MMM: FPGA-accelerated hardware simulator for resource-efficient deployment with automatic compiler

Hyunje Jo*
aleph@rebellions.ai
Rebellions
Seongnam-si, South Korea

Han-sok Suh†
hs2239@cornell.edu
Cornell Tech
New York, USA

Hyungseok Heo*
hyungseok.heo@rebellions.ai
Rebellions
Seongnam-si, South Korea

Jinseok Kim*
jinseok@rebellions.ai
Rebellions
Seongnam-si, South Korea

Hyunsung Kim*
hyunsungkim@rebellions.ai
Rebellions
Seongnam-si, South Korea

Boeui Hong*
boeui.hong@rebellions.ai
Rebellions
Seongnam-si, South Korea

Jungju Oh*
jungju.oh@rebellions.ai
Rebellions
Seongnam-si, South Korea

Sunghyun Park*
sunghyun.park@rebellions.ai
Rebellions
Seongnam-si, South Korea

Jinwook Oh*
j.oh@rebellions.ai
Rebellions
Seongnam-si, South Korea

Sunghwan Jo‡
sunghjo@synopsys.com
Synopsys Korea
Yongin-si, South Korea

Kangwook Lee‡
kangwook@synopsys.com
Synopsys Korea
Yongin-si, South Korea

Jae-sun Seo†
js3528@cornell.edu
Cornell Tech
New York, USA

Abstract

This paper proposes a novel approach to accelerating large ASIC design simulations on FPGA through a Module Multiplexing Method (MMM) and its automatic compiler. Leveraging a resource-sharing technique based on Time-Division Multiplexing (TDM), this method achieves a 93.8% reduction in resource utilization on FPGA. Compared to the conventional multiplexer-based TDM resource-sharing technique, our shift register-based TDM reduces the total sum of fanout by 85.7%, significantly alleviating routing congestion on FPGA. To scale out and accommodate the size of large Application-Specific Integrated Circuit (ASIC) designs, we are using a multi-FPGA environment to deploy our simulator. Our inter-FPGA communication strategy for MMM reduces interface link latency by utilizing TDM merged with a novel design partitioning method. The fully automated MMM compiler generates resource-efficient and FPGA-accelerated simulation systems from design sources written in various Hardware Design Languages (HDLs) such as Verilog, VHDL, HLS, and Chiesel. This method is compatible with a broad range of FPGA devices, from small on-premise boards with PCIe form-factor to server-grade hardware simulation platforms such as Synopsys ZeBu. When applied to NVIDIA's open-source machine learning accelerator, NVDLA, we could deploy NVDLA full variant on AMD U250 FPGA device, which would be impossible to fit in a FPGA without MMM method. This approach demonstrated a 3,653x acceleration in NVDLA's simulation time over Synopsys's VCS simulator on a CPU. By providing faster simulation results, it can expedite the hardware-software co-design process, benefiting both hardware and software developers. This method has already been implemented in the simulation and verification of commercial machine learning accelerator designs to evaluate calculation patterns for Large Language Models (LLMs), and we successfully emulated a 4-die 1024 TFLOPS chiplet using 144 FPGAs on Zebu5.

CCS Concepts

• **Hardware → Board- and system-level test**; **Application specific processors**; • **Computing methodologies → Simulation environments**.

Keywords

Field-Programmable Gate Arrays, hardware simulation, hardware verification, simulation accelerator

ACM Reference Format:
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. 2024. MMM: FPGA-accelerated hardware simulator for resource-efficient deployment with automatic compiler. In *Proceedings of 33rd ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '25).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

FPGA '25, Feb 27–Mar 1, 2025, Monterey, CA

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

As the CMOS process nodes continue to shrink, the total number of transistors that can be integrated into single chip has increased and large-scale ASIC simulation presents greater challenges. In this regard, Field-Programmable Gate Arrays (FPGAs) can be used for fast-prototyping and testing of ASIC hardware designs. However, due to the limited resources available on FPGA, deploying large hardware designs on FPGA is challenging. To address this issue, prior works have deployed large designs on FPGA clusters [3, 14, 18], as scaling out across multiple FPGAs becomes necessary when the target design exceeds the capacity of a single FPGA device.

