CTDM: Resource-Efficient FPGA-Accelerated Simulation of Large-Scale NPU Designs

Hyunje Jo[§*], Han-sok Suh[§†], Hyungseok Heo[*], Jinseok Kim[*], Hyunsung Kim[*], Boeui Hong[*], Jungju Oh[*], Sunghyun Park[*], Jinwook Oh[*], Sunghwan Jo[‡], Kangwook Lee[‡], and Jae-sun Seo[†]
[*] Rebellions, Seongnam-si, South Korea, [†] Cornell Tech, New York, USA
[‡] Synopsys Korea, Yongin-si, South Korea
Email: [*]{aleph, hyungseok.heo, jinseok, hyunsungkim, boeui.hong, jungju.oh, sunghyun.park, j.oh}@rebellions.ai,
[†]{hs2239, js3528}@cornell.edu, [‡]{sunghjo, kangwook}@synopsys.com

***Abstract*—This paper proposes a novel approach to accelerate large Neural Processing Unit (NPU) 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 built-in shift register primitives. It reduces FPGA resource utilization more effectively than conventional multiplexer-based Time-Division Multiplexing (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 extend the applicability of CTDM to multi-FPGA systems, we propose a block interleaving technique that hides inter-FPGA link latency and fully utilizes the pipeline in a high-speed serial I/O channel. When applied to NVIDIA Deep Learning Accelerator (NVDLA), CTDM achieved a 66% and 82% reduction in LUT and FF utilization, respectively, and enabled the successful deployment of the largest variant of NVDLA on a single AMD U250 FPGA device. This demonstrated a 3,653× 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 ZeBu 5 server.**

I. Introduction

Advances in CMOS technology have greatly increased transistor density, enabling large-scale designs such as Neural Processing Units (NPUs) for Artificial Intelligence (AI) applications. While these architectures offer tremendous capability, their size and complexity pose significant challenges for pre-silicon validation, particularly in designs with repetitive structures like large systolic arrays. Conventional CPU-based simulation is prohibitively slow, and GPUs often struggle with the irregular branching and dependencies in RTL.

ASIC-based simulation can handle such irregularities but its long development cycle delays pre-silicon validation. In contrast, FPGAs enable rapid prototyping and testing of large designs due to their flexibility and reconfigurability [1]. However, FPGAs have limited Look-Up Tables (LUTs), Flip-Flops (FFs), and interconnect bandwidth.

In this respect, HAsim [2] time-multiplexes modules to simulate many cores with a single instance on FPGA. PiMulator [3] employs FreezeTime virtualization to swap memory states for larger system emulation. ASH [4] applies the dataflow-driven method and skips inactive logic to reduce the usage of hardware resources in simulation. Although efficient, these methods often lack essential features for simulating large-scale NPUs, such as compiler-driven automation, scalability for large source designs, or the ability to run software workloads for testing AI models. FireSim [5] offers the broadest set of features among existing solutions that satisfy these criteria and is released as open source. FireAxe [6], a compiler add-on to FireSim [5], offers design partitioning and FAME-5 transformation for better resource efficiency on FPGA. However, their optimizations apply only when the hardware is written in Chisel [7], due to tight integration with the Chipyard [8] ecosystem. Designs written in hardware design languages are supported only as blackbox IPs, which disables key optimizations of FireSim/FireAxe [9]. Table I summarizes supported features across simulation methods.

Simulation features	HAsim[2]	PiMulator[3]	FireSim[5]	ASH[4]	**CTDM (ours)**
Resource saving	o	o	o	o	**o**
Compiler support	o	x	o	o	**o**
Scalability	x	△	o	o	**o**
Supported language	.v/.sv	.sv	Chisel	.v/.sv	**All**

TABLE I: Feature comparison of simulation/emulation frameworks. (Triangle mark indicates partial support)

Beyond these requirements, we also aim to optimize inter-FPGA communication, since a multi-FPGA system is inevitable when a design exceeds single-chip capacity. Prior work addresses this using Time-Division Multiplexing (TDM) to optimize bandwidth [10], [11], but often overlooks intra-FPGA routing congestion or low channel utilization.

Commercial platforms like Synopsys ZeBu [12] and Cadence Protium [13] support automatic FPGA partitioning for accelerated simulation, but are expensive and lack resource-sharing optimizations, limiting scalability.

We limit our discussion to the logic-level TDM method [14] without live reconfiguration, as it introduces configuration delay into simulation, however small. Likewise, hypervisors [15], vFPGA [16], and partial reconfiguration [17] on FPGA are outside the scope of this work.

[§]These authors contributed equally to this work.

To address the challenges above, we propose a novel resource-efficient FPGA mapping methodology to accelerate large-scale NPU simulation with limited FPGA resources, with the following key contributions:

- A Chain-based Time-Division Multiplexing (CTDM) technique employing an efficient TDM strategy for significant resource reduction, including LUTs and FFs.
- An optimized FPGA mapping methodology for CTDM-applied designs utilizing built-in primitives, effectively reducing routing congestion and logic depth.
- An automated CTDM compiler applicable to diverse source design languages and supporting a wide range of FPGA target devices for deployment.
- An inter-FPGA latency-hiding technique specifically developed to improve CTDM scalability for large-scale NPU simulation in multi-FPGA environments.

