A Resource-Efficient FPGA-Accelerated Simulation of Neural Processing Units with Chain-based Time-Division Multiplexing

Anonymous Authors

***Abstract*—This paper proposes a novel approach to accelerate large Neural Processing Unit (NPU) design simulations on FPGA through Chain-based Time-Division Multiplexing (CTDM) and its automatic compiler. CTDM replaces repeated logic patterns with a single logic pattern and register chains, which can take advantage of hardware-predefined shift register primitives. It reduces FPGA resource utilization more effectively than the conventional multiplexer-based TDM approaches by minimizing logic overhead and routing congestion. The automated CTDM compiler supports various Hardware Design Languages (HDL) including Verilog, VHDL, High-Level Synthesis (HLS), and Chisel, as well as a wide range of FPGA devices—from small on-premise boards to server-grade hardware simulators like Synopsys Zebu. To address inter-FPGA communication bottlenecks in multi-FPGA deployments, we also propose an optimized device partitioning strategy with the block interleaving technique that minimizes the FPGA link latency. When applied to NVIDIA's open-source machine learning accelerator NVDLA, CTDM achieved 66% and 82% resource reduction of LUTs and FFs, respectively, and enabled a successful deployment of the full variants of NVDLA on a single AMD U250 FPGA device. This demonstrated a 3,653x acceleration in NVDLA simulation time over the Synopsys VCS simulator on a CPU. This method has already been implemented for the simulation and verification of our proprietary NPUs. Notably, it enabled the simulation of a 4-die 1024 TFLOPS chiplet using 144 FPGAs on Zebu5.**

I. Introduction

The rapid advancements in CMOS process technology have significantly increased the transistor density achievable in modern chips, enabling the development of large and complex hardware designs such as Neural Processing Units (NPUs) for Artificial Intelligence (AI) applications. However, these advancements also present challenges for large-scale hardware simulations, particularly for designs with repetitive structures such as matrix multipliers. Such designs require substantial hardware resources, whose simulation and prototyping are critical but resource-intensive and time-consuming [1], [2].

Field-Programmable Gate Arrays (FPGAs) are widely used for rapid prototyping and testing of large hardware designs due to their flexibility and reconfigurability [3]. However, FPGAs have inherent resource constraints, such as limited Look-Up Tables (LUTs), Flip-Flops (FFs), and interconnect bandwidth. When a design exceeds the capacity of a single FPGA, multi-FPGA systems become necessary. Although these systems provide scalability, they introduce inter-FPGA communication bottlenecks that degrade performance. Prior efforts to mitigate these bottlenecks have focused on techniques such as Time-Division Multiplexing (TDM) for optimizing inter-FPGA communication bandwidth. These approaches improve system performance, but often do not address the resource utilization and routing congestion within the FPGA fabric itself [4], [5]. TDM methods can also be applied to logic modules to reduce FPGA resource usage but often exacerbate routing congestion and increase logic depth, limiting their scalability and efficiency [6], [7].

The usability of FPGA-accelerated simulation systems is crucial in hardware/software co-design workflows. Frequent design iterations require tools that can seamlessly accommodate changes without disrupting the development process. Automatic compilers play a vital role in reducing design time and enabling efficient resource optimizations. For instance, FireSim [8] and FireAxe [1] provide high-performance compiler systems with resource sharing and automatic partitioning. However, these tools do not fully address routing congestion or advanced resource utilization techniques, leaving room for further exploration. At the industrial level, server-grade systems like Synopsys Zebu [9] and Cadence Palladium [10] offer automatic partitioning and multi-user support for FPGA-accelerated simulations. However, these systems do not allow user-defined optimizations such as TDM or resource-sharing strategies, limiting their applicability for large and complex designs.

To address these challenges, we propose a novel resource-efficient FPGA mapping methodology to enable acceleration of large-scale NPU simulation by FPGA(s) with limited resources. Our key contributions include:

- We propose a Chain-based Time-Division Multiplexing (CTDM) that employs a TDM strategy for resource reduction including LUTs and FFs.
- We mapped the CTDM-applied design in FPGA using predefined primitives, thereby reducing routing congestion and logic depth.
- Our automated CTDM compiler can be applied to any source design written in HDL. It also supports a wide range of FPGA target devices for deployment.
- Using a latency-hiding technique based on the block interleaving, our multi-FPGA partitioning strategy reduces inter-FPGA bandwidth requirements and high-speed I/O (HSIO) link latency between FPGAs.
- An CPU-FPGA hybrid simulation environment was developed to offer user-friendly functionalities from commercial simulation software.

The remainder of this paper is organized as follows. Section II overviews the techniques typically utilized in modern FPGA-accelerated simulation. Section III explains how our proposed CTDM scheme works and the partitioning strategy with an inter-FPGA latency-hiding technique. Section IV presents the experimental setup and the results. Finally, Section V concludes the paper.

II. Background

This section discusses how FPGAs can be utilized to accelerate large-scale hardware design simulations. Then, we explain the techniques of TDM and optimizations for inter-FPGA communication. In addition, this section introduces commercial hardware and its corresponding tools, illustrating how the industry uses FPGAs for large-scale hardware simulation.

A. *Time-division multiplexing*

Normally, TDM is utilized in inter-FPGA communication to achieve the required bandwidth with limited physical wires. However, TDM can also be used to share a repeated logic pattern across different data streams, reducing the area of logic design at the cost of reduced hardware performance. Figure 1(a) illustrates the baseline module without TDM, which consists of four identical combinational logic patterns and their state registers, S.

For resource sharing, two types of TDM can be considered. First, we can place a memory device that can store the internal states of the combinational logic, which will be multiplexed for resource sharing (Figure 1(b)). This memory-based TDM requires storing and restoring the internal states (e.g. context switching) of a module to replicate the state and behavior of the baseline module. Thus, this requires a load-store logic and related control signals to extract the necessary context data from the module. Additionally, context switching itself requires extra cycles before the module can begin operation.

