GEM: GPU-Accelerated Emulator-Inspired RTL Simulation

Zizheng Guo[1,3], Yanqing Zhang[2], Runsheng Wang[1,3], Yibo Lin[1,3], Haoxing Ren[2]
[1]School of Integrated Circuits, Peking University, Beijing, China
[2]NVIDIA Corporation [3]Institute of Electronic Design Automation, Peking University, Wuxi, China
{gzz,yibolin,r.wang}@pku.edu.cn, {yanqingz,haoxingr}@nvidia.com

***Abstract*—In this paper, we present a GPU-accelerated RTL simulator addressing critical challenges in high-speed circuit verification. Traditional CPU-based RTL simulators struggle with scalability and performance, and while FPGA-based emulators offer acceleration, they are costly and less accessible. Previous GPU-based attempts have failed to speed up RTL simulation due to the heterogeneous nature of circuit partitions, which conflicts with the SIMT (Single Instruction, Multiple Thread) paradigm of GPUs. Inspired by the design of emulators, our approach introduces a novel virtual Very Long Instruction Word (VLIW) architecture, designed for efficient CUDA execution. We also design a flow that maps circuit logic to the architecture in a process analogous to the FPGA CAD flow. This architecture mitigates issues of irregular memory access and thread divergence, unlocking GPU potential for RTL simulation. Our solution achieves up to 64× speed-up over the best CPU simulators, democratizing high-speed RTL simulation with accessible hardware and establishing a new frontier for GPU-accelerated circuit verification.**

I. Introduction

As design complexity of VLSI circuits scale aggressively, the complexity of verifying their functionality at the Register Tranfer Level (RTL) also rockets up. However, traditional CPU-based RTL simulators face scalability bottlenecks due to limited computation power, caches, and parallelism inside multi-core execution [1]. For large-scale designs, even high-performance CPU simulators struggle to deliver the speed necessary for rapid and iterative verification.

Field Programmable Gate Array (FPGA)- and custom-processor-based emulation has emerged as a powerful solution to accelerate logic verification, achieving orders-of-magnitude speedup over CPU simulation [2], [3]. By leveraging parallelism inherent in hardware, emulators can validate large designs more efficiently than software-based simulators. Despite their advantages, FPGA- and custom-processor-based emulators [4], [5] are costly in terms of both financial costs and setup time, limiting their accessibility and adaptability. With these barriers, GPU-accelerated simulation has the potential to offer a compelling alternative, given GPUs' high degree of parallelism, availability, and versatility. Yet, despite the evident advantages, GPUs are rarely used for simulation due to specific architectural challenges.

The barriers to GPU-based simulation arise from fundamental differences between circuit characteristics and GPU architectures. First, GPUs operate under a Single Instruction, Multiple Thread (SIMT) model where threads execute the same instructions in lockstep, which is optimal for workloads with homogeneous parallelism. In contrast, digital circuits are inherently heterogeneous; each circuit partition can have a unique graph structure and functional requirements that challenge SIMT compatibility [6]. Although prior work has sought to make circuits GPU-compatible by mapping them into LUT-based gate level structures [7], [8], [9], [10], [11], [12], this approach falls short in performance compared with highly optimized CPU simulators. Additionally, some GPU-based methods simulate multiple independent testbenches in parallel [13]. While this strategy improves simulation *throughput*, it cannot help in reducing *latency* which is critical for rapid turnaround in iterative design cycles.

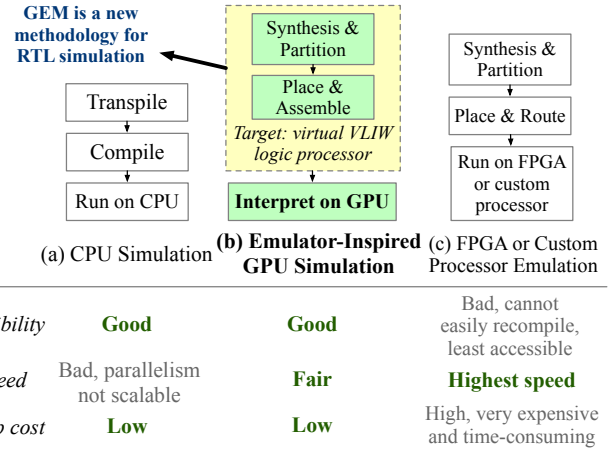

Fig. 1: GEM is a new, emulator-inspired methodology for flexible, fast, and low setup cost RTL simulation.

Another challenge lies in the irregular nature of circuit graphs, which complicates efficient data access on GPUs. GPU memory systems are optimized for throughput, relying on coalesced and regular memory accesses to achieve peak efficiency. However, circuit graphs are sparse and irregular, causing frequent irregular memory accesses that degrade GPU performance. These factors have, until now, prevented GPUs from being viable platforms for efficient, low-latency circuit simulation.

In this work, we propose a GPU-accelerated RTL simulator, **GEM**, that overcomes these challenges and redefines the potential of GPU-based circuit verification. The major contributions are summarized as follows.