II. Background

This section discusses how FPGAs can be utilized to accelerate large-scale hardware design simulations. Then, we explain TDM techniques 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. Limitations of Prior TDM Resource Sharing Techniques

To improve area efficiency on FPGAs, prior works have explored several TDM approaches, including memory-based TDM (MEM-TDM), multiplexer-based TDM (MUX-TDM), and multi-pumping TDM (MP-TDM). Although each reduces resource usage in different contexts, all present scalability and timing limitations for large datapath workloads.

MEM-TDM replicates logic virtually by saving and restoring internal states across time slices. Used in FPGA-based multi-core emulation [18], [2], this approach enables thread-level context switching, but incurs high memory and bandwidth overheads, making it inefficient for compute-bound designs with large amounts of internal state.

MUX-TDM reuses stateless combinational logic via multiplexed data paths [19], [20], [21], [22]. While effective for pipelined or asynchronous modules, it increases critical path delay and routing complexity due to dense control and muxing logic, especially in wide or replicated datapaths.

MP-TDM exploits clock-domain frequency scaling to efficiently time-share fixed-function units such as DSPs [23], [24], [25], [26], [27], [28]. This method avoids added latency, but requires strict clock alignment and does not scale well with high TDM-ratio resource sharing.

Figure 1(a) illustrates the baseline module, which consists of four identical combinational logic patterns and their state registers, S. Figure 1(b–d) summarizes the above techniques, showing how each handles data movement and control compared to the baseline. While effective within their design scopes, they scale poorly for compute-dense workloads with sequential dependencies and high interconnect demand.

To address these challenges, we propose *Chain-based TDM (CTDM)* (Figure 1(e)), which serializes logic patterns across a unidirectional chain. CTDM combines the compactness of MUX-TDM with the benefit of requiring neither state management nor multi-clock design, while enabling high-ratio TDM that is impractical in MP-TDM.

B. Large-Scale FPGA Simulation Requirements

Scaling large hardware designs across multiple FPGAs inherently requires partitioning and distributing logic, which introduces inter-FPGA communication over high-speed serial I/O (HSIO) links. These links, typically based on serializer-deserializer (SERDES) interfaces, serialize parallel data for transmission and reconstruct it at the receiver. To this end, supporting logic is needed, such as FIFOs, data alignment units, and clock domain crossing modules, to ensure data integrity and timing consistency.

As the TDM ratio in SERDES increases, allowing more logical signals to share a single physical link, pipeline depth, and communication latency also grow. Prior works mitigate this overhead by optimizing signal-to-channel assignments [29], TDM ratios [30], [10], [31], or minimizing inter-FPGA communication via partitioning algorithms [32], [33], [34].

However, these methods typically assume steady-state traffic where HSIO pipelines remain full. In practice, especially with TDM-based resource sharing, bubbles or idle cycles can occur, reducing efficiency. We observed such stalls when applying CTDM across multiple FPGAs.

To support CTDM in multi-FPGA simulation environments, Section III-C identifies the root causes of inter-FPGA communication inefficiency and introduces a latency-hiding solution.

C. Commercial Hardware Simulator

Commercial platforms like Synopsys ZeBu and Cadence Protium enable hardware-accelerated simulation of extremely

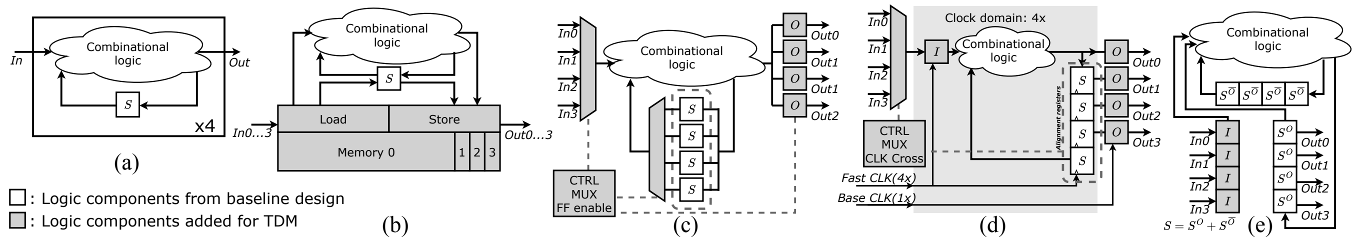

Fig. 1: TDM resource-sharing techniques: (a) baseline, (b) MEM-TDM, (c) MUX-TDM, (d) MP-TDM and (e) CTDM (ours).

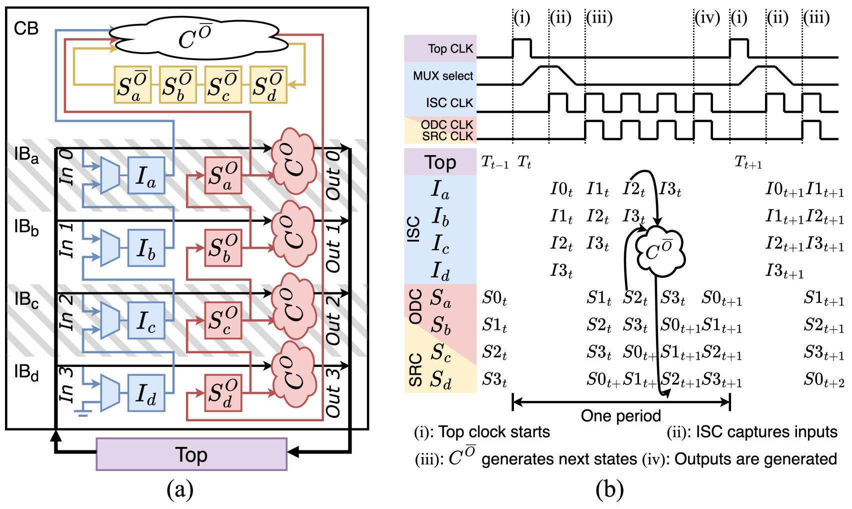
CB
$C^{\bar{O}}$
$S^{\bar{O}}_a$ $S^{\bar{O}}_b$ $S^{\bar{O}}_c$ $S^{\bar{O}}_d$
IB$_a$ IB$_b$ IB$_c$ IB$_d$
In 0 In 1 In 2 In 3
I_a I_b I_c I_d
S^O_a S^O_b S^O_c S^O_d
C^O
Out 0 Out 1 Out 2 Out 3
Top
(a)
(i) (ii) (iii) (iv)
Top CLK
MUX select
ISC CLK
ODC CLK
SRC CLK
Top
ISC
ODC
SRC
T_{t-1} T_t T_{t+1}
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 dataflow for calculating $S2_{t+1}$.

