A Fast, Accurate, and Open-Source Simulation Tool for High-Level Synthesis

Rishov Sarkar, Cong (Callie) Hao
School of Electrical and Computer Engineering, Georgia Institute of Technology
rishov.sarkar@gatech.edu, callie.hao@ece.gatech.edu

***Abstract*—High-level synthesis (HLS) tools enable developers to write software code in C, C++, or SystemC and generate hardware RTL code in Verilog or VHDL, making it easier to develop complex hardware designs. However, to evaluate these designs, developers typically still rely on slow RTL simulators that can take hours to provide feedback, especially for complex designs. We introduce LightningSim, an open-source simulation tool for HLS designs that produces "RTL-like" latency estimates with "C-like" speed. LightningSim directly operates on the LLVM intermediate representation (IR) code of an HLS design and accurately simulates a hardware design's dynamic behavior. First, it traces LLVM IR execution to capture the run-time information; second, it maps the static HLS scheduling information to the trace to simulate the dynamic behavior; third, it calculates stalls and deadlocks from inter-function interactions to get precise cycle counts. Evaluated on 33 benchmarks, LightningSim produces 99.9%-accurate timing estimates up to 146× faster than RTL simulation. Our code and benchmark scripts, written in a combination of Python and Rust, are open-source on GitHub, licensed under the AGPL-3.0 license, and were awarded badges for artifact reproducibility.**

I. Introduction

However, the latency estimation provided by HLS tools is usually far from accurate due to the lack of run-time information such as loop counts, branches, and stalls [1], [3], [9], [10]. To get cycle-accurate timing and to check for design issues such as deadlocks, full-blown RTL simulation is needed, which can take hours to days for complex designs. How to get accurate latency information as fast as possible without running RTL simulation is essential to agile hardware development and is of great interest.

To address this challenge, we notice that, HLS tools first compile source code to LLVM [5] intermediate representation (IR) operations and precisely schedule them within functions and loops, which can already provide abundant (yet partial) latency information. Latency variability can be attributed to the dynamic nature of the design, including factors such as loop iteration counts, branch conditions, and stalls in FIFOs and AXI interfaces. Fortunately, *such dynamic behaviors can be simulated through instrumentation of the C/C++ code only without invoking RTL simulation*. We can precisely infer the start cycle and cycle count of every LLVM IR instruction from an *IR execution trace* to compute the accurate latency of the entire program.

Motivated by the need for faster simulation, we propose **LightningSim** [7], [8], the first trace-based simulator based on Vitis HLS. It can provide accurate clock cycle counts for

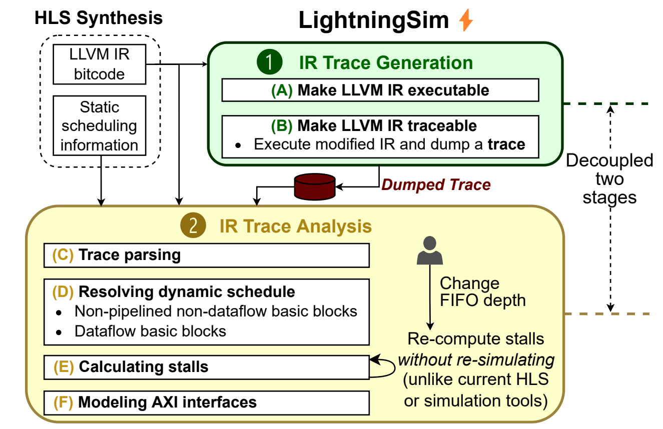
HLS Synthesis
LLVM IR bitcode
Static scheduling information
LightningSim
1 IR Trace Generation
(A) Make LLVM IR executable
(B) Make LLVM IR traceable
• Execute modified IR and dump a trace
Dumped Trace
Decoupled two stages
2 IR Trace Analysis
(C) Trace parsing
(D) Resolving dynamic schedule
• Non-pipelined non-dataflow basic blocks
• Dataflow basic blocks
(E) Calculating stalls
(F) Modeling AXI interfaces
Change FIFO depth
Re-compute stalls without re-simulating (unlike current HLS or simulation tools)

Fig. 1. An overview of LightningSim's decoupled two-stage simulation flow: IR trace generation and trace analysis.

the entire program similar (99.9%) to RTL simulation as soon as HLS finishes its front-end compilation and scheduling, even before RTL code generation.