1) Rather than simulating circuits using LUT-based methods, we introduce a technique that maps circuit logic into a specialized virtual Boolean processor with a Very Long Instruction Word (VLIW) architecture, optimized for execution on CUDA-compatible GPUs. This innovative architecture allows us to fully utilize the GPU threads even for irregular circuit structures and enabling coalesced memory accesses that are critical to high-performance GPU operation.
2) We design a mapping flow from RTL to our virtual VLIW architecture that mimics the partitioning, synthesis, and physical design flow of hardware, enabling us to interpret the circuit functionality on GPU in a way similar to the computer-aided design (CAD) flow for FPGAs.
3) In our mapping flow, we address various critical challenges on simulation performance by designing a set of novel algorithms, including (i) deep long-tailed logic mapping that minimizes synchronization, (ii) RAM mapping and depth-optimized extended AIG synthesis, (iii) width-constrained replication-efficient partitioning, and (iv) iterative timing-driven bit placement.

Our GPU-accelerated simulator achieves a remarkable 38× and

64× speed-up compared to a leading commercial tool and Verilator, respectively. This substantial performance improvement validates our approach, demonstrating a successful use of GPUs in delivering RTL simulation speed-up. GEM offers a paradigm shift in GPU-based circuit verification, with potential applications extending into areas such as hardware-software co-design and rapid prototyping. Furthermore, this software-based method democratizes high-speed RTL simulation by making it accessible on readily available and relatively inexpensive GPU hardware, bringing high-speed simulation to a broader audience of designers and researchers. To this end, we made GEM open source under a permissive Apache license[1].

II. Preliminary

In digital circuit design, simulation and emulation serve as crucial processes for verifying and evaluating functionality before hardware manufacturing. Both approaches typically work in two stages: compilation and execution.

- In the *compilation* stage, the behavioral RTL netlist, represented in languages such as Verilog or SystemVerilog, is processed and transformed into an executable simulator. This simulator models the circuit behavior, either in software or as a hardware-mapped design, which will then be used to execute various test scenarios.
- The *execution* stage involves running the compiled simulator with specific input stimuli, provided as waveforms or recorded signal patterns (e.g., VCD or FSDB format). The simulator or emulator responds to these stimuli, producing output that reveals the circuit's expected behavior, helping designers validate functionality across test cases.

One key performance metric for evaluating simulators and emulators is the simulation speed, commonly measured in *simulated cycles per second* or Hertz (Hz). Depending on the underlying methodology, simulators are typically categorized as either oblivious/full-cycle simulators or event-based simulators [1].

- Full-cycle simulators process the entire circuit in each simulation cycle, regardless of whether specific parts of the circuit experience any activity. This approach is suitable for high-throughput FPGA-based emulators because the process is consistent across cycles.
- Conversely, event-based simulators, typically CPU-based, are optimized for efficiency by selectively updating only the circuit elements that are actively switching or affected by input changes in each cycle. This selective processing can save significant computation time when only parts of the circuit change, but it is generally slower than hardware-accelerated full-cycle methods.

Previous work in logic emulation and simulation has explored various hardware and software platforms, each with distinct advantages and limitations. FPGA- or custom-processor-based emulation is known for its impressive performance, often achieving emulation speeds of 1 MHz or more [4], [5]. However, the significant cost of dedicated FPGA hardware and the extensive compilation times make this approach less accessible and time-efficient for iterative design workflows. For example, it can take days to compile a design into a FPGA-based emulator. In contrast, CPU-based simulators offer flexibility and ease of setup but are generally slow, often failing to scale well with increasing circuit complexity.

Efforts to parallelize CPU-based simulators, primarily through circuit partitioning, the Chandy-Misra-Bryant (CMB) algorithm, and dataflow optimization [14], [15], [16], [17], have met with only moderate success. Although these approaches can exploit multicore CPUs, they face significant bottlenecks: the CPU frontend often becomes overwhelmed by the sheer volume of compiled circuit code, and parallelism is constrained by the limited memory bandwidth in standard multicore CPUs, capping the potential performance gains.

Efforts to leverage GPU's data parallelism for logic simulation have aimed to address the performance gap but have generally been confined to gate-level simulation using lookup-table-based methods [7], [8], [9], [10], [11], [12]. This approach relies on GPU threads querying precomputed truth tables, which align with the SIMT architecture of GPUs. However, this method presents serious limitations: gate-level simulations are typically much slower (10–100×) than RTL-level simulations, as the latter operate on a higher abstraction level. Consequently, while previous GPU-based methods show promise at the gate level, they fail to match the performance needed for practical RTL verification tasks.

Another line of GPU-accelerated simulation instead transpiles RTL directly to CUDA code. As this results in SIMT-incompatible GPU kernels, they either choose to forfeit data parallelism completely [6] or use independent workloads to fill the data parallelism dimension [13]. These methods require special inputs such as independent stimulis or simple blocking RTL code and are thus not good at reducing simulation latency which is critical in verification turnaround.