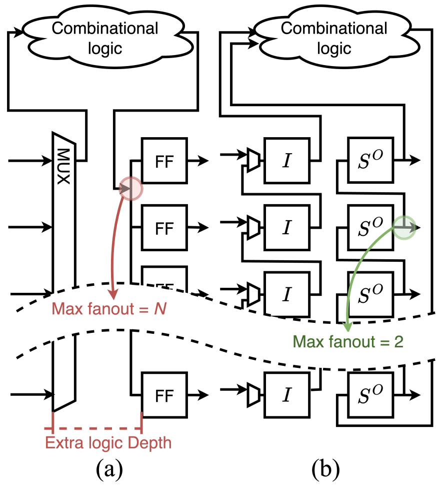

Fig. 3: Comparison of max fanout and logic depth in (a) MUX-TDM and (b) CTDM.

large designs using their respective compilers. While capable of handling tens of billions of gates, full-chip simulation often does not scale well due to the high cost of additional servers and inter-server communication bottlenecks. Additionally, their compilers support only automatic partitioning and lack resource usage optimizations. These systems use the AMD Vivado backend to map designs to FPGAs. As both ZeBu 5 server and Protium X2 use the AMD VU19P chipset, our compiler targets this architecture for CTDM-based simulation, as described in Section III-B.

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. This section details how we enabled resource savings while maintaining simulation performance.

A. Chain-based TDM

Chain-based TDM (CTDM) is a resource-sharing technique inspired by design for testability (DFT) scan chain methodologies. A scan chain inserts multiplexers (MUXes) next to the flip-flops(FFs) of the design under test (DUT), enabling test patterns to be shifted in and out of the DUT through a limited number of pins. Following a similar principle to scan chains, CTDM utilizes serialization to feed the resource-shared logic. Inputs are captured via MUXes and shifted sequentially into the shared logic, enabling it to perform computations over time that mimic the behavior of multiple distinct 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, sequentially 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 constructed by eliminating redundant logic patterns from the target modules. The remaining logic outside the target modules is named the 'top module'.

Internally, the CTDM module core consists of three FF chains, which are color coded in Figure 2 (a). This architecture reduces routing complexity via simple wiring. The chains are defined as follows:

- Input Serializer Chain (ISC): The blue chain consists of 2-to-1 MUXes, I FFs, and their connections. It serializes parallel inputs from the top module before passing them to the shared logic. The ISC performs two operations, each 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): The yellow chain 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 sequentially through SRC without requiring modification or control.
- Output Deserializer Chain (ODC): The red chain includes S^O FFs and logic C^O, 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.

Next, two CTDM components, C^O and $C^{\bar{O}}$, are derived from the original combinational logic in the target module. While C^O computes values directly for the outputs, $C^{\bar{O}}$ is the subset of the original combinational logic used to calculate the next state for all flip-flops 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. Based on their roles, the components of the CTDM module are grouped into two functional blocks, as shown in Figure 2(a).

- Compute Block (CB): $C^{\bar{O}}$ and SRC are grouped together and time-multiplexed for resource sharing. One CB represents the repeated logic patterns in the target modules.
- Interface Block (IB): Each IB, comprising one slice of ISC and ODC, interfaces between I/O and the CB. Acting as a wrapper, the IB provides interfaces identical to those of the target module. Consequently, the CTDM module can readily replace the target module. Note that multiple IBs are instantiated based on the TDM ratio (e.g., four IBs for N=4), alongside a single CB.

2) CTDM operation: Figure 2(b) provides a detailed illustration of the CTDM operating flow with a timing diagram for each clock domain. 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 $S3_{t+1}$, fill 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 substantially mitigates physical implementation complexity by minimizing 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 introduces substantial control logic, such as MUXes (e.g., the MUX in front of S in Figure 1(c)) and a finite-state machine to generate MUX select and register enable signals. In CTDM, the only added component from the baseline design is ISC to capture parallel inputs before serialization. This enables only one level increase in the logic depth when applying CTDM. In Section IV-B2, we compare CTDM and MUX-TDM with respect to routing congestion and logic depth based on the experiment results.

4) Use of FPGA primitive: Our CTDM method uses built-in FPGA primitives to implement SRC, which constitutes the largest portion of replaced logic in the original design. This crucial optimization enables maximum resource efficiency in FPGA-accelerated simulation.

