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

Jie Tong
University of Wisconsin-Madison
Madison, USA
jtong36@wisc.edu

Zhengxiong Li
University of Wisconsin-Madison
Madison, USA
zhengxiong.li@wisc.edu

Umit Yusuf Ogras
University of Wisconsin-Madison
Madison, USA
uogras@wisc.edu

Tsung-Wei Huang
University of Wisconsin-Madison
Madison, USA
tsung-wei.huang@wisc.edu

***Abstract*— 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.**

I. Introduction

Domain-specific accelerators are essential for enhancing the performance of deep learning workloads, including DNNs and transformer models, in today's AI-driven industry [1, 2]. 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 [3]. 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 [4]. 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 [5], 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 multithreaded execution. RTLflow [4], 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 [6] converts RTL designs into FIRRTL [7] and introduces a replication-aided partitioning algorithm to reduce synchronization overhead during parallel simulation. Khronos [8] and BatchSim [9] utilize the MLIR framework to analyze RTL designs and generate evaluation functions in LLVM IR. Dedup [10] introduces deduplication and targets the structural patterns of multi-core SoC designs. ScaleRTL [11] introduces a scalable deduplication and code generation approach for RTL simulation of deep learning accelerators. 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 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 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 [12], 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.

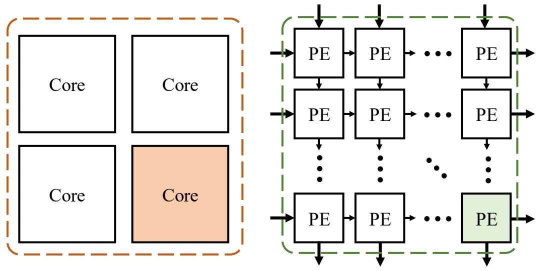

Fig. 1: Schematic of a deep learning accelerator SoC composed of multicore host CPUs and a systolic array of duplicated PEs. The heterogeneous architecture enables CPU-GPU hybrid simulation, while the duplicated components offer opportunities for simulation code reuse and reduction.

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.

II. Background and Motivation

A. 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 [5], Khronos [8], and BatchSim [9] 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 [4, 8].

While these simulators effectively capture functional behavior, they often suffer from significant code redundancy due to a lack of structural awareness. Verilator [5] and Dedup [10] 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.

B. MLIR

MLIR [12] is a modern compiler infrastructure developed to streamline the creation of new compiler components within the LLVM ecosystem [13]. 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 [7] and CIRCT [14], intermediate representations specifically designed to model RTL semantics directly.

III. HeteroRTL

Figure 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 [12] and CIRCT [14], 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.

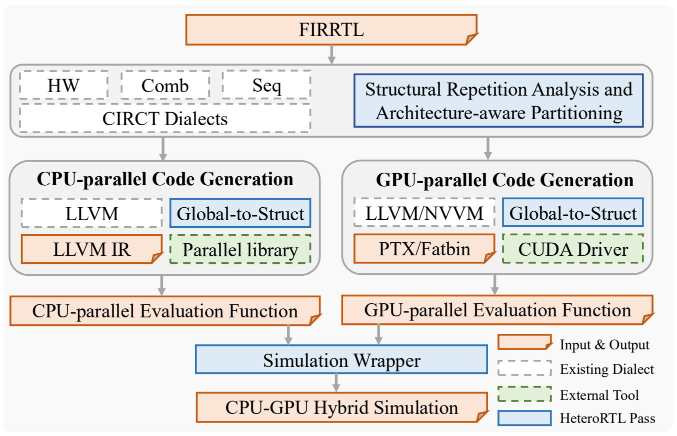
FIRRTL
HW
Comb
Seq
CIRCT Dialects
Structural Repetition Analysis and Architecture-aware Partitioning
CPU-parallel Code Generation
LLVM
Global-to-Struct
LLVM IR
Parallel library
GPU-parallel Code Generation
LLVM/NVVM
Global-to-Struct
PTX/Fatbin
CUDA Driver
CPU-parallel Evaluation Function
GPU-parallel Evaluation Function
Simulation Wrapper
CPU-GPU Hybrid Simulation
Input & Output
Existing Dialect
External Tool
HeteroRTL Pass

Fig. 2: Overview of HeteroRTL.