III. Algorithm

We present a novel GPU-accelerated RTL simulator, GEM, that addresses the fundamental mismatch between circuit simulation and GPU architecture. Our core idea is a virtual Boolean processor that acts as a reconfigurable and highly-parallel container of Boolean logic. This virtual Boolean processor is programmed with a VLIW instruction set, and is designed to fit GPU parallelism. Specifically, its bitstream can be interpreted using high performance GPU kernels on a CUDA-compatible GPU. Section III-A describes the key motivation behind the design of this architecture and how it solves the SIMT and memory access challenges. To map any input RTL design to the virtual Boolean processor, we design a process containing synthesis (Section III-B), partitioning (Section III-C), physical design (Section III-D), and finally bitstream generation and interpretation (Section III-E). This flow has a beautiful analogy to the CAD flow of FPGAs.

A. A Virtual Boolean Machine

GEM introduces a GPU-friendly virtual Boolean machine that is critical in bridging the gap between circuit functionality and GPU execution. Following are our key observations that lead to its design.

Observation 1. Every Boolean function can be implemented using only a fixed set of operators. Such an operator set is called functional complete. For example, {AND, INVERT} is a functional complete operator set. By compiling the logic into an intentionally limited set of operators, we can use GPU's built-in logic instructions to execute them, instead of having to query look-up tables from memory.

Observation 2. Irregular memory access is inherently unavoidable due to the heterogeneous nature of general circuit graph. However, irregular access inside GPU shared memory (only accessible by a local thread block) is much less costly than irregular global memory access. Rearranging the simulation so that most irregular accesses happen locally inside a thread block will thus help greatly in reducing memory overhead.

Based on Observations 1–2, our modeling of simulation is shown in Figure 2. We regard the general simulation task as a set of logic

[1]https://github.com/NVlabs/GEM

partitions, each in a reasonable size that can be handled with one Boolean processor core (i.e., one GPU thread block). Inside each partition there is an extended and-inverter graph (E-AIG) with AND gates, INVERT gates, D flip-flops (FFs), and RAM blocks. E-AIG can efficiently represent any synthesizable circuit with combinational and synchronous sequential logic. We note that although RAM blocks can be polyfilled using only FFs and decoder logic, this process is extremely costly for large RAMs and thus we introduce native support to RAM blocks in E-AIG.

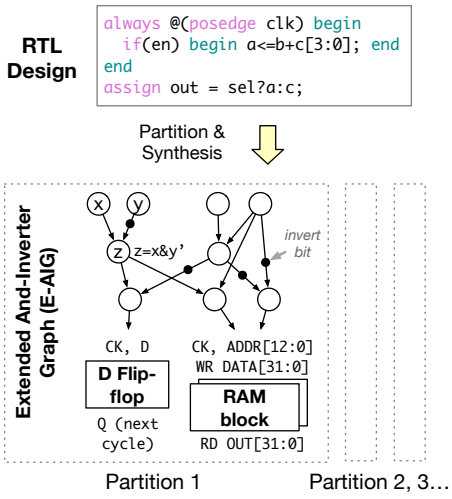

Fig. 2: GEM regards every RTL design as a set of partitions. Each partition is an extended and-inverter graph (E-AIG).

Inside the E-AIG, most of the computation lies in the combinational logic AIG. Our Boolean processor should support efficient execution of AIGs by exploiting the simplicity of gate choices and the common AIG properties.

Observation 3. In addition to thread-level parallelism, *word-level parallelism* can provide orders-of-magnitude more logic processing power. For example, suppose `a,b,c` are three 32-bit unsigned integers, calculating `r=(a AND b) XOR c` is performing 32 And-then-Invert instructions in parallel using `a,b` as inputs and `c` as a constant that encodes whether to flip each resulting bit.

Observation 4. The logic depth of an AIG can be 50–100 for common circuits. However, the gate distribution among the logic levels is extremely imbalanced. A large portion of the gates reside in a few frontier levels whereas only a few gates are accountable for the rest of the levels. We call this the *long-tailed* nature of circuit graphs.

Levelization-based GPU algorithms are frequently used to process circuit graph. They divide the circuit into a series of logic levels each containing a batch of independent calculations. Between two consecutive logic levels, there needs to be a bulk synchronization and a permutation to align the level outputs with the next-level's inputs. As a result, Observation 4 turns out to be very harmful because it leaves most of the levels underutilized and incurs large synchronization overhead. In light of Observations 3–4, we propose a boomerang-shaped executor layer as shown in Figure 3. It is the central reconfigurable logic executor in each virtual Boolean processor core, consisting of interleaved bit permutations and boomerang layers. The virtual Boolean processor core maintains up to 8,192 bits of circuit states. Inside each boomerang layer, the 8,192 bits are recursively folded using bitwise AND and then bitwise XOR with an external constant. The fold is repeated by 14 times until we get a single bit. It can be observed that even a single boomerang layer is able to simulate up to 14 levels of logic. Their shape also fits nicely with the long-tailed nature of circuit graph. Experimentally, boomerang layer reduces the number of bit permutations and synchronizations inside a GPU thread block by more than 5×.