In AMD FPGAs, SRCs are implemented using Shift Register LUT (SRL) primitives—specifically, SRL16E or SRL32E [35]. These macros reduce wiring overhead and save resources by avoiding the use of built-in flip-flops. Figure 4(a) shows the SRL structure. However, SRLs provide only a single output from the selected stage and do not support tapping intermediate outputs, making them unsuitable for components like the ISC or ODC. When the TDM ratio N is less than 16, the remaining $(16 - N)$ stages are unused and cannot be repurposed. However, a TDM ratio of 16 generally yields sufficient savings, and the replicated logic patterns in NPUs often exhibit higher parallelism, exceeding the benefits of a TDM ratio of 16.

In Intel FPGAs, CTDM uses the TriMatrix embedded memory IP [36], specifically the Memory Logic Array Block (MLAB), which is the most abundant resource type and can be configured as a shift register (Figure 4(b)). MLABs can output only at depths that are multiples of three, limiting the TDM ratio to 3, 6, 9, or 12.

Although SRLs and MLABs can be used as distributed RAM or FIFO, they are inefficient for large-capacity buffering due to routing overhead from interconnecting instances. This often leads to under-utilization of these resources, making CTDM well suited to repurpose these unused macros.

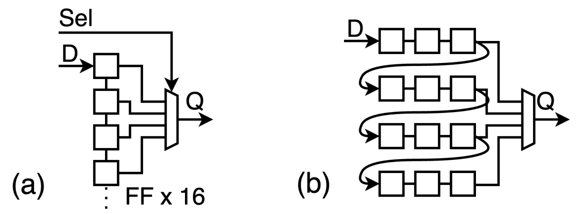

Fig. 4: FPGA shift register: (a) AMD SRL (b) Intel MLAB.

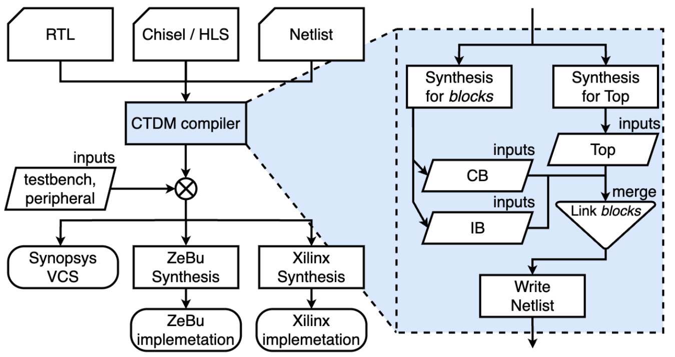

Fig. 5: Overview of CTDM compiler operation flow.

B. CTDM Compiler

The simplicity of the CTDM structure facilitates its automated insertion into baseline designs. Our CTDM compiler, implemented using Tcl scripts within the AMD Vivado Tcl interpreter, supports a wide range of FPGA devices, enabling deployment across diverse environments. Since the compiler operates on post-synthesis netlists, it supports designs written in nearly any hardware description language. As illustrated in Figure 5, this automated flow consists of three stages: (1) IB generation, (2) CB generation, and (3) top module linking.

In IB generation, the process begins by synthesizing the target module. Output state flip-flops (S^O) are identified via fan-in tracing from output ports, filtering FF cells using synthesis tool commands (e.g., Vivado, Design Compiler). These S^O are shown in red in Figure 6(a). The compiler then removes internal combinational logic and FFs (excluding S^O). A MUX and FF are inserted at each input port to build the input serialization chain. The ISC MUX input, ISC FF output, and the D/Q pins of the ODC FF (S^O) are exposed as IB ports, forming the complete IB structure in Figure 6(b).

In CB generation, the D/Q nets of S^O are connected to CB ports, and the S^O are removed. Internal FFs are replaced with enabled shift registers, implemented by adding a MUX to the D input of SRL primitives (due to the lack of a native enable pin), as shown in Figure 7(a). If the module includes SRAM, the CB increases the address folding count and inserts a shift register to latch the SRAM output due to its one-cycle latency, as shown in Figure 7(b).

In top module linking, the top module is synthesized with the target modules as black boxes. After synthesis, the compiler inserts clock control logic and the CB, replaces black boxes with IBs (Figure 6(e)), and connects ports across the top, IBs, and CB. The compiler takes the instance names of the top and target modules, along with the synthesis file list, as input. Although it does not automatically detect repetitive logic patterns for CTDM insertion, it scans the target module's internal logic to identify and insert CTDM logic.

C. CTDM in multi-FPGA environment

Although CTDM can reduce FPGA resource utilization, applying it in a multi-FPGA environment for large-scale NPU simulation presents a significant challenge. This section identifies such the challenge and proposes a solution.

1) Pipeline bubbles and inter-FPGA latency exposure: As discussed in Section II-B, inter-FPGA communication introduces considerable latency, even with applied optimizations. In TDM, the top-level control logic initiates the TDM operation on the rising edge of the clock, and logic replication is performed across multiple cycles by transmitting serialized data through a pipeline. Once multiplexing is completed, the data are returned to the top-level module, which then provides new inputs to the TDM logic for the next operation. This sequential control flow becomes more problematic when HSIO latency is introduced. Figure 8(a) illustrates how a CTDM module can be partitioned between FPGA A and B, with data transmitted over HSIO. While the first data batch completes a full round trip between FPGA B and FPGA A, the top-level module must remain idle. During this idle period, HSIO cannot transmit new data from B to A due to the lack of incoming data from the top-level module, resulting in pipeline bubbles that expose otherwise hidden communication latency.