However, introducing inter-FPGA communication in the simulation system can pose a performance bottleneck. To alleviate this problem, the authors of [13] investigated TDM based optimization with an analytical approach. This work improved system clock frequency by 7% in inter-FPGA routing, enabling the FPGA cluster to scale out up to 400K nodes. Another research focused on the FPGA I/Os using time-division-multiplexing (TDM) [15], aiming to decrease the maximum TDM ratio while satisfying the inter-FPGA communication requirements. Nonetheless, none of these works actually investigated applying TDM to the functional modules for the reducing resource utilization and relieving routing congestion. Another effort by [3] aimed to alleviate the bottleneck in inter-FPGA communication. To avoid the link latency between FPGAs, LVDS-based communication was employed; While LVDS-based communication avoids link latency, the limited I/Os on Xilinx FPGAs reduce its bandwidth. With clock synchronization, LVDS speed drops to 100 Mbps, requiring 500 differential I/Os to match the link speed of 25 Gbps of a 4-channel QSFP, making it unsuitable for large design partitioning due to bandwidth limitations. Without hardware systems that offer cable connectors for LVDS I/Os with signal integrity enhancements, like Synopsys HAPS, using LVDS for chip-to-chip communication is challenging.

Resource optimization is another major research direction for FPGAs. Researchers explored various resource sharing techniques to reduce the resource usage on FPGAs [7, 11, 16]. Sun et al. [16] attempted to share multi-cycle pipelined modules for hardware area optimization at a high level. This work employed an algorithm that selectively applies resource sharing at the module scope from a high-level hardware description; however, this approach shows limitations to implement the architectures that are already defined at RTL level. [17] explored module-wise reuse by applying multi-threading to soft-cores on FPGA to mimic multi-core behavior through resource sharing. However, this method is slow, as it requires completely storing/restoring the internal states, which is very time-consuming. Finally, TDM was applied to FPGA for resource usage reduction by [7, 11]. These TDM techniques were applied to logic and can reduce the resource usage on FPGAs by trading off performance and routing congestion. Hadjis et al. [7] presented resource sharing on FPGAs with HLS. By analyzing the type of operations and the logic resource used in specific FPGA devices, they identified which operation should be selectively multiplexed with TDM resource sharing to achieve better resource utilization. Nangia et al. [11] proposed bit-level ALU sharing with additional multiplexers for functionality selection and control logic absorption. Although TDM-based resource sharing offers significant resource savings and can be applied to already defined hardware architectures, the multiplexing clock speed cannot be increased indefinitely (e.g., overclocking), limiting the extent of resource sharing. Additionally, TDM aggregates input signals and broadcasts them to outputs after the TDM operation, which increases routing congestion and negatively impacts the performance of the FPGA design.

While the ability to deploy large ASIC designs to FPGA for accelerated simulation is important, usability is also a key aspect, as it can reduce design time in hardware/software co-design iterations. One crucial feature is automatic compiler support. Automatic simulator generation is essential for applying resource optimizations, as manual coding would be inefficient and time-consuming. Since hardware/software co-design involves frequent RTL modifications and re-evaluation, an automatic compiler can smoothly accommodate design changes without disrupting the design flow. In this regards, a number of architecture/ASIC simulators have been demonstrated to support automatic generation of their simulator system to many potential architecture designs. FireSim [9] presents a high-performance compiler system with resource sharing and automatic partitioning based on the Latency-Insensitive Bounded Dataflow Network (LI-BDN) method. This approach decouples FPGA clock cycles from the target design's clock cycles while maintaining cycle accuracy. Additionally, with the LI-BDN abstraction, FireAxe [18] was recently released, which supports automatic partitioning of target designs for multi-FPGA mapping on FireSim. Although FireSim's compiler, FireAxe, optimizes LUT usage through multi-threading, it does not yet support routing congestion optimization or advanced resource utilization techniques, leaving room for further exploration in FPGA-accelerated simulation systems with compiler support.