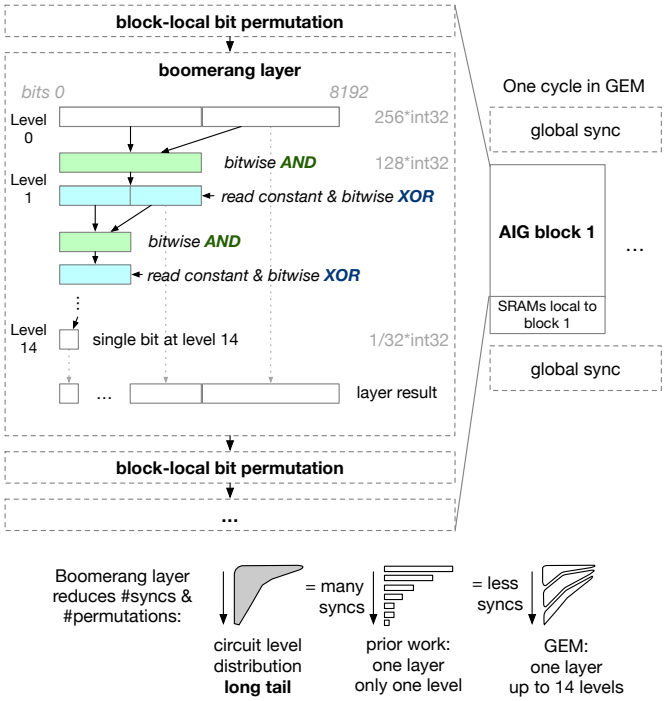

Fig. 3: The boomerang-shaped executor layer in GEM can greatly reduce the number of bit permutations and synchronizations for deep and long-tailed logic.

B. Synthesis

To compile an RTL design for GEM execution, our first step is to map it to the E-AIG format. While AIG is a widely used format in logic synthesis research, there is no existing flow that synthesizes RTL to *extended* AIG with clocked FFs and RAM blocks. The main challenges are two fold: (1) The behavioral RAM constructs in RTL netlist need to be identified and mapped to our fixed RAM block type (13 bits address and 32 bits data) to minimize thread divergence. For a general RAM in RTL, multiple such RAM blocks need to be instantiated and adapter logics need to be introduced automatically. (2) As we will show in later sections, our simulator requires high-quality synthesis results in order to execute efficiently. Specifically, we require the depth of AIG to be as low as possible.

To meet the above requirements, we develop a synthesis flow that exploits two existing synthesis flows for FPGA and ASIC, as shown in Figure 4. We use open-source Yosys synthesizer [18] to deal with RAM mapping. We create a fake FPGA target platform and define our available RAM block. Yosys will take the definition and handle the RAM mapping. Then without further LUT mapping, we can write out the intermediate RTL netlist that only has the RAMs mapped. Next, we handle the RAM-mapped RTL with another ASIC synthesis flow. This time we give the synthesizer a fake ASIC library that only contains AND, OR, INV, and FF gates. This fake library has a simple timing model that defines AND and OR gate delays as

1ps and INV gates as 0ps. Timing-driven synthesis is thus equivalent to depth optimization. In our experiments, we found commercial ASIC synthesizers outperform Yosys in this second step.

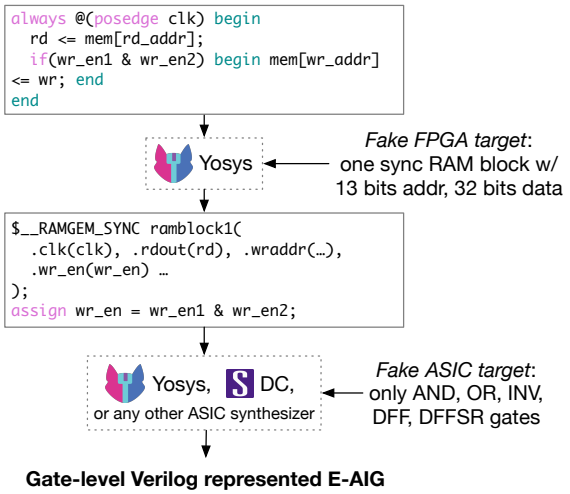

Fig. 4: We exploit existing FPGA and ASIC synthesis flows to efficiently transform RTL design into E-AIG.

C. Partitioning

Partitioning the hardware to multiple Boolean processor cores is a crucial step in our simulator because it ensures locality for data movement. As GPUs do not have efficient inter-block communication, we should ideally make partitions independent of each other and only communicate once per simulated cycle. The goal is made possible with a recent CPU-based parallel simulator RepCut [17], whose basic idea is to allow some duplicated logic in order to remove inter-partition dependency. To adapt the idea to a GPU simulator, we face two new challenges.

Firstly, the number of GPU thread blocks is much larger than the number of CPU cores. However, the replication cost (i.e., the relative size of duplicated logic over the original circuit size) grows quickly with the increased granularity of partitions. For example, RepCut [17] reports that only 1.30% cost is needed to partition a design into 8 threads, but the cost rises to 10.95% when 48 threads are used. In our experiments, we found this cost quickly surges to over 200% when we have to partition a design into 216 blocks, which is a minimal requirement to fully utilize a modern GPU.