Second, multiplexers (MUXes) can be used to pump multiple data streams into a shared logic resource for TDM (Figure 1(c)). This MUX-TDM scheme does not introduce additional cycles for context switching, enabling seamless data flow even with shared TDM logic. However, it requires the insertion of MUXes at every point where input/output serialization or deserialization occurs. These additional MUXes can increase FPGA resource utilization, as large MUXes must be nested to optimize performance. Furthermore, high fanout on the output side can exacerbate wiring congestion, which will be discussed in detail in Section III-A3.

To address these challenges associated with the other two types of TDM, we propose chain-based TDM (CTDM), which eliminates the need for large MUXes and complex control logic (Figure 1(d)). The proposed CTDM scheme also avoids context switching, ensuring that no additional cycles are introduced to the logic operation. The detailed explanation of CTDM will be given in Section III-A.

B. *Optimizing inter-FPGA communication*

To scale a design across multiple FPGAs, a large-scale hardware design must be partitioned and distributed across different FPGAs, requiring inter-FPGA communication via HSIO. HSIO including serializer-deserializer (SERDES) breaks the data into parallel words, serializes them for transmission, and reverses the process at the receiver, all of which require specialized FIFOs, alignment logic, and clock-domain crossings. Each pipeline stage, especially as the TDM ratio in SERDES increases, introduces more latency. Additionally, clock data recovery at the receiver adds overhead through phase-locked loops (PLLs) and other digital logic. Collectively, these factors significantly affect the overall link latency. In addition, SERDES can increase the logical bandwidth [11], but may still fall short of the bandwidth requirements of the partitioned design. To ease these problems, we use CTDM with our partitioning strategy to reduce HSIO link latency while meeting the partitioned design's bandwidth requirements. This will be discussed in more detail in Section III-B.

C. *Commercial hardware simulator*

For simulating large hardware designs, various commercial hardware platforms are available, including Synopsys Zebu, which can be used with the Zebu compiler to handle extremely large designs. Zebu is a hardware emulation platform equipped with 48 Xilinx VU19P FPGAs, enabling accelerated verification and validation of complex SoC designs. Although

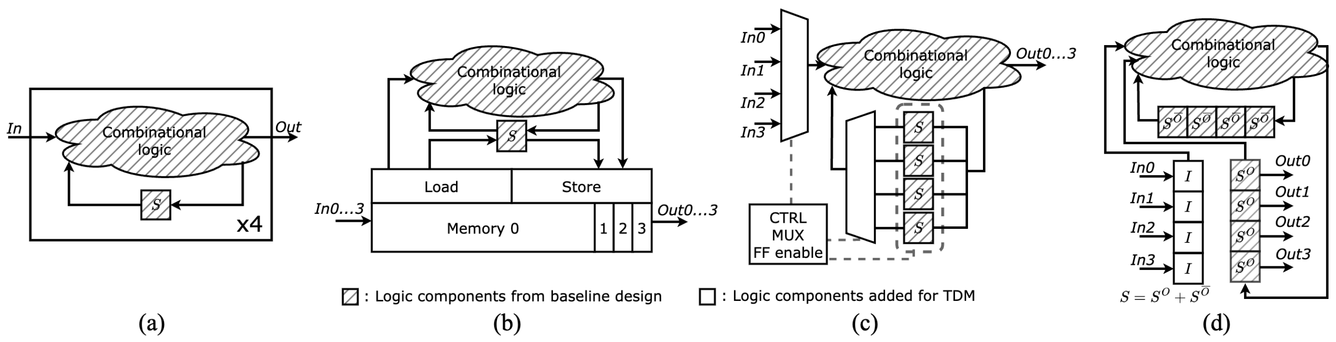

Fig. 1: TDM resource-sharing techniques (for a TDM ratio of 4): (a) baseline, (b) memory-based TDM, (c) MUX-based TDM, and (d) chain-based TDM (ours).

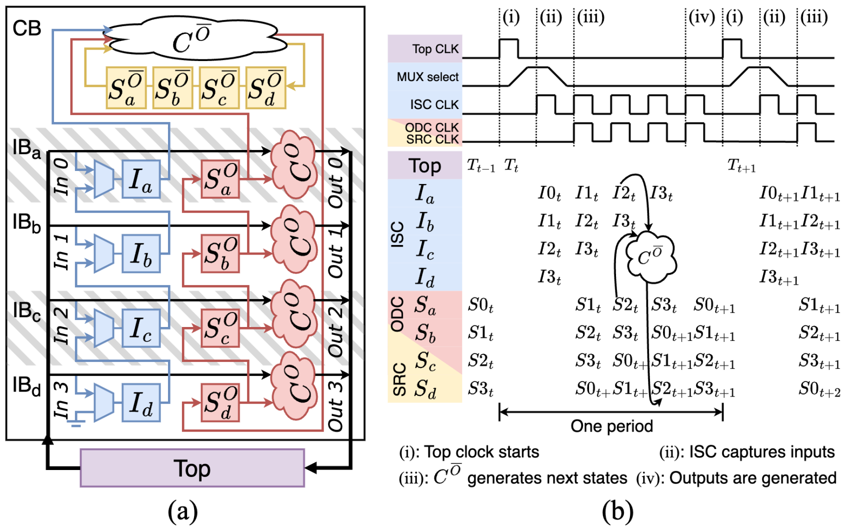
CB
$C^{\bar{O}}$
$S_a^{\bar{O}}$ $S_b^{\bar{O}}$ $S_c^{\bar{O}}$ $S_d^{\bar{O}}$
IB$_a$ IB$_b$ IB$_c$ IB$_d$
In 0 In 1 In 2 In 3
I_a I_b I_c I_d
S_a^O S_b^O S_c^O S_d^O
C^O
Out 0 Out 1 Out 2 Out 3
Top
(a)
(i) (ii) (iii) (iv) (i) (ii) (iii)
Top CLK
MUX select
ISC CLK
ODC CLK
SRC CLK
Top
ISC
ODC
SRC
One period
(i): Top clock starts
(ii): ISC captures inputs
(iii): $C^{\bar{O}}$ generates next states
(iv): Outputs are generated
(b)

Fig. 2: Detailed structure and operation of CTDM: (a) three chains and two combination logic in composition and (b) timing diagram from t to $t+1$, with $C^{\bar{O}}$ illustrating data flow for calculating $S2_{t+1}$.