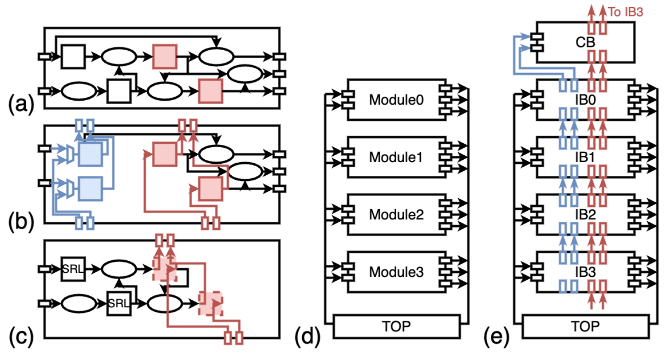

Fig. 6: CTDM compilation process: From the target module (a) to IB (b) and CB (c) generation. At a higher level, baseline design (d) will be converted to CTDM design (e).

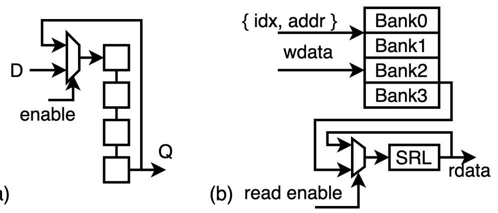

Fig. 7: CTDM Primitives: (a) Enabled Shift Register (b) Modified SRAM for address folding and an output shift register for timing alignment.

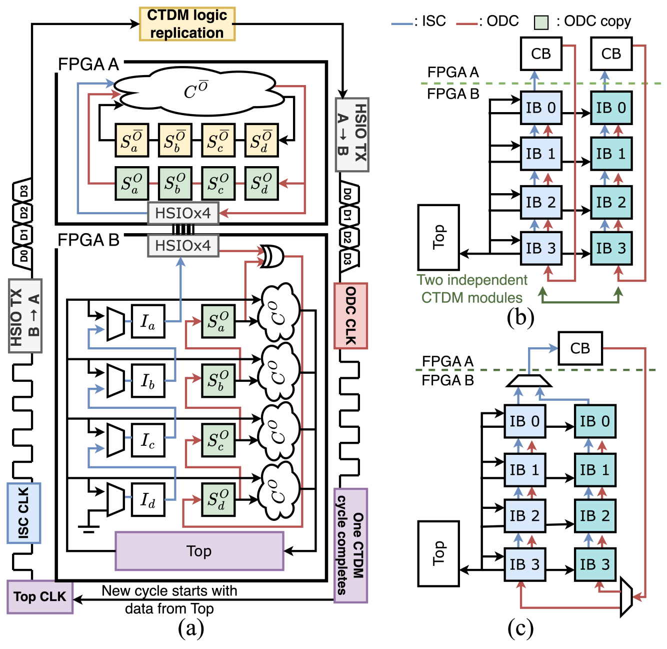

Fig. 8: CTDM partitioning for HSIO latency hiding (a) with ODC FF copy on each FPGA, (b) without IB interleaving, and (c) with IB interleaving.

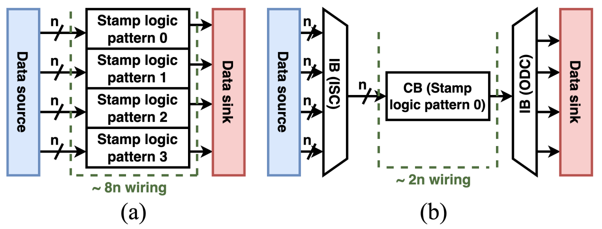

Fig. 9: Partitioning cut wiring count reduction in CTDM: (a) baseline, (b) cut made in between CB and IB.

This behavior undermines the benefits of CTDM by creating visible latencies in the inter-FPGA dataflow.

2) CTDM Partitioning for reducing inter-FPGA wiring: In most cases where manual partitioning is used, cuts within the CTDM module can be avoided, thereby preventing the performance degradation described earlier. In fact, due to the reduced wiring between CB and IBs compared to the baseline design (Figure 9(a)), placing a cut between CB and IBs is the optimal partitioning strategy (Figure 9(b)). Automatic partitioning tools, such as the ZeBu compiler, also tend to insert cuts between CB and IB, as they are unaware of potential performance penalties. In most cases, this remains a favorable design choice because it minimizes the wiring required between FPGAs. Figure 8(a) illustrates how this cut can be implemented using four HSIO channels. Note that we have two identical copies of ODC FF in both FPGA A and B devices. This makes an immediate calculation in CB possible when receiving input values from ISC in FPGA A.

3) IB Interleaving for Latency Hiding: To mitigate performance degradation while retaining the wiring efficiency of inter-FPGA communication, interleaving IBs from two independent CTDM operations can hide HSIO latency. The root cause of incomplete HSIO pipeline utilization is data unavailability: the top module cannot send new data until it receives output from the previous CTDM cycle. To address this, we introduce two parallel pipelines that share the same CB, allowing better HSIO channel utilization. Figure 8(b) shows two independent CTDM operations each using one HSIO channel, and Figure 8(c) illustrates latency hiding by interleaving two IB groups using a shared CB and a single HSIO channel in an alternating ("ping-pong") manner. Figure 10(a) shows the timing and latency breakdown of a CTDM operation partitioned across FPGA A (CB) and FPGA B (IBs). At each top clock edge, data enters the CTDM module, travels from FPGA B to A and back through HSIO, and returns to the top module (the A-to-B return path is omitted for simplicity). Figure 10(b) depicts a baseline CTDM timing without ODC FF copy and IB interleaving. Four data frame transmissions from ISC to CB in each channel are shown between top clock edges, leaving idle gaps (bubbles) that decrease overall simulation throughput. Figure 10(c) shows how alternating data transmission between two IB groups via MUX selection keeps the HSIO link fully utilized while using only one HSIO channel. This ping-pong scheduling fills the pipeline and hides latency, enabling a higher top clock frequency and improved simulation performance.