To scale RepCut to over 200 partitions, our solution is to extend it to multiple stages as shown in Figure 5. We cut the circuit graph at one or more levels in the middle. We treat the nodes at the cut level as endpoints and run RepCut separately for each stage. We found that with the cost of 1 additional synchronization, we reduce the replication cost from 200% to less than 3% when partitioning a 500K gates design into 216 blocks. For even larger designs, more stages might be needed but it is easy to strike a balance between synchronization and replication costs with heuristics.

Secondly, the goal of partitioning to Boolean processors is different from the original RepCut, which aims at balancing the size of partitions. Instead, we require that all partitions are mappable to the boomerang-shaped executor layer. This effectively constrains the *width* (8192 bits, see Figure 3) instead of *total size* of partitions. It is difficult to modify a hypergraph partitioner's objective to logic widths as this metric does not have nice additive property. Instead, we make no change to the partitioner itself and keep the original size

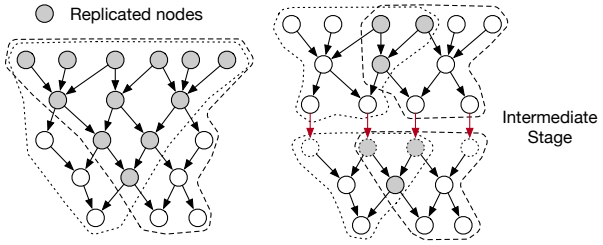

Fig. 5: By introducing one additional stage, the logic duplication of RepCut can be remarkably reduced, unlocking enough parallelism for GPU utilization.

objective. We run an additional postprocessing algorithm after the partitioner to align the objectives to our logic width need. We show our idea in Algorithm 1, which is based on empirically trying to merge the resulting partitions after a round of excessive partitioning. After running the algorithm, it is easy to guarantee that each partition has at least 50% effective bit utilization.

Algorithm 1: Partition merging

```
Partition the design excessively so that each partition is
  mappable;
for each partition p do
    Sort other unvisited partitions by overlap size with p;
    for partition q with large-to-small overlap do
        Try merging q with p, if the result is mappable,
          commit the merge;
```

D. Logic Placement

Given an E-AIG partition, our logic placement algorithm implements the logic onto a series of reconfigurable boomerang layers (Figure 3). Our objective is to map all AIG nodes while reducing the total number of boomerang layers required. In this section, we present an iterative timing-driven placement algorithm in GEM to map AIG to boomerang layers.

We show our basic idea of bit mapping in Figure 6 using a 4-level small boomerang layer as an example. We start with an empty boomerang layer as Figure 6 (1). Then, we choose a bit from E-AIG at logic level 4 and map it to the empty slot in boomerang executor at level 4. To do so, we recursively map the inputs of the being-mapped bit to upper levels of the executor. In the worst case, the fan-in cone of the being-mapped bit forms a perfectly balanced binary tree, thus occupying all bits of a boomerang layer. However, such worst case never happens in practice, as most bits in AIG have a pair of inputs that reside in different logic levels (i.e., imbalanced). As a result, there leaves a lot of free space after mapping one bit, and such vacant bits can be used to map other bits as shown in Figure 6 (2)–(4).

Based on the above bit mapping primitive, we use Algorithm 2 to map a whole AIG partition into a series of boomerang layers. We iteratively try to map the fan-in frontier of current AIG subgraph into a new empty layer. After the layer is full, we remove the nodes that are already realized by current layers, create a new AIG subgraph of remaining nodes, and repeat the process until all nodes are mapped. We define timing criticality of a node as its logic depth on the reversed AIG subgraph. To reduce the number of layers, we prioritize the mapping of nodes on timing-critical paths (Figure 6 (5)

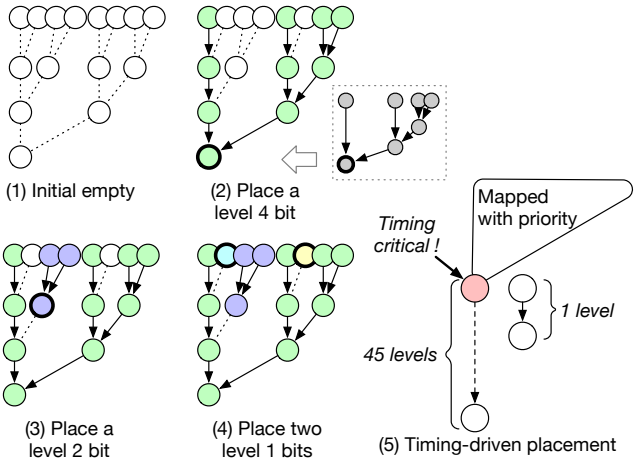

Fig. 6: Example of a placement iteration (1)–(4) and illustration of timing-driven bit placement (5).

and Algorithm 2 lines 7–8) according to the constantly updated timing criticality of nodes.

