout = ""} {
  llvm.func @PE(%arg0: !llvm.ptr<i8>) {
      %0 = nvvm.read.ptx.sreg.tid.x : i32
      %1 = nvvm.read.ptx.sreg.ctaid.x : i32
      %2 = nvvm.read.ptx.sreg.ntid.x : i32
      ...
      %10 = llvm.getelementptr %arg0[%9] : (!llvm.ptr<i8>, i64) -> !
   llvm.ptr<i8>
      ...
      llvm.return
  }
}
```

Listing 3.4: Example GPU-based RTL evaluation code in LLVM and NVVM Dielact.

RTL simulation typically runs for thousands of cycles. If we use stream-based execution, repetitive kernel launches will accumulate significant overhead. To mitigate this issue, we leverage CUDA Graph [95] to merge successive kernel calls into a single simulation task graph to reduce kernel launch overhead and improve GPU-based simulation performance.

3.4 Experimental Results

We evaluate the performance of ScaleRTL on four deep learning accelerator RTL designs: Conv2D [65], GEMM [65], Gemmini [30], and SIGMA [122]. Experiments are conducted on a 64-bit Linux machine with an Intel i5-13500 CPU and an NVIDIA RTX A4000 GPU. CPU code generation utilizes LLVM 17's `clang` and `llc` compilers, while GPU code generation employs CUDA Toolkit 12.6, targeting compute capability 8.6. All code is compiled with the `-O2` optimization flag. In the following sections, we refer to ScaleRTL with CPU code generation as $ScaleRTL_C$ and ScaleRTL with GPU code generation as $ScaleRTL_G$. We consider Verilator [127], Khronos [146], and BatchSim [130] as baseline CPU-based simulators. Verilator and BatchSim are configured with 4 threads, while Khronos runs in single-threaded mode as it doesn't support parallelism. All simulations use a single input stimulus; therefore, we do not include the GPU-based RTL simulator RTLflow [97], as it is designed for batch-stimulus scenarios, which is a different scope of work. We also exclude ESSENT [8] and its successors [134, 135], as they encounter out-of-memory errors during code generation. To ensure consistency, all simulation results are averaged over five runs.

3.4.1 Code Generation and Compilation Results

Table 3.1 presents the end-to-end compilation time and generated executable size for Conv2D, GEMM, Gemmini, and SIGMA across different RTL simulators. The end-to-end compilation time includes the transformation from RTL source code to

Table 3.1: Comparison of compilation time (T) and generated executable size for Conv2D, GEMM, Gemmini, and SIGMA among different RTL simulators.

Design	#PEs	Verilator		Khronos		BatshSim		ScaleRTL_C		ScaleRTL_G	
		T(s)	**Size(MB)**	**T(s)**	**Size(MB)**	**T(s)**	**Size(MB)**	**T(s)**	**Size(MB)**	**T(s)**	**Size(MB)**
Conv2D	2^7	9	0.4	6	0.2	3	0.6	2	0.08	4	1.1
	2^9	13	1.1	103	0.8	20	2.4	2	0.08	4	1.1
	2^{11}	39	3.8	2633	3.5	302	9.6	2	0.08	4	1.1
	2^{13}	163	15	41428	14	7796	39	2	0.08	4	1.1
GEMM	2^7	25	0.3	2	0.1	3	0.4	1	0.05	4	1.1
	2^9	48	0.9	27	0.5	11	2	1	0.05	4	1.1
	2^{11}	224	3.1	1135	2.3	139	7.7	1	0.05	4	1.1
	2^{13}	2053	12	72477	9	2751	31	1	0.05	4	1.1
Gemmini	2^5	83	2.3	118	0.7	38	0.8	25	0.3	4	1.2
	2^7	380	8.5	1897	3	592	3.3	33	0.3	5	1.2
	2^9	1621	34	26183	12	9439	13	33	0.3	5	1.2
	2^{11}	17893	132	357498	47	92673	52	32	0.3	5	1.2
SIGMA	2^5	41	0.7	7	0.2	9	0.3	3	0.1	8	1.4
	2^7	94	2.3	99	0.9	59	1.3	3	0.1	10	1.4
	2^9	443	8.7	1552	3.9	1053	5	3	0.1	10	1.4
	2^{11}	4920	35	22248	16	10969	20	4	0.1	11	1.4

simulation code (C++ or LLVM IR) and the subsequent compilation and linking process to generate the final binary. For baseline simulators (Verilator, Khronos, and BatchSim), their inability to detect repetitive components leads to significant redundant code generation and compilation overhead. As the number of PEs increases, both compilation time and executable size grow proportionally. Even worse, compiling designs with thousands of PEs can take several hours to days, which could significantly hamper the turnaround time of hardware designs.

In contrast, ScaleRTL_C and ScaleRTL_G complete compilation in just a few seconds, achieving up to 70,000× compilation speedup compared to the baselines. This improvement comes from ScaleRTL's ability to detect repetitive components in deep learning accelerator designs, generating evaluation functions only for PEs and other critical units, and invoking them with the corresponding data structures at

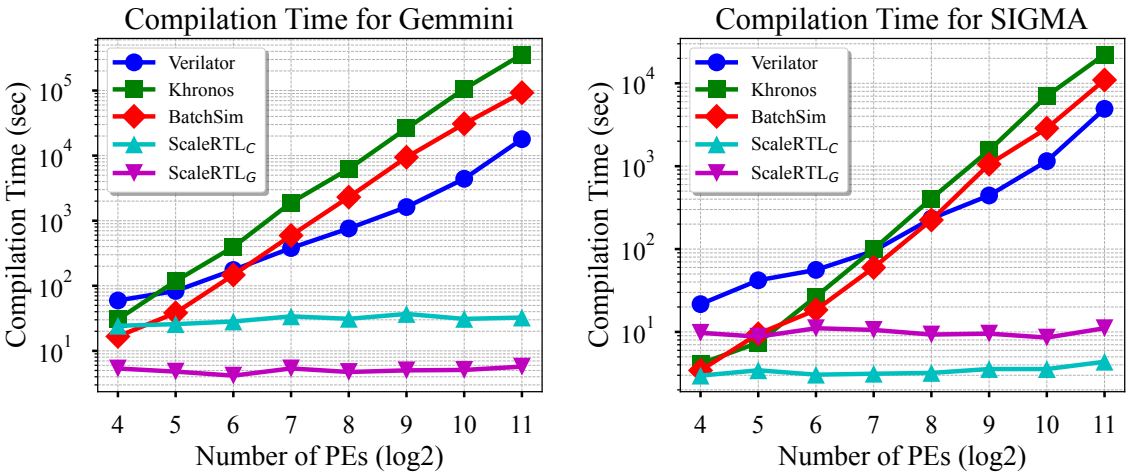


Figure 3.4: Compilation time of Gemmini and SIGMA accelerators among different RTL simulators as the number of PEs increases exponentially.

runtime.

To demonstrate the scalability of ScaleRTL's compilation time, Figure 3.4 shows the compilation time of Gemmini and SIGMA on different RTL simulators as the number of PEs increases. The results clearly indicate that ScaleRTL achieves sublinear overhead growth, even as the design size increases exponentially. This trend highlights ScaleRTL's efficiency and scalability in handling deep learning accelerator RTL simulations, even for large-scale designs.

3.4.2 Overall Simulation Performance Comparison

Figure 3.5 shows the simulation speedup of Conv2D, GEMM, Gemmini, and SIGMA on different RTL simulators, over the baseline Verilator. For small-scale designs, ScaleRTL$_C$ and ScaleRTL$_G$ do not outperform other simulators, as the baseline simulators can fit the RTL design within the cache and apply optimizations for

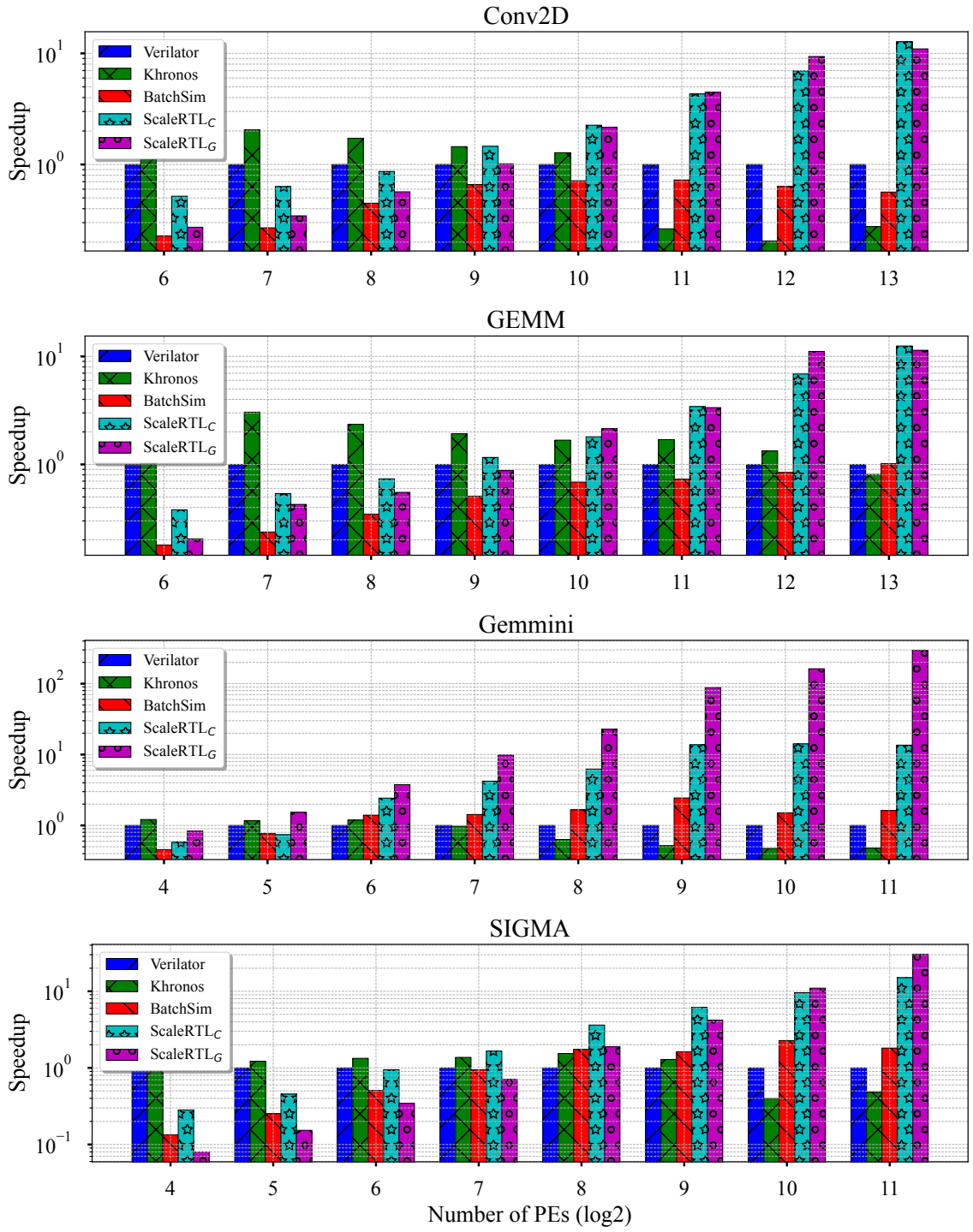


Figure 3.5: Overall simulation speedup of Conv2D, GEMM, Gemmini, and SIGMA on different RTL simulators as the number of PEs increases exponentially. Speedup is measured relative to the baseline Verilator.

higher efficiency. However, for mid-scale to large-scale designs, ScaleRTL$_C$ and ScaleRTL$_G$ exhibit increasing speedup as the design size grows. This is because ScaleRTL$_C$ evaluates components by passing pointers to structs, improving data locality and reducing synchronization overhead. Additionally, ScaleRTL$_G$ employs a block of threads to evaluate identical components, which exploits highly parallel SIMT execution on the GPU. Consequently, ScaleRTL$_C$ achieves a 12×–15× speedup, and ScaleRTL$_G$ achieves an 11×–300× speedup at the largest design sizes.

3.4.3 CPU and GPU Simulation Runtime Analysis

Figure 3.6 shows the simulation time of Gemmini and SIGMA on different RTL simulators as the number of PEs increases exponentially. All CPU-based simulators, including ScaleRTL$_C$, exhibit linear or superlinear simulation growth because CPU threads are limited, and the total executed instructions scale proportionally with the design size. In contrast, ScaleRTL$_G$ exhibits sublinear growth. For instance, the simulation time for Gemmini remains around 0.1 seconds, even as the size increases from 2^4 to 2^{11}. This is because GPU consists of multiple streaming multiprocessors (SMs), each capable of managing thousands of threads. As a result, GPU-based simulation benefits from latency hiding through context switching and achieves higher concurrency. This underscores ScaleRTL's efficiency and scalability in deep learning accelerator RTL simulation, especially for large-scale designs.

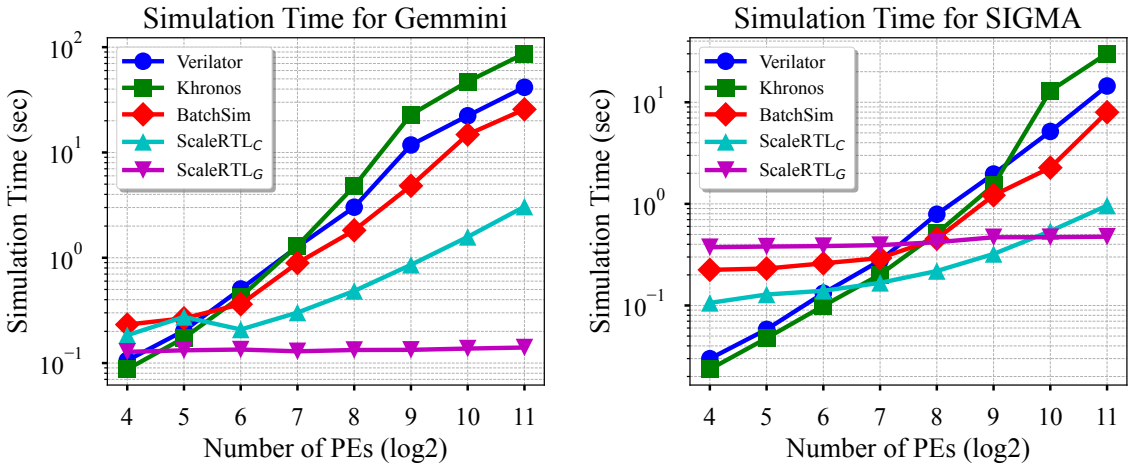
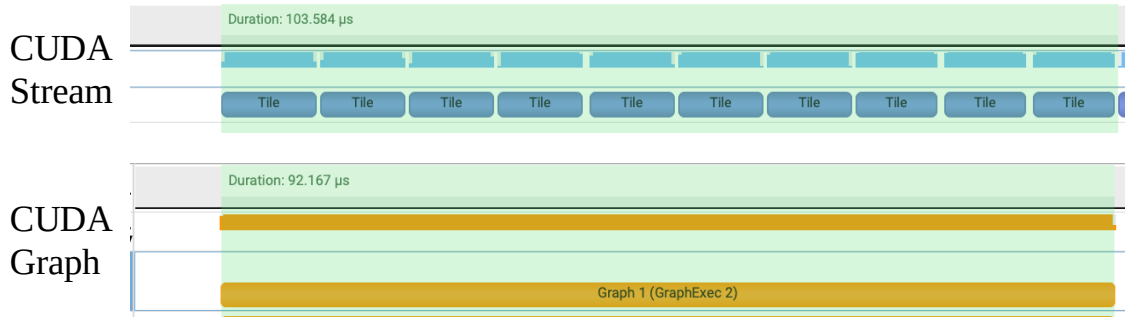


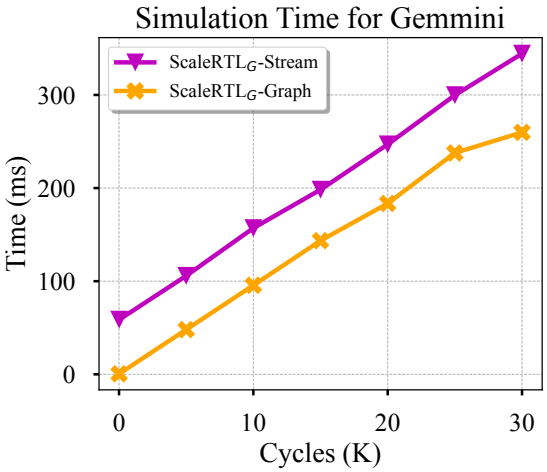
Figure 3.6: Simulation time of Gemmini and SIGMA accelerators on different RTL simulators as the number of PEs increases exponentially.

3.4.4 Performance Result of CUDA Graph

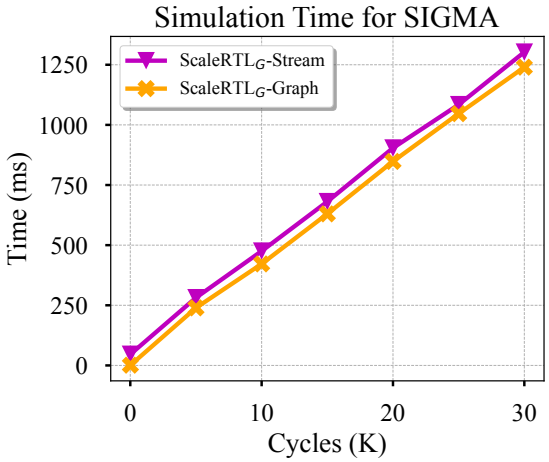
Figure 3.7 compares the performance of CUDA Stream-based and CUDA Graph-based execution. Since GPU-based simulation involves consecutive kernel calls, connecting these kernels into a graph is crucial to reducing kernel launch overhead. Figure 3.7b and 3.7c illustrate simulation time over increasing cycles for both GPU-based approaches on the large Gemmini and SIGMA designs. CUDA Graph-based simulation consistently outperforms stream-based simulation across all evaluated scenarios. For instance, in the Gemmini design, CUDA Graph-based simulation reduces execution time by a consistent 60 milliseconds compared to stream-based execution.



(a) Profiling results: The CUDA graph consists of 10 kernels and incurs less overhead compared to stream-based execution.



(b) Gemmini (2^{11} PEs)



(c) SIGMA (2^{11} PEs)

Figure 3.7: Simulation results comparing CUDA Stream-based and CUDA Graph-based execution.

3.5 Conclusion

This chapter presents *ScaleRTL*, a scalable and unified code generation flow that automatically produces optimized parallel RTL simulations for deep learning accelerators. Built atop the MLIR infrastructure, ScaleRTL identifies repetitive design patterns, reduces code size, accelerates compilation, and generates efficient parallel

simulation executables for both CPU and GPU targets. Compared to state-of-the-art RTL simulators, ScaleRTL achieves a compilation speedup of three to five orders of magnitude and up to 15× and 300× simulation speedup on CPU and GPU, respectively.