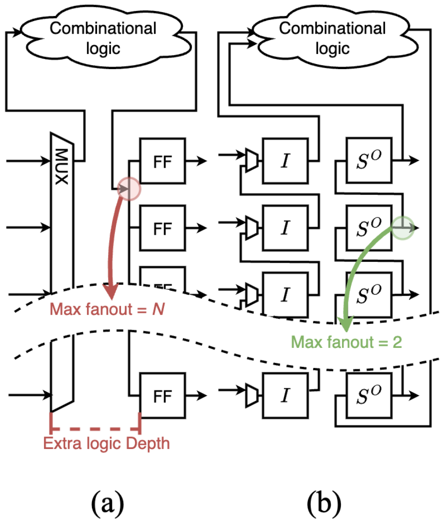

Fig. 3: Comparison of max fanout and logic depth in (a) MUX-TDM and (b) CTDM.

multiple users can share these FPGA resources, this is often impractical because large designs—often exceeding billions of logic gates—can quickly consume all available capacity. Moreover, Zebu lacks optimization for FPGA resource sharing and inter-FPGA latency, offering only automatic partitioning through the Zebu compiler. In Section IV-B3, we compare our proposed inter-FPGA latency-hiding approach and partitioning strategy to this auto-partition capability of existing tools.

III. FPGA Mapping Optimization of NPU

To map our NPU onto an FPGA device, we use chain-based TDM and inter-FPGA latency hiding with our partitioning strategy. The former focuses on resource savings, while the latter addresses performance optimization. Finally, we developed a CPU–FPGA interface that allows FPGA logic to run within a commercial simulator, enabling the simulator's debugging features to be used alongside the FPGA accelerated simulation.

A. Chain-based TDM

Chain-based TDM (CTDM) is a resource-sharing technique inspired by design-for-test (DFT) scan-chain methodologies. A scan chain adds MUXes to the FFs in the design-under-test (DUT), connecting the output of each FF to the input of the next FF. It leverages serialized data flow to insert and extract test patterns to the DUT using a limited number of input/output pins. Similarly, in CTDM, the serialized data flow is used to deliver data to a resource-shared logic. The inputs are captured by MUXes and sequentially shifted to the resource-shared logic, enabling it to perform a series of computations that mimics the behavior of multiple modules. To apply CTDM, repeatedly used modules in the resource-intensive design are selected as the target modules, regardless of their clock domains or hierarchy levels. The rest of the design, excluding the target modules, halts its clock during computation, waiting for the CTDM-applied modules to complete its processing. The CTDM module, generated by our compiler, serially processes the inputs of the target modules and deserializes the results, ensuring equivalent functionality to the original design.

1) Components of CTDM: As shown in Figure 2, the CTDM module is created from the target modules by applying CTDM and inserting the corresponding components. The remaining logic outside the target modules is named as the top module. The components in the CTDM module are composed of three FF chains and two combinational logic modules.

CTDM uses FF chains to emulate the behavior of multiple target modules by modifying states. It consists of three chains: one for input delivery, one for state modification, and one for output generation.

- Input Serializer Chain (ISC): ISC (the blue components in Figure 2(a)) consists of 2-to-1 MUXes, I FFs, and their connections, which are used to serialize parallel inputs from the top module and pass them into the shared logic. The ISC performs two operations, controlled by the MUX select signal. First, it captures input values from the top module. Second, it passes these values along the FF chain to the shared logic.
- State Register Chain (SRC): SRC (the yellow components in Figure 2(a)) represents $S^{\bar{O}}$ FFs, which transfer the original states to the shared combinational logic. $S^{\bar{O}}$ represents FFs that do not directly produce *Out* through the combinational logic. The states move linearly and sequentially without any modification or control.
- Output Deserializer Chain (ODC): ODC (the red components in Figure 2(a)) includes S^O FFs and logic C^O, and it is used to produce the original target outputs. S^O represents FFs that directly produce *Out* through the combinational logic. C^O refers to the subset of the original combinational logic that is directly connected to

Out. The serialized computation results are deserialized by ODC to generate output values simultaneously.

These FF chains have simple wiring, which reduces the wiring complexity in FPGA design. Among them, the SRC can be replaced with a dedicated shift register primitive since there is no value change or usage for intermediate FFs. AMD Shift Register LUT (SRL) [12] utilizes predefined shift register macros that do not consume the FPGA's built-in flip-flops, reducing netlist overhead and improving performance. However, replacing ISC or ODC with SRL is not feasible because SRL does not provide access to the intermediate wires between the internal FFs. This limitation prevents the insertion of MUXes or the generation of parallel outputs after deserialization.

In the CTDM module, there are two types of combinational logic components which are derived from the combinational logic in the target module: C^O and $C^{\bar{O}}$. Unlike C^O in ODC, $C^{\bar{O}}$ is the subset of the original combinational logic that is used to update S $(= S^O + S^{\bar{O}})$. Note that, unlike S^O and $S^{\bar{O}}$, C^O and $C^{\bar{O}}$ are not mutually exclusive and may share some common logic gates.

Depending on their roles, the components of the CTDM module are grouped into two functional blocks (Figure 2(a)).

- Compute Block (CB): $C^{\bar{O}}$ and SRC are grouped together and time-multiplexed for resource sharing. Since one CB corresponds to the repeated logic patterns in the target modules, this is the part where the most resource saving comes from in CTDM.
- Interface Block (IB): We group one slice of ISC and one slice of ODC as one IB, which works as an interface between input/output and CB. Unlike CB, multiple IBs are required for the CTDM module; when a TDM ratio N is 4, four IBs and one CB are used. The interfaces of IBs are identical to those of the target module, therefore the CTDM module can seamlessly replace the target modules. Note that C^O in ODC should be replicated for each IB due to its use in parallel output generation.