Algorithm 2: Iterative multi-boomerang-layers mapping

```
1  Input AIG;
2  Initialize node set frontier as input ports of the AIG;
3  while AIG not all mapped do
4      Make a new empty boomerang layer;
5      for Level i in boomerang layer from bottom to top do
6          while Level i is not full do
7              Update timing criticality (reverse logic depth) of
                the remaining AIG;
8              Choose the most timing-critical unmapped node
                in AIG with logic level i, and map it to the
                boomerang layer;
9      Add the boomerang layer to the list of layers;
10     Update frontier to the input ports of the current
        unrealized AIG subgraph;
```

E. Bitstream Generation and CUDA Interpretation

The synthesized and placed circuit from previous sections are then transformed into a binary format that will be loaded and interpreted by our GEM CUDA kernel. In one way, this process is analogous to the bitstream generation of FPGA designs as we are encoding and serializing the wiring (boomerang layer setups) of a reconfigurable hardware. In another way, this process might also be called a binary assembler of software programs, as the resulting bitstream will be interpreted on a GPU by a software-only approach like inside a virtual machine.

We design a domain-specific instruction set architecture (ISA) for GEM to assemble the Boolean processor programs. This is a VLIW ISA that has 3 instruction length variations: 8192, 16384, and 32768 bit. It is designed for a GPU thread block with fixed 256 threads to load and interpret with high throughput. A 8192-bit GEM ISA instruction is loaded by 256 threads in lockstep by performing one coalesced 32-bit global memory read. For 16384- or 32768-bit instructions, 256 threads similarly perform a coalesced 64- or 128-bit global memory read, respectively. All these memory loads are fully coalesced and natively supported in CUDA.

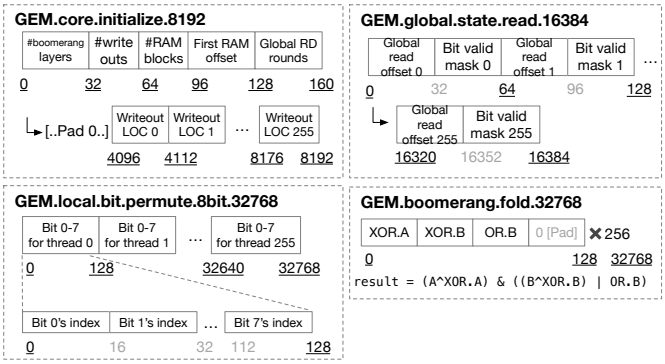

Fig. 7: Instruction bit layout: initialization, global state reading, local bit permutation, and boomerang folding.

TABLE I: Design statistics and GEM mapping results.

Design	#E-AIG Gates	#Levels	#Stages	#Layers	#Parts	Bitstream
NVDLA	668,746	62	1	9	52	11.2 MB
RocketChip	346,687	82	1	13	39	9.2 MB
Gemmini	1,831,381	148	1	19	143	44.4 MB
OpenPiton1	682,646	66	2	10	119	18.4 MB
OpenPiton8	5,479,795	66	2	13	947	162.4 MB

Due to page limit, we cannot include a complete ISA documentation. Figure 7 shows some examples of instruction bit layout. 8192-bit instruction is used to initialize a thread block with necessary information, including the number of boomerang layers, RAMs, and state size of a partition. This is followed by a few 16384-bit instructions to read the input ports of the E-AIG from global memory, only once per cycle. Then, bit permutations and boomerang folding constants are provided with interleaved 32768-bit instructions. The bit permutations are encoded as a compressed form of source bit locations indexed within 8192 thread-local bits in GPU shared memory. The boomerang folding provides 3 constants, `XOR.A`, `XOR.B`, `OR.B` for each thread, controlling the behavior of the word-parallel AND gates. The `OR.B` is used to bypass operand B to implement dashed lines in Figure 6 (4).

In the CUDA kernel implementation, we note two important optimizations. The first is the use of wide global read instructions and aligned structs that allow coalescing to work correctly. The second is the use of cooperative groups [19] to implement device-level synchronization in cycle and stage boundaries in order to bypass kernel launching overhead.

IV. Experimental Results

We implement GEM from scratch in Rust (for the mapping flow) and CUDA (for the VLIW interpreter kernel). We evaluate GEM's performance on a variety of open-source RTL benchmarks listed in Table I, including NVDLA [20] (a deep learning accelerator), RocketChip [21] (a RISC-V CPU), Gemmini [22] (another deep learning accelerator), and an internal multi-core CPU design derived from OpenPiton [23]. For Chisel designs like RocketChip and Gemmini, we generate RTL using Chipyard [24]. We use official benchmark workloads provided by these designs for evaluation.

Table I lists the size of the designs. To estimate the actual RTL circuit footprint, we show the number of logic gates and logic levels after our GEM synthesis (Section III-B). Table I also lists the statistics after finishing the GEM mapping flow, including the number of RepCut stages, boomerang layers, partitions, and the size of the bitstream. The number of boomerang layers is 6–8× smaller

TABLE II: Simulation speed (Hz) and speed-up (compared to GEM-A100) comparison between GEM, a commercial tool (Comm.), Verilator, and GL0AM. NVDLA only simulates correctly with an old 3.912 version of Verilator without multi-threading [20].