In this work, Jie Tong was the primary contributor, leading the majority of the research and development efforts. Wan-Luan Lee, Umit Y. Ogras, and Tsung-Wei Huang supervised the project, offering guidance and oversight throughout the project. All authors contributed to the preparation and review of the final manuscript.

Chapter 4

HeteroRTL: A Scalable Code Generation Flow for Heterogeneous Parallel RTL Simulation using MLIR

As hardware design complexity increases, efficient Register Transfer Level (RTL) simulation becomes critical for reducing the long runtime of design and verification. Although several parallel RTL simulators have been developed, they often suffer from long compilation times and slow simulation performance, especially for large-scale heterogeneous architectures and deep learning SoC designs that exhibit repetitive and hierarchical structures. These limitations arise because existing simulators fail to effectively map heterogeneous architectures onto CPU-GPU platforms, resulting in underutilized compute resources. In addition, they repeatedly regenerate and recompile redundant code, missing the opportunity to exploit the structural

parallelism inherent in deep learning accelerators. To address these challenges, we propose *HeteroRTL*, a scalable code generation flow that produces hybrid CPU-GPU parallel RTL simulators for heterogeneous deep learning accelerator SoCs. Built on the MLIR infrastructure, HeteroRTL analyzes RTL designs, partitions the simulation between CPU and GPU targets, identifies structural repetition to reduce compilation overhead, and generates efficient simulation executables. Compared to state-of-the-art simulators, HeteroRTL achieves compilation speedups of three to five orders of magnitude and delivers up to 9× and 122× simulation speedups across various designs.

4.1 Overview

Domain-specific accelerators are essential for enhancing the performance of deep learning workloads, including DNNs and transformer models, in today's AI-driven industry [30, 70]. Register Transfer Level (RTL) simulation is a critical step in hardware design and verification, used to validate functionality prior to physical implementation through tasks such as regression testing, debugging, and design space exploration. As accelerators evolve, their design complexity continues to grow. For instance, the systolic array size in Google's TPU has increased from 128×128 to 256×256 in the latest TPU v6e [1]. Consequently, RTL simulation has become increasingly time-consuming. Recent studies report that simulation can take several hours to days to achieve coverage closure when validating deep learning accelerators [97]. Therefore, accelerating RTL simulation is essential for

managing growing design complexity and meeting the fast-paced time-to-market requirements of the accelerator industry.

To overcome the prohibitive runtimes of RTL simulation, researchers have introduced various parallel simulation techniques. One prominent example is Verilator [127], a widely adopted open-source RTL simulator that transpiles hardware description languages (HDLs) into C++ using abstract syntax trees (ASTs), and employs disjoint-set-based partitioning to enable multi-threaded execution. RTLflow [97], built on top of Verilator, targets GPU acceleration by translating RTL code into CUDA, but requires thousands of input stimuli to outperform CPU-based simulators. RepCut [134] converts RTL designs into FIRRTL [64] and introduces a replication-aided partitioning algorithm to reduce synchronization overhead during parallel simulation. Khronos [146] and BatchSim [130] utilize the MLIR framework to analyze RTL designs and generate evaluation functions in LLVM IR. Dedup [135] introduces deduplication techniques in RTL simulation code generation, targeting the structural patterns of multi-core SoC designs. While these approaches improve performance, they have largely evolved independently, resulting in fragmented toolchains and missed opportunities for shared infrastructure. As a result, developing new RTL simulation algorithms remains time-consuming and error-prone, often involving redundant engineering efforts and reimplementation of common optimization techniques.

However, prior research on parallel RTL simulation has primarily focused on generic RTL designs, without addressing the unique characteristics of large-scale heterogeneous architectures and deep learning SoCs. These approaches suffer from

two major limitations. First, they do not effectively map heterogeneous architectures onto CPU-GPU simulation platforms, resulting in underutilized compute resources. As illustrated in Figure 4.1, a deep learning SoC typically features a heterogeneous architecture composed of multicore host CPUs and a systolic array of duplicated processing elements (PEs). Running such a simulation on CPUs alone fails to exploit the fine-grained parallelism well-suited to GPUs. Conversely, executing the entire simulation on GPUs underutilizes the hardware for complex CPU cores, which are fewer in number than GPU warp sizes, and may overconsume registers and memory, leading to suboptimal GPU performance. Second, existing simulators do not take advantage of structural redundancy. Even when designs contain homogeneous logic elements, they generate separate evaluation code for each instance. This results in substantial inefficiencies, as the same code is repeatedly compiled instead of

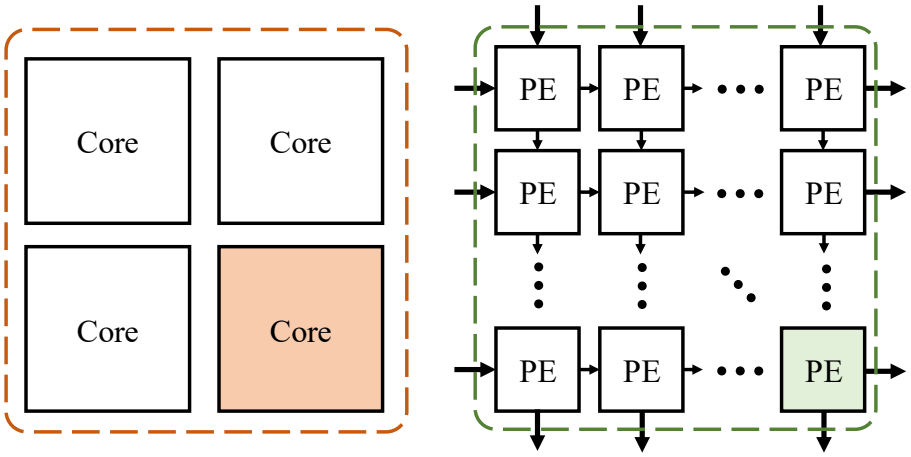


Figure 4.1: Schematic of a deep learning accelerator SoC composed of multicore host CPUs and a systolic array of duplicated PEs. The heterogeneous architecture enables CPU-GPU hybrid simulation, while the duplicated components offer opportunities for simulation code reuse and reduction.

being reused, failing to leverage the structural parallelism present in deep learning accelerators.

To address these challenges, we propose *HeteroRTL*, a scalable code generation flow that produces hybrid CPU-GPU parallel RTL simulators targeting heterogeneous deep learning accelerator SoCs. Unlike prior works, HeteroRTL introduces an architecture-aware partitioning method that identifies heterogeneous components and structurally parallel modules in the RTL design. It partitions the system into two parts: complex host cores are simulated on the CPU, while the systolic array is offloaded to the GPU to exploit massive parallelism. This partitioning improves load balancing and maximizes compute resource utilization. In addition, HeteroRTL detects structural repetition to significantly reduce compilation overhead. By reusing generated and compiled evaluation functions during simulation, it avoids the redundant code generation commonly found in traditional compilers and simulators. To support a unified code generation flow for hybrid CPU and GPU simulation, HeteroRTL is built on top of the multi-level intermediate representation (MLIR) framework [85], which provides flexible dialects and transformation capabilities. HeteroRTL emits evaluation functions in LLVM IR, then lowers them to native binary code for CPU execution and PTX code for GPU execution. It also generates simulation wrappers to invoke and coordinate hybrid simulation tasks across CPU and GPU platforms. We summarize our technical contributions as follows:

- We propose a code generation flow that produces hybrid CPU-GPU parallel RTL simulators for heterogeneous deep learning accelerator SoCs.

- We develop an architecture-aware partitioning method that separates heterogeneous components and structurally parallel modules for efficient CPU and GPU execution.
- We design a scalable code generation approach that detects structural repetition and eliminates redundant code, significantly reducing compilation overhead.

We evaluate HeteroRTL on a set of deep learning accelerator SoC RTL designs. Compared to state-of-the-art simulators, HeteroRTL achieves compilation speedups of three to five orders of magnitude and delivers up to 9× and 122× simulation speedups across various designs.

4.2 Background and Motivation

4.2.1 RTL Simulation

RTL designs are typically described using hardware description languages (HDLs) such as SystemVerilog or Chisel. For simulation, these designs are translated into intermediate representations like C++ or LLVM IR, integrated into a simulation framework, and compiled into executable binaries. To achieve cycle-accurate simulation and parallel execution, full-cycle simulators such as Verilator [127], Khronos [146], and BatchSim [130] are commonly used. These tools represent RTL designs as directed graphs, referred to as *RTL graphs*, where nodes correspond to logic elements and edges capture data dependencies. Each simulation cycle involves

evaluating the RTL graph by propagating input values through logic elements to compute outputs. This process is repeated thousands to millions of times to ensure functional correctness [97, 146].

While these simulators effectively capture functional behavior, they often suffer from significant code redundancy due to a lack of structural awareness. Verilator [127] and Dedup [135] offer only limited support for deduplication in RTL simulation code generation. Verilator operates at the level of low-level SystemVerilog statements and does not recognize or optimize larger structural patterns. Dedup focuses on multi-core SoC-style designs, emphasizing heterogeneity and connectivity, but does not address the scalability requirements of deep learning accelerators with highly repetitive architectures.

4.2.2 MLIR

MLIR [85] is a modern compiler infrastructure developed to streamline the creation of new compiler components within the LLVM ecosystem [84]. It offers a rich set of composable abstractions, such as operations, types, attributes, and regions, that enable the representation of programs at multiple levels of abstraction. MLIR also allows developers to define custom dialects and transformation passes, facilitating unified optimization workflows across diverse input languages and target platforms. To preserve the original design intent and retain high-level structural information, we build HeteroRTL on top of FIRRTL [64] and CIRCT [2], intermediate representations specifically designed to model RTL semantics directly.

4.3 HeteroRTL

Figure 4.2 presents an overview of the proposed HeteroRTL framework. At a high level, HeteroRTL compiles RTL source code written in FIRRTL into simulation executables targeting both CPU and GPU platforms. The framework is built on top of MLIR [85] and CIRCT [2], which provide reusable dialects and compilation passes for general-purpose optimization and hardware modeling. HeteroRTL consists of four key components: structural repetition analysis and architecture-aware partitioning, CPU-parallel code generation, GPU-parallel code generation, and CPU-GPU hybrid simulation generation.

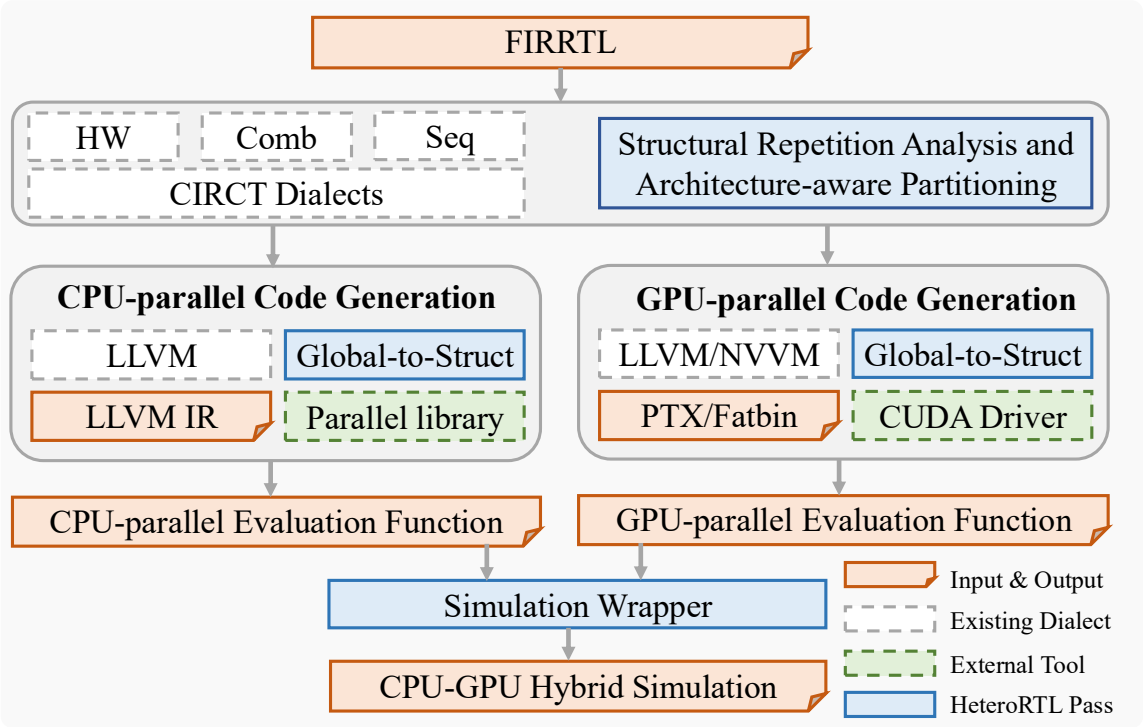


Figure 4.2: Overview of HeteroRTL.

4.3.1 Structural Analysis and Architecture-aware Partitioning

The RTL simulation code generation process begins by using CIRCT tools to lower the FIRRTL source design into CIRCT dialects, such as hw, seq, and comb. Listing 4.1 shows an example of a deep learning accelerator SoC represented in the hw dialect.

```
module {
  hw.module @DL_SoC(%arg0: i32, ...) -> i32 {
    ...
    %Core_0.io_data_, ... = hw.instance "Core_0" @Core(clock: %clock
   : i1, ...) -> (io_data_: i32, ...)
    %Core_1.io_data_, ... = hw.instance "Core_1" @Core(clock: %clock
   : i1, ...) -> (io_data_: i32, ...)
    ...
    %PE_0.io_data_, ... = hw.instance "PE_0" @PE(clock: %clock: i1,
   ...) -> (io_data_: i16, ...)
    %PE_1.io_data_, ... = hw.instance "PE_1" @PE(clock: %clock: i1,
   ...) -> (io_data_: i16, ...)
    ...
  }
}
```

Listing 4.1: Example Deep Learning SoC Design in HW Dialect.

Unlike generic RTL designs, deep learning accelerator SoCs exhibit a heterogeneous architecture consisting of a cluster of host CPU cores and a systolic array composed of replicated processing elements (PEs). To exploit this structure, we introduce a method that analyzes the architectural heterogeneity and partitions the design for subsequent CPU and GPU code generation. Within each partition,

the layout is highly homogeneous, as cores and PEs are often instantiated repetitively. From a hardware perspective, these components operate in parallel and can therefore be simulated concurrently. To construct a highly parallel simulator, we leverage this structural parallelism by analyzing the design, identifying repetitive components, and extracting them from the top-level module. We implement this analysis as a custom MLIR pass that inspects the hardware module hierarchy. The pass identifies the top-level module using a method that computes both direct and flattened instance counts, and returns a mapping of each module to its total number of instantiations in a fully flattened design. This enables us to isolate and extract frequently repeated modules, which can then be simulated efficiently as parallel instances.

4.3.2 CPU-parallel Simulation Code Generation

Following the analysis of architectural heterogeneity and repetitive structures, we decompose the deep learning accelerator RTL design into distinct modules and apply a series of intermediate representation (IR) transformations. For CPU-parallel code generation, we target the host CPU cores, which are typically large and complex. Due to their instruction-heavy behavior, these modules are well-suited for CPU-based simulation. Using MLIR, we lower these components from the hw dialect to the LLVM dialect, enabling efficient parallel simulation on the CPU. Listing 4.2 illustrates this transformation flow.

```
module attributes {llvm.data_layout = ""} {
  ...
```

```
  llvm.mlir.global linkonce_odr @clock() : i1
  llvm.mlir.global linkonce_odr @reset() : i1
  ...
  llvm.func @Core() {
    ...
    %25 = llvm.mlir.addressof @reset : !llvm.ptr<i1>
    %26 = llvm.load %25 : !llvm.ptr<i1>
    ...
    llvm.store %30, %31 : !llvm.ptr<i16>
    llvm.return
  }
}
```

Listing 4.2: Example core evaluation code in LLVM Dialect.

In the LLVM dialect, internal states are commonly allocated as global variables in the data segment. When lowered to LLVM IR and compiled into an object file, each evaluation function, such as `@Core`, is statically linked to these globals. In a deep learning accelerator SoC with tens of cores and thousands of PEs, this leads to redundant compilation of identical logic for each instance, resulting in excessive code duplication and inflated binary size. To address this inefficiency, we introduce a simulation model that separates data from computation. Instead of binding evaluation functions to global variables, we encapsulate all state variables within a struct and pass a pointer to this struct as an argument. This transformation, known as the `Global-to-Struct` pass, promotes function reuse across instances and significantly reduces both compilation time and executable size.

Listing 4.3 illustrates an evaluation function that takes a pointer to a struct as its argument, with the struct itself defined in a header file. During code generation, we record the byte offsets of all variables within the struct to ensure correct memory access. This guarantees that the evaluation function can compute the correct addresses and access the corresponding data reliably. By decoupling the function from its internal state, we compile the evaluation logic once and allocate multiple struct instances at runtime. This design enables concurrent invocation of the same function on different data, reducing data hazards and minimizing synchronization overhead. With both the evaluation function and the struct definition in place, we leverage OpenMP to execute cycle-level parallel simulation efficiently across CPU threads.

```
// LLVM Dialect
module attributes {llvm.data_layout = ""} {
  llvm.func @Core(%arg0: !llvm.ptr<i8>) {
    %0 = llvm.mlir.constant(0 : i64) : i64
    %1 = llvm.getelementptr %arg0[%0] : (!llvm.ptr<i8>, i64) -> !
   llvm.ptr<i8>
    %2 = llvm.bitcast %1 : !llvm.ptr<i8> to !llvm.ptr<i16>
    ...
    llvm.return
  }
}
// C++ header file
typedef struct EvalContext {
  // Field 0 - Original global: @data - Byte offset: 0
  char data[8];
```

```
  ...
} EvalContext;
void Core(EvalContext* ctx);
```

Listing 4.3: Example core evaluation code in LLVM dialect with a struct pointer as an argument, and the corresponding struct defined in a C++ header file.

4.3.3 GPU-parallel Simulation Code Generation

Building on the analysis of architectural heterogeneity and structural repetition, we partition the deep learning accelerator SoC RTL design into separate modules and apply a series of intermediate representation (IR) transformations. For GPU-parallel code generation, we target the processing elements (PEs) in systolic arrays, which are typically simple and compute-light. Due to their data-parallel nature and regular structure, these modules are well suited for GPU-based simulation. Unlike prior work [132] that leverages the GPU dialect for simulation code generation, we found that relying solely on the GPU dialect limits flexibility in kernel control and host-side optimization. To overcome this limitation, we design a custom host-side CUDA code generator that programmatically invokes CUDA driver APIs to load modules, manage device memory, and launch kernels. On the device side, similar to CPU-parallel code generation, we emit evaluation functions in the LLVM dialect.

Given the GPU's ability to launch thousands of threads executing the same kernel in a SIMT model, we first allocate a contiguous block of device memory to store struct instances. Each thread must compute the correct address of its assigned struct, which requires calculating both the base address and the byte

offset of each field. These offsets are precomputed during code generation to ensure correct memory access at runtime. Listing 4.4 provides an example of a GPU evaluation kernel written in the NVVM dialect, where thread and block IDs are used to compute global memory addresses. Once the LLVM and NVVM dialects are generated, we invoke the LLVM static compiler `llc` to lower the code to PTX. To avoid the overhead of just-in-time (JIT) compilation, where the GPU compiles PTX to SASS upon first execution, we use the PTX assembler `ptxas` to compile the PTX into architecture-specific SASS binaries. These are packaged as fatbins, which improve performance and maintain compatibility across different GPU architectures.