At the industry level, systems such as Synopsys ZeBu and Cadence Palladium offer automatic partitioning of large hardware designs across multiple FPGA chipsets and provide FPGA-accelerated simulation methods, making them both fast and user-friendly. However, these server-grade simulators are extremely expensive, and despite multi-user tenancy, sharing them efficiently is challenging, as server slots are often occupied by large design simulations. Furthermore, design optimization techniques like TDM or resource sharing are not supported, as they use consolidated clock system that does not allow for user intervention, preventing users from implementing their own optimizations.

To resolve these issues, we propose MMM (Module Multiplexing Method) and its automatic compiler to generate custom FPGA-accelerated simulation system with the following key contributions:

- MMM employs a resource TDM strategy that reduces LUTs and Flip-Flops (FFs) resource utilization with small fan-out requirement which helps easing routing congestion.
- Unlike traditional TDM methods, MMM can be applied at a module level, even when the module contains a complex internal state changes and output logic that depends on both input and the module's current state.
- Our MMM partitioning strategy reduces inter-FPGA bandwidth requirements and reduces QSFP link latency between FPGAs using a latency-hiding technique.
- The MMM compiler can be applied to any source design written in HDL. It also supports a wide range of FPGA-based target devices for deployment.

The remainder of this paper is organized as follows. Section 2 gives the preliminaries on the techniques typically utilized in latest FPGA-accelerated simulators like how logic multiplexing can be done on FPGAs and introduction on optimization of inter-FPGA communication. Section 3 explains how our MMM strategy works with differenciation from other TDM methods. In Section 4 and Section 5, we explain our two target design, commercial chip2 we designed and NVDLA which we used for evaluating the performance and benefit of our simulation accelerator system. Section 6 presents the experimental setup and results, and Section 7 concludes the paper.

2 Preliminaries

This section explains how FPGAs can be used for accelerating large hardware design simulations. The technique of TDM and optimization on inter-FPGA communication will be explained.

2.1 Time-Division Multiplexing

Time-Division Multiplexing (TDM) is commonly used for transmitting and receiving multiple independent data streams over a shared signal path, typically in telecommunications. However, on FPGAs, this technique can also be utilized to share common logic resources across different data streams, reducing the area of the logic design at the cost of reduced hardware performance. Figure 1 illustrates how the TDM method can be used for resource sharing on FPGA.

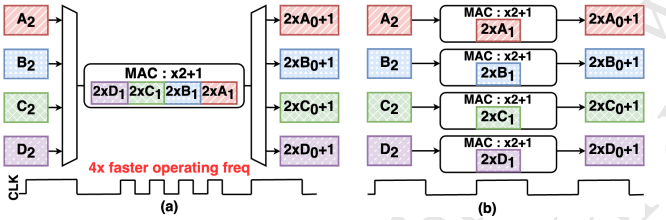

Figure 1: (a) MAC module with TDM ratio 4 applied. (b) equivalent module before TDM application.

When the maximum operating frequency of the target hardware is F_{spec}, the logic utilizing TDM must operate at N_{fold} times slower, where N_{fold} is the multiplexing or TDM ratio. This typically results in a design that is N_{fold} times slower than a non-TDM design on the same hardware.

TDM method is integrated into AMD's Deep Learning Processing Unit (DPU) IP design, which uses DSPs on FPGAs [1]. With the Virtex Ultrascale+ DSP48 primitive's maximum frequency at 775 MHz [2], applying a TDM folding factor of 16 limits system performance to 48.44 MHz. Additionally, applying TDM at the module level is challenging due to dependencies on the module's internal state and inputs. Furthermore, clock-crossing logic and multiplexers (MUXes) must be inserted to maintain interleaved data order, requiring manual identification and insertion of logic at specific points in the design.

2.2 Optimizing inter-FPGA communication

To scale a design across multiple FPGAs, a monolithic design must be partitioned and distributed across different FPGAs, requiring inter-FPGA communication via LVDS-based connections or high-speed interfaces like QSFP. While both enable high-speed communication, LVDS has bandwidth limitations due to limited I/O counts, and QSFP introduces link latency that degrades system performance. TDM can increase logical bandwidth in both cases [22], but it may still fall short of the partitioned design's bandwidth requirements. To ease this requirement, partitioning algorithms can be used [4, 5, 21], but this is a complex graph partitioning problem that requires detailed analysis and extra computation. Instead, we use MMM to combine our novel partitioning strategy with TDM bandwidth reduction, reducing QSFP link latency while meeting the partitioned design's bandwidth needs. This will be discussed further in Section 3.4.