2) CTDM operation: Figure 2(b) illustrates how CTDM operates with a timing diagram for each clock domain. All clocks in the diagram originate from the same clock source, but are gated according to their purpose. In cycle (i), the operation begins by launching the top clock (CLK) outside of the CTDM module. The top module uses the output of the CTDM module to generate its state for the next cycle, which updates the input value entering the CTDM module. In cycle (ii), the MUX select goes high to capture the values from *In 0* to *In 3*.

In the cycles denoted as (iii), $C^{\bar{O}}$ calculates the value of the input to $S_d^{\bar{O}}$ and S_d^O using the output values of I_a, $S_a^{\bar{O}}$, and S_a^O. This process is repeated for the number of iterations of the TDM ratio, N. As cycles progress, the values at the ODC shift positions, and after the N iterations, the first calculated value, $S0_{t+1}$, reaches the head of the ODC, S_a^O. At this point, the calculated values of $C^{\bar{O}}$, $S0_{t+1}$ to $S3_{t+1}$, fills the ODC.

In cycle (iv), the output of the CTDM module is then calculated in C^O using the values of the ODC, S_a^O to S_d^O, and these results are fed back to the top module for further processing. This marks the end of one CTDM period, with a new period beginning on the rise edge of the next top clock cycle. The pipeline operation of SRC, $S_a^{\bar{O}}$ to $S_d^{\bar{O}}$, follows the same timing with ODC as shown in Figure 2(b).

3) Routing congestion and logic depth: Compared to MUX-TDM, CTDM reduces implementation complexity by minimizing the routing congestion and logic depth. In MUX-TDM, the fanout increases as the data pipeline branches out at the point where resource sharing ends (Figure 3(a)). The large fanout in MUX-TDM causes high routing congestion and restricts the TDM ratio, thereby limiting the amount of resource reduction that is achievable. In contrast, the output in CTDM has a maximum fanout of only 2, owing to the chained structure (Figure 3(b)). It is important to note that the maximum fanout in CTDM remains unchanged regardless of the TDM ratio.

Furthermore, MUX-TDM adds a significant amount of combinational logic for controlling the dataflow, such as MUXes to order data to each data pipeline stage (e.g. MUX in front of S in Figure 1(c)). In CTDM, we only need to insert SRLs to hold intermediate values for multiple data pipelines ($S^{\bar{O}}$ in Figure 2(a)). Since SRLs can be implemented using dedicated cells in FPGA, no wiring is required between pipeline stages [13]. A smaller amount of added combinational logic in CTDM results in an increase of just one level in the logic depth. In Section IV-B2, we compare CTDM and MUX-TDM with respect to routing congestion and logic depth based on the experiment results.

4) CTDM compiler: Thanks to the simple structure, we can automate the insertion of CTDM into a baseline design. To apply CTDM to the target design, we implemented a compiler based on TCL scripts, which works within AMD's Vivado TCL interpreter. This allows our proposed method to be deployed across multiple FPGA-based devices with diverse environments. Figure 4 shows the entire CTDM compiler procedure and how it can be deployed to different target designs.

The CTDM compiler is divided into three main processes: (1) IB generation, (2) CB generation, and (3) linking them to the top module. (1) In the IB generation stage, our compiler removes the original internal combinational logic and FFs,

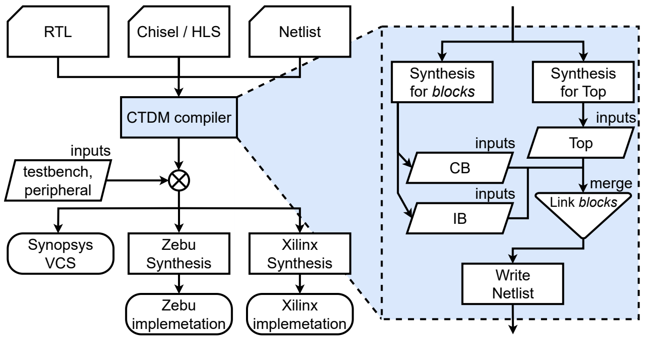

Fig. 4: Overview of CTDM compiler operation flow.

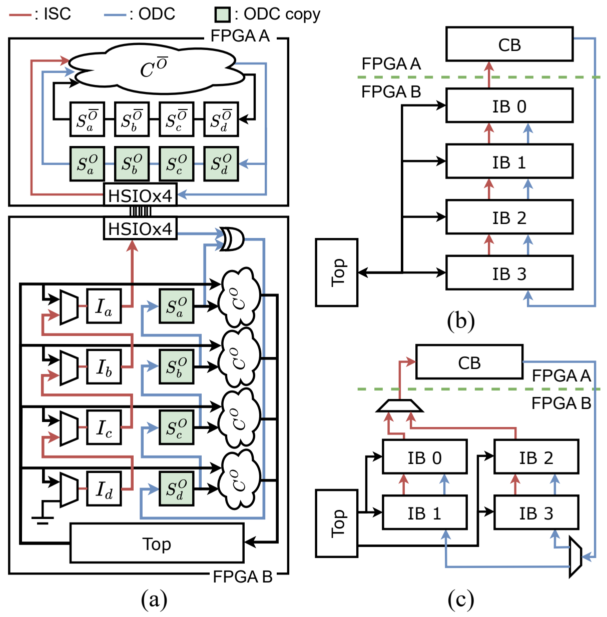

Fig. 5: CTDM partitioning for HSIO latency hiding (a) with ODC copy on each FPGA, (b) without IB interleaving, and (c) with IB interleaving.

then creates a chain of FFs for ISC and ODC. (2) In the CB generation stage, S^O are removed and replaced with ports that are later connected to IB, while $S^{\bar{O}}$ are replaced with the SRC using the SRL primitive. (3) In the top module link process, our compiler runs the synthesis in top module with the target module replaced by black boxes. Once the synthesis is complete, the clock control logic and CB are inserted into the top module, and the black boxes are replaced with IBs. The ports in the top module, IB, and CB are then stitched together, followed by the netlist export process (.v, .edif). The input data for the CTDM compiler includes the top module name, the target module names, and the synthesis file list. CTDM can be applied to multiple modules, and by simply adding target module names to the compiler, and the tasks will automatically run in parallel.