```
module attributes {llvm.data_layout = ""} {
  llvm.func @PE(%arg0: !llvm.ptr<i8>) {
      %0 = nvvm.read.ptx.sreg.tid.x : i32
      %1 = nvvm.read.ptx.sreg.ctaid.x : i32
      %2 = nvvm.read.ptx.sreg.ntid.x : i32
      ...
      %11 = llvm.getelementptr %arg0[%10] : (!llvm.ptr<i8>, i64) ->
   !llvm.ptr<i8>
      ...
      llvm.return
  }
}
```

Listing 4.4: Example GPU-based PE evaluation code in LLVM and NVVM Dielact.

4.3.4 CPU-GPU Hybrid Simulation Generation

After generating simulation functions for both CPU-parallel and GPU-parallel modules, we construct a unified simulation wrapper to enable hybrid execution across CPU and GPU platforms. This wrapper coordinates the simulation of heterogeneous components, executing host cores on the CPU and processing elements (PEs) on the GPU, in a single simulation cycle. To achieve this, we assign two host threads: one responsible for launching the CPU-side simulation function and the other for invoking the GPU kernel through the CUDA driver API. These threads are folded into a lightweight runtime framework that synchronizes execution using a barrier at each end of the simulation cycle to ensure correctness and data consistency. Since our simulation scenario involves no shared inputs or outputs to simplify our simulation model, no explicit data transfer between host and device memory is required.

During each simulation cycle, the CPU thread invokes the evaluation functions for complex cores using OpenMP, while the GPU thread asynchronously launches the evaluation kernel to simulate thousands of parallel PEs. This hybrid execution model leverages the strengths of both CPU and GPU: the CPU efficiently handles control-heavy, instruction-rich host cores, while the GPU executes lightweight, massively parallel PEs with high throughput. By balancing workloads and minimizing idle compute resources, the hybrid simulation framework improves scalability and simulation efficiency for heterogeneous deep learning accelerator SoCs.

4.4 Experimental Evaluation

We evaluate the performance of HeteroRTL on four deep learning SoC RTL designs, each integrating multiple RISC-V Rocket [7] cores as the processing host and paired with one of the following accelerators: Conv2D [65], GEMM [65], Gemmini [30], or SIGMA [122]. Evaluations are performed on a 64-bit Linux machine with an Intel i5-13500 CPU and an NVIDIA RTX A4000 GPU. CPU code is generated using LLVM 17's `clang` and `llc`, while GPU code generation leverages CUDA Toolkit 12.6 with compute capability 8.6. All generated code is built with the `-O2` optimization level.

4.4.1 Baseline

We compare HeteroRTL against three CPU-based RTL simulators: Verilator [127], Khronos [146], and BatchSim [130]. Verilator and BatchSim are executed with four threads enabled, while Khronos operates in a single-threaded configuration due to its lack of parallel execution support. Since our experiments focus on single-input stimulus scenarios, we exclude RTLflow [97], a GPU-based simulator specifically optimized for batch-driven workloads. ESSENT [8] and its successors [134, 135] are also excluded, as they fail to complete code generation due to out-of-memory errors. To ensure consistency, all simulation results are averaged over five runs.

4.4.2 Code Generation and Compilation Results

Table 4.1 shows the end-to-end compilation time and executable size for Conv2D, GEMM, Gemmini, and SIGMA across various RTL simulators. The compilation time includes both the transformation from RTL source to simulation code (C++ or LLVM IR) and the final compilation and linking steps to produce the executable. Baseline simulators fail to identify repetitive components, leading to redundant code generation and substantial compilation overhead. In contrast, HeteroRTL compiles in just a few seconds, achieving a three to five order-of-magnitude speedup. This efficiency stems from HeteroRTL's ability to detect structural repetition in heterogeneous architectures and deep learning accelerators, generating evaluation functions only for critical components (e.g., cores and PEs) and reusing them at runtime via corresponding data structures.

To demonstrate HeteroRTL's scalability, Figure 4.3 shows the compilation time of GEMM across RTL simulators as the number of PEs and Rocket cores increases. HeteroRTL exhibits sublinear growth in compilation overhead, even as design size scales exponentially. This trend underscores HeteroRTL's efficiency and scalability for large-scale RTL simulation of heterogeneous architectures and deep learning accelerators.

4.4.3 Overall Simulation Speedup

Figure 4.5 and Figure 4.6 show the simulation speedup of Conv2D, GEMM, Gemmini, and SIGMA on various RTL simulators under different configurations of Rocket cores and PEs, relative to the baseline Verilator. In Figure 4.5, HeteroRTL

Table 4.1: Compilation time (T) and executable size for Conv2D, GEMM, Gemmini, and SIGMA across different Rocket core and PE configurations.

Design	**#Cores**	**#PEs**	Verilator		Khronos		BatchSim		HeteroRTL	
			T(s)	**Size(MB)**	**T(s)**	**Size(MB)**	**T(s)**	**Size(MB)**	**T(s)**	**Size(MB)**
Conv2D	4	2^7	25	0.9	16	0.6	11	1.1	5	1.1
	4	2^{11}	55	4.3	2643	3.8	310	10	5	1.1
	32	2^7	150	3.5	229	2.8	79	5.1	7	1.1
	32	2^{11}	180	6.9	2856	6.1	378	14	7	1.1
GEMM	4	2^7	41	0.9	12	0.5	11	1.0	5	1.1
	4	2^{11}	240	3.6	1145	2.6	148	8.2	5	1.1
	32	2^7	167	3.4	226	2.7	79	4.9	7	1.1
	32	2^{11}	365	6.2	1359	4.9	215	12	7	1.1
Gemmini	4	2^7	396	9	1907	3.3	601	3.8	7	1.2
	4	2^{11}	17909	132	357508	47	92682	52	7	1.2
	32	2^7	521	11	2121	5.6	669	7.8	8	1.2
	32	2^{11}	18034	135	357721	49	92750	56	8	1.2
SIGMA	4	2^7	111	2.8	109	1.3	68	1.8	12	1.4
	4	2^{11}	4936	35	22258	16	10977	20	12	1.4
	32	2^7	236	5.4	323	3.5	136	5.8	13	1.4
	32	2^{11}	5062	38	22472	18	11045	24	14	1.4

demonstrates increasing speedup as deep learning accelerator designs scale from small to large sizes. This improvement is driven by HeteroRTL's strategy of assigning thread blocks to evaluate identical components (PEs) in parallel, leveraging SIMT execution on the GPU to effectively hide latency. As a result, it achieves a speedup of 9× to 122× for the largest designs.

In Figure 4.6, HeteroRTL outperforms other simulators, achieving up to 80× speedup. This gain is attributed to its use of pointer-based struct passing, which enhances data locality and reduces synchronization overhead. Additionally, HeteroRTL adopts a hybrid CPU-GPU co-simulation strategy that balances the workload: complex heterogeneous cores are simulated on the CPU, while simpler PEs

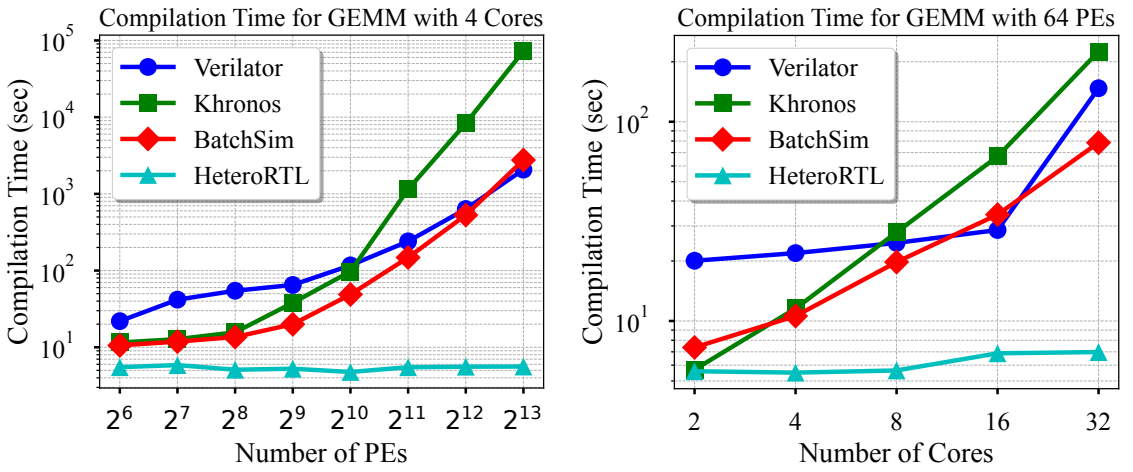


Figure 4.3: Compilation time of GEMM accelerators across different RTL simulators under varying Rocket core and PE configurations.

are handled in parallel on the GPU, reducing pressure on both sides and achieving better load balancing.

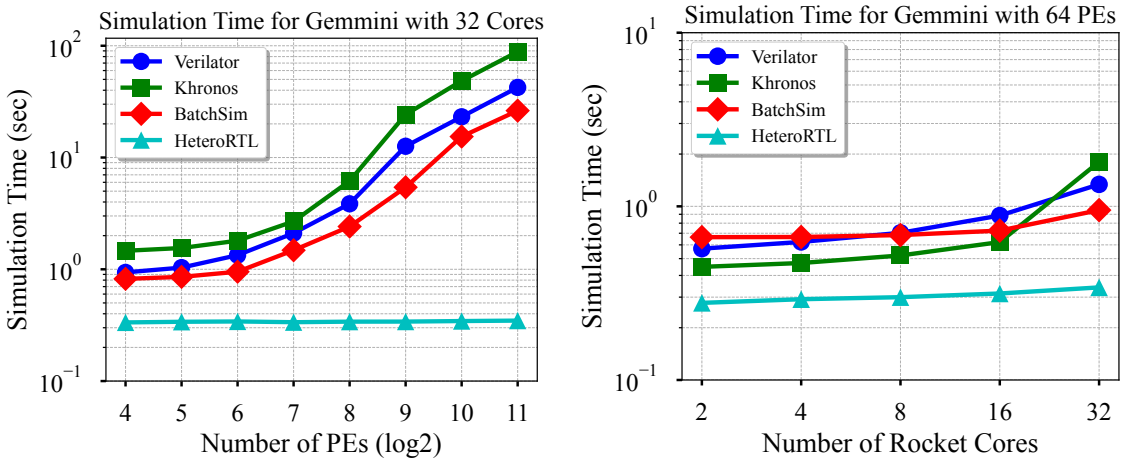


Figure 4.4: Simulation time of Gemmini on various RTL simulators with different numbers of PEs and Rocket core configurations.

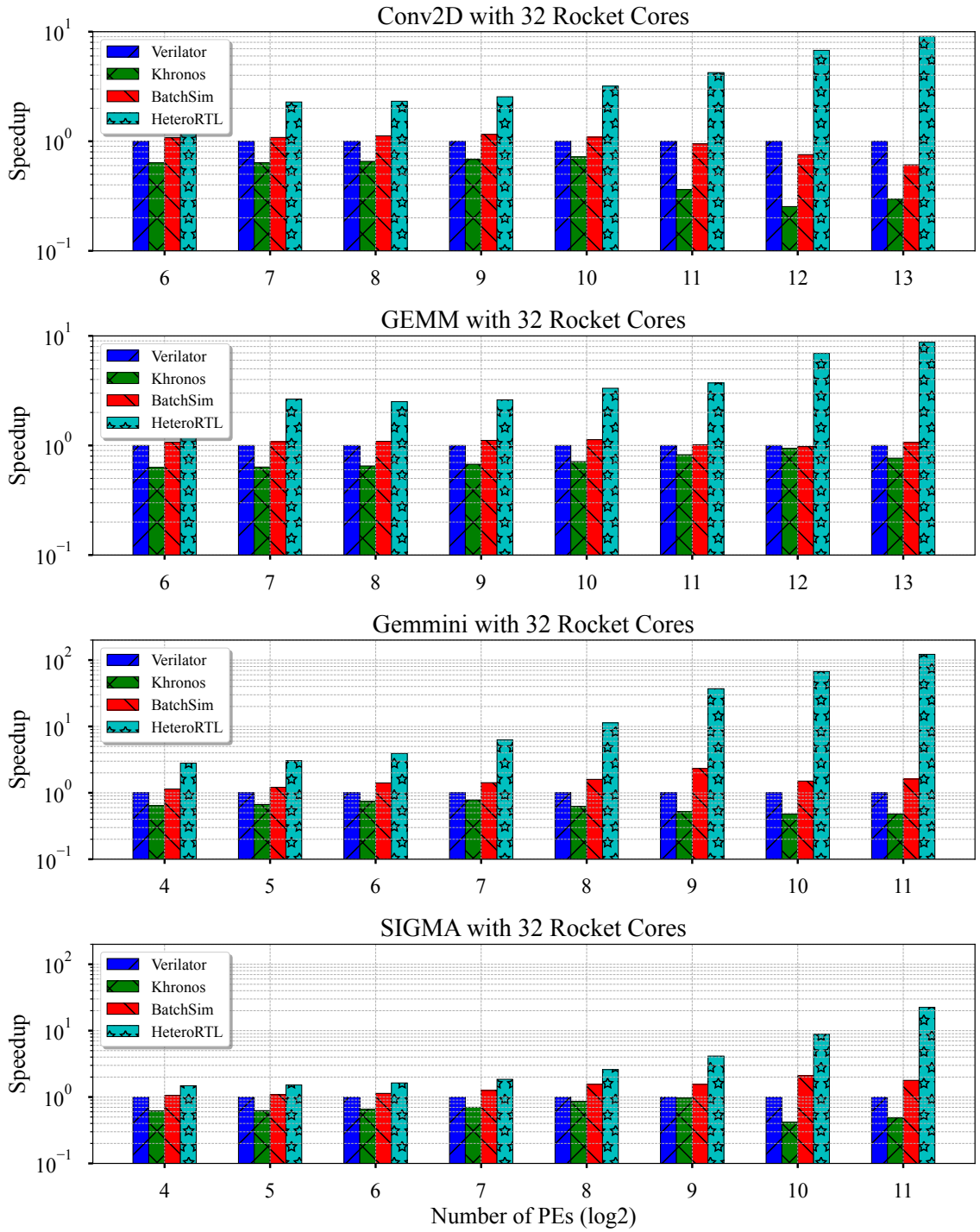


Figure 4.5: Overall simulation speedup of Conv2D, GEMM, Gemmini, and SIGMA on various RTL simulators with 32 Rocket cores while the number of PEs increases exponentially.

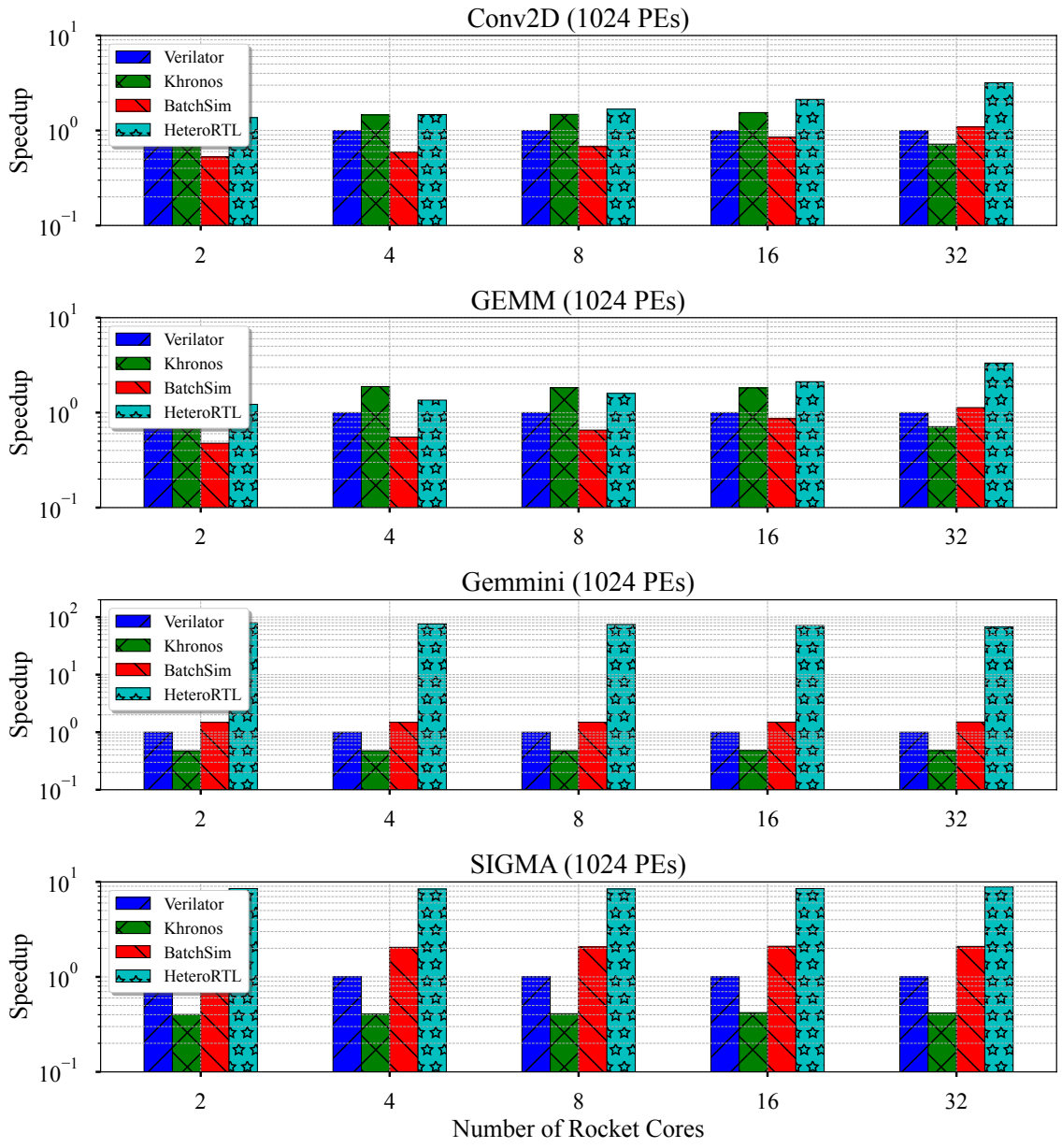


Figure 4.6: Overall simulation speedup of Conv2D, GEMM, Gemmini, and SIGMA on various RTL simulators with 1024 PEs while the number of cores increases.