In addition to being able to start early, the most significant advantage of LightningSim is a novel **decoupled** two-stage simulation: ❶ **IR trace generation** and ❷ **IR trace analysis**, as shown in Fig. 1. First, the IR *trace generation* step takes in the generated LLVM IR by HLS, enables its execution and tracing by completing undefined functions on-the-fly, and then executes it on CPU to collect a *trace*, i.e., a full execution history of LLVM IR instructions. Second, the *trace analysis* step maps the static scheduling information extracted from HLS onto the generated dynamic trace to calculate the resulting latency. This step captures run-time and complex behaviors of the program, including pipelined loops, dataflow, FIFO/AXI stalls, branches, etc. Decoupling the trace generation and analysis can enable *incremental simulation*: if the hardware configuration, such as FIFO depth, is changed, it is only necessary to rerun trace analysis, rather than rerunning the entire HLS synthesis and trace generation. This is a unique advantage compared to existing HLS tools and previous simulators: if the simulation is based on the generated RTL, once the FIFO depth changes, then the entire RTL code needs to be re-generated and re-simulated.

We highlight our contributions and advantages over existing approaches as follows:

- **"C-like" simulation speed with "RTL-like" accuracy.**

LightningSim applies lightweight instrumentation to sequential LLVM code, which retains nearly the same performance as simply compiling and running the C/C++ program on a CPU. Moreover, LightningSim achieves almost the same cycle count as RTL simulation obtains. With 33 benchmarks, including state-of-the-art machine learning accelerators, LightningSim obtains 99.9% accuracy in clock cycles compared to RTL simulation; the speedup is usually one to two orders of magnitude, up to 146×.

- **Rigorous dynamic behavior modeling.** LightningSim faithfully mimics dynamic program behaviors, including FIFO stalls, deadlocks, pipelined loops, dataflow, branches, etc., using the static scheduling information provided by HLS and the dynamic IR execution trace.
- **Decoupled trace-based analysis for incremental simulation.** Thanks to decoupled trace generation and analysis, LightningSim can quickly and flexibly change simulated hardware parameters *ex post facto*, after a single simulation run. For instance, to our best knowledge, LightningSim is the **first** tool that can simulate any FIFO depth from just one simulation, enabling LightningSim to modify or suggest optimal FIFO depths and detect deadlock without rerunning HLS synthesis.
- **Parallelizable with HLS synthesis.** To our best knowledge, LightningSim is the first tool that can start simulation before the completion of HLS synthesis or even scheduling. The *IR trace generation* step can start as soon as the front-end LLVM compilation finishes, which happens before HLS scheduling, binding, and RTL generation.
- **Timing model for external memory accesses.** FPGA designs typically use AXI protocol to read and write from external memory, the DRAM. LightningSim includes a highly accurate timing model for such accesses.
- **Open-source with push-button ease of use.** LightningSim is publicly available on GitHub[1] and, following installation, one can easily apply it to existing Vitis HLS projects using a single Python command without rerunning HLS synthesis.

II. Background

We identify two prior works focusing on HLS simulation acceleration: FastSim [1] and FLASH [3]. FastSim translates the generated Verilog code back to equivalent C++ and optimizes the constructs that frequently appear such as finite state machines (FSMs) for faster simulation. FLASH uses the scheduling information extracted from HLS synthesis together with the input C/C++ code to generate a *FIFO communication cycle-accurate* (FCCA) C model.

Both of these approaches rely on the information *after* HLS scheduling or RTL generation. By contrast, we identify an opportunity for a novel approach: **map HLS scheduling information directly onto LLVM IR execution,** which allows simulation to start as soon as the IR is generated and enables our decoupled trace-based approach.

[1]https://github.com/sharc-lab/LightningSim

III. Overview and Challenges

LightningSim's two decoupled stages are *IR trace generation* and *IR trace analysis*. First, in trace generation, LightningSim compiles the LLVM IR to an executable binary and runs it to generate an execution trace. Second, in trace analysis, LightningSim analyzes the trace and hierarchically calculates the latency of the entire program from instructions to basic blocks and to functions. We identify the following major challenges during the two stages:

1) **Enabling execution of LLVM IR.** While the LLVM project provides the backend infrastructure capable of compiling the IR code to native machine code for popular CPU architectures [5], the IR code generated by HLS is intended only for internal use and thus cannot be directly compiled into a software executable due to undefined or dummy functions. We must define or redefine such functions on-the-fly to enable execution.
2) **Enabling tracing of LLVM IR.** The scheduling information provided by the HLS tool is given in terms of the hardware start and end stages (roughly analogous to clock cycles) of each LLVM IR instruction within a function (corresponding to a hardware module). In order to accurately count clock cycles, the execution order of the LLVM IR instructions must be tracked, as well as any other events that can affect timing and stalls such as reads from and writes to FIFOs and AXI interfaces.
3) **Dynamic schedule resolution.** HLS only provides static scheduling for each module but does not account for dynamic program behavior. During RTL generation, each *stage* in the static schedule is mapped to one state of a finite state machine (FSM), which does not execute sequentially: the FSM can skip states because of branches and loops with multiple iterations. This is further complicated by pipelined loops, where the execution of stages across loop iterations can overlap. Therefore, we must recalculate the dynamic scheduling information based on the static schedule and the instruction execution trace.
4) **Stall calculation.** Each stage of the dynamic schedule corresponds to one clock cycle except in the case of *stalls*. Stalls can occur on attempts to read from an empty FIFO, write to a full FIFO, or read from an AXI interface that is not ready, etc. Therefore, we must efficiently account for cross-module interactions caused by each FIFO and AXI interface such that we can quickly calculate the total clock cycle count including stall cycles.
5) **AXI modeling.** Stall calculation is particularly challenging for external memory accesses through AXI interfaces, which have complex internal behavior, making it difficult to accurately predict the timing of AXI transactions; FLASH [3] does not have a timing model for this.

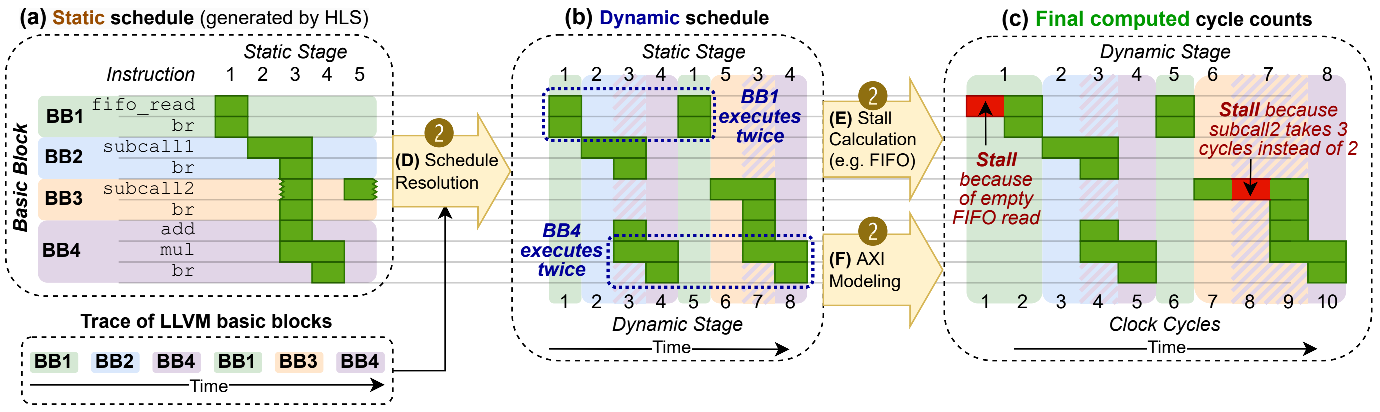

Fig. 2. An example of LightningSim's trace analysis process for a single function call. Using the trace of LLVM basic blocks generated during simulation (§IV-B) and static schedule data from HLS synthesis, schedule resolution (§IV-D) resolves a dynamic schedule where dynamic stages increase monotonically over time. Then, stall calculation (§IV-E) accounts for any stall cycles.

We must design a model for AXI transactions that is as close as possible to the HLS-generated RTL design.

In the next section, we discuss how LightningSim addresses these challenges.

IV. LightningSim Techniques

A. Making the IR Executable

HLS front-end compilation produces LLVM IR code to be used for scheduling, binding, and RTL generation. LightningSim extracts the IR code and uses the LLVM project's built-in code generation backend to compile the IR code to native machine code for the host CPU architecture, such as x86 and x86-64 [5]. As the HLS-generated LLVM IR code is not intended for CPU machine code generation, there are undefined functions and missing intrinsics that LightningSim defines or re-implements *on-the-fly*.

B. Making the IR Traceable

To create an accurate representation of hardware timing, LightningSim needs to understand the specific LLVM IR instructions that were executed and in what order during CPU simulation. Instructions are organized in *basic blocks*, which are groups of instructions with a single entry point and a single exit point, which can be traced to determine all executed instructions [2].