B. Inter-FPGA communication optimization

While CTDM can reduce resource utilization, it can also offer a performance improvement in simulation by hiding inter-FPGA latency when a large design is partitioned across multiple FPGAs. This section presents how we optimized inter-FPGA communication with CTDM and highlights how our method differs from traditional approaches.

1) Partitioning strategy for reducing inter-FPGA wiring: With CTDM, we employ a partitioning method to reduce the inter-FPGA bandwidth requirement. A compute engine in an NPU consumes many LUTs and wires due to its high complexity. Making a partitioning cut inside the compute engine is not desirable, since the bandwidth requirement between partitions will surpass the number of physical wires available in inter-FPGA communication. Instead, we cut the CTDM-applied design between CB and IBs as shown in Figure 5(a).

2) IB interleaving for latency hiding: Inter-FPGA latency from HSIO interfaces can impair the simulation performance. This latency comes from SERDES, clock domain crossing (CDC) logic, and link clock synchronizers in each FPGA. By interleaving IBs that communicate with the other FPGAs, we can effectively hide this latency.

Note that we have two identical copies of ODC in both 'FPGA A' and 'FPGA B' devices in Figure 5(a). This makes an immediate calculation in CB possible when receiving input values from ISC in 'FPGA A'. It reduces HSIO RX latency in 'FPGA A', since we only need to send the content of ISCs. However, on IB in 'FPGA B', we still need to wait for the completion of the previous transaction for incoming ODC content, and this prohibits 'FPGA B' from sending new input values due to the current transaction. To avoid this problem, we can bundle two IB modules into a set and send data to CBs in a ping-pong manner using two IB module bundles. With this optimization, the waiting time for ODC data arrival will be shorter and the other IB set can start new transactions to send data without waiting. This can hide both the HSIO transmitter (TX) and receiver (RX) latency on 'FPGA B'. Figure 5(b) and Figure 5(c) show CTDM partitioning methods without and with IB interleaving, respectively.

Figure 6(a) shows the timing diagram for the traditional partitioning method. Upon the rising edge of the top clock, data is transferred to the target module, and the target module sends output data back to the top module. Figure 6(b) shows the timing diagram for CTDM partitioning without ODC copy and IB interleaving, where four ISC transmissions to CB are placed between the rising edges of the top clock. The response data from CB must also be transmitted to the top module to complete one trip of data transaction; hence, the HSIO TX and RX latencies are still visible. Figure 6(c) illustrates the timing diagram in inter-FPGA data transfer for IB modules, where IB0-IB1 and IB2-IB3 alternate sending data through HSIO via MUX selection. The MUX with the red lines in Figure 5(c) alternates the selection between the IB bundles, selecting the ISC to accept the data. Similarly, CB's output data are received through HSIO by selecting the destination IB bundles through the MUX with blue lines in Figure 5(c). By alternating between IB bundles and sending data in a ping-pong manner, HSIO latency is hidden, allowing the top clock frequency to increase.

C. CPU-FPGA interface for user-friendliness

FPGA-based simulations offer faster simulation speed and more programmability compared to CPU-based simulations, but they provide only limited access to the internal signals even with the existence of an integrated logic analyzer (ILA) and FPGA readback functionality for debugging and signal probing. On the other hand, software simulation on the CPU offers full visibility to the internal signals of the chip design and has other features such as assertion, breakpoints, and waveform generation. To address this, we developed a hybrid hardware simulation system that offloads compute units and computation-intensive logic to FPGAs, while simulating the

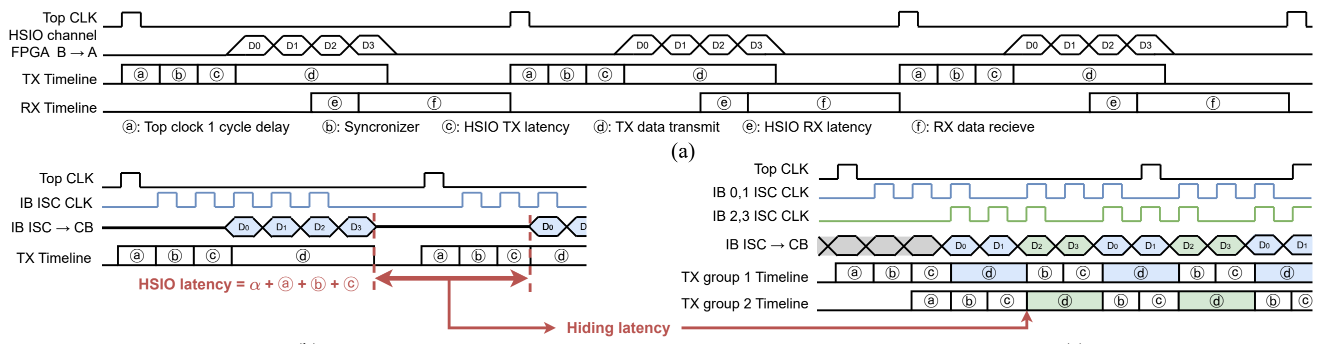

Fig. 6: Our inter-FPGA latency-hiding method illustrated in the timing diagram: (a) the latency breakdown for a typical HSIO, (b) CTDM without IB interleaving, and (c) CTDM with IB interleaving.

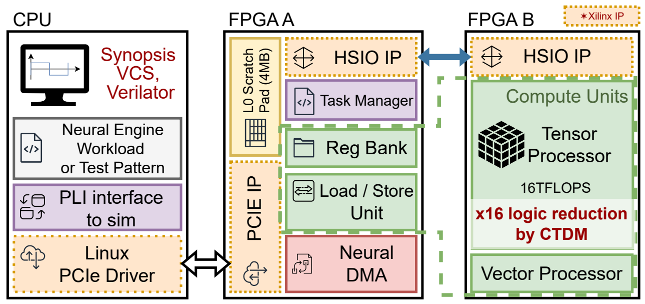

Fig. 7: Overview of CPU-FPGA hybrid simulation with CTDM.