4.4.4 Simulation Runtime Analysis

Figure 4.4 presents the simulation time of Gemmini on various RTL simulators across different PE counts and Rocket core configurations. All baseline simulators exhibit superlinear growth in simulation time due to their CPU-based execution. With limited CPU threads and instruction counts scaling proportionally to design size, the simulation imposes significant memory and compute pressure on the CPU. In contrast, HeteroRTL demonstrates sublinear growth. As shown in Figure 4.4, the simulation time remains around 0.5 seconds with 32 Rocket cores, even as the number of PEs increases from 2^4 to 2^{11}. This performance comes from offloading deep learning accelerator simulation to the GPU. The GPU consists of multiple streaming multiprocessors (SMs), each capable of handling thousands of threads. This enables latency hiding through context switching and allows massive concurrency, significantly improving simulation throughput.

Figure 4.4 also shows that HeteroRTL continues to outperform other simulators as the number of Rocket cores increases from 2 to 32. This is due to the effective load balancing of HeteroRTL's hybrid simulation model: complex heterogeneous cores (e.g., Rocket cores) are assigned to the CPU, which is better suited for their instruction-heavy behavior, while simpler, massively parallel PEs are evaluated on the GPU. As a result, HeteroRTL's hybrid CPU-GPU co-simulation approach achieves high performance and scalability for heterogeneous architectures with integrated deep learning accelerators.

4.5 Conclusion

This chapter presents *HeteroRTL*, a unified and scalable code generation flow for hybrid CPU-GPU RTL simulation. By introducing an architecture-aware partitioning method and a structural deduplication strategy, HeteroRTL effectively maps complex host cores to CPUs and highly parallel processing elements to GPUs. Built on top of the MLIR infrastructure, HeteroRTL enables modular, parallel code generation and simulation, significantly improving performance and resource utilization. Our evaluation shows that HeteroRTL achieves up to five orders of magnitude compilation speedup and delivers up to 122$\times$ simulation speedup compared to state-of-the-art simulators. These results underscore the benefits of leveraging structural parallelism and heterogeneous hardware to accelerate RTL simulation. In the future, we plan to extend HeteroRTL to support additional features such as dynamic scheduling. We also aim to scale it to support larger and more diverse SoC designs.

In this work, Jie Tong was the primary contributor, leading the majority of the research and development efforts. Zhengxiong Li, Umit Y. Ogras, and Tsung-Wei Huang supervised the project, offering guidance and oversight throughout the project. All authors contributed to the preparation and review of the final manuscript.

Chapter 5

MQL: ML-Assisted Queuing Latency Analysis for Data Center Networks

Data center network (DCN) performance analysis is becoming increasingly critical due to the growing data center scale and proliferation of latency-critical applications. Packet-level simulators, the de-facto performance evaluation tools, allow accurate modeling of the network and protocols, but they are extremely slow [115, 109, 103]. Simulation of large-scale DCNs with thousands of nodes can take days, making meaningful design space exploration impractical. Analytical techniques, such as queuing theory, can mitigate the scalability problem and offer high accuracy when specific workload assumptions are satisfied. However, their accuracy may decline as these assumptions break, and execution times explode unless designed carefully.

To address these challenges, we propose a novel and scalable performance analysis methodology that combines two powerful techniques. First, it uses queuing

theory and the maximum entropy (ME) principle to approximate the waiting time in each queue in a DCN. It then finds the end-to-end latency of each flow using traffic input, routing algorithm, and network parameters. This ME-based queuing model can approximate the latency under generalized exponential input traffic and general service distributions. Since its accuracy can degrade as traffic diverges from input and service time assumptions, the second step of the proposed methodology learns and corrects the systematic errors using a regression tree. The resulting ML-assisted technique achieves less than 3% modeling error on average compared to ns-3 simulations. Moreover, the speedup over ns-3 ranges from 100× to 9000× on DCNs with 128 to 1024 nodes.

5.1 Overview

Due to the increasing demand for internet services, millions of users worldwide rely heavily on data centers to serve their computational needs. Data centers provide shared access to data, applications, storage, and compute resources. They comprise of many compute and storage nodes structured in a particular topology, such as a fat-tree [5]. As the data volumes and amount of managed services explode, data centers scale out to satisfy the growing requirements.

Data center networks (DCN) must deliver low latency and virtually unlimited bandwidth to maximize the quality of service (QoS) under cost (e.g., equipment cost) constraints. Therefore, DCN architects spend substantial effort designing the network topology, routing algorithms, and protocols. Packet-level simulators,

such as ns-3 [3] and OMNet++ [4], are used to evaluate the performance of these factors. Simulators provide flexibility in modeling different protocols and a high degree of observability, facilitating debugging and achieving high fidelity with real DCN hardware. However, the advantages of a simulator come at the expense of simulation speed. Our evaluations show that ns-3 simulations of a DCN with a 1024-node fat-tree topology and 100 Mbps links take up to a week. Consequently, simulation-based design space exploration is not practical, considering that numerous simulations with different parameters are required.

Notoriously slow simulation speeds motivated the deployment of analytical approaches to estimate network performance [9, 12, 116, 105, 106, 107, 113, 114]. The most prominent example is the queuing theory, which led to a large family of analytical models that model the queuing delay under different input traffic and service time distributions. These models are then employed to approximate the delay through complex switches and eventually through networks composed of these switches. Queuing latency models are effective when the workload and service time assumptions match real-life behavior. For example, the M/M/1 queue model in Kendall's notation [72] implies Poisson input arrivals, exponentially distributed service time, one server, and infinite-sized queues. The analytical equations for this model match almost perfectly with actual measurements when the Poisson arrival, exponential service time, and infinite-sized queue assumptions hold. The rich queuing theory literature [9, 12, 76, 78, 77] provides tailored models to more general cases, such as general traffic arrival and service time distributions and a finite-sized queue with K slots. However, the model accuracies degrade with

higher mismatch between the model assumptions and real-life behavior, and with non-trivial interaction between tandem queues. Furthermore, knowing the precise input traffic and service time distributions is unrealistic.

Motivated by the shortcomings of classical queuing theory, recent works proposed employing deep learning techniques for DCN performance analysis [142, 29, 138, 123]. For example, MimicNet [142] proposes a hybrid technique that combines simulation with deep learning. It observes that users are often interested in detailed performance analysis of one cluster. Hence, they simulate the cluster of interest using OMNet++ and model the rest of the DCN using a deep neural network (DNN) trained offline. While speeding up the DCN performance analysis, this approach loses the network-level observability by abstracting all remaining clusters. Similarly, DeepQueueNet [138] models each device in the network as a DNN. The proposed models approximate the delay of each packet in the incoming packet stream and produce an outgoing stream of packets. Then, it composes these DNNs in one-to-one correspondence with the network structure. Treating the switches as black boxes and modeling them with DNNs requires substantial training data. Furthermore, trying to achieve high accuracy can lead to overfit and overkill since this approach replaces the well-known structure of DCN switches with generic DNNs instead of exploiting them. Finally, the composability of these DNNs is not proven theoretically. These emerging approaches aim to replace the well-established analytical techniques completely, while traditional queuing models fail to exploit the rich set of simulation data and emerging machine learning (ML) techniques. In contrast to these two extremes, we propose to build a new family of

ML-assisted queuing latency (MQL) analysis approaches by leveraging queuing theory rather than resorting to black-box deep learning techniques. We base our novel approach on the following insights:

Key insight 1: Queuing theory can produce fast, accurate, and scalable models for many workload scenarios.

Key insight 2: Current methodologies already compare the accuracy of analytical techniques to simulations and produce a massive dataset. We can learn from this data where the analytical models fall short.

Key insight 3: A suite of lightweight ML techniques, such as regression analysis, can be used to learn and correct the analysis errors.

We propose a new MQL methodology using the insights listed above. The proposed MQL method first develops analytical latency models based on queuing network discipline deemed appropriate for the target scenario. This work employs the maximum entropy (ME) model to derive mathematical expressions for each queue in the target DCN. Then, the input flows, topology, and routing algorithm are used to find the end-to-end flow latencies. In this work, the ns-3 simulator is used as the golden truth to compare the accuracy between the analytical model and simulation. At this point, the proposed MQL methodology performs a more extensive analysis than simply measuring and comparing the accuracy. From the comparison, we identify the regions where the analytical models fall short. These shortcomings can happen systematically due to the network structure, such as tandem queues. We use these systematic errors to extract the features that are highly correlated with the underlying analysis of the analytical model. The last step

of the proposed MQL methodology is modeling the systemic errors and adding them to the analytical estimates. While the MQL methodology can be used with any ML technique, this work uses Regression Trees (RT).

In summary, this work makes the following contributions:

- Demonstrates the first ML-assisted queuing theory-based technique that can handle a large-scale (>1000 nodes) network of queues, for modern DCN protocols,
- An automated tool that generates an executable performance model (queuing analysis and RT) for a given DCN,
- The ability to provide detailed observability (e.g., individual queuing delay, occupancies, and tier-level visibility) without relying on any communication pattern and topology assumptions,
- Extensive simulation studies with synthetic traffic and network traces that demonstrate less than 3% error on average, 100× to 9000× speed up over ns-3, and scalability to 1024-node fat-tree.

The rest of this chapter is organized as follows. Section 5.2 presents a background on data center networks and the motivation for this work. The proposed MQL approach is described in Section 5.3. Section 5.4 presents extensive experimental evaluations and comparisons with state-of-the-art approaches. We review the related work in Section 5.5. Finally, Section 5.6 concludes the work with directions for future work.

5.2 Background and Motivation

Data centers are evolving rapidly with new paradigms and emerging innovations, such as resource disaggregation and low-diameter topologies, to meet the unprecedented growth of data center computing and enable new system-level architectures [129]. Similarly, application disaggregation approaches disintegrate monolithic applications into small microservices or functions with communication overheads [128]. CPU tasks are increasingly offloaded to special-purpose accelerators, often linked over the network, including FPGA resource pooling [147]. In addition, memory and storage are being disaggregated, opening up new system-level architectures to explore [102, 89].

From a DCN perspective, integrated silicon photonics [110] delivers breakthrough performance, enabling high bandwidth and low latency interconnection. It has generated renewed interest and innovations in low-diameter (shorter path-length) topologies [83, 137, 11, 74, 75]. These benefits enable exploring large-scale topologies (>10K end points) [108] and numerous permutations. Hence, there is a critical need for fast performance models to explore the vast design space of new architectures and data center topologies enabled by these innovations.

This work develops fast performance models that generalize to any topology [10]. While exploring new topologies is a crucial use case, we initially demonstrate the proposed technique on fat-tree topologies. Since fat-trees are widely used in DCN and HPC systems [136, 108], they provide the baseline for new DCNs "to beat." Figure 5.1 shows an example of a three-tier fat-tree topology with 16 compute nodes and twenty k-port switches arranged hierarchically in three layers.

In addition to the data center network, the proposed technique can also be applied to model the emerging interchiplet communication architectures, enabled by advanced packaging [124, 126, 125, 79].

A well-established approach for fast network modeling is queuing theory [10, 29]. Each compute node and switch in the DCN is represented by one or more queues, while the interconnection of compute nodes and switches is modeled as a network of queues (shown in blue), as illustrated in Figure 5.1. The internal structure of switches is modeled via different techniques, such as output-queue (OQ), with queues only on the output ports, and combined input/output-queued (CIOQ) switch with queues on both the input and output ports [25]. OQ switches have a simple scheduling policy which is easier to model, but they are impractical due to the high-speed crossbar. CIOQ switches, on the other hand, are practical to implement but are harder to model due to their more complex scheduling policies. For modeling a switch's performance, [25] shows that the simpler OQ switch is equivalent to CIOQ+CCF (critical cell first IO scheduling) in its input/output

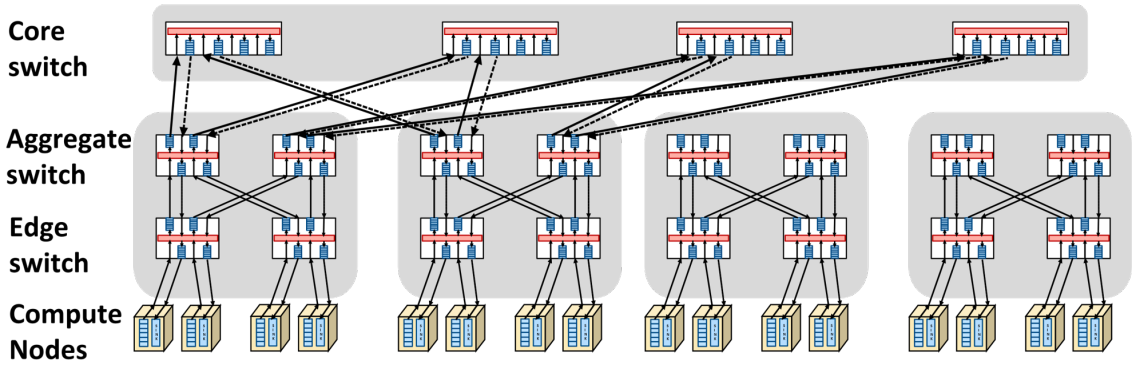


Figure 5.1: Queuing theory representation of 16-node fat-tree

timing characteristics. Thus, we use the OQ switch as the modeling abstraction.

There are different versions of queuing theory-based analytical models. This work uses the principle of maximum entropy (ME) because it handles bursty traffic and generalized service distributions quite well by generating the least biased distribution that matches the constraints. We first find the mean queue occupancies and then calculate the waiting time using Little's law [101]. One must also model the interconnection of different queues and flows going through the queues following the routing algorithm. We use the decomposition method to find the inputs to each queue [121], as detailed in Section 5.3. Finally, the accuracy of analytical models can degrade when the real traffic diverges from the assumed parameters. Instead of completely switching to a machine-learning (ML)-based approach, we harness the power of queuing theory and improve on it using a light-weight ML-based correction technique, as described in Section 5.3.5.

5.3 MQL: ML-Assisted Queuing Latency Analysis

5.3.1 MQL Overview

We set the following goals while scaling to thousands of nodes and providing visibility of internal queue utilization:

- High accuracy in estimating end-to-end packet latency and round-trip delay,
- Fast and lightweight estimation, scaling to DCNs with thousands of nodes,
- Tier- and queue-level visibility (e.g., observing individual queue utilizations),

- Generality to support different workloads and protocols.

Offline Phase: The first two components of MQL leverage the current DCN performance evaluation practice, as illustrated in Figure 5.2. It develops analytical performance models for the topologies, workloads, and network protocols of interest. Then, the accuracy of these models is compared against simulations. The common practice does not offer any systematic technique to learn the accuracy mismatches and use them for correction. MQL elaborates on this step by systematically analyzing the modeling error as a function of the simulation and analytical parameters. For example, the error may be a function of the queue within a particular switch, data rate, or any input used by analytical models. Therefore, the final component, which provides ML assistance, extracts the most prominent features. Then, it trains a regression model that estimates the error as a function of these features.

Online Use: MQL first runs the analytical performance model to determine the end-to-end latencies. Then, it adds the error estimate to its results as a correction factor.

The proposed MQL methodology can be implemented with any performance analysis and ML technique. The following section describes the specific methods used in this paper.

5.3.2 Modeling Assumptions and Target Illustrative DCN

The proposed MQL methodology can be applied to arbitrary DCN topologies and routing algorithms. This work demonstrates MQL on the three-level fat-tree

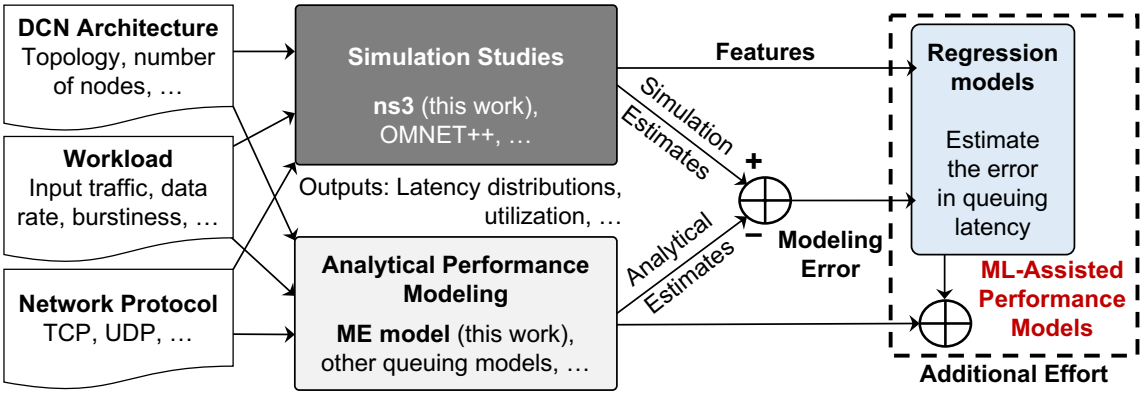


Figure 5.2: Overview of the proposed MQL methodology.

topology (shown in Figure 5.1) with up to 1024 nodes (16 pods) due to its popularity. It is validated with fixed and equal-cost multi-path routing (ECMP) [43] due to its wide usage and complexity.

Since real-world traffic can be bursty, we model the arrival process at each queue based on the Generalized Exponential (GE) distribution [78, 77], which can also handle other distributions like Poisson. The queues accept packets one at a time. Hence, multiple flows (a stream of packets per source-destination pair) can merge and decompose while entering and leaving queues. This general behavior suggests that even if the individual flows follow a specific distribution at the input, the output stream of packets can follow an unknown distribution. Similarly, we do not make any assumptions about the packet size distributions and thus select the Generalized distribution to model the service time. The channel (link) between switches and nodes in Figure 5.1 is modeled as a server for the output queue on the corresponding port. Therefore, we start with a GE/G/1 model and extend it to GE/G/1/N, where N is the finite queue length.

5.3.3 Analytical Queuing Models

This work uses ME-based queuing models to illustrate the proposed MQL methodology due to its accuracy and scalability. Since we focus on the network latency, we use the traffic from each node entering the DCN as the primary input. The ME-based models with generalized exponential traffic employ the first two moments: the average arrival rate (λ_i) and squared coefficient of variation ($C_{A_i}^2$) of each input flow, as summarized in Table 5.1. These inputs are propagated to the switches in the fat-tree using a decomposition method [121]. Then, the ME models are used to compute the delay in each queue in the target DCN, as described next.

Decomposition method

The flows entering the DCN from the node queues go through multiple merges and separations as they travel to their destination. Since we used a generalized exponential model, the sum of arrival rates alone is insufficient, unlike the Poisson distribution assumption. Figure 5.3 illustrates two flows entering the same queue. The packets from these flows are stored in their arrival order, which is arbitrary. Hence, we need to estimate the first and second orders of the models, i.e., the arrival rate and squared coefficient of variation. Following the derivation in the decomposition model given in [121], we find the squared coefficient of variation of inter-arrival times of merged flow as the weighted average of the incoming squared coefficient of variations, as shown in Figure 5.3. Similarly, the squared coefficient of variation of inter-departure times (C_D^2) of the merged flows is found using the decomposition approach illustrated in Figure 5.3. The output flows are split to

Table 5.1: Summary of notations used in this work.