2.3 CPU Hosted Hardware Simulation with FPGA Acceleration

The recent surge in developing accelerators for machine learning applications has driven the need for architectures with thousands of computing cores and massive parallelism. As chip designs grow, simulating them on CPUs becomes increasingly difficult due to the immense computational demands. FPGAs offer a solution with faster hardware simulation speeds and programmability compared to CPU-based simulations. However, deploying large designs entirely onto FPGAs is challenging. To address this, we developed a hybrid hardware simulation system that offloads compute unit and computation-intensive logic to FPGAs, while simulating the rest on a CPU in a conventional manner with software simulator. In our simulator, the CPU serves as the functional model, and the FPGA works as timing model, balancing computational load. Despite introducing inter-device delays, such as PCIe latency, this approach improves scalability and allows for simulating larger designs with fewer FPGA resources. Our system interfaces with software SoC design simulators (e.g., Verilator, Synopsys VCS), leveraging their debugging capabilities while accelerating high-load simulations on the FPGA. To mitigate inter-device latency, we developed a custom Linux driver for efficient data transfer, achieving near bare-metal performance. Figure 2 provides an overview of our FPGA-accelerated simulation system.

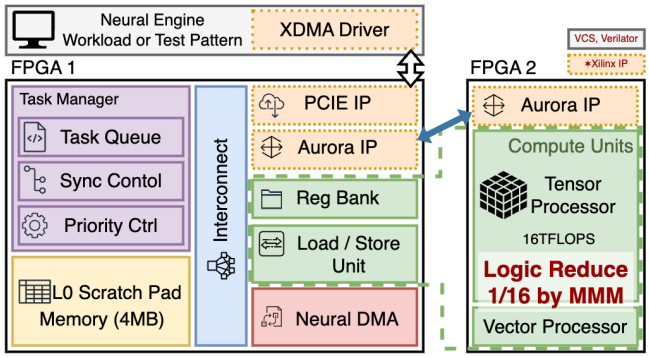

Figure 2: Overview of FPGA accelerated simulator system with MMM.

3 Module Multiplexing Method

The Module Multiplexing Method (MMM) is a resource-sharing technique based on TDM. Instead of using MUXes, MMM employs a shift register chain, allowing data to flow consistently through the pipeline. This eliminates the need for MUXes and associated control logic before pipeline registers, simplifying TDM usage on FPGAs. This also enables the development of an automatic compiler, as complex control logic for managing data order is no longer required. In the following sections, we will explain our technique with an example case, which uses TDM ratio of four.

3.1 Proposed MMM Structure

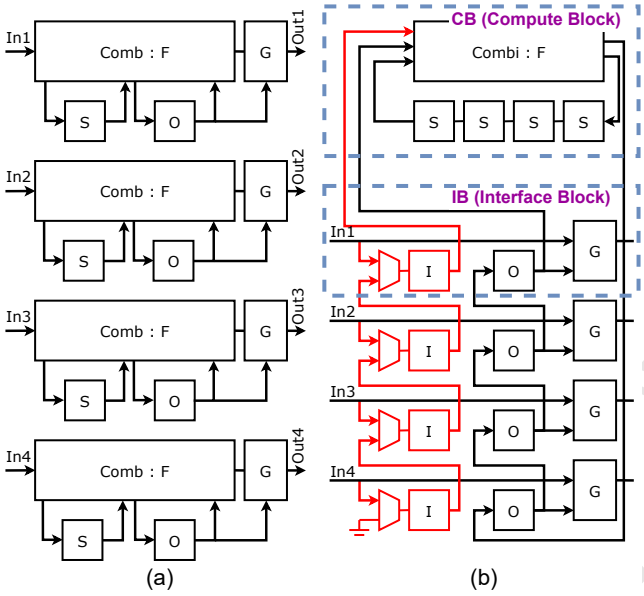

Figure 3: (a) Original design with four parallel modules. (b) MMM applied structure.