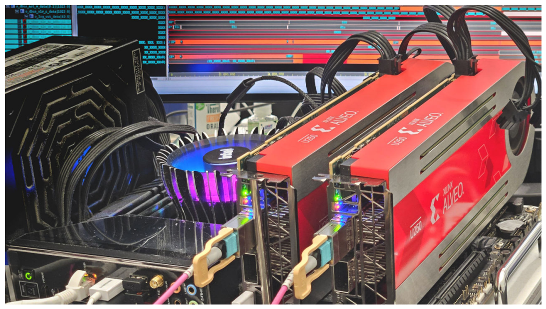

Fig. 8: Neural engine simulation system with two FPGAs.

rest on a CPU in a conventional manner with a software RTL simulator (Figure 7).

In our simulation system, the CPU-FPGA connection is established through PCIe, which introduces inter-device delays. However, this approach improves scalability and enables the simulation of larger designs with fewer FPGA resources. To minimize inter-device latency, we developed a custom Linux driver for efficient data transfer, achieving performance near bare-metal levels. Since our system directly interfaces with software RTL simulators (e.g. Verilator, Synopsys VCS), we can leverage their user-friendly debugging features while accelerating high-load simulations on the FPGA.

IV. Evaluation

A. Experimental setup

We used three simulation cases, neural engine (NE), neural engine with a system on a chip (NE+SoC), and NVDLA [14] to evaluate our proposed method. When on-premise FPGA was used, the test system has Intel's i9-12900 CPU with 32 GB DRAM, and FPGA cards are attached to the PCIe 3.0 x16 slot. NPU designs were mapped to FPGA using AMD Vivado software, with our automatic compiler for applying CTDM operating within Vivado's TCL interpreter. For mapping designs to Synopsys Zebu [9], we utilized Synopsys's Zebu compiler. To optimize performance and resource utilization, we applied CTDM with a TDM ratio of 16 in all test cases.

1) Neural Engine (NE): We applied our proposed method to our proprietary NPU chip's simulation. Neural Engine (NE) is the core of our proprietary chip that performs multiply-accumulate (MAC) operations for deep learning workloads. NE features a 4MB scratch pad and provides 16 TFLOPS of compute performance. Its architecture consists of vector processors, tensor processors, and load/store units, all using repeated MAC operations. The tensor processor is composed of 128 MAC units, while the vector processor includes 16 MAC units. Both have a repeated logic pattern to which CTDM can be applied. To simulate NE, we used a hybrid system composed of a CPU and two on-premise AMD U250 FPGAs. The CPU-FPGA hybrid simulation was conducted by creating a C++ interface in the Synopsys VCS simulator, allowing data transfer with the U250 FPGA through a PCIe driver. The PCIe driver is developed using AMD's XDMA IP [15], which is published as an open source Linux driver. Figure 8 shows the CPU-FPGA hybrid simulation system that runs NE.

2) Neural Engine with System-On-Chip (NE+SoC): In our NPU design, each die contains a total of 16 neural engines (NEs) and four dies that form a single chiplet. In addition, this chiplet includes a system-on-chip (SoC) with NEs. The SoC includes components such as an SRAM buffer, a DMA engine, a PCIe controller, and a network-on-chip (NoC). The SoC occupies a significant area on the chip, comparable to the NE within the NPU. Consequently, simulating this system requires more resources than in the NE case, necessitating additional FPGAs to map the design. To simulate NE+SoC on FPGA, we used Synopsys Zebu5.

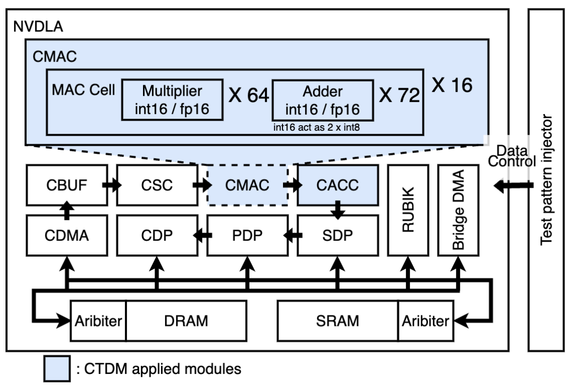

Fig. 9: NVDLA architecture deployed on FPGA for simulation acceleration.

3) NVIDIA deep learning accelerator (NVDLA): We also tested our methods in open-source NVDLA [14] to fairly demonstrate the effectiveness of CTDM. In the test, the largest configuration *nv_full*, which includes 2,048 INT8 MACs, 1,024 MACs (INT16, FP16), and a 512 KB buffer, was used. Figure 9 shows the abstracted structure of NVDLA, where data flow through internal modules for computation. Among these, CTDM was applied to the convolution MAC array (CMAC) and the convolution accumulator (CACC). The CMAC consists of 16 MAC cells and each of the MAC cells consists of 64 16-bit multipliers and 72 adders. To simulate NVDLA, we used a single U250 device.

B. Result and analysis

1) Resource utilization: In all three experiments across NE, NE+SoC, and NVDLA, a significant reduction in the use of LUT and FF was observed. For the NE experiment, Figure 10 illustrates the LUT usage for each CTDM component in the original baseline implementation, the DSP-mapped version, the MUX-TDM version, and the proposed CTDM-applied version. Additionally, it shows the LUT capacity limit for the VU19P and U250 FPGA devices. The DSP-mapped version of the NE architecture is the result of hard mapping specific combinational logics into DSPs and some registers into block RAMs for resource reduction. It required four engineers working for six months to result in a 20% reduction in resource usage. In comparison, CTDM achieved 71% reduction in just 12 hours of automated compiler run.