Design	Test Name	Comm.	Verilator		GL0AM	GEM		Comm./GEM	Verilator/GEM		GL0AM/GEM
			8 Threads	1 Thread	A100	A100	3090		8 Threads	1 Thread	
NVDLA	dc_...6x3x76x270_int8_0	2,956	N/A	1,010	2,175	65,385	55,716	22.12	N/A	64.76	30.06
	dc_...6x3x76x16_int8_0	4,712	N/A	1,060	3,534	65,385	55,716	13.88	N/A	61.69	18.50
	img_51x96x4..._int8_0	7,848	N/A	1,169	8,213	65,385	55,716	8.33	N/A	55.93	7.96
	cdp_8x8x32_lrn3_int8_2	1,683	N/A	1,512	7,443	65,385	55,716	38.85	N/A	43.24	8.79
	pdp_...max_int8_0	3,391	N/A	1,555	8,353	65,385	55,716	19.28	N/A	42.04	7.83
RocketChip	dhrystone	7,262	9,517	4,639	7,275	52,403	51,695	7.22	5.51	11.30	7.20
	mt-memcpy	11,672	8,845	4,790	6,584	52,403	51,695	4.49	5.92	10.94	7.96
	pmp	4,955	8,220	4,529	6,034	52,403	51,695	10.58	6.38	11.57	8.68
	qsort	6,764	8,342	4,657	7,142	52,403	51,695	7.75	6.28	11.25	7.34
	spmv	11,305	7,534	4,719	7,420	52,403	51,695	4.64	6.96	11.10	7.06
Gemmini	tiled_matmul_ws_full_C	5,188	9,638	2,460	11,618	25,608	17,889	4.94	2.66	10.41	2.20
	tiled_matmul_ws_perf	13,205	10,554	2,537	13,227	25,608	17,889	1.94	2.43	10.09	1.94
OpenPiton1	ldst_quad2	13,871	5,355	3,415	8,400	36,583	31,339	2.64	6.83	10.71	4.36
	fp_mt_combo0	10,569	5,402	3,358	7,303	36,583	31,339	3.46	6.77	10.90	5.01
	asi_notused_priv	5,167	5,025	3,157	4,624	36,583	31,339	7.08	7.28	11.59	7.91
OpenPiton8	ldst_quad2	4,820	1,078	315	5,172	7,285	4,694	1.51	6.76	23.14	1.41
	fp_mt_combo0	7,666	1,080	316	7,203	7,285	4,694	0.95	6.74	23.06	1.01
	asi_notused_priv	1,441	1,004	306	1,920	7,285	4,694	5.05	7.25	23.85	3.79
Average Speed-up				-				9.15	5.98	24.87	7.72

than the logic depth (e.g., reduced from 148 to 19 for Gemmini). We note that the GEM bitstream is a very concise format for circuit logic. It takes only 162.4 MB of GPU memory to store the whole assembled GEM bitstream (Section III-E) even for our largest design OpenPiton8 which has over 5 million logic gates and is over 800 MB in flattened gate-level Verilog. As a result, even lowest-end GPUs have enough GPU memory to simulate large designs with GEM.

We compare the performance of GEM against a set of strong baselines, including a leading commercial tool[2], Verilator latest 5.028 [14], and the current state-of-the-art GPU-accelerated gate-level logic simulator GL0AM [12]. All experiments are run on 48 cores of Intel Xeon Gold 6136 CPU. We run Verilator with up to 8 threads as we observe that 16-threaded Verilator is only 80%–95% the speed of 8 threads. This performance *degradation* highlights the inherent scalability bottleneck of CPU parallelism on highly-complex real designs as also identified by prior works. We run the commercial tool with default single core similar to the settings in [12], [11] as multi-threaded version is unstable and crashes during some simulations. We evaluate GEM on both one NVIDIA A100 and one NVIDIA RTX 3090. A100 demonstrates GEM's current peak performance and 3090 shows its performance with a more accessible GPU alternative.

Table II gives a comprehensive performance comparison. We are on average 9.15×, 5.98×, and 24.87× faster than the leading commercial tool, 8-threaded Verilator and 1-threaded Verilator respectively. The peak speed-ups happen on the deep-learning accelerator NVDLA where we are 64.76× and 38.85× faster than Verilator 1 thread and the commercial tool. GEM on 3090 has comparable performance with A100 except on the largest design, OpenPiton8, with the highest resource pressure on GPU.

The current version of GEM has **limitations** which we plan to address in the future. For example, we found that the speed-up ratio of OpenPiton8 is inferior to OpenPiton1. This is because the workload of OpenPiton8 does not keep all 8 cores busy. Reported by the commercial tool, we observe 8,612 signal events per cycle for 1 core but only 28,789 events (3.3×) for 8 cores. As an oblivious full-cycle simulator, GEM has a consistent simulation speed for any stimuli, whereas our baselines are event-based simulators that run faster if the design is not actively switching. In the future, we plan to explore event-based pruning in GEM. NVDLA shows the best speed-up GEM can achieve because all RAMs inside it are mapped to E-AIG RAM blocks, but the other 4 designs have RAMs with *asynchronous* read ports that can only be implemented inefficiently with FFs and decoder logic. As asynchronous RAMs are not available in ASIC or FPGA deployment, we regard NVDLA's performance as representative for real-world designs.

[2]We do not disclose the name of the commercial tool due to license agreements.

