Scalable Code Generation for RTL Simulation of Deep Learning Accelerators With MLIR

Jie Tong, Wan-Luan Lee, Umit Yusuf Ogras, and Tsung-Wei Huang(✉)

University of Wisconsin–Madison, Madison, WI, USA
{jtong36,wanluan.lee,uogras,tsung-wei.huang}@wisc.edu

Abstract. 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.

Keywords: RTL simulation · MLIR · GPU code generation

1 Introduction

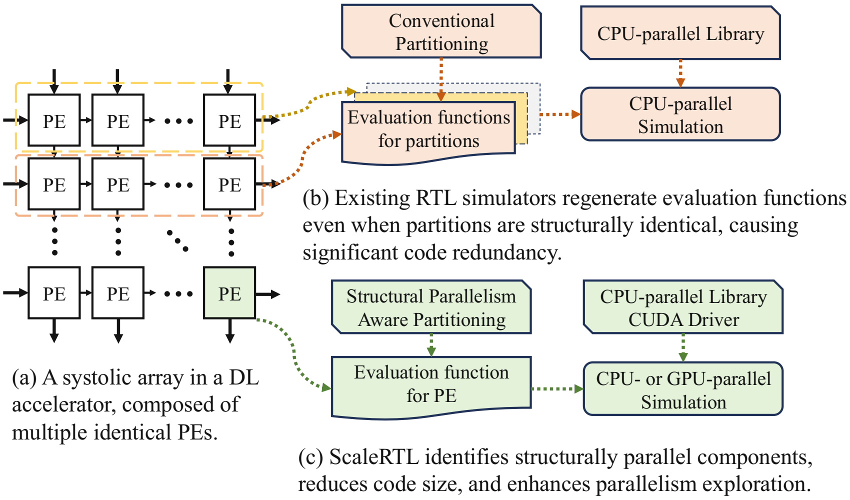
ASIC accelerators play a critical role in boosting the performance of deep learning backbone applications, such as GEMM, DNNs, and transformers in the modern AI industry [2]. To validate the functionality of a hardware design before physical implementation, *Register Transfer Level* (RTL) simulation 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 [10]. Thus, accelerating RTL simulation is critical for managing increasing design complexity and meeting short time-to-market demands in the accelerator market.

W. Nagel et al. (Eds.): Euro-Par 2025, LNCS 15900, pp. 1–14, 2025.
https://doi.org/10.1007/978-3-031-99854-6_2

To mitigate the runtime challenge of RTL simulation, researchers have proposed various parallel RTL simulation algorithms. For example, Verilator [12], 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 [15] converts RTL source code to FIRRTL [5] and introduces a replication-aided partitioning algorithm to reduce synchronization overhead in parallel simulation. Khronos [17] and BatchSim [13] parse RTL designs using MLIR and generate evaluation functions through LLVM IR. RTLflow [10], 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* [12], 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 Fig. 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 [12] and Dedup [16] offer limited support for deduplication in RTL simulation code generation. Verilator [12] focuses on small SystemVerilog statements and does not handle full structural components, while Dedup [16] targets multi-core SoC-style designs that emphasize heterogeneity and connectivity, rather than scalability of deep learning accelerators.

To tackle these challenges, we introduce *ScaleRTL*, a scalable code generation flow that automatically generates optimized parallel RTL simulators for deep learning accelerators. Figure 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


PE
PE
PE
Conventional Partitioning
Evaluation functions for partitions
CPU-parallel Library
CPU-parallel Simulation
Structural Parallelism Aware Partitioning
Evaluation function for PE
CPU-parallel Library CUDA Driver
CPU- or GPU-parallel Simulation

(a) A systolic array in a DL accelerator, composed of multiple identical PEs.

(b) Existing RTL simulators regenerate evaluation functions even when partitions are structurally identical, causing significant code redundancy.

(c) ScaleRTL identifies structurally parallel components, reduces code size, and enhances parallelism exploration.

Fig. 1. Comparison of existing RTL simulation approaches and ScaleRTL.

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) project [7], 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 accelerating 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.

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

2 Background and Motivation

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 [12], Khronos [17], and BatchSim [13], 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. This evaluation process is repeated thousands to millions of times to validate the design's functionality [10].

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 [10] 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 [3,4,9,12,13,15,17]. 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.

2.2 MLIR

MLIR [7] 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 [5] 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 source code, allowing for greater opportunities to exploit data parallelism.

3 ScaleRTL

Figure 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 [7] 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 [8] to further enhance the performance of GPU-based simulation.