For the NE+SoC experiment, we used Zebu5 for the simulation. Without LUT reduction, simulation was impossible since it requires 204 AMD VU19P FPGAs. However, after applying CTDM, we implemented it using 144 FPGAs which is the amount of FPGA chips included in three units of Zebu5 server. For NVDLA, the reduction ratio is 66.5% for LUT and 82.4% for FF. To the best of our knowledge, our work is the first to deploy the full variant of NVDLA on FPGA without removing its modules. The only other work [16] deployed *nv_full* on Amazon EC2 FPGA cloud but removed the convolution engines for INT16 and FP16. Table I and Table II present the resource count comparison in the FPGA deployment of NE and NVDLA across the baseline, MUX-TDM, and CTDM schemes.

TABLE I: Resource count comparison of baseline design, MUX-TDM, and CTDM for NE and NVDLA.

Type	Design	Baseline	MUX-TDM	CTDM
LUT	NE	5.55M	2.64M (48%)	1.54M (28%)
LUT	NVDLA	3.31M	2.71M (82%)	1.11M (34%)
FF	NE	2.15M	2.24M (104%)	1.21M (56%)
FF	NVDLA	6.17M	6.17M (100%)	1.09M (18%)

TABLE II: LUT count comparison of baseline design, MUX-TDM, and CTDM for NE and NVDLA subunits.

Design	Subunit	Baseline	MUX-TDM	CTDM
NE	ld, st	291k	405k (139%)	84k (29%)
NE	vector0	419k	166k (40%)	65k (16%)
NE	vector1	521k	194k (37%)	81k (16%)
NE	tensor	3,539k	1,394k (42%)	606k (17%)
NVDLA	CMAC	728k	123k (17%)	102k (14%)

2) Routing congestion and logic depth: Comparing the three conditions—baseline, MUX-TDM, and CTDM—shows that applying CTDM reduces both the total net count and the total fanout. These two reductions help reduce routing congestion. We tested three configurations on the U250 FPGA: baseline 16 MAC cells, MUX-TDM, and CTDM. Due to routing failure, *nv_full* with MUX-TDM could not fit, so we implemented only the NVDLA CMAC for evaluation.

From the experiments, we obtained the following metrics: a total sum of fanout, the highest fanout, the maximum logic

TABLE III: Detailed results in resource and routing reduction of NVDLA CMAC.

	Baseline		MUX-TDM	CTDM
# of Instances	1	16	16	16
Total # of nets	62k	993k	206k	116k
Sum of fanout	345k	5,515k	1,022k	787k
Max. fanout	14k	14k	26k	14k
Max. logic depth	34	34	37	35
# of max. logic depth nets	128	2048	2048	128

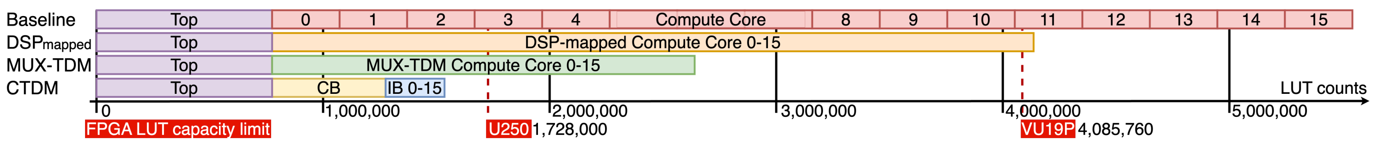

Fig. 10: LUT reduction result for NE.

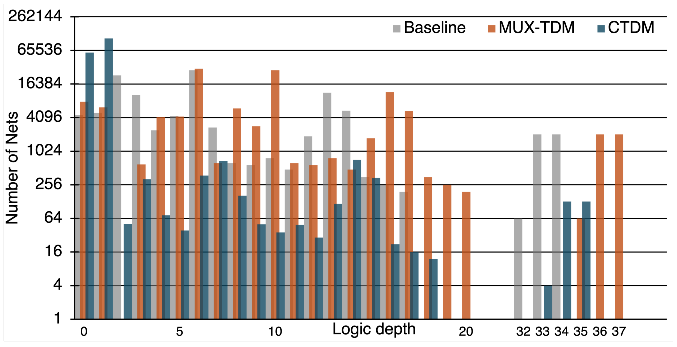

Fig. 11: NVDLA CMAC logic depth histogram.

depth, and the number of nets with the highest logic depth. Table III shows that CTDM reduces the net count by 44% and the sum of fanout by 23% compared to MUX-TDM.

Compared to baseline, CTDM increased the maximum logic depth by 1 but reduced the net count from 2,048 to 128 at this depth. In contrast, MUX-TDM maintained the same net count but increased the logic depth by 3. CTDM reduced the number of nets with the maximum logic depth by a factor of 16, corresponding to the TDM ratio used in the experiment.

Figure 11 shows the histogram of logic depth for the CMAC in baseline, MUX-TDM, and CTDM schemes. The histogram analysis shows that MUX-TDM increases the maximum logic depth of the baseline by 3, while CTDM increases it only by 1 and reduces the occurrences by 16 times. This reduction is consistent except at depth 0 and 1, due to shift registers inserted between combinational logics. Logic depth affects critical path delay in static timing analysis (STA), while the number of wires affects routing congestion. In this context, CTDM achieves better results in reducing routing congestion compared to MUX-TDM, allowing *nv_full* to fit on a single U250 FPGA.

3) Frequency optimization via latency-hiding: In comparison experiments with a commercial simulator, applying CTDM resulted in a higher operating frequency. To evaluate our inter-FPGA latency-hiding method, we used Zebu compiler's auto-partitioning for Zebu5 and compared it to our proposed latency-hiding approach combined with our partitioning strategy. The target design used here is NE. In each experiment, simulation performance is limited by the top clock, which depends on the end-to-end HSIO latency. We used this slowest clock to evaluate the effectiveness of our latency-hiding technique. Table IV shows the results of various FPGA-based devices to simulate NE without SoC.

CTDM on Zebu can reduce FPGA usage by 1/5, freeing up resources for other users. This enables a trade-off between performance and resource utilization in FPGA simulation, supporting simulations with fewer FPGAs and freeing up additional FPGA units for multi-tenancy on expensive server simulators like Zebu. For on-premise FPGA setups (e.g. U250), we compared CTDM with a baseline configuration that uses the same partitioning method as ours but without any TDM techniques, requiring deployment on five U250 boards. Our CTDM achieves 397 kHz using only two FPGAs, surpassing the baseline performance of 318 kHz while using fewer FPGAs (Table IV).