LightningSim employs a custom LLVM pass that iterates through all basic blocks and adds a call to a tracing function to identify the execution of these blocks.

C. Trace Parsing

The executed and generated *trace data* is "flat," i.e., dumped sequentially in an unstructured form. LightningSim transforms the flat trace into a hierarchical structure of function calls, each with FIFO, AXI, and nested sub-call instructions.

D. Resolving the Dynamic Schedule

Dynamic schedule resolution uses the **static schedule** generated by HLS and the parsed trace to determine the **dynamic schedule** for each instruction and function, as shown in Fig. 2.

Each instance of a basic block (BB) in the trace can be linked to a set of "static stages" in the HLS-generated schedule and a set of "dynamic stages" based on simulation behavior. LightningSim handles three cases separately—non-pipelined non-dataflow BBs, pipelined BBs, and dataflow BBs—following specific rules to replicate the hardware behavior for each case.

E. Calculating Stalls

The resulting dynamic schedule consists of dynamic stages, which are usually one cycle each. However, two issues remain. First, some stages may take more than one cycle to complete due to stalls, for instance, if a FIFO is not ready by the time the HLS design expects it to be. Second, within any function, stalls may delay the start of any sub-calls, such that the timing of a callee depends on its caller.

To solve these issues globally throughout the simulation, LightningSim compiles all modules' dynamic schedules into a *directed acyclic graph* (DAG) where nodes represent the dynamic stages within each module and edges represent cross-module "happens-before" constraints, whose lengths are numbers of clock cycles. For instance, we add edges to enforce a minimum delay between writes in the producer function and reads in the consumer function.

We can then quickly calculate the total number of clock cycles as the length of the longest path in the DAG, which fully accounts for all stalls thanks to edges that increase the path length. Furthermore, for incremental simulation with changed FIFO depths, we can rapidly recalculate the longest path with new edge lengths without having to rerun any other part of the simulation.

F. Modeling AXI Interfaces

The HLS-generated design incorporates its own AXI controllers for each AXI interface, which mediates requests and responses on all AXI buses between the HLS design and the rest of the system. These controllers are used, for example, to break long AXI bursts generated internally from the HLS design into multiple specification-compliant AXI

TABLE I
SPEED AND ACCURACY METRICS OF LIGHTNINGSIM ON VARIOUS BENCHMARKS.