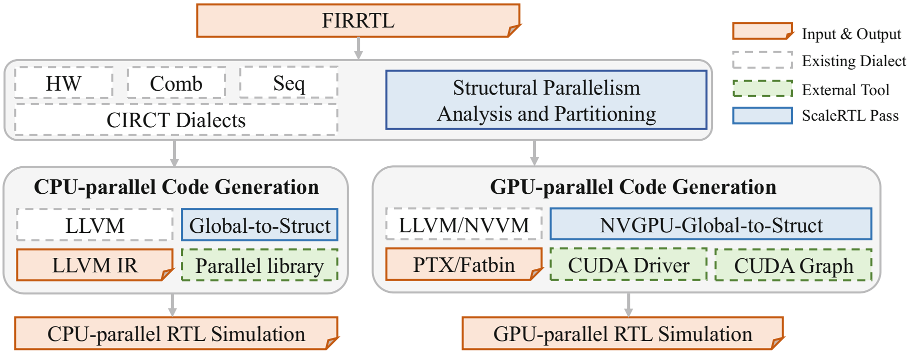


Fig. 2. Overview of ScaleRTL.

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 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 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.

3.2 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 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 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 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. 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 GPU-Parallel Simulation Code Generation

Figure 3 illustrates the GPU code generation process in ScaleRTL. Unlike prior work [14], which uses the GPU dialect to generate GPU-based simulation code, 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 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.