A. 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 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 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.

B. 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 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 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 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 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.

C. 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 [15] 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 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: Example GPU-based PE evaluation code in LLVM and NVVM Dielact.

D. 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.

IV. Experimental Evaluation

We evaluate the performance of HeteroRTL on four deep learning SoC RTL designs, each integrating multiple RISC-

V Rocket [16] cores as the processing host and paired with one of the following accelerators: Conv2D [17], GEMM [17], Gemmini [1], or SIGMA [18]. 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.

A. Baseline

We compare HeteroRTL against three CPU-based RTL simulators: Verilator [5], Khronos [8], and BatchSim [9]. 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 [4], a GPU-based simulator specifically optimized for batch-driven workloads. ESSENT [19] and its successors [6, 10] 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.

B. Code Generation and Compilation Results

Table I 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.

TABLE I: 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

To demonstrate HeteroRTL's scalability, Figure 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.

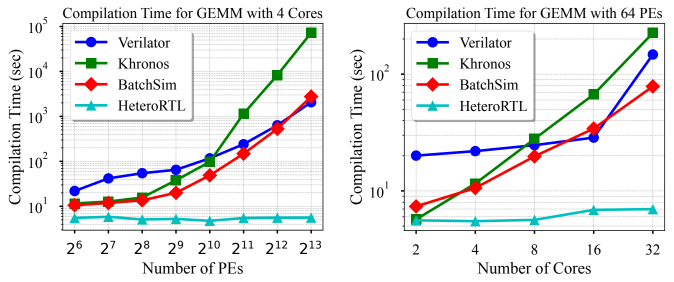

Fig. 3: Compilation time of GEMM accelerators across different RTL simulators under varying Rocket core and PE configurations.

C. Overall Simulation Speedup

Figure 4 and Figure 5 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, HeteroRTL 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 5, 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 are handled in parallel on the GPU, reducing pressure on both sides and achieving better load balancing.

D. Simulation Runtime Analysis

Figure 6 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 6, the simulation time remains around 0.5 seconds

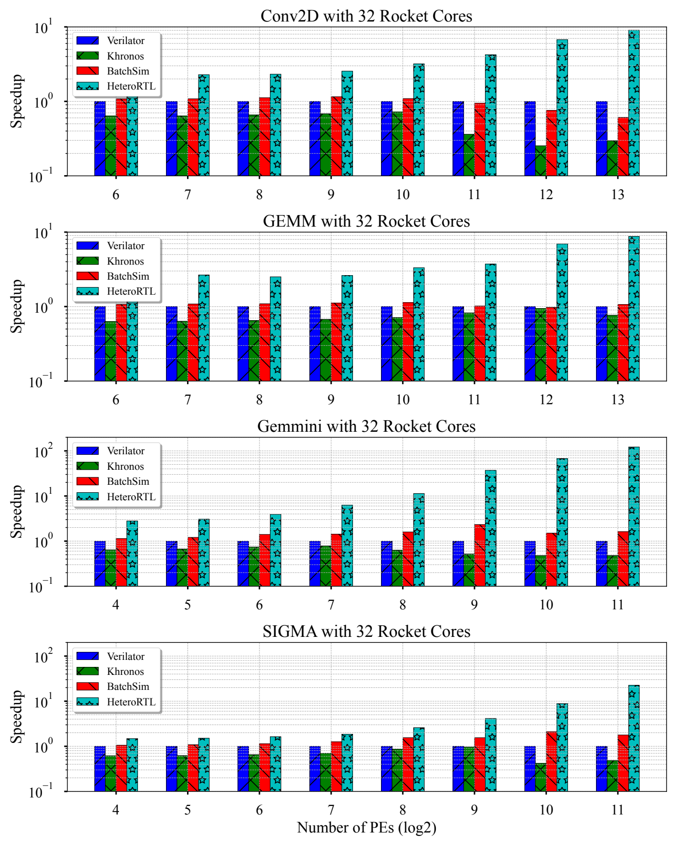

Fig. 4: Overall simulation speedup of Conv2D, GEMM, Gemmini, and SIGMA on various RTL simulators with 32 Rocket cores while the number of PEs increases exponentially.