	Features					Time (s)			Cycles		
Benchmark	**C**	**P**	**D**	**F**	**A**	**Cosim**	**LS**	**LS Inc**	**Cosim**	**LS**	**HLS**
Fixed-point square root [11]	✗	✓	✗	✗	✗	27.25	4.97 (5.5×)	0.00 ms	30	30 (±0.0%)	32 (+6.7%)
FIR filter [11]	✗	✓	✗	✗	✗	20.12	2.43 (8.3×)	0.00 ms	172	172 (±0.0%)	174 (+1.2%)
Fixed-point window conv [11]	✗	✓	✗	✗	✗	28.30	3.69 (7.7×)	0.00 ms	35	35 (±0.0%)	37 (+5.7%)
Floating point conv [11]	✗	✓	✗	✗	✗	49.78	2.42 (20.6×)	0.00 ms	35	35 (±0.0%)	37 (+5.7%)
Arbitrary precision ALU [11]	✗	✗	✗	✗	✗	24.17	2.12 (11.4×)	0.00 ms	36	36 (±0.0%)	36 (±0.0%)
Parallel loops [11]	✓	✓	✗	✗	✗	26.81	2.34 (11.4×)	0.00 ms	32	32 (±0.0%)	34 (+6.2%)
Imperfect loops [11]	✓	✓	✗	✗	✗	25.80	2.24 (11.5×)	0.00 ms	34	34 (±0.0%)	36 (+5.9%)
Loop with max bound [11]	✗	✓	✗	✗	✗	24.76	2.25 (11.0×)	0.00 ms	31	31 (±0.0%)	33 (+6.5%)
Perfect nested loops [11]	✗	✓	✗	✗	✗	24.76	2.27 (10.9×)	0.00 ms	406	406 (±0.0%)	408 (+0.5%)
Pipelined nested loops [11]	✗	✓	✗	✗	✗	24.92	2.23 (11.2×)	0.00 ms	405	405 (±0.0%)	405 (±0.0%)
Sequential accumulators [11]	✓	✓	✗	✗	✗	26.59	2.29 (11.6×)	0.00 ms	32	32 (±0.0%)	34 (+6.2%)
Accumulators + asserts [11]	✓	✓	✗	✗	✗	27.13	2.30 (11.8×)	0.00 ms	33	33 (±0.0%)	259 (+684.8%)
Accumulators + dataflow [11]	✓	✓	✓	✗	✗	27.26	2.29 (11.9×)	0.00 ms	31	31 (±0.0%)	33 (+6.5%)
Static memory example [11]	✓	✓	✗	✗	✗	33.23	2.18 (15.2×)	0.00 ms	66	66 (±0.0%)	70 (+6.1%)
Pointer casting example [11]	✗	✓	✗	✗	✗	32.55	2.15 (15.1×)	0.00 ms	408	408 (±0.0%)	410 (+0.5%)
Double pointer example [11]	✓	✓	✗	✗	✗	31.70	2.14 (14.8×)	0.00 ms	25	25 (±0.0%)	29 (+16.0%)
AXI4 master [11]	✓	✓	✗	✗	✓	21.06	2.19 (9.6×)	0.00 ms	178	177 (–0.6%)	176 (–1.1%)
AXIS w/o side channel [11]	✗	✓	✗	✗	✗	19.12	2.06 (9.3×)	0.00 ms	52	51 (–1.9%)	53 (+1.9%)
Multiple array access [11]	✗	✓	✗	✗	✗	24.32	2.18 (11.2×)	0.00 ms	252	252 (±0.0%)	254 (+0.8%)
Resolved array access [11]	✓	✓	✗	✗	✗	24.36	2.20 (11.1×)	0.00 ms	131	131 (±0.0%)	133 (+1.5%)
URAM with ECC [11]	✓	✓	✗	✗	✗	22.07	2.21 (10.0×)	0.00 ms	115	115 (±0.0%)	121 (+5.2%)
Fixed-point Hamming [11]	✗	✓	✗	✗	✗	33.28	2.37 (14.1×)	0.00 ms	259	259 (±0.0%)	261 (+0.8%)
Unoptimized FFT [4]	✓	✓	✗	✗	✗	153.53	2.78 (55.2×)	1.85 ms	261,781	261,150 (–0.2%)	?
Multi-stage FFT [4]	✓	✓	✓	✗	✗	61.43	2.67 (23.0×)	0.00 ms	3,770	3,772 (+0.1%)	?
Huffman encoding [4]	✓	✓	✓	✗	✗	46.89	2.63 (17.8×)	0.01 ms	10,283	10,272 (–0.1%)	?
Matrix multiplication [4]	✗	✓	✗	✗	✗	26.33	2.61 (10.1×)	0.00 ms	1,036	1,036 (±0.0%)	1,038 (+0.2%)
Parallelized merge sort [4]	✓	✓	✓	✗	✗	48.79	2.27 (21.5×)	0.00 ms	131	131 (±0.0%)	139 (+6.1%)
Vector add with stream [12]	✓	✓	✓	✓	✓	27.21	4.48 (6.1×)	0.36 ms	4,261	4,261 (±0.0%)	4,242 (–0.4%)
FlowGNN GIN [6]	✓	✓	✓	✓	✓	4219.85	28.86 (146.2×)	2.62 ms	260,359	260,337 (–0.0%)	216,007 (–17.0%)
FlowGNN GCN [6]	✓	✓	✓	✓	✓	534.33	30.95 (17.3×)	19.67 ms	112,836	112,561 (–0.2%)	68,170 (–39.6%)
FlowGNN GAT [6]	✓	✓	✓	✓	✓	838.24	41.63 (20.1×)	14.15 ms	17,282	17,282 (±0.0%)	16,102 (–6.8%)
FlowGNN PNA [6]	✓	✓	✓	✓	✓	3285.45	30.50 (107.7×)	8.54 ms	344,206	344,206 (±0.0%)	?
FlowGNN DGN [6]	✓	✓	✓	✓	✓	996.13	26.89 (37.0×)	1.80 ms	110,710	110,710 (±0.0%)	93,846 (–15.2%)

Features indicate whether benchmark has: **C**: sub-**c**alls to other functions; **P: p**ipelined loops; **D: d**ataflow regions; **F: F**IFO streams; **A: A**XI interfaces.
Cosim: Time for C/RTL co-simulation. **LS:** Time for LightningSim end-to-end simulation (vs. C/RTL co-simulation).
LS Inc.: Time for LightningSim to incrementally re-calculate stalls after changing FIFO depths.
Cosim Cycles: Clock cycle counts reported by C/RTL co-simulation.
LS Cycles: Clock cycles counts reported by LightningSim (vs. C/RTL co-simulation).
HLS Cycles: Clock cycles counts reported by HLS synthesis report, when available (vs. C/RTL co-simulation).

bursts that do not cross 4 KB address boundaries. As the AXI controllers are implemented only in the generated RTL, LightningSim uses a detailed model to calculate the stall cycles for AXI I/O events in a manner as close as possible to the complex RTL-described behavior of the AXI controllers.