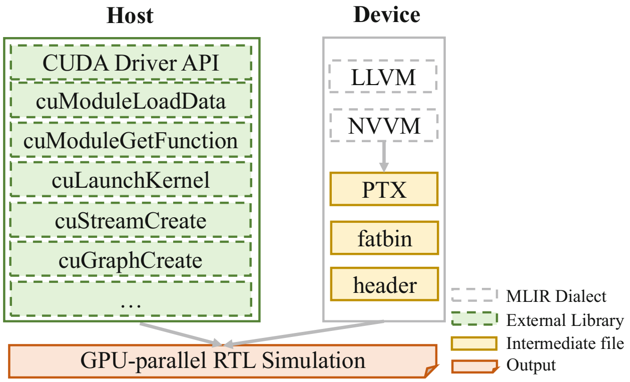


Fig. 3. GPU code generation flow in ScaleRTL.

```
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 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 [8] to merge successive kernel calls into a single simulation task graph to reduce kernel launch overhead and improve GPU-based simulation performance.

4 Experimental Results

We evaluate the performance of ScaleRTL on four deep learning accelerator RTL designs: Conv2D [6], GEMM [6], Gemmini [2], and SIGMA [11]. 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 [12], Khronos [17], and BatchSim [13] 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 [10], as it is designed for batch-stimulus scenarios, which is a different scope of work. We also exclude ESSENT [1] and its successors [15, 16], as they encounter out-of-memory errors during code generation. To ensure consistency, all simulation results are averaged over five runs.

4.1 Code Generation and Compilation Results

Table 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 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$_\mathrm{C}$ and ScaleRTL$_\mathrm{G}$ 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 runtime.

To demonstrate the scalability of ScaleRTL's compilation time, Fig. 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.

Table 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

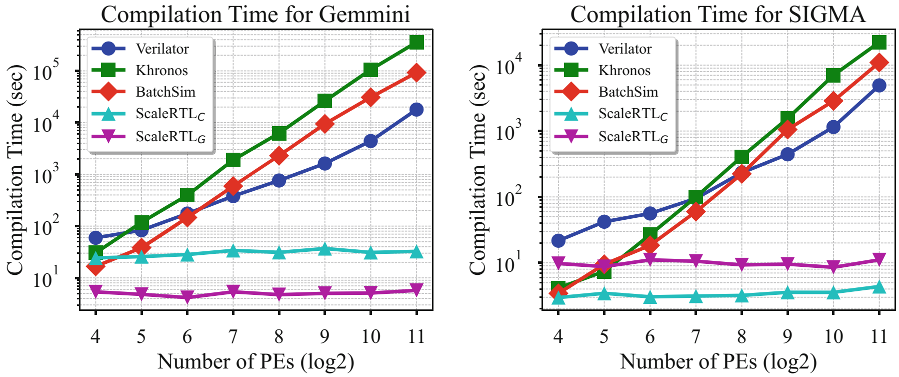


Fig. 4. Compilation time of Gemmini and SIGMA accelerators among different RTL simulators as the number of PEs increases exponentially.

4.2 Overall Simulation Performance Comparison

Figure 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 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

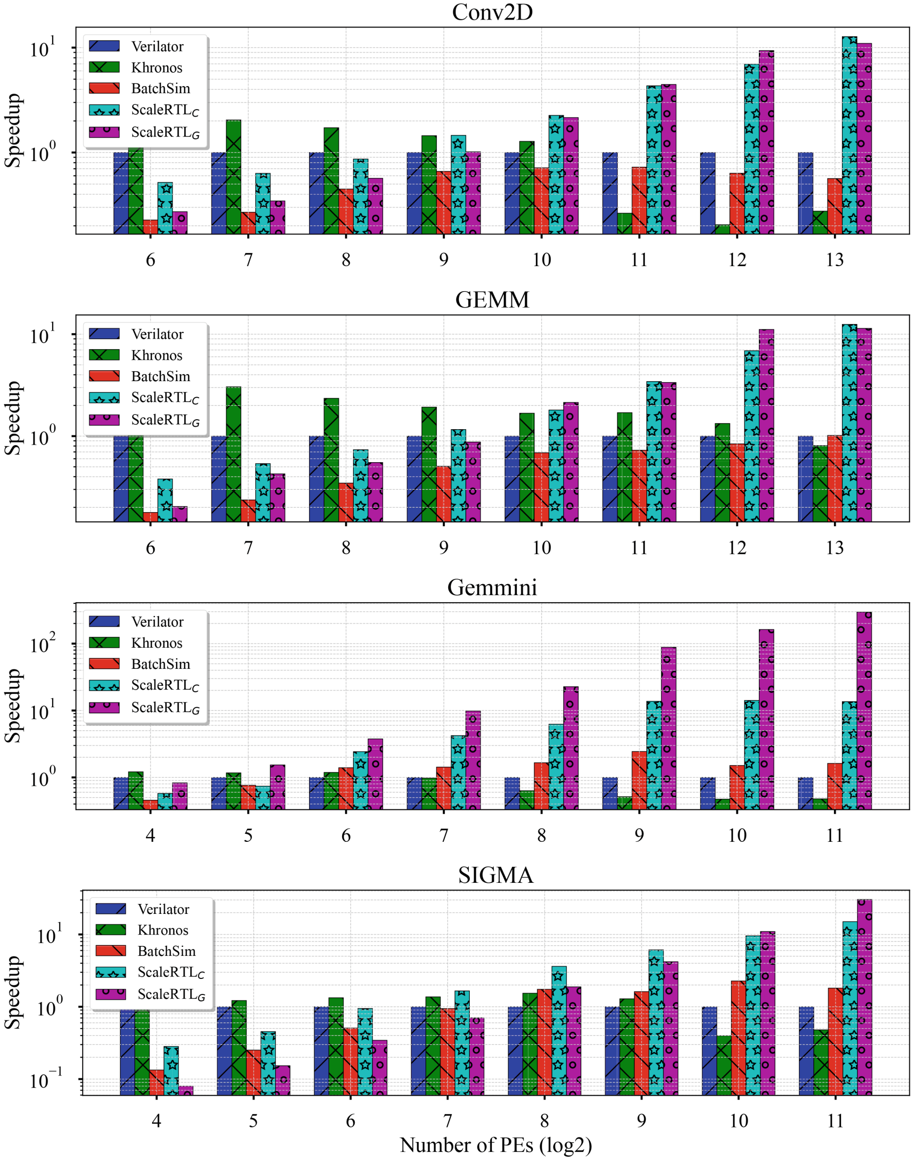


Fig. 5. Overall simulation speedup of Conv2D, GEMM, Gemmini, and SIGMA on different RTL simulators as the number of PEs increases exponentially. Speedup is measured relative to the baseline Verilator.

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.

4.3 CPU and GPU Simulation Runtime Analysis

Figure 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 s, 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.

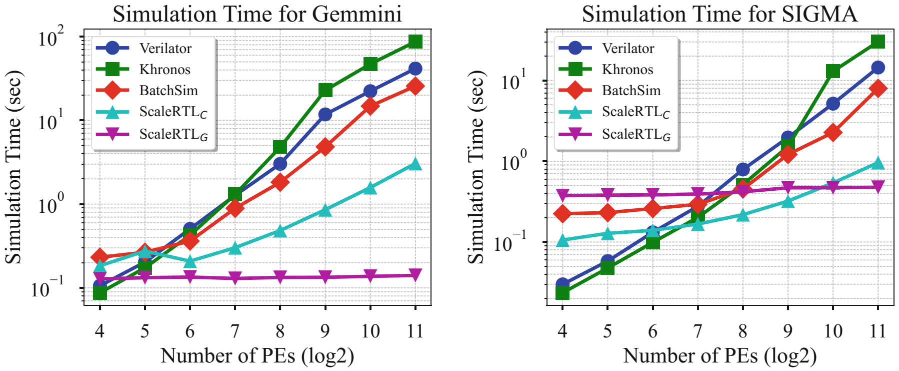
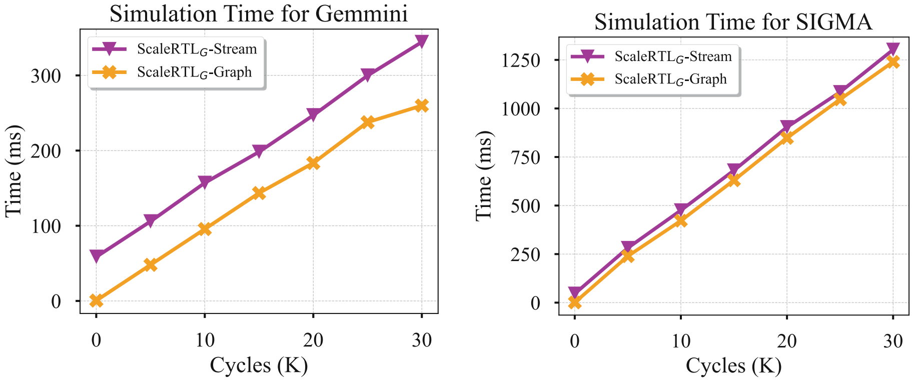


Fig. 6. Simulation time of Gemmini and SIGMA accelerators on different RTL simulators as the number of PEs increases exponentially.

4.4 Performance Result of CUDA Graph

Figure 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 7a and 7b 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) Gemmini (2^{11} PEs) (b) SIGMA (2^{11} PEs)

Fig. 7. Simulation results comparing CUDA Stream-based and CUDA Graph-based execution.

5 Conclusion

This paper 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. Future work includes integrating ScaleRTL with MLIR and CIRCT to support the compiler community in exploring RTL simulation research.