4) Simulation speedup: By selecting specific workloads and comparing the CPU-based approach against the proposed CTDM, a significant improvement in simulation speed was observed. We compared the simulation performance of the Synopsys VCS running on a CPU with that of our CPU-FPGA hybrid system. For the NE case, the simulation took 372.8 seconds on the CPU using the VCS simulator, whereas it completed in just 0.6 second on our CPU-FPGA hybrid simulator. This demonstrates a 621× speedup in simulation time with our proposed method. The workload used in this test is our matrix multiplication test pattern for our chip's functionality verification. For NVDLA, the simulation on the CPU using the VCS simulator took 0.88 hours, while the FPGA-only simulation system completed it in just 0.867 second, achieving a 3,653× speedup for the AlexNet experiment.

V. Conclusion

This paper investigated mapping large NPU designs with repeated logic patterns to FPGAs in a resource-efficient manner. In simulating a large NPU design, the application of TDM for resource sharing is crucial due to the limited resources available on FPGAs. Using a novel chain-based TDM (CTDM) technique, we achieved 66.5% reduction in LUT utilization and 82.4% reduction in FF utilization in FPGA mapping of the NVDLA design. Furthermore, by incorporating the inter-FPGA latency-hiding technique with the partitioning strategy, the implementation of our own NPU core design in U250 demonstrated a 24.8% improvement in the operating frequency with fewer FPGA boards. Finally, applying this methodology to the Zebu5 FPGA simulation server for a massive 4-chiplet NPU, we successfully mapped the entire design using only 144 VU19P FPGA chips. This approach is now actively used for the hardware simulation and software development of our proprietary NPU.

TABLE IV: Operating frequency of the proprietary chip's neural engine on two FPGA-based devices.

Device	Partitioning	Latency-hiding	CTDM ratio	Top freq. (kHz)	FPGA counts	LUT usage (% per FPGA)
Zebu	Auto	No	1:1	57.8	VU19P×5	5.5M (134%)
Zebu	No	No	1:16	11.4	VU19P×1	1.6M (39%)
U250	No opt.	No	1:1	318	XCU250×5	5.5M (318%)
U250	CTDM	No	1:16	97.4	XCU250×2	1.6M (92%)
U250	CTDM	Yes	1:16	397	XCU250×2	1.6M (92%)

References

[1] J. Whangbo, E. Lim, C. L. Zhang, K. Anderson, A. Gonzalez, R. Gupta, N. Krishnakumar, S. Karandikar, B. Nikolić, Y. S. Shao, and K. Asanović, "FireAxe: Partitioned FPGA-Accelerated Simulation of Large-Scale RTL Designs," in *ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 501–515.

[2] S. Asaad, R. Bellofatto, B. Brezzo, C. Haymes, M. Kapur, B. Parker, T. Roewer, P. Saha, T. Takken, and J. Tierno, "A Cycle-Accurate, Cycle-Reproducible Multi-FPGA system for Accelerating Multi-Core Processor Simulation," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2012, pp. 153–162.

[3] J. Ributzka, Y. Hayashi, F. Chen, and G. R. Gao, "DEEP: an Iterative FPGA-based Many-Core Emulation System for Chip Verification and Architecture Research," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2011, pp. 115–118.

[4] C.-W. Pui, G. Wu, F. Y. Mang, and E. F. Young, "An Analytical Approach for Time-Division Multiplexing Optimization in Multi-FPGA based Systems," in *ACM/IEEE International Workshop on System Level Interconnect Prediction (SLIP)*, 2019, pp. 1–8.

[5] L. Sun, L. Guo, and P. Huang, "System-Level FPGA Routing for Logic Verification with Time-Division Multiplexing," in *International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, 2021, pp. 210–218.

[6] S. Hadjis, A. Canis, J. Anderson, J. Choi, K. Nam, S. Brown, and T. Czajkowski, "Impact of FPGA Architecture on Resource Sharing in High-Level Synthesis," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2012, p. 111–114.

[7] R. Nangia and N. K. Shukla, "Resource Utilization Optimization with Design Alternatives in FPGA based Arithmetic Logic Unit Architectures," *Procedia Computer Science*, vol. 132, pp. 843–848, 2018.

[8] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra *et al.*, "FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud," in *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 29–42.

[9] Synopsis, "Zebu server 5: High-capacity emulator for fast soc bring-up," [url] https://www.synopsys.com/verification/emulation/zebu-server.html, 2024 (accessed January 17, 2025).

[10] Cadence, "Palladium emulation," [url] https://www.cadence.com/en_US/home/tools/system-design-and-verification/emulation-and-prototyping/palladium.html, 2024 (accessed January 17, 2025).

[11] P. Zou, Z. Lin, X. Shi, Y. Wu, J. Chen, J. Yu, and Y.-W. Chang, "Time-division multiplexing based system-level fpga routing for logic verification," in *ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[12] AMD, "Ultrascale architecture configurable logic block user guide," [url] https://docs.amd.com/v/u/en-US/ug574-ultrascale-clb, p. 47, 2017 (accessed November 18, 2024).

[13] ——, "Xilinx wp271 saving costs with the srl16e white paper," [url] https://docs.amd.com/v/u/en-US/wp271, 2008 (accessed October 06, 2024).

[14] Nvidia, "Nvidia deep learning accelerator (nvdla) open source project," [url] https://nvdla.org, 2018 (accessed September 20, 2024).

[15] Xilinx, "Xilinx DMA Subsystem for PCI Express (XDMA), year = 2019 (accessed September 24, 2024), howpublished = "[url] https://github.com/Xilinx/dma_ip_drivers/tree/master/XDMA/linux-kernel"."

[16] F. Farshchi, Q. Huang, and H. Yun, "Integrating NVIDIA deep learning accelerator (NVDLA) with RISC-V SoC on FireSim," in *Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*, 2019, pp. 21–25.