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 6 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.

V. Conclusion

This paper 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,

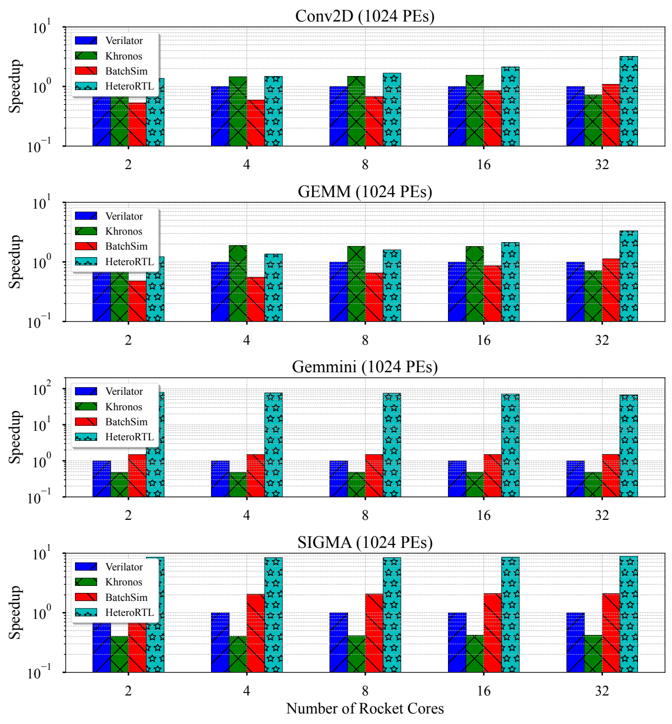

Fig. 5: Overall simulation speedup of Conv2D, GEMM, Gemmini, and SIGMA on various RTL simulators with 1024 PEs while the number of cores increases.

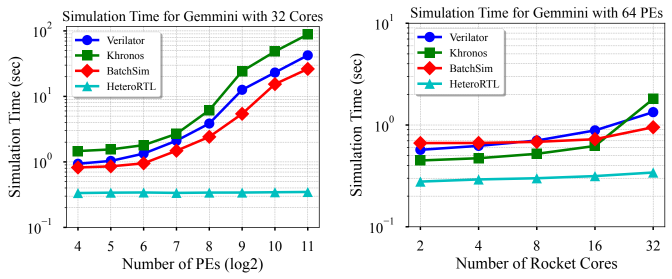

Fig. 6: Simulation time of Gemmini on various RTL simulators with different numbers of PEs and Rocket core configurations.

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× 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. Inspired by our prior work on task graph parallelism [4, 20–92], we plan to extend HeteroRTL to support additional features such as dynamic graph scheduling.

Acknowledgment

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

References

[1] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard 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.

[2] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian 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.

[3] TPU Architecture. https://cloud.google.com/tpu/docs/system-architecture-tpu-vm. [Online; last accessed 16-May-2025.].

[4] Dian-Lun Lin, Haoxing Ren, Yanqing Zhang, Brucek Khailany, and Tsung-Wei 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.

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

[6] Haoyuan Wang and Scott 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.

[7] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim 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.

[8] Kexing Zhou, Yun Liang, Yibo Lin, Runsheng Wang, and Ru 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.

[9] Jie Tong, Liangliang Chang, Umit Yusuf Ogras, and Tsung-Wei Huang. BatchSim: Parallel RTL Simulation using Inter-cycle Batching and Task Graph Parallelism. In *2024 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 789–793. IEEE, 2024.

[10] Haoyuan Wang, Thomas Nijssen, and Scott 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.

[11] Jie Tong, Wan-Luan Lee, Umit Yusuf Ogras, and Tsung-Wei 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.

[12] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr 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.

[13] Chris Lattner and Vikram 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.

[14] CIRCT. https://circt.llvm.org/. [Online; last accessed 16-May-2025.].

[15] Tiago Trevisan Jost, Arun Thangamani, Raphaël Colin, Vincent Loechner, Stéphane Genaud, and Bérenger Bramas. GPU Code Generation of Cardiac Electrophysiology Simulation with MLIR. In *European Conference on Parallel Processing*, pages 549–563. Springer, 2023.