V. Conclusion

This paper presents GEM, a novel emulator-inspired methodology for GPU-accelerated RTL simulation. GEM bridges the fundamental discrepancy between GPU's SIMT data parallelism and the heterogeneous and irregular circuit logics, by designing a CUDA-interpretable virtual VLIW logic processor and a mapping flow analogous to the CAD flow of FPGAs, with novel algorithms addressing deep long-tailed logic, RAM mapping, width-constrained partitioning, timing-driven bit placement, etc. GEM achieves superior performance compared to cutting-edge commercial and open-source simulators.

The framework of GEM is extensible and many improvements are possible as future works, including native arithmetic operations, multi-GPU support, CUDA software pipelining, 4-state simulation, etc. We will open-source GEM for democratizing high-performance RTL verification as well as fostering further research.

VI. Acknowledge

This work is supported in part by the Natural Science Foundation of Beijing, China (Grant No. Z230002), the National Natural Science Foundation of China (Grant No. T2293701), and the 111 project (B18001).

REFERENCES

[1] L.-T. Wang, Y.-W. Chang, and K.-T. T. Cheng, *Electronic design automation: synthesis, verification, and test*. Morgan Kaufmann, 2009.

[2] M. D. Moffitt, G. E. Günther, and K. A. Pasnik, "Place and route for massively parallel hardware-accelerated functional verification," in *Proc. ICCAD*, 2013, pp. 466–472.

[3] W. N. Hung and R. Sun, "Challenges in Large FPGA-based Logic Emulation Systems," in *Proc. ISPD*. Monterey California USA: ACM, 2018, pp. 26–33.

[4] "Synopsys ZeBu Server," https://www.synopsys.com/verification/emulation.html.

[5] "Cadence Palladium," https://www.cadence.com/en_US/home/tools/system-design-and-verification/emulation-and-prototyping/palladium.html.

[6] H. Qian and Y. Deng, "Accelerating RTL simulation with GPUs," in *Proc. ICCAD*. San Jose, CA, USA: IEEE, 2011, pp. 687–693.

[7] D. Chatterjee, A. DeOrio, and V. Bertacco, "GCS: High-performance gate-level simulation with GPGPUs," in *Proc. DATE*. IEEE, 2009, pp. 1332–1337.

[8] D. Chatterjee, A. Deorio, and V. Bertacco, "Gate-level simulation with GPU computing," *ACM TODAES*, vol. 16, no. 3, pp. 1–26, 2011.

[9] Y. Zhu, B. Wang, and Y. Deng, "Massively parallel logic simulation with GPUs," *ACM TODAES*, vol. 16, no. 3, pp. 1–20, 2011.

[10] Y. Zhang, H. Ren, A. Sridharan, and B. Khailany, "GATSPI: GPU accelerated gate-level simulation for power improvement," in *Proc. DAC*. IEEE, 2022.

[11] Z. Guo, Z. Zhang, X. Jiang, W. Li, Y. Lin, R. Wang, and R. Huang, "General-purpose gate-level simulation with partition-agnostic parallelism," in *Proc. DAC*. ACM, 2023.

[12] Y. Zhang, H. Ren, and B. Khailany, "GL0AM: GPU Logic Simulation Using 0-Delay and Re-simulation Acceleration Method," in *Proc. ICCAD*. New York, NY, USA: IEEE, 2024, pp. 1–9.

[13] D.-L. Lin, H. Ren, Y. Zhang, and T.-W. Huang, "From rtl to cuda: A gpu acceleration flow for rtl simulation with batch stimulus," in *Proc. ICPP*, 2022.

[14] W. Snyder, "Verilator 4.0: open simulation goes multithreaded," in *Open Source Digital Design Conference (ORConf)*, 2018.

[15] S. Beamer and D. Donofrio, "Efficiently Exploiting Low Activity Factors to Accelerate RTL Simulation," in *Proc. DAC*. San Francisco, CA, USA: IEEE, 2020, pp. 1–6.

[16] K. Zhou, Y. Liang, Y. Lin, R. Wang, and R. Huang, "Khronos: Fusing Memory Access for Improved Hardware RTL Simulation," in *Proc. MICRO*. Toronto ON Canada: ACM, 2023, pp. 180–193.

[17] H. Wang and S. Beamer, "RepCut: Superlinear Parallel RTL Simulation with Replication-Aided Partitioning," in *Proc. ASPLOS*. ACM, 2023, pp. 572–585.

[18] "Yosys." [Online]. Available: https://yosyshq.net/yosys/

[19] M. Harris and K. Perelygin, "Cooperative groups: Flexible cuda thread programming," 2017. [Online]. Available: https://developer.nvidia.com/blog/cooperative-groups/

[20] "Nvidia deep learning accelerator," 2017. [Online]. Available: https://nvdla.org/

[21] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," UC Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[22] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanovic, B. Nikolic, and Y. S. Shao, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Proc. DAC*, 2021.

[23] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, M. Matl, and D. Wentzlaff, "Openpiton: An open source manycore research framework," *SIGARCH Comput. Archit. News*, vol. 44, no. 2, p. 217–232, 2016.

[24] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.