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

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

Liangliang Chang
Arizona State University
Tempe, USA
lchang21@asu.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 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.**

***Index Terms*—RTL simulation, parallel simulation, task graph parallelism**

I. Introduction

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 [1]. 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 throughout 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 [2, 3]. 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[1], the leading open-source RTL simulator, transpiles (source-to-source compiles) an input RTL source (verilog) into optimized C++ simulation code through abstract syntax tree (AST) traversals. ESSENT[4] enhances the simulation performance by partitioning an input RTL graph to several subgraphs with similar activities for load balancing. RTLflow[2] 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 [3, 5, 6], RTLflow significantly improves the simulation throughput performance. RepCut[7] 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[8] 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.

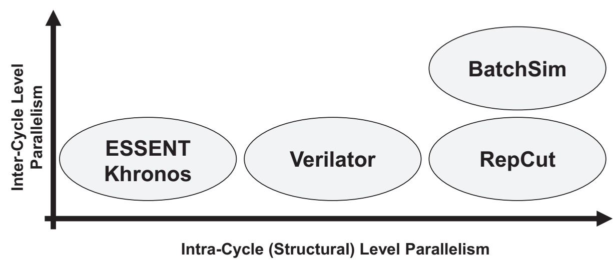

Fig. 1. BatchSim explores both intra- and inter-cycle parallelism to significantly improve the performance of parallel RTL simulation.

Despite improved simulation performance, existing simulators are largely limited to *single-cycle* simulation (see Figure 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

2159-3477/24/$31.00 ©2024 IEEE
DOI 10.1109/ISVLSI61997.2024.00155

RTL simulation can incur certain threading overhead at each cycle, such as scheduling tasks, synchronization, and dynamic load balancing [9]. 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 of *batch* or *inter-cycle* simulation can bring significant yet untapped performance advantages to parallel RTL simulation.

This paper 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.

II. Background and Motivation

A. 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 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 1. A C++ code snippet for full-cycle RTL simulation.

B. 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 1. Notably, ESSENT[4] and Khronos[8] operate on a single-threaded model, whereas Verilator[1] and RepCut[7] 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 computation 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: 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.

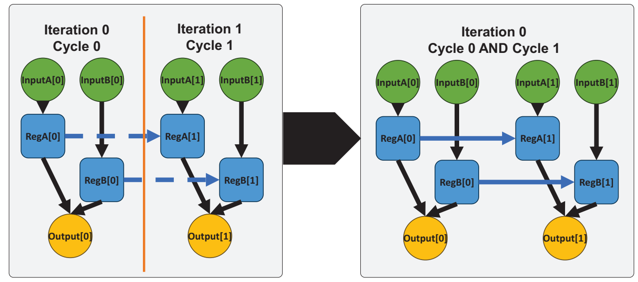

Fig. 2. BatchSim batches consecutive cycle graphs and merges them into a multi-cycle computation graph.

III. BatchSim

Figure 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[10] to accommodate various RTL designs and generates MLIR[11] dialects to leverage existing code generation and optimization passes. BatchSim incorporates the IR compilation infrastruc-

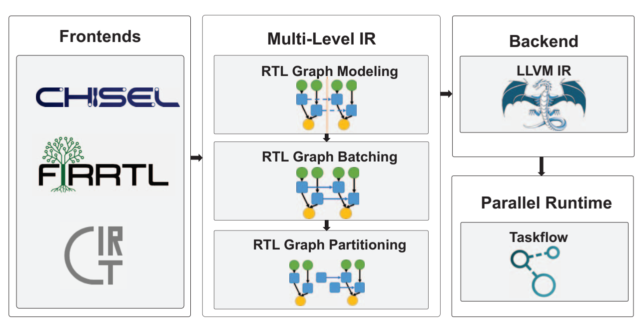

Fig. 3. Overview of BatchSim.

ture and RTL graph modeling from Khronos[8], integrates our inter-cycle graph batching pass, and adopts the graph partitioning method from RepCut[7]. It also employs the code emitting capabilities of the LLVM backend. Additionally, BatchSim utilizes the advanced parallel runtime, Taskflow[9, 12–17], to facilitate multithreading task scheduling and synchronization, enhancing its efficiency and scalability.

A. Inter-Cycle Graph Batching

We utilize the internal data structure of the multi-level IR to handle the RTL design evaluation, which in MLIR[11] 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. 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. An RTL evaluation IR after the inter-cycle batching pass.

B. Parallel Runtime

After completing the inter-cycle batching and graph partition passes, we build a *task graph* to describe the simulation workload. Figure 4 shows a simulation task graph example. Based on this task graph, we can initiate the multi-threaded simulation. In BatchSim, we utilize Taskflow[12, 13], 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 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.