IV. Evaluation

A. Experimental setup

We evaluated our proposed method using three designs: a neural engine (NE), NE with a system-on-chip (NE+SoC), and NVDLA [37]. When PCIe FPGA is used, the test system has Intel's i9-12900 CPU with 32 GB DRAM. For mapping designs to Synopsys ZeBu 5 server [12], we utilized Synopsys's ZeBu compiler. In all cases, a CTDM TDM ratio of 16 was applied to optimize performance and resource usage.

To fairly compare our CTDM approach with the MUX-TDM design proposed in [21], we adopted their function merging and coarse-grained resource sharing strategy to maximize resource savings. In addition, their method reduces the number of multiplexers and demultiplexers inserted through MUX propagation and sharing optimizations that merge redundant logic. Additionally, we observed that demultiplexers can often be avoided by using registers with enable signals to latch outputs conditionally, thereby conserving valuable MUX resources on the FPGA.

1) Neural Engine (NE): 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

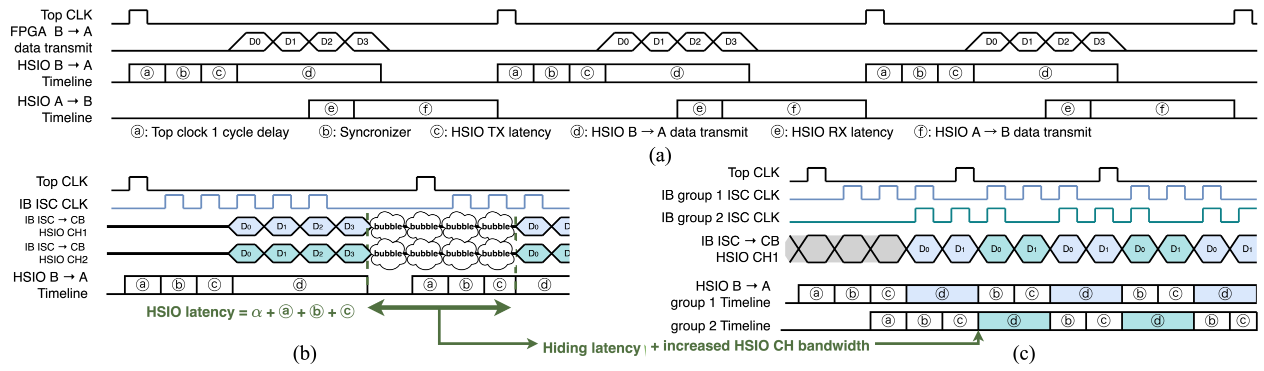

Fig. 10: Our inter-FPGA latency-hiding method illustrated in the timing diagram: (a) the latency breakdown for a typical HSIO, (b) CTDM without IB interleaving, and (c) CTDM with IB interleaving.

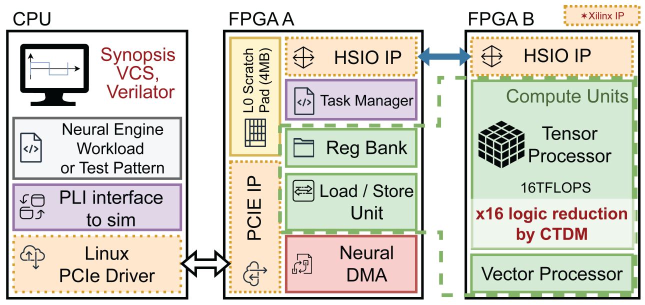

Fig. 11: Overview of NE partitioning on FPGA with CTDM.

provides 16 TFLOPS of compute performance. Its architecture consists of vector processors, tensor processors, and load/store units. 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 system composed of a CPU and two AMD U250 FPGAs (Figure 11).

2) Neural Engine with System-On-Chip (NE+SoC): In the larger version of our NPU design, each die contains 16 NEs, and four dies are combined to form a single chiplet. Also, 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). To simulate NE+SoC on FPGA, we used Synopsys ZeBu 5.

3) NVIDIA deep learning accelerator (NVDLA): To demonstrate CTDM's effectiveness, we also used the open-source NVDLA [37]. In this test, the largest NVDLA configuration *nv_full*, which includes 2,048 INT8 MACs, 1,024 MACs (INT16, FP16), and a 512 KB buffer, was used.

CTDM was applied to the convolution MAC array (CMAC) and the convolution accumulator (CACC) of NVDLA's architecture. 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 12 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 (ZeBu) and U250 FPGA devices. In the early stage of our research, we mapped arithmetic operations onto DSP blocks to reduce LUT usage on

TABLE II: 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 III: 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%)

the FPGA. This DSP-mapped version of the design required four engineers working for six months and resulted in only a 20 percent reduction in resource utilization. In comparison, CTDM achieved a 71% of LUT reduction in less than 12 hours using an automated compiler flow.

For the NE+SoC experiment, we used ZeBu 5 server for simulation. Without LUT reduction, simulation was impossible since it requires 204 AMD VU19P FPGAs. However, with CTDM, we implemented it using 144 FPGAs, the exact number of FPGA chips included in three units of ZeBu 5.