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

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

[18] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar 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.

[19] Scott Beamer and David Donofrio. Efficiently Exploiting Low Activity Factors to Accelerate RTL Simulation. In *2020 DAC*, pages 1–6. IEEE, 2020.

[20] Cheng-Hsiang Chiu, Chedi Morchdi, Chih-Chun Chang, Cunxi Yu, Yi Zhou, and Tsung-Wei Huang. Optimizing CUDA Graph Scheduling with Reinforcement Learning: A Case Study in SSTA Propagation. In *ACM/IEEE International Symposium on Machine Learning for CAD (MLCAD)*, 2025.

[21] Jie Tong, Wan-Luan Lee, Umit Yusuf Ogras, and Tsung-Wei 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.

[22] Chih-Chun Chang and Tsung-Wei Huang. Statistical Timing Graph Scheduling Algorithm for GPU Computation. In *ACM/IEEE Design Automation Conference (DAC)*, 2025.

[23] Wan-Luan Lee, Shui Jiang, Dian-Lun Lin, Che Chang, Boyang Zhang, Yi-Hua Chung, Ulf Schlichtmann, Tsung-Yi Ho, , and Tsung-Wei Huang. iG-kway: Incremental k-way Graph Partitioning on GPU. In *ACM/IEEE Design Automation Conference (DAC)*, 2025.

[24] Yi-Hua Chung, Shui Jiang, Wan Luan Lee, Yanqing Zhang, Haoxing Ren, Tsung-Yi Ho, and Tsung-Wei 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.

[25] Shui Jiang, Yi-Hua Chung, Chih-Chun Chang, Tsung-Yi Ho, and Tsung-Wei 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.

[26] Serhan Gener, Sahil Hassan, Liangliang Chang, Chaitali Chakrabarti, Tsung-Wei Huang, Umit Ograss, , and Ali 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.

[27] Che Chang, Boyang Zhang, Cheng-Hsiang Chiu, Dian-Lun Lin, Yi-Hua Chung, Wan-Luan Lee, Zizheng Guo, Yibo Lin, and Tsung-Wei Huang. PathGen: An Efficient Parallel Critical Path Generation Algorithm. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2025.

[28] Boyang Zhang, Che Chang, Cheng-Hsiang Chiu, Dian-Lun

Lin, Yang Sui, Chih-Chun Chang, Yi-Hua Chung, Wan-Luan Lee, Zizheng Guo, Yibo Lin, and Tsung-Wei 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.

[29] Wan-Luan Lee, Dian-Lun Lin, Cheng-Hsiang Chiu, Ulf Schlichtmann, and Tsung-Wei Huang. HyperG: Multilevel GPU-Accelerated k-way Hypergraph Partitioner. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2025.

[30] Cheng-Hsiang Chiu, Chedi Morchdi, Yi Zhou, Boyang Zhang, Che Chang, and Tsung-Wei Huang. Reinforcement Learning-generated Topological Order for Dynamic Task Graph Scheduling. In *IEEE High-performance and Extreme Computing Conference (HPEC)*, 2024.

[31] Chih-Chun Chang, Boyang Zhang, and Tsung-Wei Huang. GSAP: A GPU-Accelerated Stochastic Graph Partitioner. In *ACM International Conference on Parallel Processing (ICPP)*, pages 565–575, 2024.

[32] Zizheng Guo, Zuodong Zhang, Wuxi Li, Tsung-Wei Huang, Xizhe Shi, Yufan Du, Yibo Lin, Runsheng Wang, and Ru Huang. HeteroExcept: Heterogeneous Engine for General Timing Path Exception Analysis. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, pages 1–9, 2024.

[33] Shui Jiang, Rongliang Fu, Lukas Burgholzer, Robert Wille, Tsung-Yi Ho, and Tsung-Wei 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.

[34] Jie Tong, Liangliang Chang, Umit Yusuf Ogras, and Tsung-Wei 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.

[35] Che Chang, Cheng-Hsiang Chiu, Boyang Zhang, and Tsung-Wei Huang. Incremental Critical Path Generation for Dynamic Graphs. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 771–774, 2024.