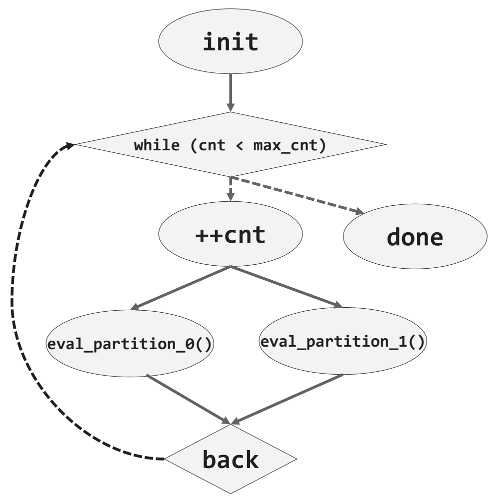

Fig. 4. A task graph for the RTL simulation, which is parallelized through Taskflow [12, 13].

```
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 3. Taskflow code for Figure 4.

IV. Evaluation

A. Evaluation Setup

We evaluate BatchSim's performance on large industrial designs, Gemmini[18], SIGMA[19], RocketChip[20], and BOOM[21], as listed in Table I. 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.

B. Baseline

We consider Khronos[8] as our baseline to evaluate the performance of BatchSim in terms of inter-cycle batching and

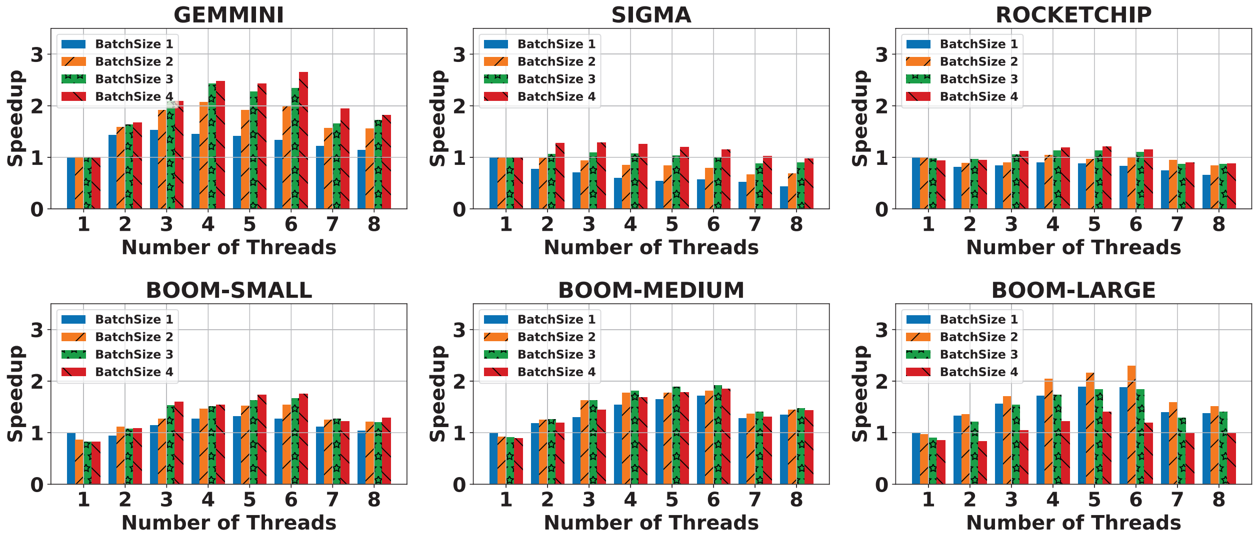

Fig. 5. Speedup improvement of BatchSim across varying thread counts and batch sizes.

TABLE I
EVALUATED BENCHMARKS

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

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 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.

C. Performance Comparison

Figure 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 the 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.

V. CONCLUSION AND FUTURE WORK

This paper 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 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 [22–36], we plan to also leverage the power of GPU to accelerate BatchSim.

REFERENCES

[1] Wilson Snyder. Verilator 4.0: Open Simulation Goes Multi-threaded. In *ORConf*, 2018.
[2] 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 *Proceedings of the 51st International Conference on Parallel Processing*, pages 1–12, 2022.
[3] 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)*, 2024.
[4] Scott Beamer and David Donofrio. Efficiently Exploiting Low Activity Factors to Accelerate RTL Simulation. In *2020 DAC*, pages 1–6. IEEE, 2020.

[5] Dian-Lun Lin and Tsung-Wei Huang. Efficient GPU Computation using Task Graph Parallelism. In *European Conference on Parallel and Distributed Computing (Euro-Par)*, 2021.
[6] 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)*, 2021.
[7] 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.
[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] 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)*, 2020.
[10] CIRCT:Circuit IR Compilers and Tools. https://circt.llvm.org/. [Online; last accessed 20-April-2024.].
[11] 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.
[12] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2022.
[13] 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.
[14] Tsung-Wei Huang, Yibo Lin, Chun-Xun Lin, Guannan Guo, and Martin Wong. Taskflow: A General-purpose Parallel Task Programming System at Scale. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2021.
[15] 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)*, 2023.
[16] 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.
[17] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin Wong. A Modern C++ Parallel Task Programming Library. In *ACM Multimedia Conference (MM)*, 2019.
[18] 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.
[19] 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.
[20] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, et al. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro*, 40(4):10–21, 2020.
[21] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste 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.
[22] 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 International Conference on Computer-aided Design (ICCAD)*, 2020.
[23] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. Accelerating Static Timing Analysis using CPU-GPU Heterogeneous Parallelism. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)*, 2023.
[24] 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.
[25] G. Guo, Tsung-Wei Huang, Y. Lin, Z. Guo, S. Yellapragada, , and M. D. F. Wong. A gpu-accelerated framework for path-based timing analysis. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)*, 2023.
[26] 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)*, 2020.
[27] 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)*, 2021.
[28] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. GPU-accelerated Path-based Timing Analysis. In *IEEE/ACM Design Automation Conference (DAC)*, 2021.
[29] 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.
[30] 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.
[31] 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)*, 2021.
[32] Tsung-Wei Huang, Guannan Guo, Chun-Xun Lin, and Martin D. F. Wong. OpenTimer v2: A New Parallel Incremental Timing Analysis Engine. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2021.
[33] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. OpenTimer v2: A Parallel Incremental Timing Analysis Engine. *IEEE Design and Test (DAT)*, 2021.
[34] 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)*, 2023.
[35] 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.
[36] Dian-Lun Lin and Tsung-Wei Huang. Accelerating Large Sparse Neural Network Inference using GPU Task Graph Parallelism. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2022.