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 [38] deployed *nv_full* on the Amazon EC2 FPGA cloud using FireSim [5], but removed the INT16 and FP16 convolution engines and wrapped the NVDLA RTL as a black box, leaving FireSim's optimization functionalities unavailable to NVDLA. In our experiment, their stripped-down NVDLA only takes 479k LUTs and 351k FFs on U250 FPGA without any resource sharing applied. Table II and Table III present the comparison of resource counts in the FPGA deployment of NE and NVDLA in the baseline, MUX-TDM, and CTDM schemes.

2) Routing congestion and logic depth: Comparing the three conditions—baseline, MUX-TDM, and CTDM—shows that CTDM reduces both total net count and total fanout. These two reductions help reduce routing congestion. We tested three configurations on the U250 FPGA: baseline 16 MAC

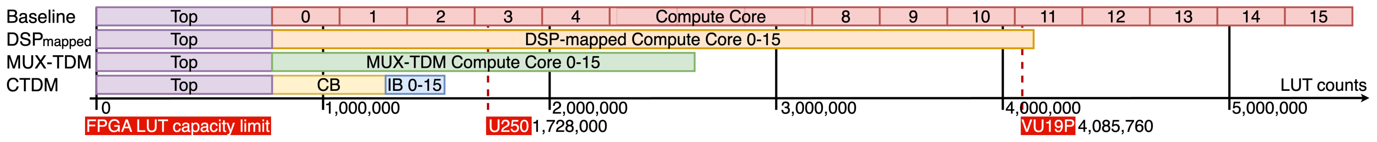

Fig. 12: LUT reduction result for NE.

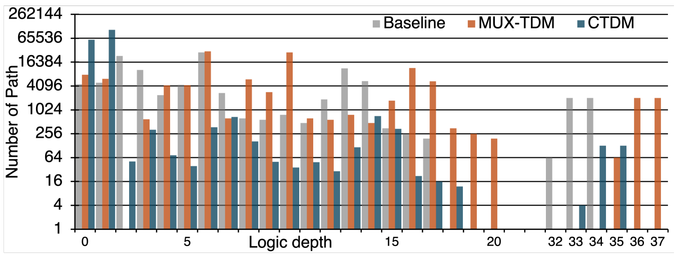

Fig. 13: NVDLA CMAC logic depth histogram.

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 depth, and the number of nets with the highest logic depth. Table IV 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 13 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) Multi-FPGA Simulation with CTDM: To evaluate CTDM in a multi-FPGA environment, we compared on-premise FPGAs (e.g., U250) and Synopsys ZeBu using the NE design. On ZeBu, we applied CTDM without latency hiding and relied on its automatic partitioning. On U250, we compared CTDM against a baseline using the same partitioning but without any TDM techniques.

In all cases, simulation speed was measured by the slowest clock. As shown in Table V, CTDM on U250 achieved 397 kHz, outperforming ZeBu without CTDM (57.8 kHz). Section II-C notes that, ZeBu limits user-level optimization and prevents customized inter-FPGA communication.

Despite a theoretical 16x slowdown from its TDM ratio of 16, CTDM reduced FPGA usage on ZeBu by 5x with only a 5x performance penalty. This illustrates a performance–resource trade-off: CTDM enables resource savings and multi-tenancy on server simulators, but performance benefits are more significant in user-controlled, on-premise environments.

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 CTDM method on FPGA. For the NE case, the simulation took 372.8 seconds on the CPU using the VCS simulator, whereas it completed in just 0.6 seconds on our FPGA-accelerated 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 took 0.88 hours, while the FPGA simulation completed it in just 0.867 seconds, achieving a 3,653× speedup for the AlexNet experiment. Furthermore, we compared our approach to [38] for running quantized YOLOv3 in INT8 format. FireSim with NVDLA processed one frame in 0.133 seconds, whereas our method took 0.358 seconds, which is 2.69 times slower, but with 43% fewer LUT resources without modifying NVDLA.

TABLE IV: 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

TABLE V: Operating frequency of NE deployed on two FPGA-based devices.

Device	CTDM applied	Top freq. (kHz)	FPGA counts	LUT count (% per one FPGA)
ZeBu	N (1:1)	57.8	VU19P×5	5.5M (134%)
ZeBu	Y (16:1)	11.4	VU19P×1	1.6M (39%)
U250	N (1:1)	318	XCU250×5	5.5M (318%)
U250	Y (16:1)	97.4	XCU250×2	1.6M (92%)
U250	Y (16:1) + latency opt	397	XCU250×2	1.6M (92%)

V. Conclusion

This paper investigated mapping large NPU designs with repeated logic patterns to FPGAs in a resource-efficient manner. Using a novel chain-based TDM (CTDM) technique, we achieved a 66.5% reduction in LUT utilization and an 82.4% reduction in FF utilization in FPGA mapping of the NVDLA design. Furthermore, to extend CTDM to multi-FPGA simulation, inter-FPGA latency-hiding technique with the partitioning strategy was proposed. The implementation of our own NPU core design in U250 demonstrated a 24.8% improvement in the operating frequency with fewer FPGA boards. Ultimately, we successfully applied CTDM to a massive 4-die chiplet NPU on ZeBu 5 server, mapping the entire design onto just 144 VU19P FPGA chips.

References

[1] 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.

[2] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer, "Hasim: Fpga-based high-detail multicore simulation using time-division multiplexing," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, 2011, pp. 406–417.

[3] S. Mosanu, M. N. Sakib, T. Tracy, E. Cukurtas, A. Ahmed, P. Ivanov, S. Khan, K. Skadron, and M. Stan, "Pimulator: A fast and flexible processing-in-memory emulation platform," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2022, pp. 1473–1478.