λ_i	Injection rate of flow-i
ρ_i	Link utilization of flow-i
p_i	Probability of flow-i split when leaving the queue
$C_{A_i}^2$	Squared coefficient of variation of inter-arrival time for flow-i
$C_{D_i}^2$	Squared coefficient of variation of inter-departure time for flow-i
$C_{S_i}^2$	Squared coefficient of variation of service time for flow-i
λ	Injection rate for merged flow
ρ	Link utilization for merged flow
C_A^2	Squared coefficient of variation of inter-arrival time for merged flow
C_D^2	Squared coefficient of variation of inter-departure time for merged flow
C_S^2	Squared coefficient of variation of service time for merged flow
$\langle n_i \rangle$	Mean queue length of flow-i in an infinite-sized queue
$\langle n_i \rangle_N$	Mean queue length of flow-i in a finite-sized(N) queue
W_i	Average waiting time of flow-i

enter the downstream queues. Therefore, we calculate the C_D^2 of the split flow and the probability of splitting based on the number of downstream queues using the equation in Phase 3 of Figure 5.3. These split $C_{D_i}^2$ will be the $C_{A_i}^2$ to the downstream queues. Furthermore, the decomposition method is computed in one pass, making our approach scalable.

GE/G/1 Maximum Entropy model

The Maximum Entropy(ME) method approximates the networks when queues achieve equilibrium [12, 78]. For fat-tree topology, with the first-come-first-served (FCFS) queuing discipline and a single server, we adopt the GE/G/1 ME model (generalized exponential arrival process, and generalized service process) proposed

in [78, 77]. The proposed approach traverses the network from source to destination for each flow in the workload, following the routing algorithm. During this process, it uses the decomposition process to find the mean arrival rate (λ_i), utilization (ρ_i), squared coefficient of variation ($C^2_{A_i}$), and the coefficient of variation of the service time ($C^2_{S_i}$) at each queue. Then, it uses the GE/G/1 ME model to find the mean queue length of each flow i ($\langle n_i \rangle$) in an infinite-sized queue as:

$$\langle n_i \rangle = \frac{\rho_i}{2}(C^2_{A_i} - 1) + \frac{\sum_{k=1}^{N} \frac{\lambda_i}{\lambda_k} \rho_k^2 (C^2_{A_k} + C^2_{S_k})}{1 - \rho} \tag{5.1}$$

Finally, the waiting time (queuing delay) of flow-i becomes:

$$W_i = \frac{\langle n_i \rangle - \rho_i}{\lambda_i} \tag{5.2}$$

GE/G/1/N Maximum Entropy model

For the finite-sized queue model, we adopt the treatment of queue occupancy (and delay) presented in [76]. We list the key steps here for completeness. Derivations can be found in [76]. We start constructing the analytical model for the finite-sized queue by first assuming infinite queues. With this assumption, we first find the mean queue length of each flow i ($\langle n_i \rangle$) in an infinite-sized queue using Equation 5.1. Then, we find the Lagrangian coefficient x by using the mean infinite queue length $\langle n_i \rangle$ from Equation 5.1 [76].

$$x = \frac{\langle n_i \rangle - \rho_i}{\langle n_i \rangle} \tag{5.3}$$

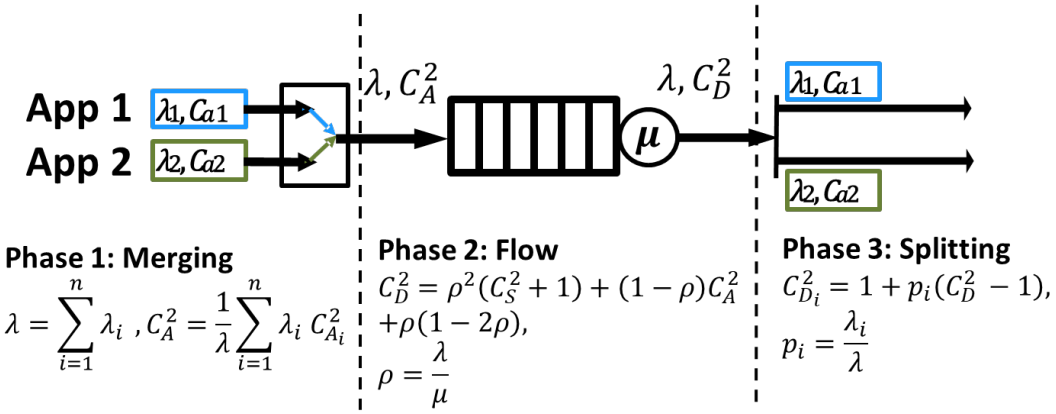


Figure 5.3: Decomposition method: Phase 1 merges multiple flows into a single flow. Phase 2 computes the coefficient of variation of departure processes. Phase 3 splits the merged flow to derive the individual departure processes.

The Lagrangian coefficient x is then used to find the mean finite queue length given by Equation 5.4, where N is the finite size of the queue:

$$\langle n_i \rangle_N = \frac{\rho_i}{1 - \rho_i^2 x^{N-1}} \left\{ \frac{1 - x^N}{1 - x} - N\rho_i x^{N-1} \right\} \tag{5.4}$$

Finally, it computes the finite mean occupancy ($\langle n_i \rangle_N$) and the waiting time (queuing delay) of flow-i as:

$$W_i = \frac{\langle n_i \rangle_N - \rho_i}{\lambda_i} \tag{5.5}$$

5.3.4 End-to-End and Round-Trip Latency Modeling

Using the λ_i, ρ_i, $C_{A_i}^2$ of the flows entering a queue, we find the queuing delay of that queue. Then, the squared coefficient of variation of inter-departure time ($C_{D_i}^2$) is calculated by the decomposition model, which uses the squared coefficient of variation of inter-arrival time ($C_{A_i}^2$) of the upstream queues. Therefore, the

algorithm iterates over every queue that has its inputs ready and computes the corresponding delay.

The next step is finding the end-to-end latency of each flow by summing up the queuing delays and the service times. The proposed approach achieves this objective using the fat-tree size, routing algorithm, and the characteristics of each flow(λ_i, ρ_i, $C_{D_i}^2$, $C_{S_i}^2$, mean packet size, queuing delay of host queues). The output is the average end-to-end latency for each flow, as shown in the pseudo-code in Algorithm 1. Finally, we compute the average round-trip time (RTT) by adding the end-to-end latency of data and their corresponding acknowledgment packets.

5.3.5 ML-Based Correction Technique

While the ME model applies to any network that decomposes into a network of queues, it may fail to generalize to all scenarios. Therefore, MQL augments the

Algorithm 1: End-to-end latency computation

```
Input: Fat-tree size, link bandwidth, flow metadata (flow ID, source, destination),
  characteristics for each flow (λ_i, C²_{D_i}, C²_{S_i}, mean packet size, queuing delay of host
  queues)
Output: Average end-to-end latency for each flow
foreach queue ready to be processed do
    I = number of flows in the queue
    for i = 1:I do
        Compute W_i using GE/G/1 and GE/G/1/N ME model
        Compute C²_{D_i} using decomposition model
        Populate flow characteristics of the upstream queues
    end
end
foreach flow in the traffic do
    Traverse all the queues throughout the flow's path
    Aggregate queuing delay and link delay
end
```

ME model with a correction factor obtained by ML models. We first provide two examples when ME models perform poorly and then present the proposed ML-based correction technique.

Sample scenarios in which ME models fail to generalize

The packets pass through a sequence of queues as the packets traverse the DCN. For example, packets from the first node (far left) to the last node (far right) in Figure 5.1 pass through queues in the edge, aggregate, and core switches. This tandem arrangement of queues shapes the packet inter-arrival times as a function of their link service times. When packets of different sizes pass through the same link, the tandem nature causes the smaller packets to experience higher queuing latency than their larger counterparts. The ME models fail to capture this effect, resulting in poor latency estimations.

Similar to the packet size distribution, the communication protocol plays a crucial role in the manner the packets are transported through the network. For instance, the User Datagram Protocol (UDP) sends packets into the network irrespective of the size, while the Transmission Control Protocol (TCP) divides the packets into segments based on a preset threshold size [120]. Similarly, the TCP sends an acknowledgement packet, which are typically significantly smaller than the data packets. The resulting bimodal packet size distribution, combined with the tandem effect, degrades the accuracy.

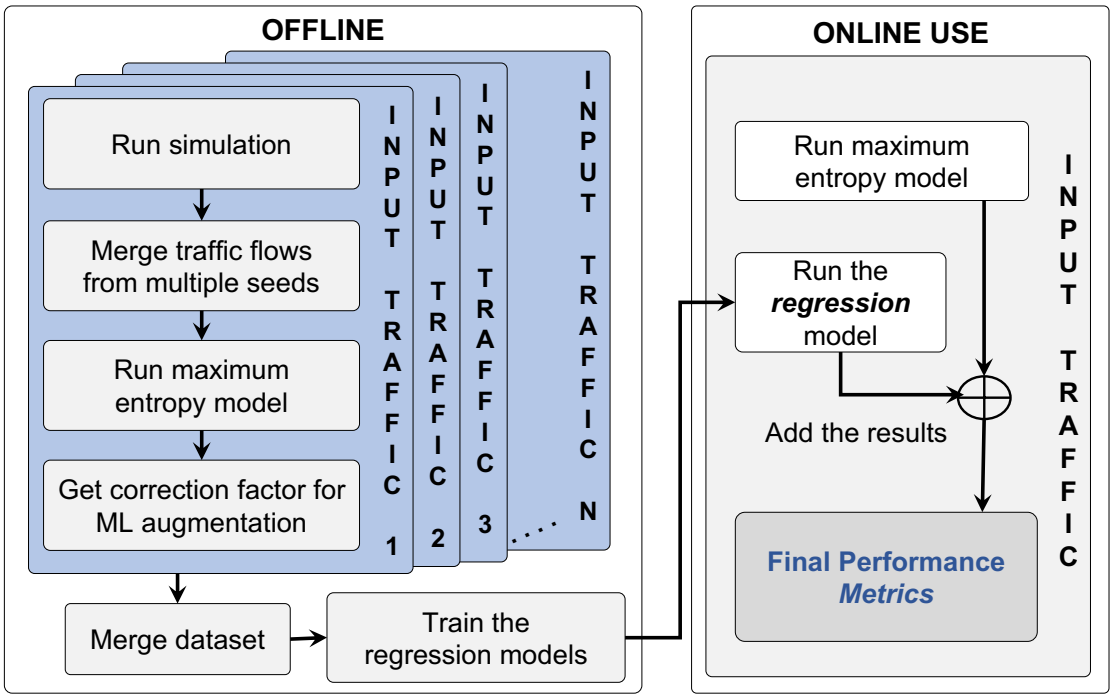


Figure 5.4: Workflow of the ML-assistance component in the MQL framework.

Proposed ML-based Assistance to Queuing Models

The ML-assistance component of the proposed MQL framework is presented in Figure 5.4. First, we run simulations with representative configurations (network sizes) and input traffic (packet arrival and size distributions, data rates, and traffic types). Simulations with multiple random seed values help in eliminating randomness effects. Then, MQL obtains the correction factors by comparing the expected latencies from the simulation and the ME models. Since each queue type (e.g., edge-up, core-down, aggregate-up) in the fat-tree observes a different traffic pattern, we use a regression model for each queue type. The input features (described in Section 5.3.5) and the correction factors are aggregated to obtain a merged training

dataset. Since the systematic errors are continuous quantities, any regression-based ML model can be used in this stage. We use regression trees (RT) due to their accuracy and explainable structure. To avoid overfitting and minimize inference latency, we empirically set the maximum depth of RTs to 12. The RT uses 11 input features (shown in Table 5.2) and predicts one real-valued output. MQL employs one RT model for each type of queue (e.g., edge, aggregate, core) in the network, regardless of the number of nodes, thereby providing excellent scalability across network sizes. Finally, we utilize the scikit-learn library to train RTs. Building a consolidated set of training samples allows MQL to generalize to different traffic patterns. Then, MQL trains generalizable regression models, which concludes the one-time offline process. Finally, the runtime step uses the pre-trained regression models to accurately estimate the latency.

Features for the ML-based Regression Model

End-to-end latencies are functions of DCN topology, the traffic arrival distribution, packet size distribution, data rates, and routing patterns. Since our goal is to estimate these delays accurately, we systematically construct their input features with the following attributes:

1. They must demonstrate a strong correlation with the target quantity to be estimated,

2. They cannot depend on any parameter or quantity we cannot obtain at runtime (e.g., information from simulation),

Table 5.2: List of the input features constructed in a particular queue for the regression model.

Input Feature	Mathematical Representation
Data rate of flow i	λ_i
Link utilization of flow i	ρ_i
Total utilization of a link leaving switch s	$\rho_{total,s}$
Co-efficient of inter-arrival time of flow i	$C^2_{A,i}$
Co-efficient of service time for flow i	$C^2_{S,i}$
Packet size of flow i	P_i
Link occupancy indicator of flow i	$1\ /\ (1-\rho_i)$
Link occupancy indicator of switch s	$1\ /\ (1-\rho_{total,s})$
Data rate/link occup. indicator of flow i	$\lambda_i\ /\ (1-\rho_i)$
Data rate/link occupancy indicator of switch s for flow i	$\frac{\lambda_i}{1-\rho_{total,s}}$
Queue occupancy indicator of flow i	$(C^2_{A,i}+C^2_{S,i})\,/\,(1-\rho_{total,s})$

3. The overheads to compute them must be minimal.

To satisfy these requirements, we methodically architect the 11 input features to the regression model as shown in Table 5.2 for each queue type. The queuing models described in Section 5.3.3 use data rate (λ), link utilization (ρ, and ρ_{total}), and second-order moments of inter-arrival (C^2_A) and service times (C^2_S) in latency estimation. The proposed MQL framework exploits the information to include these parameters as input features. In addition, we include input features, such as link occupancy and queue occupancy indicators, since they typically appear in analytical models (e.g., Equation 5.1). These features satisfy *Attribute 1* both intuitively (as described here) and empirically (demonstrated in Section 5.4). Basing the input features on quantities computed by the ME model is highly desirable since we reuse

the information already computed, thereby simultaneously catering to *Attribute 2* and *Attribute 3*.

5.4 Experimental Evaluation

This section first describes the experimental setup. Section 5.4.2 and Section 5.4.3 evaluate the proposed MQL approach with synthetic traffic and real-world traces, respectively. Section 5.4.4 presents the execution time of the MQL models, their speedup w.r.t simulation, and comparisons with approaches from the literature. Finally, Section 5.4.5 compares the round-trip latency of MQL with state-of-the-art approaches.

5.4.1 Experimental Setup and Methodology

DCN Topology: While the MQL methodology is applicable to other topologies, this work focuses on the widely used fat-tree topology. Fat-tree topologies are represented by the number of layers and a parameter K, which determines the number of pods [5]. The number of nodes for a K-ary fat-tree is $(K^3/4)$. In this work, we use three-layer fat-trees with $K \in \{4, 8, 12, 16\}$, leading to sizes listed in Table 5.3.

Workloads used for Evaluation: We utilize both synthetic and real-world traces to evaluate MQL. Synthetic traffic includes *three* different traffic types: all-to-all (each node in the DCN sends packets to every other node), broadcast (only one node (source) sends packets to all other nodes), and incast (all nodes send packets to

one node (destination)), as summarized in Table 5.3 along with other parameters. The real-world trace is Anarchy [119].

Simulation Environment and Other Parameters: We performed simulations with ns-3, a discrete-event network simulator [3]. NS-3 provides packet-level visibility. It also allows users to configure various parameters such as the number of source nodes and destination nodes in the DCN, routing patterns, mean flow sizes, simulation time, network protocol, and FIFO and queue sizes. We perform 30-second simulations with a warmup of an additional 10 seconds to ensure the inputs to the simulation are representative of the steady state. The queue sizes are set to 128 to represent the finite buffer scenario.

The simulations and analytical models are executed on an Intel® Xeon® Gold 6336Y CPU at 2.40GHz with 36 MB cache (OS: SUSE Linux Enterprise Server 12

Table 5.3: A summary of the experimental setup used for evaluations in this paper.

Parameter	Values Evaluated in this Paper
3-Layer Fat-Tree Topology	Number of nodes: 16, 128, 432, 1024
Workloads	Synthetic and Real Traces
Synthetic Traffic Patterns	All-to-all, broadcast, incast
Traffic Arrival Distributions	Poisson and Generalized Exponential (GE)
Synthetic Packet Size Dist.	500 B; uniform (500B with 1% variation)
Synthetic Workload Data Rates	Low (link utilization of 25%) Medium (link utilization of 50%) High (link utilization of 75%)
Real Trace	Anarchy
Link Bandwidth	100 Mbps
Protocol	TCP, UDP
Queue	FIFO, 128 Packets Capacity

SP5, compiler version: g++/gcc 11.1.0).

ML-based Assistance: The ML-based regression models must generalize to unseen scenarios for the application of the proposed framework at a larger scale. Thus, we randomly pick 60% of the data to train the regression models, and then evaluate all configurations. In this particular work, we use the regression tree model with a maximum depth of 12 [118].

Comparison Metrics: We use the normalized Wasserstein distance [133] and mean absolute percentage error (MAPE) for accuracy evaluations. The Wasserstein distance compares probability distributions based on the theory of optimal mass transport, and is a measure of the distance between two distributions. MAPE is a measure of the average absolute error observed between simulation and MQL estimates.

State-of-the-Art Approaches Chosen for Comparison: We identify a combination of ML- and queuing theory-based approaches, namely DeepQueueNet [138], MimicNet [142], and RouteNet [27] for comparisons. Section 5.5 discusses the significance of these approaches. Comparisons with the state-of-the-art approaches are presented in Section 5.4.5.

Protocols: We evaluated the proposed MQL methodology using both UDP and TCP. UDP is a connectionless protocol with no congestion control mechanism. Thus, the flow distribution with UDP is less complicated than TCP. Since we obtain high accuracy (overall less than 10% MAPE) with MQL, the rest of this paper focuses on the TCP results.