V. Experiments and Results

We use 33 benchmarks to comprehensively evaluate LightningSim's accuracy and performance, consisting of small sample designs [11], [12], popular HLS algorithms [4], and complex GNN accelerators [6]. Table I shows our results.

LightningSim achieves 5.5–146.2× speedup comparing with C/RTL cosimulation. Larger designs such as FlowGNN models tend to achieve the largest speedups.

Furthermore, LightningSim achieves its largest speedups in incremental stall recalculation, where the simulation is re-executed with changed FIFO depths. This takes under 20 milliseconds in all test cases, enabling hardware designers to evaluate deadlocks and stalls in streaming dataflow designs with unprecedented speed.

LightningSim produces cycle count results with 99.9% accuracy on average, in comparison to the cycle counts produced by C/RTL cosimulation. LightningSim achieves 100% accuracy for 26 out of 33 cases. In the remaining 7 cases, two have 1.9% and 0.6% error but both are only 1 cycle off; the other five are all within 0.2% error.

VI. Open-Source

LightningSim is developed with usability, reproducibility, and open-source principles in mind. The code for our simulation tool itself, written in Python and Rust, is publicly available on GitHub and licensed under the AGPL-3.0 open-source license. We also package LightningSim as a conda package, allowing easy installation on user systems using a single conda command. Several usage examples are available as part of a tutorial on our documentation site.[2]

Both of our publications for LightningSim [7], [8] have been validated for reproducibility, each earning three artifact evaluation badges certifying that our code and benchmarks are available, evaluated, and results reproduced.

We believe that ensuring that anyone can use and reproduce works such as LightningSim is extremely important for EDA research, and we hope to set an example for others in the field.

[2]https://lightningsim-doc.readthedocs.io/en/latest/tutorial/index.html

References

[1] M. Abderehman, J. Patidar, J. Oza, Y. Nigam, T. A. Khader, and C. Karfa, "FastSim: A fast simulation framework for high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 5, pp. 1371–1385, May 2022.

[2] T. Ball and J. R. Larus, "Optimally profiling and tracing programs," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 4, pp. 1319–1360, Jul. 1994.

[3] Y.-K. Choi, Y. Chi, J. Wang, and J. Cong, "FLASH: Fast, parallel, and accurate simulator for HLS," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 4828–4841, Dec. 2020.

[4] R. Kastner, J. Matai, and S. Neuendorffer, "Parallel programming for FPGAs," May 2018.

[5] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, Mar. 2004, pp. 75–86.

[6] R. Sarkar, S. Abi-Karam, Y. He, L. Sathidevi, and C. Hao, "FlowGNN: A dataflow architecture for real-time workload-agnostic graph neural network inference," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. Montreal, QC, Canada: IEEE, Feb. 2023, pp. 1099–1112.

[7] R. Sarkar and C. Hao, "LightningSim: Fast and accurate trace-based simulation for high-level synthesis," in *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Marina Del Rey, CA, USA: IEEE, May 2023, pp. 1–11.

[8] R. Sarkar, R. Paul, and C. Hao, "LightningSimV2: Faster and scalable simulation for high-level synthesis via graph compilation and optimization," in *2024 IEEE 32nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Orlando, FL, USA: IEEE, May 2024, pp. 104–114.

[9] A. Sohrabizadeh, Y. Bai, Y. Sun, and J. Cong, "Enabling automated FPGA accelerator optimization using graph neural networks," Nov. 2021.

[10] N. Wu, Y. Xie, and C. Hao, "IRONMAN: GNN-assisted design space exploration in high-level synthesis via reinforcement learning," in *Proceedings of the 2021 on Great Lakes Symposium on VLSI*, ser. GLSVLSI '21. New York, NY, USA: Association for Computing Machinery, Jun. 2021, pp. 39–44.

[11] Xilinx, "Basic examples for Vitis HLS," GitHub, Apr. 2021.

[12] ——, "Vitis accel examples' repository," GitHub, Aug. 2022.