[4] F. Elsabbagh, S. Sheikhha, V. A. Ying, Q. M. Nguyen, J. S. Emer, and D. Sanchez, "Accelerating rtl simulation with hardware-software co-design," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 153–166.

[5] 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.

[6] 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.

[7] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Proceedings of the 49th annual design automation conference*, 2012, pp. 1216–1225.

[8] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton *et al.*, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *Ieee Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[9] U. Berkeley, "Firesim manual - including verilog ip," [url] https://docs.fires.im/en/latest/Advanced-Usage/Generating-Different-Targets.html, 2024 (accessed April 20, 2025).

[10] 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.

[11] 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.

[12] 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).

[13] Cadence, "Protium enterprise prototyping," [url] https://www.cadence.com/en_US/home/tools/system-design-and-verification/emulation-and-prototyping/protium.html, 2024 (accessed April 19, 2025).

[14] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A time-multiplexed fpga," in *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No. 97TB100186)*. IEEE, 1997, pp. 22–28.

[15] J. Ma, G. Zuo, K. Loughlin, X. Cheng, Y. Liu, A. M. Eneyew, Z. Qi, and B. Kasikci, "A hypervisor for shared-memory fpga platforms," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 827–844.

[16] B. Ramhorst, D. Korolija, M. J. Heer, J. Dann, L. Liu, and G. Alonso, "Coyote v2: Raising the level of abstraction for data center fpgas," *arXiv preprint arXiv:2504.21538*, 2025.

[17] AMD, "Vivado design suite user guide: Dynamic function exchange," [url] https://docs.amd.com/r/en-US/ug909-vivado-partial-reconfiguration, 2025 (accessed August 9, 2025).

[18] E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, K. Mai, and B. Falsafi, "Protoflex: Towards scalable, full-system multiprocessor simulations using fpgas," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 2, no. 2, pp. 1–32, 2009.

[19] J. Hansen and M. Singh, "A fast hierarchical approach to resource sharing in pipelined asynchronous systems," in *2012 IEEE 18th International Symposium on Asynchronous Circuits and Systems*. IEEE, 2012, pp. 57–64.

[20] Q. Si and B. Carrion Schaefer, "Pepa: performance enhancement of embedded processors through hw accelerator resource sharing," in *Proceedings of the Great Lakes Symposium on VLSI 2023*, 2023, pp. 23–28.

[21] T.-H. Juang, C. Schlaak, and C. Dubach, "Let coarse-grained resources be shared: Mapping entire neural networks on fpgas," *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 5s, pp. 1–23, 2023.

[22] Y. Gorbounov and H. Chen, "Achieving high efficiency: Resource sharing techniques in artificial neural networks for resource-constrained devices," in *Journal of Physics: Conference Series*, vol. 2719, no. 1. IOP Publishing, 2024, p. 012005.

[23] A. Canis, J. H. Anderson, and S. D. Brown, "Multi-pumping for resource reduction in fpga high-level synthesis," in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2013, pp. 194–197.

[24] B. Ronak and S. A. Fahmy, "Minimizing dsp block usage through multi-pumping," in *2015 International Conference on Field Programmable Technology (FPT)*. IEEE, 2015, pp. 184–187.

[25] ——, "Multipumping flexible dsp blocks for resource reduction on xilinx fpgas," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 9, pp. 1471–1482, 2016.

[26] S. A. Alam and O. Gustafsson, "Implementation of time-multiplexed sparse periodic fir filters for frm on fpgas," in *2011 IEEE International Symposium of Circuits and Systems (ISCAS)*. IEEE, 2011, pp. 661–664.

[27] G. Brignone, M. T. Lazarescu, and L. Lavagno, "A dsp shared is a dsp earned: Hls task-level multi-pumping for high-performance low-resource designs," in *2023 IEEE 41st International Conference on Computer Design (ICCD)*. IEEE, 2023, pp. 551–557.

[28] R. Shi, Y. Ding, X. Wei, H. Li, H. Liu, H. K.-H. So, and C. Ding, "Ftdl: A tailored fpga-overlay for deep learning with high scalability," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[29] M. Inagi, Y. Takashima, Y. Nakamura, and A. Takahashi, "Optimal time-multiplexing in inter-fpga connections for accelerating multi-fpga prototyping systems," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 91, no. 12, pp. 3539–3547, 2008.

[30] 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.

[31] S.-H. Liou, S. Liu, R. Sun, and H.-M. Chen, "Timing driven partition for multi-fpga systems with tdm awareness," in *Proceedings of the 2020 International Symposium on Physical Design*, 2020, pp. 111–118.

[32] Ü. V. Çatalyürek and C. Aykanat, "Patoh (partitioning tool for hypergraphs)." 2011.

[33] M.-H. Chen, Y.-W. Chang, and J.-J. Wang, "Performance-driven simultaneous partitioning and routing for multi-fpga systems," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 1129–1134.

[34] D. Zheng, X. Zang, and M. D. Wong, "Topopart: a multi-level topology-driven partitioning framework for multi-fpga systems," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2021, pp. 1–8.

[35] 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).

[36] Intel, "Stratix 10 embedded memory ip references - shift register (ram-based) intel fpga ip," [url] https://cdrdv2.intel.com/v1/dl/getContent/826816?fileName=ug-s10-memory-683423-826816.pdf, p. 119, 2024 (accessed April 13, 2025).

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

[38] 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.