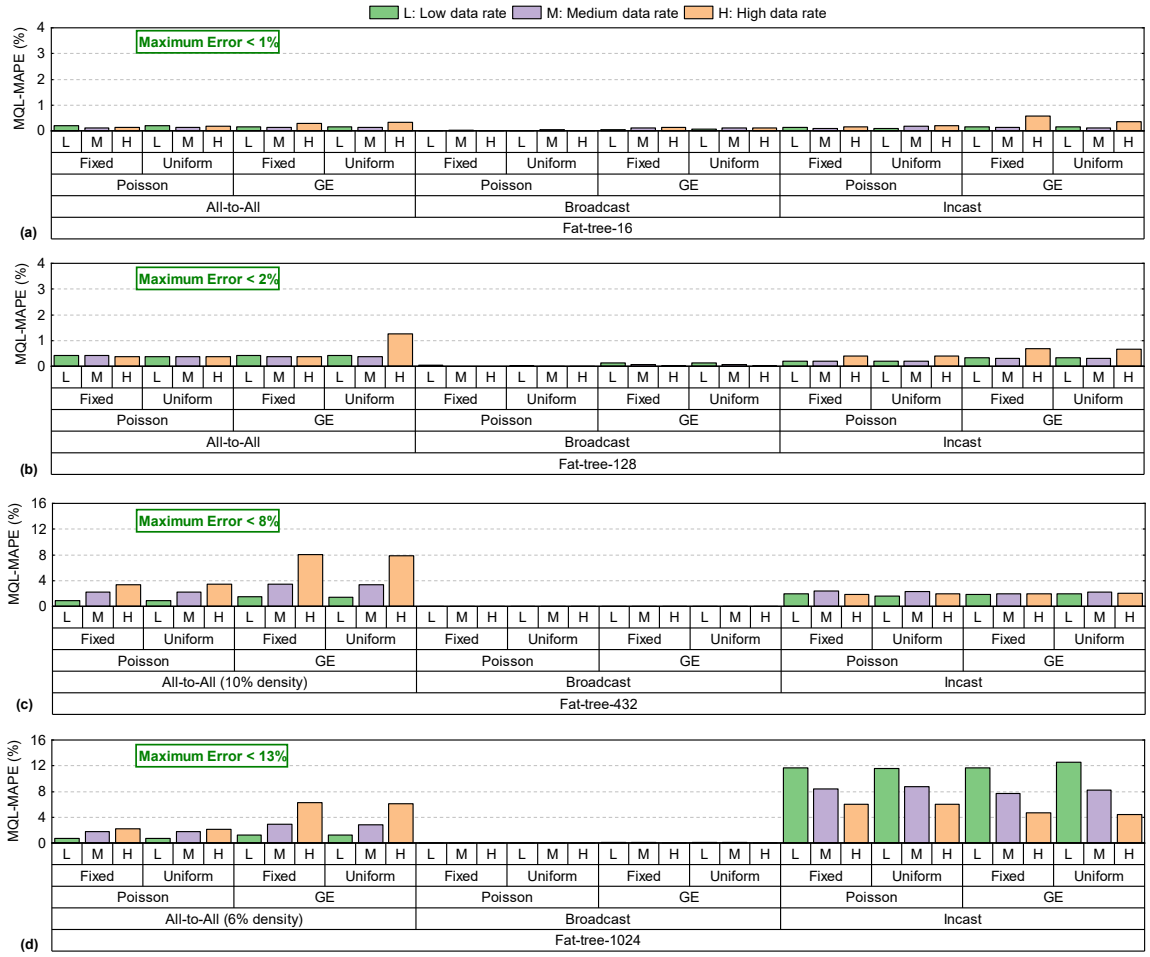


Figure 5.5: MAPE (%) of the round-trip latency achieved by MQL on all-to-all, incast and broadcast traffic for (a) Fat-Tree-16, (b) Fat-Tree-128, (c) Fat-Tree-432 and (d) Fat-Tree-1024 with different types of packet arrival distributions, packet size distributions, and data rates.

5.4.2 Evaluations with Synthetic Traffic

This section presents extensive evaluations to compare the proposed MQL framework to ns-3 simulations using synthetic traffic. Figure 5.5(a), (b), (c), and (d) present the MAPE results for fat-tree-16, fat-tree-128, fat-tree-432, and fat-tree-1024,

respectively. We sweep three data rates (low, medium, and high) with uniform distribution of packet size (fixed and uniform as defined in Table 5.3) in all cases. The entire all-to-all traffic simulations for fat-tree-432 and fat-tree-1024 take prohibitively long simulation times (over 5–9 days for one random seed only) since they simulate 373K flows and 2M flows, respectively. As a result, we reduced the number of flows by using a density parameter that uniformly selects a subset of active hosts participating in the all-to-all communication [71]. Considering the simulation time, we set the density parameters for fat-tree-432 and fat-tree-1024 as 10% and 6%, respectively.

The broadcast traffic is the simplest pattern since packet injection is only from one source. Furthermore, the packets entering the network are serialized. Our MQL framework models this scenario very accurately, with average MAPE always less than 1%, as shown in Figure 5.5. Hence, in the following discussion we only cover all-to-all and incast patterns.

Fat-tree-16 Results: MQL achieves an MAPE of less than 1% for Poisson all-to-all traffic. Unlike Poisson, GE traffic can produce bursty traffic, which is more complex to model. The maximum error with GE packet arrivals, even in the medium and high data rates, remains less than 2%, with an average MAPE of 0.9%. The incast traffic pattern is highly complex to model since several flows merge into one queue. A combination of the ME and ML models in the MQL framework effectively captures this behavior and achieves an average MAPE of 0.9%, with the highest being under 2%.

Fat-tree-128 Results: Similar to the analysis for Fat-tree-16, MQL achieves an average

MAPE of 1% for all-to-all Poisson packet arrivals and $< 2\%$ MAPE for GE arrivals. MQL models incast for Fat-tree-128 accurately, with an average error less than 0.9%.

Fat-tree-432 Results: The number of flows and complexity of latency estimation grow with increasing fat-tree size. Besides scaling to these large sizes, the combination of the ME and ML models in the MQL framework perform well with an average error of less than 3% MAPE for the incast traffic and less than 8% MAPE for the all-to-all (10% density) traffic.

Fat-tree-1024 Results: The large number of flows in a 1024 node fat-tree, especially merging into a single queue in incast traffic, severely complicates the modeling. The proposed MQL models result in higher error compared to lower network sizes, with an average MAPE of 8% for incast patterns. However, we note that MQL still enables rapid design space exploration when compared to ns-3, which takes over a week to simulate a reasonable workload duration.

Cumulative Distribution Function of Round-Trip Latency: We must ensure that the RTT throughput distribution is close to the ground truth, as opposed to an averaged value such as the normalized Wasserstein distance or MAPE. Figure 5.6 presents the cumulative distribution function (CDF) of the RTT for all-to-all traffic in fat-tree-16 and fat-tree-128. We observe that MQL achieves high fidelity with the simulation ground truth in the RTT spectrum for both fat-tree-16 and fat-tree-128.

Error Reduction: As anticipated, the MQL demonstrates a significant improvement in error reduction compared to the ME model alone. Upon evaluating all of the synthetic workloads, we calculated the difference of MAPE between the ME model and MQL. The results indicate that the average error reduction is 7.1%, with a

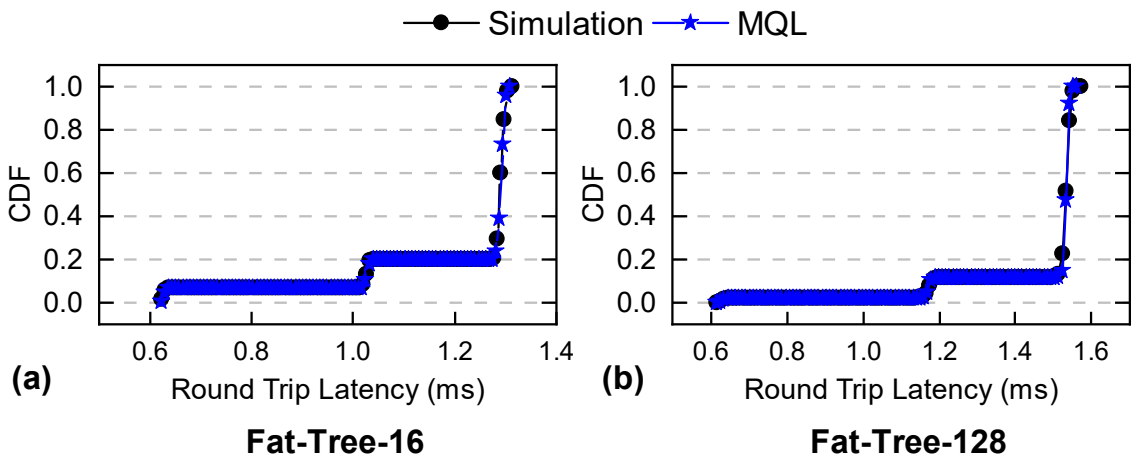


Figure 5.6: A comparison of the round-trip latency (RTT) (in milliseconds) cumulative distribution function (CDF) between simulation and MQL models for all-to-all traffic in (a) Fat-tree-16 and (b) Fat-tree-128, respectively.

variance of 1.2%.

Results of 2-tier fat-tree: In addition to the 3-tier fat-tree, we also evaluated MQL on a proprietary 2-tier fat-tree. The 2-tier version offers a significantly larger level of parallelism by using additional pairs of parallel links between the same edge-core switch pairs. Hence, it is structurally different than the conventional 3-tier fat-tree. Our results indicate that we achieved a comparable accuracy (overall less than 9%) on a 128-node fat-tree when simulating all-to-all, incast, and broadcast synthetic traffic under UDP.

5.4.3 Evaluations with Real-World Traces

Synthetic traffic may often over-constrain the system with traffic that does not represent realistic scenarios. Therefore, we also evaluate a real-world public trace, Anarchy [119]. This trace provides time stamps of the packets injected into the

Table 5.4: Evaluations with the Anarchy [119] trace.

Size	MAPE(%)	avgRTT(w_1)	p99RTT(w_1)
Fat-tree-16	0.23	0.007	0.009
Fat-tree-128	0.46	0.029	0.034
Fat-tree-432	7.86	0.052	0.058
Fat-tree-1024	0.37	0.040	0.047

DCN from 16 hosts, including the packet sizes and destinations. We mapped the source and destinations to a 16-node fat-tree (i.e., four pods). The round-trip times match almost perfectly with ns-3 simulations with 0.09% MAPE even without any ML assistance, as shown in the first row of Table 5.4. Consequently, the RT adds a negligible correction factor, maintaining the accuracy.

To analyze scalability, we also extended this trace into 128 nodes (i.e., eight pods) by replicating each flow and randomly reassigning its source and destination to different nodes. We repeat this flow replication and reassigning process until all 128 nodes are assigned a source or destination. Similarly, we expanded the original trace to 432- and 1024-node fat-trees. Table 5.4 shows the MAPE and RTT normalized Wasserstein distances in fat-tree-128, fat-tree-432, and fat-tree-1024. All of them achieve less than 8% MAPE and very small Wasserstein distances. Furthermore, Figure 5.7 displays the CDF of RTT for real-world traces on fat-tree-16, fat-tree-128, fat-tree-432, and fat-tree-1024. These plots demonstrate that MQL achieves good traffic generality and accurate results.

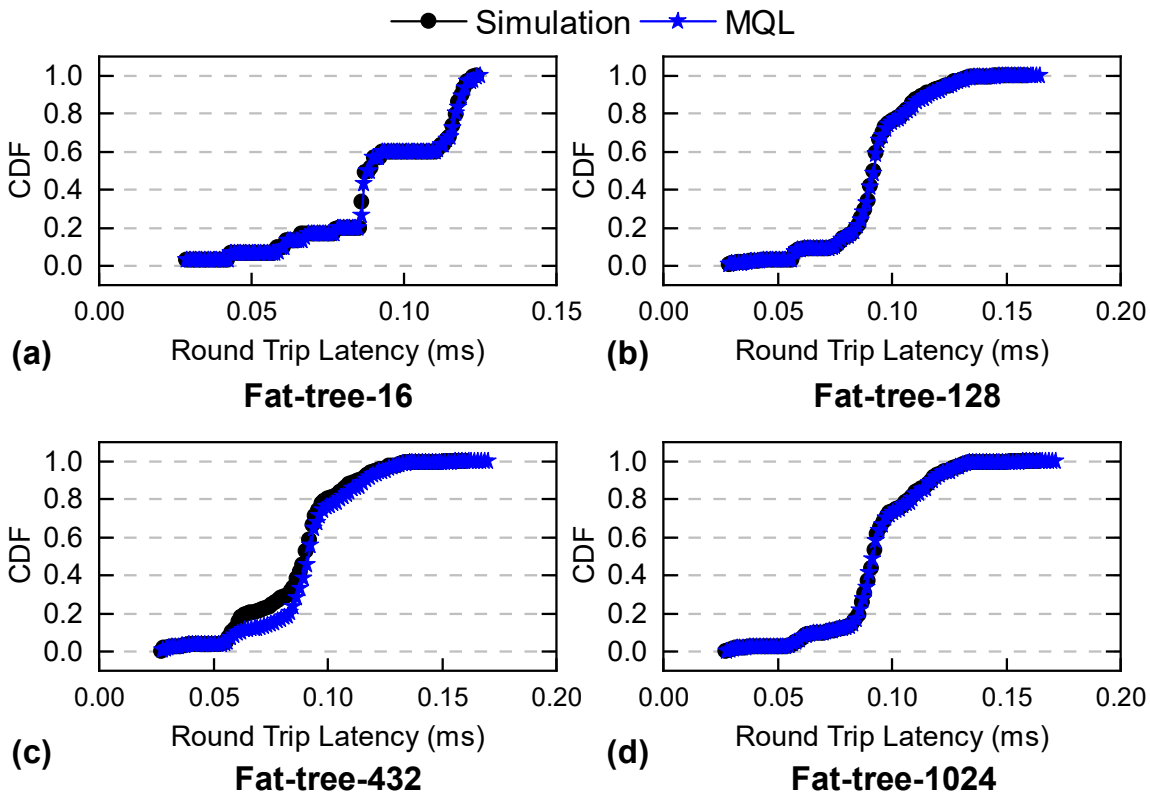


Figure 5.7: A comparison of the cumulative distribution function (CDF) of the round-trip time (RTT) (or latency) in milliseconds between simulation and MQL for the real-world trace Anarchy on (a) fat-tree-16, (b) fat-tree-128, (c) fat-tree-432, and fat-tree-1024, respectively.

5.4.4 Scalability and MQL Execution Time Analysis

This section compares the execution time speedup of the proposed MQL analytical models to corresponding ns-3 simulations (40-second simulation including a 10-second warmup).

Since the fat-tree-1024 all-to-all simulations take an extremely long time to complete, we compare the runtime based on a 10-second-long simulation, whose results are not used for accuracy analysis due to the small number of packets. The

Table 5.5: Execution time of the MQL models, speedup w.r.t simulations A2A: all-to-all, IC: incast, BC: broadcast

Size	Traffic	Exec. Time	Speedup w.r.t Sim	Size	Traffic	Exec. Time	Speedup w.r.t Sim
16	BC	0.002s	22092	432	BC	2.420s	471
16	IC	0.002s	19198	432	IC	2.440s	532
16	A2A	0.028s	29111	432	A2A	13m17s	400
16	Anarchy	0.008s	6375	432	Anarchy	1.046s	9730
128	BC	0.083s	2246	1024	BC	9.159s	220
128	IC	0.090s	2429	1024	IC	36.71s	89
128	A2A	47.14s	1367	1024	A2A	1h49m	115
128	Anarchy	0.634s	456	1024	Anarchy	8.293s	5317

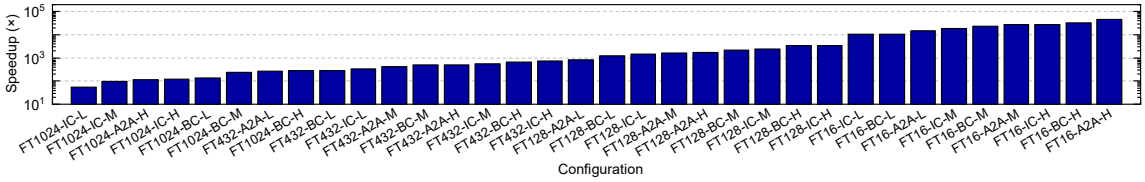


Figure 5.8: Speedup of the proposed MQL framework when compared to ns-3 simulations for different configurations of tree sizes, traffic type, and data rates represented by FT{size}-{traffic type}-{data rate}. Sizes vary between 16, 128, 432, and 1024. Traffic types vary between all-to-all (A2A), incast (IC), and broadcast (BC). Data rates vary between low (L), medium (M), and high (H).

speedup is highest for fat-tree-16 at over four orders of magnitude in the best case and over three on average, as shown in Figure 5.8. Since the number of flows increases with the increase in the tree, the ME model component of MQL takes longer, while the ML component takes a constant amount of time. We emphasize that MQL uses the same regression models across sizes and configurations and achieves similar execution times across workloads. Even for fat-tree-1024, MQL achieves a speedup of 89× or higher. The benefits during rapid DCN design space exploration multiply since simulations need to be repeated for multiple random seeds.

5.4.5 Comparison with State-of-the-Art Approaches

This section compares the proposed MQL approach to three state-of-the-art approaches: DeepQueueNet [138], RouteNet [123], and MimicNet [142]. Table 5.6 lists the normalized Wasserstein distances [133] (lower is better) between the RTT of these approaches and simulations for synthetic traffic in fat-tree-16 and fat-tree-128. We compare the w_1 distance of average RTT indicated by avgRTT(w_1), and the 99th percentile RTT w_1 distances indicated by p99RTT(w_1).

For fat-tree-16 using synthetic traffic with Poisson distribution arrivals, MQL achieves an avgRTT(w_1) better than competitive approaches. The proposed MQL approach also achieves a lower 99th percentile w_1 distance (i.e., higher accuracy) for all configurations. Similarly, MQL outperforms the state-of-the-art approaches in terms of the avgRTT(w_1) and p99RT(w_1) for fat-tree-128, providing the best-in-class performance estimation models. We could not include comparison with larger fat-trees since the other approaches limit their evaluations to networks with 128 nodes. In contrast, we report evaluations with substantially larger network sizes, demonstrating the proposed MQL approach's scalability.

5.5 Related Work

Packet-level simulations, such as ns-3 [3] and OMNet++ [4], provide high accuracy and fine-grain visibility and are versatile, handling different topologies, network protocols, queueing disciplines, and routing algorithms. However, they are too slow to scale to a large number of nodes (1k-10k+ nodes) required for data centers [144,

Table 5.6: A comparison of normalized Wasserstein distances of RTT (avgRTT(w_1)) and 99th percentile RTT (p99RTT(w_1)) between DeepQueueNet [138], MimicNet [142], RouteNet [27] and our proposed MQL framework for synthetic traffic.

avgRTT (w_1)				
Size	**DeepQueueNet**	**RouteNet**	**MimicNet**	**MQL (Ours)**
Fat-tree-16	0.0086	0.6737	0.0090	**0.0025**
Fat-tree-128	0.0133	0.9824	0.0172	**0.0077**
p99RTT (w_1)				
Size	**DeepQueueNet**	**RouteNet**	**MimicNet**	**MQL (Ours)**
Fat-tree-16	0.0145	0.9723	0.0135	**0.0021**
Fat-tree-128	0.0532	0.6397	0.0194	**0.0109**

138]. Therefore, prior work focuses on speeding up network simulation and creating fast network performance models.

Part of the challenge in speeding up network simulators is the almost non-existent opportunities for parallelization. Parsimon [144] observes that large-scale data centers are provisioned such that congestion events rarely occur, and when they do occur, they happen at different points along the path and at different times. Hence, the modeling of the interdependence between queues is a second-order effect. Breaking this dependency enabled the authors to speed up the simulation by decomposing the problem into a large number of parallel, independent single-link simulations. While their approach handles cases with limited congestion, design space exploration also requires identifying solutions that satisfy highly congested workloads, especially for deep learning workloads.