In Figure 3a, four identical modules are shown, consisting of combinational logic and Flip-Flops (FFs) to hold internal states of the logic. Within a single module, the module is divide into two combinational logic. If the portion of combinational logic is involved with output generation, it is marked as logic G and the rest of the logic is named as logic F. Logic F gets inputs from the parallel input source, $In1 \sim 4$, state register S_t and pre-output register O_t to generate an input to the pre-output register O_{t+1} and the next state of logic F, S_{t+1}. On the output side, G gets inputs from $In1 \sim 4$ and pre-output register O_t to generate output of the module, $Out1 \sim 4$. In summary, the relationship between each component of the logic can be formulated as Equation (1).

$$(S_{t+1}, O_{t+1}) = F(In, S_t, O_t)$$
$$Out = G(In, O_t) \quad (1)$$

To begin with, we select one 'target module' from four identical modules and apply our resource-sharing technique. MMM then creates three types of shift register chains and two types of sub-blocks using this target module. The configuration of MMM is shown in Figure 3b. Below explains each components of MMM.

- Input serializer chain: To share logic F across multiple data pipelines, input data is conveyed through an input serializer chain, shown as the red logic in Figure 3b. Using input-selecting MUXes, the input value is captured and the serialized input data is pushed to the shared logic F.
- State register chain: To store an internal state of F, original state registers, S, are reconstructed as a shift registers. In AMD FPGA, this can be mapped to Shift Register Look-up table (SRL) primitive. This saves wiring resources by eliminating the connection between FFs.
- Output deserializer chain: To preserve the behavior of the original parallel modules, logic G cannot be shared, and an original instance of logic G must be maintained for each data pipeline. The output shift register chain O is fed back to logic F when the output of current cycle is required for generating new outputs in the next cycle (e.g., in cases of accumulation).
- CB: Compute Block (CB) is a resource-shared compute logic with logic F and a state register chain. This submodule occupies the largest area on the FPGA, and the most significant resource savings are achieved by applying TDM to this submodule.
- IB: Interface Block (IB) composed of an single FF from input serializer chain, one FF from output deserializer logic, and one logic G. IB captures input and output from previous cycle, and deliver it to CB to generate output value. In MMM with folding ratio of, $N_{fold} = 4$, we need four IBs and one CB logic.

3.2 MMM Operation

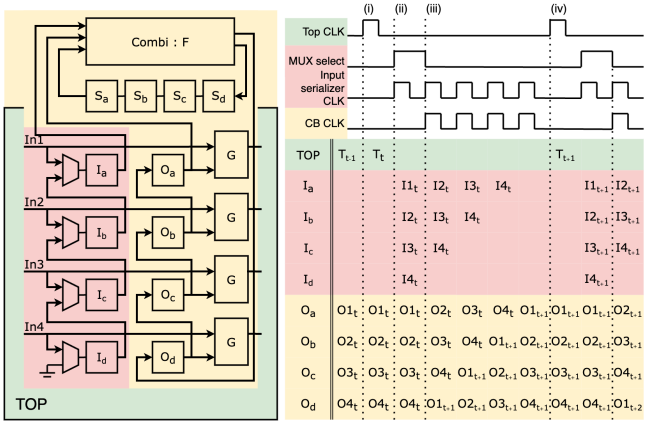

Figure 4: Clocks and pipeline operation during t to $t+1$ for MMM operation.

Figure 4 shows how MMM operates with timing diagram for each clock domain. All clocks in the diagram originate from the same source but are gated based on their purpose. The TOP module is defined as the remaining logic in the design, excluding the MMM

module. The operation begins by launching the TOP clock (CLK) outside of the MMM module at cycle (i). The TOP module uses the output from MMM module to generate its state for the next cycle, which updates the input value entering MMM module. To pass this input value to IB, the MUX select is adjusted to store the input value in the input serializer chain, which then captures the value at cycle (ii). In the next cycle, CB calculates the value of the input of two register chain, S_d and O_d, using the output values of register chains, S_a and O_a at cycle (iii). This process is repeated for the number of folding iterations, N_{fold}. In each cycle, the newly calculated value from CB is added to the end of the output deserializer chain and state register chain. As cycles progress, the values at output deserializer chain shift positions, and after the N_{fold} iterations, the first calculated value, $O1_{t+1}$, reaches to the head of output deserializer chain, O_a. At this point, CB calculated values, $O1_{t+1}$~$O4_{t+1}$, fills the output chain. The output of MMM module is then computed using values in output chain, O_a~O_d, and these results are fed back to the TOP module for further processing at cycle (iv). The pipeline operation of the shift register chain, S_a~S_d, follows the same timing with output deserializer chain as shown in Figure 4.