[36] Cheng-Hsiang Chiu and Tsung-Wei 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.

[37] Dian-Lun Lin, Tsung-Wei Huang, Joshua San Miguel, and Umit 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.

[38] Wan Luan Lee, Dian-Lun Lin, Tsung-Wei Huang, Shui Jiang, Tsung-Yi Ho, Yibo Lin, and Bei Yu. G-kway: Multilevel GPU-Accelerated k-way Graph Partitioner. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2024.

[39] Che Chang, Tsung-Wei Huang, Dian-Lun Lin, Guannan Guo, and Shiju Lin. Ink: Efficient Incremental k-Critical Path Generation. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2024.

[40] Boyang Zhang, Dian-Lun Lin, Che Chang, Cheng-Hsiang Chiu, Bojue Wang, Wan Luan Lee, Chih-Chun Chang, Donghao Fang, and Tsung-Wei Huang. G-PASTA: GPU Accelerated Partitioning Algorithm for Static Timing Analysis. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2024.

[41] Shiju Lin, Guannan Guo, Tsung-Wei Huang, Weihua Sheng, Evangeline Young, and Martin Wong. GCS-Timer: GPU-Accelerated Current Source Model Based Static Timing Analysis. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2024.

[42] Tsung-Wei Huang, Boyang Zhang, Dian-Lun Lin, and Cheng-Hsiang Chiu. Parallel and Heterogeneous Timing Analysis: Partition, Algorithm, and System. In *ACM International Symposium on Physical Design (ISPD)*, pages 51–59, 2024.

[43] Zizheng Guo, Tsung-Wei Huang, Jin Zhou, Cheng Zhuo, Yibo Lin, Runsheng Wang, and Ru Huang. Heterogeneous Static Timing Analysis with Advanced Delay Calculator. In *IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*, 2024.

[44] Chedi Morchdi, Cheng-Hsiang Chiu, Yi Zhou, and Tsung-Wei 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.

[45] Cheng-Hsiang Chiu, Dian-Lun Lin, and Tsung-Wei Huang. Programming Dynamic Task Parallelism for Heterogeneous EDA Algorithms. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, pages 1–8, 2023.

[46] Chih-Chun Chang and Tsung-Wei Huang. uSAP: An Ultra-Fast Stochastic Graph Partitioner. In *IEEE High-performance and Extreme Computing Conference (HPEC)*, pages 1–7, 2023.

[47] Shiu Jiang, Tsung-Wei Huang, and Tsung-Yi Ho. GLARE: Accelerating Sparse DNN Inference Kernels with Global Memory Access Reduction. In *IEEE High-performance and Extreme Computing Conference (HPEC)*, 2023.

[48] Shiu Jiang, Tsung-Wei Huang, and Tsung-Yi 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.

[49] Dian-Lun Lin, Yanqing Zhang, Haoxing Ren, Shih-Hsin Wang, Brucek Khailany, and Tsung-Wei Huang. GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2023.

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

[51] Elmir Dzaka, Dian-Lun Lin, and Tsung-Wei Huang. Parallel And-Inverter Graph Simulation Using a Task-graph Computing System. In *IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSw)*, 2023.

[52] Guannan Guo, Tsung-Wei Huang, and Martin D. F. Wong. Fast STA Graph Partitioning Framework for Multi-GPU Acceleration. In *IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*, 2023.

[53] Guannan Guo, Tsung-Wei Huang, Y. Lin, Z. Guo, S. Yellapragada, and Martin 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.

[54] Zizheng Guo, Tsung-Wei Huang, and Yibo 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.

[55] Tsung-Wei Huang and Leslie Hwang. Task-parallel Programming with Constrained Parallelism. In *IEEE High-Performance Extreme Computing Conference (HPEC)*, 2022.

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

[57] Cheng-Hsiang Chiu and Tsung-Wei Huang. Composing Pipeline Parallelism using Control Taskflow Graph. In *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 283–284, 2022.

[58] Cheng-Hsiang Chiu and Tsung-Wei Huang. Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms. In *ACM/IEEE Design Automation Conference (DAC)*,

pages 1388–1389, 2022.

[59] Tsung-Wei Huang and Yibo 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.