In addition to simulators, prior work creates fast network performance models via pure analytical, pure ML, and hybrid techniques. A well-established performance analysis approach is queuing-theoretic (QT) estimators. They are fast, but

their assumptions, including the Poisson arrival process, FIFO queueing discipline, and queue independence, can lead to unacceptable accuracy in some realistic use cases [9, 12, 76, 78, 77], which we address in this work. QT-based performance analysis approaches have also been applied to networks-on-chip (NoC). However, these models are typically limited to a few 100s of nodes, with a 16x16 2D mesh being the largest [104, 73, 117], unlike our approach that scales to over 1000 nodes.

RouteNet-Erlang [29] observes that Graph Neural Networks (GNNs) capture the underlying graph structure of computer networks. It trains on two small networks (10s) of nodes with various traffic communication patterns and queuing disciplines. However, it requires tuning hyperparameters which is empirical. The tunable hyperparameters question the scalability of the approach to larger networks. Finally, they do not present DCN performance evaluations with network protocols and congestion control algorithms, which are essential for DCNs.

MimicNet [142] uses deep learning to speed up simulation by combining deep learning with simulation. They simulate one cluster and use a deep neural network-trained model to estimate the remaining clusters. The technique relies on symmetry in both the topology and the traffic (e.g., symmetric bisection bandwidth), thereby limiting its applicability to many real-world traffic patterns and topologies.

DeepQueueNet [138] models each device in the network as a DNN that adds a delay to each packet in the incoming packet stream and produces an outgoing stream of packets. It composes these DNNs in one-to-one correspondence with the network structure, but there is no formal guarantee that the DNN can be composed to model the whole network. DeepQueueNet targets packet-level visibility, enabled

by its detailed simulations.

QT-RouteNet [27] combines a queueing theoretic model with a GNN in two steps. First, it runs a simplistic queueing model (M/M/1/B) and extracts features, such as predicted latency, to use in training. Then, it combines with path and link features to train the RouteNet GNN model [123]. Using RouteNet/RouteNet-Erlang at its core, it suffers from similar limitations: tuning the hyperparameters, which is empirical, generalizability, and long traning time. Moreover, it does not present DCN evaluations and learns non-interpretable black-box models, like other purely ML-based approaches.

In summary, existing models suffer from long training times, generalisability and scalability. In contrast, our MQL approach needs no empirical hyperparameters, making the approach generalizable. Moreover, it leverages the data from simulations that are performed to validate the analytical performance models (as part of the regular design flows). To the best of our knowledge, it is the first approach that provides ML assistance to QT-based performance analysis.

5.6 Conclusion

Data centers provide shared access to computing, storage, and memory resources for large organizations that serve millions of users. Efficient, accurate, and scalable performance analysis techniques are critical for rapid design exploration efforts, enabling architectural optimizations. To address these challenges, we proposed MQL, a novel and scalable performance analysis methodology that combines a

queuing theory-based maximum entropy principle and an ML-based assistance technique to correct systematic errors. MQL achieves a minimum of $\sim 80\times$ and up to four orders of magnitude speedup compared to simulations using the discrete-event ns-3 framework. With the evaluations used in this paper, MQL estimates the latencies with less than 3% error on average for DCNs with 16 to 1024 nodes. Future directions include demonstrating the approach on topologies other than fat-tree and validating MQL's modeling accuracy to ensure it is scalable when the queue buffer resources are highly constrained.

In this work, Jie Tong and Shruti Yadav Narayana contributed equally to this project, leading the majority of the research and development efforts. Anish Krishnakumar and Nuriye Yildirim assisted with simulation automation and experimental analysis. Emily Shriver, Mahesh Ketkar, and Umit Y. Ogras supervised the project, providing guidance and oversight. All authors contributed to result discussions and participated in the preparation and review of the final manuscript.

Chapter 6

Conclusion of the Dissertation

This dissertation has addressed the growing need for scalable and efficient simulation and modeling techniques in modern computing systems, particularly as the complexity of deep learning accelerators and data center networks continues to rise. Traditional RTL simulators and packet-level network simulators struggle to meet the demands of performance, scalability, and turnaround time required in both academic and industrial workflows. To overcome these limitations, this research introduced two complementary contributions: a compiler-based framework for accelerating RTL simulation and a machine-learning-assisted analytical modeling approach for network performance estimation.

On the RTL simulation front, the dissertation presented a unified compiler infrastructure that leverages structural parallelism, task graph execution, as well as architecture-aware partitioning. *BatchSim* introduced inter-cycle batching to reduce per-cycle overhead and improve parallelism across time. *ScaleRTL* advanced

this further by detecting and reusing structurally identical modules, significantly reducing redundant code generation and achieving compilation speedups of up to five orders of magnitude. *HeteroRTL* extended the framework to heterogeneous CPU-GPU simulation platforms by introducing target-aware partitioning strategies, enabling fine-grained parallelism and hybrid execution. Together, these contributions establish a robust, extensible simulation pipeline capable of handling modern accelerator-scale RTL designs.

In parallel, the dissertation introduced *MQL*, an analytical performance modeling technique that addresses the scale and speed limitations of packet-level simulators. By combining queuing theory with the Maximum Entropy principle and augmenting it with regression tree learning, MQL provides accurate latency estimations, within 3% of cycle-accurate simulations, while achieving orders-of-magnitude speedups. MQL is well-suited for analyzing large-scale data center topologies and offers detailed observability into queue and tier-level behavior, enabling practical early-stage performance evaluation.

In summary, this dissertation has advanced the state of the art in simulation and modeling for both RTL and network domains. It demonstrates how compiler infrastructure and lightweight machine learning can be leveraged to build scalable, high-performance tools that meet the challenges of contemporary system design. These tools offer immediate benefits in accelerating verification cycles, supporting architecture exploration, and enabling rapid performance feedback.

Future work may explore broader applications and extensions of the ideas presented in this dissertation. For instance, inspired by our success in task graph

parallelism [131, 17, 87, 26, 69, 31, 15, 141, 88, 24, 18, 42, 68, 130, 13, 21, 99, 86, 14, 140, 100, 63, 41, 111, 23, 16, 66, 67, 98, 47, 28, 36, 35, 40, 48, 46, 97, 20, 19, 49, 145, 96, 112, 45, 39, 34, 139, 22, 95, 33, 38, 81, 58, 59, 61, 55, 62, 93, 94, 44, 37, 32, 92, 91, 80, 54, 56, 57, 53, 90, 52, 82, 51], we plan to accelerate RTL simulation using task graphs on both CPUs and GPUs. While the proposed RTL simulation framework and network modeling technique each address distinct challenges, the underlying principles of structural analysis, parallel execution, and lightweight abstraction can be applied to other domains of system design and performance evaluation. As system designs continue to grow in scale and complexity, advancing scalable and efficient simulation methodologies will remain critical. The foundations laid in this dissertation offer a platform for future research that continues to close the gap between simulation fidelity and practical usability.

Bibliography

[1] TPU Architecture. `https://cloud.google.com/tpu/docs/system-architecture-tpu-vm`. [Online; last accessed 30-July-2025.].

[2] CIRCT: Circuit IR Compilers and Tools. `https://circt.llvm.org/`. [Online; last accessed 30-July-2025.].

[3] ns-3 | a discrete-event network simulator. `https://www.nsnam.org`. [Online; last accessed 30-July-2025.].

[4] OMNeT++ Discrete Event Simulator. `https://omnetpp.org`. [Online; last accessed 30-July-2025.].

[5] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. *ACM SIGCOMM Computer Communication Review*, 38(4):63–74, 2008.

[6] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, et al. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro*, 40(4):10–21, 2020.

[7] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, et al. The Rocket Chip Generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 4:6–2, 2016.

[8] S. Beamer and D. Donofrio. Efficiently Exploiting Low Activity Factors to Accelerate RTL Simulation. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[9] D. Bertsekas and R. Gallager. *Data Networks (2nd Ed.)*. Prentice-Hall, Inc., USA, 1992. ISBN 0132009161.

[10] D. Bertsekas and R. Gallager. *Data Networks*. Athena Scientific, 2021.

[11] M. Besta and T. Hoefler. Slim Fly: A Cost Effective Low-Diameter Network Topology. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 348–359, 2014.

[12] G. Bolch, S. Greiner, H. De Meer, and K. S. Trivedi. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*. John Wiley & Sons, 2006.

[13] C. Chang, C.-H. Chiu, B. Zhang, and T.-W. Huang. Incremental Critical Path Generation for Dynamic Graphs. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 771–774, 2024.

[14] C. Chang, T.-W. Huang, D.-L. Lin, G. Guo, and S. Lin. Ink: Efficient Incremental k-Critical Path Generation. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2024.

[15] C. Chang, B. Zhang, C.-H. Chiu, D.-L. Lin, Y.-H. Chung, W.-L. Lee, Z. Guo, Y. Lin, and T.-W. Huang. PathGen: An Efficient Parallel Critical Path Generation Algorithm. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2025.

[16] C.-C. Chang and T.-W. Huang. uSAP: An Ultra-Fast Stochastic Graph Partitioner. In *IEEE High-performance and Extreme Computing Conference (HPEC)*, pages 1–7, 2023.

[17] C.-C. Chang and T.-W. Huang. Statistical Timing Graph Scheduling Algorithm for GPU Computation. In *ACM/IEEE Design Automation Conference (DAC)*, 2025.

[18] C.-C. Chang, B. Zhang, and T.-W. Huang. GSAP: A GPU-Accelerated Stochastic Graph Partitioner. In *ACM International Conference on Parallel Processing (ICPP)*, pages 565–575, 2024.

[19] C.-H. Chiu and T.-W. Huang. Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1388–1389, 2022.

[20] C.-H. Chiu and T.-W. Huang. Composing Pipeline Parallelism using Control Taskflow Graph. In *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 283–284, 2022.

[21] C.-H. Chiu and T.-W. Huang. An Experimental Study of Dynamic Task Graph Parallelism for Large-Scale Circuit Analysis Workloads. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 766–770, 2024.

[22] C.-H. Chiu, D.-L. Lin, and T.-W. Huang. An Experimental Study of SYCL Task Graph Parallelism for Large-Scale Machine Learning Workloads. In *International Workshop of Asynchronous Many-Task systems for Exascale (AMTE)*, pages 468–479, 2021.

[23] C.-H. Chiu, D.-L. Lin, and T.-W. Huang. Programming Dynamic Task Parallelism for Heterogeneous EDA Algorithms. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, pages 1–8, 2023.

[24] C.-H. Chiu, C. Morchdi, Y. Zhou, B. Zhang, C. Chang, and T.-W. Huang. Reinforcement Learning-generated Topological Order for Dynamic Task Graph Scheduling. In *IEEE High-performance and Extreme Computing Conference (HPEC)*, 2024.

[25] S.-T. Chuang, A. Goel, N. McKeown, and B. Prabhakar. Matching Output Queueing with a Combined Input/Output-Queued Switch. *IEEE Journal on Selected Areas in Communications*, 17(6):1030–1039, 1999.

[26] Y.-H. Chung, S. Jiang, W. L. Lee, Y. Zhang, H. Ren, T.-Y. Ho, and T.-W. Huang. SimPart: A Simple Yet Effective Replication-aided Partitioning Algorithm for Logic Simulation on GPU. In *International European Conference on Parallel and Distributed Computing (Euro-Par)*, 2025.

[27] B. K. de Aquino Afonso and L. Berton. QT-RouteNet: Improved GNN Generalization to Larger 5G Networks by Fine-Tuning Predictions from Queuing Theory. *arXiv e-prints*, pages arXiv–2207, 2022.

[28] E. Dzaka, D.-L. Lin, and T.-W. Huang. Parallel And-Inverter Graph Simulation Using a Task-graph Computing System. In *IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSw)*, 2023.

[29] M. Ferriol-Galmés, K. Rusek, J. Suárez-Varela, S. Xiao, X. Shi, X. Cheng, B. Wu, P. Barlet-Ros, and A. Cabellos-Aparicio. RouteNet-Erlang: A Graph Neural Network for Network Performance Evaluation. In *IEEE Conference on Computer Communications*, pages 2018–2027, 2022.

[30] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, et al. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 769–774. IEEE, 2021.

[31] S. Gener, S. Hassan, L. Chang, C. Chakrabarti, T.-W. Huang, U. Ograss, , and A. Akoglu. A Unified Portable and Programmable Framework for Task-Based Execution and Dynamic Resource Management on Heterogeneous Systems. In *ACM International Workshop on Extreme Heterogeneity Solutions (ExHET)*, 2025.

[32] G. Guo, T.-W. Huang, C.-X. Lin, and M. Wong. An Efficient Critical Path Generation Algorithm Considering Extensive Path Constraints. In *ACM/IEEE Design Automation Conference* (*DAC*), pages 1–6, 2020.

[33] G. Guo, T.-W. Huang, Y. Lin, and M. Wong. GPU-accelerated Path-based Timing Analysis. In *IEEE/ACM Design Automation Conference* (*DAC*), pages 721–726, 2021.

[34] G. Guo, T.-W. Huang, Y. Lin, and M. Wong. GPU-accelerated Critical Path Generation with Path Constraints. In *IEEE/ACM International Conference on Computer-Aided Design* (*ICCAD*), pages 1–9, 2021.

[35] G. Guo, T.-W. Huang, Y. Lin, Z. Guo, S. Yellapragada, and M. Wong. A GPU-Accelerated Framework for Path-Based Timing Analysis. In *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems* (*TCAD*), pages 4219–4232, 2023.

[36] G. Guo, T.-W. Huang, and M. D. F. Wong. Fast STA Graph Partitioning Framework for Multi-GPU Acceleration. In *IEEE/ACM Design, Automation and Test in Europe Conference* (*DATE*), 2023.

[37] Z. Guo, T.-W. Huang, and Y. Lin. GPU-accelerated Static Timing Analysis. In *IEEE/ACM International Conference on Computer-Aided Design* (*ICCAD*), 2020.

[38] Z. Guo, T.-W. Huang, and Y. Lin. A Provably Good and Practically Efficient Algorithm for Common Path Pessimism Removal in Large Designs. In *IEEE/ACM Design Automation Conference* (*DAC*), pages 3466–3478, 2021.

[39] Z. Guo, T.-W. Huang, and Y. Lin. HeteroCPPR: Accelerating Common Path Pessimism Removal with Heterogeneous CPU-GPU Parallelism. In *IEEE/ACM International Conference on Computer-Aided Design* (*ICCAD*), 2021.

[40] Z. Guo, T.-W. Huang, and Y. Lin. Accelerating Static Timing Analysis using CPU-GPU Heterogeneous Parallelism. In *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems* (*TCAD*), pages 4973–4984, 2023.

[41] Z. Guo, T.-W. Huang, J. Zhou, C. Zhuo, Y. Lin, R. Wang, and R. Huang. Heterogeneous Static Timing Analysis with Advanced Delay Calculator. In *IEEE/ACM Design, Automation and Test in Europe Conference* (*DATE*), 2024.

[42] Z. Guo, Z. Zhang, W. Li, T.-W. Huang, X. Shi, Y. Du, Y. Lin, R. Wang, and R. Huang. HeteroExcept: Heterogeneous Engine for General Timing Path Exception Analysis. In *IEEE/ACM International Conference on Computer-aided Design* (*ICCAD*), pages 1–9, 2024.

[43] C. Hopps and D. Thaler. Multipath Issues in Unicast and Multicast Next-Hop Selection. RFC 2991, Nov. 2000. URL `https://www.rfc-editor.org/info/rfc2991`.

[44] T.-W. Huang. A General-purpose Parallel and Heterogeneous Task Programming System for VLSI CAD. In *IEEE/ACM International Conference on Computer-aided Design* (*ICCAD*), pages 1–2, 2020.

[45] T.-W. Huang. TFProf: Profiling Large Taskflow Programs with Modern D3 and C++. In *IEEE International Workshop on Programming and Performance Visualization Tools* (*ProTools*), pages 1–6, 2021.

[46] T.-W. Huang. Enhancing the Performance Portability of Heterogeneous Circuit Analysis Programs. In *IEEE High-Performance Extreme Computing Conference* (*HPEC*), 2022.

[47] T.-W. Huang. qTask: Task-parallel Quantum Circuit Simulation with Incrementality. In *IEEE International Parallel and Distributed Processing Symposium* (*IPDPS*), pages 746–756, 2023.

[48] T.-W. Huang and L. Hwang. Task-parallel Programming with Constrained Parallelism. In *IEEE High-Performance Extreme Computing Conference* (*HPEC*), 2022.

[49] T.-W. Huang and Y. Lin. Concurrent CPU-GPU Task Programming using Modern C++. In *IEEE International Workshop on High-level Parallel Programming Models and Supportive Environments* (*HIPS*), pages 588–597, 2022.

[50] T.-W. Huang and M. Wong. OpenTimer: A High-Performance Timing Analysis Tool. In *IEEE/ACM International Conference on Computer-Aided Design* (*ICCAD*), pages 895–902, 2015.

[51] T.-W. Huang, M. Wong, D. Sinha, K. Kalafala, and N. Venkateswaran. A Distributed Timing Analysis Framework for Large Designs. In *IEEE/ACM Design Automation Conference* (*DAC*), pages 1–6, 2016.

[52] T.-W. Huang, C.-X. Lin, and M. Wong. DtCraft: A Distributed Execution Engine for Compute-intensive Applications. In *IEEE/ACM International Conference on Computer-aided Design* (*ICCAD*), pages 757–764, 2017.

[53] T.-W. Huang, C.-X. Lin, G. Guo, and M. Wong. A General-purpose Distributed Programming System using Data-parallel Streams. In *ACM Multimedia Conference* (*MM*), pages 1360–1363, 2018.

[54] T.-W. Huang, C.-X. Lin, , and M. Wong. Distributed Timing Analysis at Scale. In *ACM/IEEE Design Automation Conference* (*DAC*), pages 1–2, 2019.

[55] T.-W. Huang, C.-X. Lin, G. Guo, and M. Wong. Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++. In *IEEE International Parallel and Distributed Processing Symposium* (*IPDPS*), 2019.

[56] T.-W. Huang, C.-X. Lin, G. Guo, and M. Wong. Essential Building Blocks for Creating an Open-source EDA Project. In *ACM/IEEE Design Automation Conference* (*DAC*), pages 1–4, 2019.

[57] T.-W. Huang, C.-X. Lin, and M. Wong. DtCraft: A High-performance Distributed Execution Engine at Scale. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (*TCAD*), pages 1070–1083, 2019.

[58] T.-W. Huang, G. Guo, C.-X. Lin, and M. D. F. Wong. OpenTimer v2: A New Parallel Incremental Timing Analysis Engine. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (*TCAD*), 2021.

[59] T.-W. Huang, C.-X. Lin, and M. Wong. OpenTimer v2: A Parallel Incremental Timing Analysis Engine. In *IEEE Design and Test* (*DAT*), 2021.

[60] T.-W. Huang, Y. Lin, C.-X. Lin, G. Guo, and M. Wong. Taskflow: A General-purpose Parallel Task Programming System at Scale. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (*TCAD*), 2021.

[61] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. In *IEEE Transactions on Parallel and Distributed Systems* (*TPDS*), pages 1303–1320, 2022.