Acknowledgments. This project is supported by NSF grants 2235276, 2349144, 2349143, 2349582, and 2349141.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Beamer, S., Donofrio, D.: Efficiently exploiting low activity factors to accelerate RTL simulation. In: 2020 57th ACM/IEEE DAC, pp. 1–6. IEEE (2020)
2. Genc, H., et al.: Gemmini: enabling systematic deep-learning architecture evaluation via full-stack integration. In: DAC 2021, pp. 769–774. IEEE (2021)
3. Huang, T.W., Lin, D.L., Lin, C.X., Lin, Y.: TaskFlow: a lightweight parallel and heterogeneous task graph computing system. IEEE Trans. Parallel Distr. Syst. **33**(6) (TPDS) (2022)
4. Huang, T.W., Wong, M.: OpenTimer: a high-performance timing analysis tool. In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD) (2015)
5. Izraelevitz, A., et al.: Reusability is FIRRTL ground: hardware construction languages, compiler frameworks, and transformations. In: 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 209–216. IEEE (2017)
6. Jia, L., Luo, Z., Lu, L., Liang, Y.: TensorLib: a spatial accelerator generation framework for tensor algebra. In: 2021 58th ACM/IEEE Design Automation Conference (DAC), pp. 865–870. IEEE (2021)
7. Lattner, C., et al.: MLIR: scaling compiler infrastructure for domain specific computation. In: CGO 2021, pp. 2–14. IEEE (2021)
8. Lin, D.L., Huang, T.W.: Efficient GPU computation using task graph parallelism. In: European Conference on Parallel and Distributed Computing (Euro-Par) (2021)
9. Lin, D.L., Huang, T.W., Miguel, J.S., Ogras, U.: TaroRTL: accelerating RTL simulation using coroutine-based heterogeneous task graph scheduling. In: Euro-Par 2024 (2024)
10. Lin, D.L., Ren, H., Zhang, Y., Khailany, B., Huang, T.W.: From RTL to CUDA: a GPU acceleration flow for RTL simulation with batch stimulus. In: ACM International Conference on Parallel Processing (ICPP) (2022)
11. Qin, E., et al.: SIGMA: a sparse and irregular GEMM accelerator with flexible interconnects for DNN training. In: 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 58–70. IEEE (2020)
12. Snyder, W.: Verilator: open simulation goes multithreaded. In: ORConf (2018)
13. Tong, J., Chang, L., Ogras, U.Y., Huang, T.W.: BatchSim: parallel RTL simulation using inter-cycle batching and task graph parallelism. In: 2024 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), pp. 789–793. IEEE (2024)
14. Trevisan Jost, T., Thangamani, A., Colin, R., Loechner, V., Genaud, S., Bramas, B.: GPU code generation of cardiac electrophysiology simulation with MLIR. In: European Conference on Parallel Processing, pp. 549–563. Springer (2023)
15. Wang, H., Beamer, S.: RepCut: superlinear parallel RTL simulation with replication-aided partitioning. In: Proceedings of the 28th ACM International Conference on ASPLOS, vol. 3, pp. 572–585 (2023)
16. Wang, H., Nijssen, T., Beamer, S.: 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, vol. 4, pp. 79–93 (2024)
17. Zhou, K., Liang, Y., Lin, Y., Wang, R., Huang, R.: Khronos: fusing memory access for improved hardware RTL simulation. In: Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 180–193 (2023)