[60] Kexing Zhou, Zizheng Guo, Tsung-Wei Huang, and Yibo 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.

[61] Dian-Lun Lin and Tsung-Wei 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.

[62] McKay Mower, Luke Majors, and Tsung-Wei Huang. Taskflow-San: Sanitizing Erroneous Control Flow in Taskflow Programs. In *IEEE Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, 2021.

[63] Tsung-Wei 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.

[64] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. HeteroCPPR: Accelerating Common Path Pessimism Removal with Heterogeneous CPU-GPU Parallelism. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2021.

[65] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. GPU-accelerated Critical Path Generation with Path Constraints. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–9, 2021.

[66] Yasin Zamani and Tsung-Wei Huang. A High-Performance Heterogeneous Critical Path Analysis Framework. In *IEEE High-Performance Extreme Computing Conference (HPEC)*, 2021.

[67] Cheng-Hsiang Chiu, Dian-Lun Lin, and Tsung-Wei 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.

[68] Dian-Lun Lin and Tsung-Wei Huang. Efficient GPU Computation using Task Graph Parallelism. In *European Conference on Parallel and Distributed Computing (Euro-Par)*, pages 435–450, 2021.

[69] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. GPU-accelerated Path-based Timing Analysis. In *IEEE/ACM Design Automation Conference (DAC)*, pages 721–726, 2021.

[70] Zizheng Guo, Tsung-Wei Huang, and Yibo 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.

[71] Kuan-Ming Lai, Tsung-Wei Huang, Pei-Yu Lee, and Tsung-Yi 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.

[72] Tsung-Wei Huang, Guannan Guo, Chun-Xun Lin, and Martin 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.

[73] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. OpenTimer v2: A Parallel Incremental Timing Analysis Engine. In *IEEE Design and Test (DAT)*, 2021.

[74] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. In *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, pages 1303–1320, 2022.

[75] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019.

[76] Tsung-Wei Huang, Dian-Lun Lin, Yibo Lin, and Chun-Xun 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.

[77] Chun-Xun Lin, Tsung-Wei Huang, and Martin Wong. An Efficient Work-Stealing Scheduler for Task Dependency Graph. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 64–71, 2020.

[78] Dian-Lun Lin and Tsung-Wei Huang. A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism. In *IEEE High-performance and Extreme Computing Conference (HPEC)*, 2020.

[79] Tsung-Wei 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.

[80] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. GPU-accelerated Static Timing Analysis. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2020.

[81] Guannan Guo, Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. An Efficient Critical Path Generation Algorithm Considering Extensive Path Constraints. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[82] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin Wong. A Modern C++ Parallel Task Programming Library. In *ACM Multimedia Conference (MM)*, pages 2284–2287, 2019.

[83] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin Wong. An Efficient and Composable Parallel Task Programming Library. In *IEEE High-performance and Extreme Computing Conference (HPEC)*, 2019.

[84] Kuan-Ming Lai, Tsung-Wei Huang, and Tsung-Yi Ho. A General Cache Framework for Efficient Generation of Timing Critical Paths. In *ACM/IEEE Design Automation Conference (DAC)*, 2019.

[85] Tsung-Wei Huang, Chun-Xun Lin, , and Martin Wong. Distributed Timing Analysis at Scale. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1–2, 2019.

[86] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. Essential Building Blocks for Creating an Open-source EDA Project. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1–4, 2019.

[87] Tsung-Wei Huang, Chun-Xun Lin, and Martin 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.

[88] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. A General-purpose Distributed Programming System using Data-parallel Streams. In *ACM Multimedia Conference (MM)*, pages 1360–1363, 2018.

[89] Chun-Xun Lin, Tsung-Wei Huang, Ting Yu, and Martin Wong. A Distributed Power Grid Analysis Framework from Sequential Stream Graph. In *ACM Great Lakes Symposium on VLSI (GLSVLSI)*, pages 183–188, 2018.

[90] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. DtCraft: A Distributed Execution Engine for Compute-intensive Applications. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, pages 757–764, 2017.

[91] T.-Y. Lai, Tsung-Wei Huang, , and Martin 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.

[92] Tsung-Wei Huang, Martin 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.