[62] T.-W. Huang, D.-L. Lin, Y. Lin, and C.-X. Lin. Taskflow: A General-purpose Parallel and Heterogeneous Task Programming System. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (*TCAD*), 2022.

[63] T.-W. Huang, B. Zhang, D.-L. Lin, and C.-H. Chiu. Parallel and Heterogeneous Timing Analysis: Partition, Algorithm, and System. In *ACM International Symposium on Physical Design* (*ISPD*), pages 51–59, 2024.

[64] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, et al. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design* (*ICCAD*), pages 209–216. IEEE, 2017.

[65] L. Jia, Z. Luo, L. Lu, and Y. Liang. TensorLib: A Spatial Accelerator Generation Framework for Tensor Algebra. In *2021 58th ACM/IEEE Design Automation Conference* (*DAC*), pages 865–870. IEEE, 2021.

[66] S. Jiang, T.-W. Huang, and T.-Y. Ho. GLARE: Accelerating Sparse DNN Inference Kernels with Global Memory Access Reduction. In *IEEE High-performance and Extreme Computing Conference (HPEC)*, 2023.

[67] S. Jiang, T.-W. Huang, and T.-Y. Ho. SNICIT: Accelerating Sparse Neural Network Inference via Compression at Inference Time on GPU. In *ACM International Conference on Parallel Processing (ICPP)*, pages 51–61, 2023.

[68] S. Jiang, R. Fu, L. Burgholzer, R. Wille, T.-Y. Ho, and T.-W. Huang. FlatDD: A High-Performance Quantum Circuit Simulator using Decision Diagram and Flat Array. In *ACM International Conference on Parallel Processing (ICPP)*, pages 388–399, 2024.

[69] S. Jiang, Y.-H. Chung, C.-C. Chang, T.-Y. Ho, and T.-W. Huang. BQSim: GPU-accelerated Batch Quantum Circuit Simulation using Decision Diagram. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2025.

[70] N. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles, et al. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In *Proceedings of the 50th annual international symposium on computer architecture*, pages 1–14, 2023.

[71] S. A. Kassing. Static Yet Flexible: Expander Data Center Network Fabrics. Master's thesis, ETH-Zürich, 2017.

[72] D. G. Kendall. Some Problems in the Theory of Queues. *Journal of the Royal Statistical Society: Series B (Methodological)*, 13(2):151–173, 1951.

[73] A. E. Kiasari, Z. Lu, and A. Jantsch. An Analytical Latency Model for Networks-on-Chip. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(1):113–123, 2012.

[74] J. Kim, W. J. Dally, and D. Abts. Flattened Butterfly: A Cost-Efficient Topology for High-Radix Networks. In *Proceedings of the Annual International Symposium on Computer Architecture*, pages 126–137, 2007.

[75] J. Kim, W. Dally, S. Scott, and D. Abts. Cost-Efficient Dragonfly Topology for Large-Scale Systems. *IEEE Micro*, 29(1):33–40, 2009.

[76] D. D. Kouvatsos. Maximum Entropy and the G/G/1/N Queue. *Acta Informatica*, 23(5):545–565, 1986.

[77] D. D. Kouvatsos. Entropy Maximisation and Queueing Network Models. *Annals of Operations Research*, 48(1):63–126, 1994.

[78] D. D. Kouvatsos, P. H. E. Georgatsos, and N. M. Tabet-Aouel. *A Universal Maximum Entropy Algorithm for General Multiple Class Open Networks with Mixed Service Disciplines*. Springer US, Boston, MA, 1989. ISBN 978-1-4613-0533-0. doi: 10.1007/978-1-4613-0533-0_26. URL https://doi.org/10.1007/978-1-4613-0533-0_26.

[79] G. Krishnan, A. A. Goksoy, S. K. Mandal, Z. Wang, C. Chakrabarti, J.-s. Seo, U. Y. Ogras, and Y. Cao. Big-Little Chiplets for In-Memory Acceleration of DNNs: A Scalable Heterogeneous Architecture. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pages 1–9, 2022.

[80] K.-M. Lai, T.-W. Huang, and T.-Y. Ho. A General Cache Framework for Efficient Generation of Timing Critical Paths. In *ACM/IEEE Design Automation Conference (DAC)*, 2019.

[81] K.-M. Lai, T.-W. Huang, P.-Y. Lee, and T.-Y. Ho. ATM: A High Accuracy Extracted Timing Model for Hierarchical Timing Analysis. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 278–283, 2021.

[82] T.-Y. Lai, T.-W. Huang, , and M. Wong. Libabs: An Effective and Accurate Macro-modeling Algorithm for Large Hierarchical Designs. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, pages 1–6, 2017.

[83] K. Lakhotia, M. Besta, L. Monroe, K. Isham, P. Iff, T. Hoefler, and F. Petrini. PolarFly: A Cost-Effective and Flexible Low-Diameter Topology. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 146–160. IEEE Computer Society, 2022.

[84] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[85] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

[86] W. L. Lee, D.-L. Lin, T.-W. Huang, S. Jiang, T.-Y. Ho, Y. Lin, and B. Yu. G-kway: Multilevel GPU-Accelerated k-way Graph Partitioner. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2024.

[87] W.-L. Lee, S. Jiang, D.-L. Lin, C. Chang, B. Zhang, Y.-H. Chung, U. Schlichtmann, T.-Y. Ho, , and T.-W. Huang. iG-kway: Incremental k-way Graph Partitioning on GPU. In *ACM/IEEE Design Automation Conference (DAC)*, 2025.

[88] W.-L. Lee, D.-L. Lin, C.-H. Chiu, U. Schlichtmann, and T.-W. Huang. HyperG: Multilevel GPU-Accelerated k-way Hypergraph Partitioner. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2025.

[89] S. Legtchenko, H. Williams, K. Razavi, A. Donnelly, R. Black, A. Douglas, N. Cheriere, D. Fryer, K. Mast, A. D. Brown, A. Klimovic, A. Slowey, and

A. Rowstron. Understanding Rack-Scale Disaggregated Storage. In *9th USENIX Workshop on Hot Topics in Storage and File Systems* (*HotStorage 17*), 2017.

[90] C.-X. Lin, T.-W. Huang, T. Yu, and M. Wong. A Distributed Power Grid Analysis Framework from Sequential Stream Graph. In *ACM Great Lakes Symposium on VLSI* (*GLSVLSI*), pages 183–188, 2018.

[91] C.-X. Lin, T.-W. Huang, G. Guo, and M. Wong. An Efficient and Composable Parallel Task Programming Library. In *IEEE High-performance and Extreme Computing Conference* (*HPEC*), 2019.

[92] C.-X. Lin, T.-W. Huang, G. Guo, and M. Wong. A Modern C++ Parallel Task Programming Library. In *ACM Multimedia Conference* (*MM*), pages 2284–2287, 2019.

[93] C.-X. Lin, T.-W. Huang, and M. Wong. An Efficient Work-Stealing Scheduler for Task Dependency Graph. In *IEEE International Conference on Parallel and Distributed Systems* (*ICPADS*), pages 64–71, 2020.

[94] D.-L. Lin and T.-W. Huang. A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism. In *IEEE High-performance and Extreme Computing Conference* (*HPEC*), 2020.

[95] D.-L. Lin and T.-W. Huang. Efficient GPU Computation using Task Graph Parallelism. In *European Conference on Parallel and Distributed Computing* (*Euro-Par*), pages 435–450, 2021.

[96] D.-L. Lin and T.-W. Huang. Accelerating Large Sparse Neural Network Inference using GPU Task Graph Parallelism. In *IEEE Transactions on Parallel and Distributed Systems* (*TPDS*), pages 3041–3052, 2022.

[97] D.-L. Lin, H. Ren, Y. Zhang, B. Khailany, and T.-W. Huang. From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus. In *ACM International Conference on Parallel Processing* (*ICPP*), pages 1–12, 2022.

[98] D.-L. Lin, Y. Zhang, H. Ren, S.-H. Wang, B. Khailany, and T.-W. Huang. GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2023.

[99] D.-L. Lin, T.-W. Huang, J. S. Miguel, and U. Ogras. TaroRTL: Accelerating RTL Simulation using Coroutine-based Heterogeneous Task Graph Scheduling. In *International European Conference on Parallel and Distributed Computing (Euro-Par)*, pages 151–166, 2024.

[100] S. Lin, G. Guo, T.-W. Huang, W. Sheng, E. Young, and M. Wong. GCS-Timer: GPU-Accelerated Current Source Model Based Static Timing Analysis. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2024.

[101] J. D. Little and S. C. Graves. Little's Law. In *Building Intuition*, pages 81–100. Springer, 2008.

[102] L. Liu, W. Cao, S. Sahin, Q. Zhang, J. Bae, and Y. Wu. Memory Disaggregation: Research Problems and Opportunities. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pages 1664–1673, 2019.

[103] S. K. Mandal, R. Ayoub, M. Kishinevsky, and U. Y. Ogras. Analytical Performance Models for NoCs with Multiple Priority Traffic Classes. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–21, 2019.

[104] S. K. Mandal, R. Ayoub, M. Kishinevsky, M. M. Islam, and U. Y. Ogras. Analytical Performance Modeling of NoCs under Priority Arbitration and Bursty Traffic. *IEEE Embedded Systems Letters*, 13(3):98–101, 2020.

[105] S. K. Mandal, A. Krishnakumar, R. Ayoub, M. Kishinevsky, and U. Y. Ogras. Performance Analysis of Priority-Aware NoCs with Deflection Routing under Traffic Congestion. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–9, 2020.

[106] S. K. Mandal, J. Tong, R. Ayoub, M. Kishinevsky, A. Abousamra, and U. Y. Ogras. Theoretical Analysis and Evaluation of NoCs with Weighted Round-Robin Arbitration. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2021.

[107] S. K. Mandal, S. Y. Narayana, R. Ayoub, M. Kishinevsky, A. Abousamra, and U. Y. Ogras. Fast Performance Analysis for NoCs With Weighted Round-Robin Arbitration and Finite Buffers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 31(5):670–683, 2023.

[108] P. Maniotis, L. Schares, B. G. Lee, M. A. Taubenblatt, and D. M. Kuchta. Toward Lower-Diameter Large-Scale HPC and Data Center Networks with Co-Packaged Optics. *Journal of Optical Communications and Networking*, 13(1): A67–A77, 2020.

[109] R. Marculescu, U. Y. Ogras, L.-S. Peh, N. E. Jerger, and Y. Hoskote. Outstanding Research Problems in NoC Design: System, Microarchitecture, and Circuit Perspectives. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 28(1):3–21, 2008.

[110] C. Minkenberg, N. Farrington, A. Zilkie, D. Nelson, C. P. Lai, D. Brunina, J. Byrd, B. Chowdhuri, N. Kucharewski, K. Muth, A. Nagra, G. Rodriguez, D. Rubi, T. Schrans, P. Srinivasan, Y. Wang, C. Yeh, and A. Rickman. Reimagining Datacenter Topologies with Integrated Silicon Photonics. *Journal of Optical Communications and Networking*, 10(7):126–139, 2018. doi: 10.1364/JOCN.10.00B126.

[111] C. Morchdi, C.-H. Chiu, Y. Zhou, and T.-W. Huang. A Resource-efficient Task Scheduling System using Reinforcement Learning. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 89–95, 2024.

[112] M. Mower, L. Majors, and T.-W. Huang. Taskflow-San: Sanitizing Erroneous Control Flow in Taskflow Programs. In *IEEE Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, 2021.

[113] S. Y. Narayana, S. K. Mandal, R. Ayoub, M. M. Islam, M. Kishinevsky, and U. Y. Ogras. Fast Analysis Using Finite Queuing Model for Multilayer NoCs. *IEEE Design & Test*, 40(6):112–124, 2023.

[114] S. Y. Narayana, E. Shriver, K. O'Neal, N. Yildirim, K. Begaliyeva, and U. Y. Ogras. Similarity-Based Fast Analysis of Data Center Networks. *IEEE Design & Test*, 40(6):100–111, 2023.

[115] S. Y. Narayana, J. Tong, A. Krishnakumar, N. Yildirim, E. Shriver, M. Ketkar, and U. Y. Ogras. MQL: ML-Assisted Queuing Latency Analysis for Data Center Networks. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 50–60. IEEE, 2023.

[116] U. Y. Ogras and R. Marculescu. *Modeling, analysis and optimization of network-on-chip communication architectures*, volume 184. Springer Science & Business Media, 2013.

[117] U. Y. Ogras, P. Bogdan, and R. Marculescu. An Analytical Approach for Network-on-Chip Performance Analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(12):2001–2013, 2010. doi: 10.1109/TCAD.2010.2061613.

[118] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[119] A. Petlund, P. Halvorsen, P. F. Hansen, T. Lindgren, R. Casais, and C. Griwodz. Network Traffic from Anarchy Online: Analysis, Statistics and Applications: A Server-Side Traffic Trace. In *Proceedings of the 3rd Multimedia Systems Conference*, pages 95–100, 2012.

[120] J. Postel. The TCP Maximum Segment Size and Related Topics. RFC 879, Nov. 1983. URL `https://www.rfc-editor.org/info/rfc879`.

[121] G. Pujolle and W. Ai. A Solution for Multiserver and Multiclass Open Queueing Networks. *INFOR: Information Systems and Operational Research*, 24(3): 221–230, 1986.

[122] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 58–70. IEEE, 2020.

[123] K. Rusek, J. Suárez-Varela, P. Almasan, P. Barlet-Ros, and A. Cabellos-Aparicio. RouteNet: Leveraging Graph Neural Networks for Network Modeling and Optimization in SDN. *IEEE Journal on Selected Areas in Communications*, 38(10):2260–2270, 2020.

[124] H. Sharma, S. K. Mandal, J. R. Doppa, U. Y. Ogras, and P. P. Pande. SWAP: A Server-Scale Communication-Aware Chiplet-Based Manycore PIM Accelerator. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(11):4145–4156, 2022.

[125] H. Sharma, S. K. Mandal, J. R. Doppa, U. Ogras, and P. P. Pande. Achieving Datacenter-scale Performance through Chiplet-based Manycore Architectures. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2023.

[126] H. Sharma, L. Pfromm, R. O. Topaloglu, J. R. Doppa, U. Y. Ogras, A. Kalyanraman, and P. P. Pande. Florets for Chiplets: Data Flow-aware High-Performance and Energy-efficient Network-on-Interposer for CNN Inference Tasks. *ACM Transactions on Embedded Computing Systems*, 22(5s):1–21, 2023.

[127] W. Snyder. Verilator 4.0: Open Simulation Goes Multithreaded. In *ORConf*, 2018.

[128] A. Sriraman and A. Dhanotia. Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale. In *Proceedings of*

the International Conference on Architectural Support for Programming Languages and Operating Systems, pages 733–750, 2020.

[129] M. Taubenblatt, P. Maniotis, and A. Tantawi. Optics enabled networks and architectures for data center cost and power efficiency. *Journal of Optical Communications and Networking*, 14(1):A41–A49, 2022.

[130] J. Tong, L. Chang, U. Y. Ogras, and T.-W. Huang. BatchSim: Parallel RTL Simulation using Inter-cycle Batching and Task Graph Parallelism. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 789–793, 2024.

[131] J. Tong, W.-L. Lee, U. Y. Ogras, and T.-W. Huang. Scalable Code Generation for RTL Simulation of Deep Learning Accelerators with MLIR. In *International European Conference on Parallel and Distributed Computing (Euro-Par)*, 2025.

[132] T. Trevisan Jost, A. Thangamani, R. Colin, V. Loechner, S. Genaud, and B. Bramas. GPU Code Generation of Cardiac Electrophysiology Simulation with MLIR. In *European Conference on Parallel Processing*, pages 549–563. Springer, 2023.

[133] L. N. Vaserstein. Markov Processes over Denumerable Products of Spaces, Describing Large Systems of Automata. *Problemy Peredachi Informatsii*, 5(3): 64–72, 1969.

[134] H. Wang and S. Beamer. RepCut: Superlinear Parallel RTL Simulation with Replication-Aided Partitioning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 572–585, 2023.

[135] H. Wang, T. Nijssen, and S. Beamer. Don't Repeat Yourself! Coarse-Grained Circuit Deduplication to Accelerate RTL Simulation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, pages 79–93, 2024.

[136] Y. Wang, D. Dong, and F. Lei. Understanding Node Connection Modes in Multi-Rail Fat-tree. *Journal of Parallel and Distributed Computing*, 2022.

[137] K. Wen, P. Samadi, S. Rumley, C. P. Chen, Y. Shen, M. Bahadori, K. Bergman, and J. Wilke. Flexfly: Enabling a Reconfigurable Dragonfly through Silicon Photonics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 166–177, 2016.

[138] Q. Yang, X. Peng, L. Chen, L. Liu, J. Zhang, H. Xu, B. Li, and G. Zhang. DeepQueueNet: Towards Scalable and Generalized Network Performance Estimation with Packet-Level Visibility. In *Proceedings of the ACM SIGCOMM Conference*, pages 441–457, 2022.

[139] Y. Zamani and T.-W. Huang. A High-Performance Heterogeneous Critical Path Analysis Framework. In *IEEE High-Performance Extreme Computing Conference (HPEC)*, 2021.

[140] B. Zhang, D.-L. Lin, C. Chang, C.-H. Chiu, B. Wang, W. L. Lee, C.-C. Chang, D. Fang, and T.-W. Huang. G-PASTA: GPU Accelerated Partitioning Algorithm for Static Timing Analysis. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2024.

[141] B. Zhang, C. Chang, C.-H. Chiu, D.-L. Lin, Y. Sui, C.-C. Chang, Y.-H. Chung, W.-L. Lee, Z. Guo, Y. Lin, and T.-W. Huang. iTAP: An Incremental Task Graph Partitioner for Task-parallel Static Timing Analysis. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2025.

[142] Q. Zhang, K. K. Ng, C. Kazer, S. Yan, J. Sedoc, and V. Liu. MimicNet: Fast Performance Estimates for Data Center Networks with Machine Learning. In *Proceedings of the ACM SIGCOMM Conference*, pages 287–304, 2021.

[143] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. In *Fourth Workshop on Computer Architecture Research with RISC-V*, volume 5, pages 1–7, 2020.

[144] K. Zhao, P. Goyal, M. Alizadeh, and T. E. Anderson. Scalable Tail Latency Estimation for Data Center Networks. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 685–702, 2023.

[145] K. Zhou, Z. Guo, T.-W. Huang, and Y. Lin. Efficient Critical Paths Search Algorithm using Mergeable Heap. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 190–195, 2022.

[146] K. Zhou, Y. Liang, Y. Lin, R. Wang, and R. Huang. Khronos: Fusing Memory Access for Improved Hardware RTL Simulation. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 180–193, 2023.

[147] Z. Zhu, A. X. Liu, F. Zhang, and F. Chen. FPGA Resource Pooling in Cloud Computing. *IEEE Transactions on Cloud Computing*, 9(02):610–626, 2021.