3.3 MMM Compiler

This compiler, based on TCL scripts, works automatically with AMD's Vivado to convert the target design into MMM-applied logic, allowing it to function across multiple designs and diverse environments. Figure 5 shows the entire procedure of MMM and how they can be deployed to different target designs. Algorithm 1 shows the pseudo-code for the initial part of the IB generating process for obtaining output FF and removing internal cell.

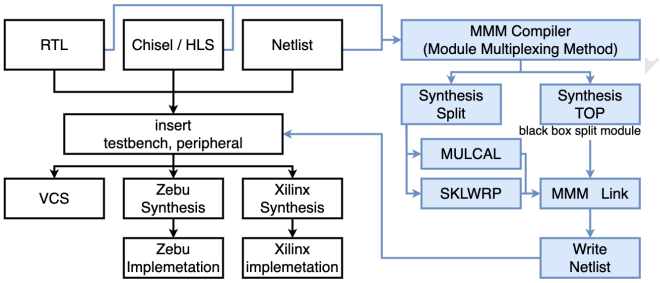

Figure 5: Overview of MMM application flow.

The MMM compiler is divided into three main processes: (1) IB generation, (2) CB generation, and (3) linking them to the top module. **(1) In IB Generation stage,** IB removes the original internal combinational logic and FFs, then creates a chain of FFs for the input serializer with MUX and output de-serializer. **(2) In CB Generation,** the pre-output FFs O are removed and replaced to the ports later to be connected to IB, while the state registers S are replaced with state register chain using AMD SRL primitive. **(3) In top module link process,** our compiler runs synthesis on the top module with the target module replaced by black boxes. Once synthesis is complete, 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(.v, .edif) process. The MMM compiler gets the top module name, target module names, and the synthesis file list. Multiple modules can be folded, and by just adding target module names to the compiler, it will automatically run the tasks in parallel. However, in case the target module contains parallel modules nested in a 2D structure, applying MMM is not currently supported but is planned for future development.

Algorithm 1 Pseudocode of finding/removing output-related FFs

```
find = GetOutPort(netlists)
for item in find do
    if item == FF then
        output_ff.insert(FF)
    else
        connected_primitives =
        netlist.GetConnectedPrimitive(item.in)
        find.insert(connected_primitives)
    end if
end for

all_ff = GetAllFF(netlist)
for item in all_ff do
    if item is not in output_ff then
        netlist.RemoveFF(item)
    end if
end for
```

3.4 MMM FPGA Split

In order to partition large designs across multiple FPGAs, MMM offers significant advantages. This section introduces two methods for partitioning FPGAs when applying MMM and explains how these differ from traditional approaches. In a Neural Processing Unit (NPU), the compute engine uses many LUTs and wires due to its complexity. Typically, the output bus-width in compute engine is smaller than that of other module because of internal accumulation or reduction. With the repeating logic patterns in the compute engine, we can apply resource sharing by selecting the compute engine as the target module in our MMM.

Traditional Partition. Figure 7a shows traditional FPGA partitioning without applying MMM. In this setup, each identical target module is assigned to a separate FPGA, while the TOP module, excluding the target modules, is assigned to another FPGA. Since we have only five FPGAs in this scenario, the TOP module uses full four High-Speed Input Output (HSIO) interfaces, while each FPGA with a target module is connected with only one HSIO to communicate with the TOP module. Figure 6a shows the timing diagram for the traditional partitioning method. Upon the rising clock of the TOP module, data is transferred to the target module. At the same time, target module needs to send output data back to the TOP module. Here, TOP clock only goes high once data transmission to and from the target module is completed.

MMM Basic Partition. In MMM basic partitioning, the CB and output register chain are placed on one FPGA, while another FPGA contains the TOP module with four Interface Blocks (IBs). Figure 7b illustrates how the modules are partitioned and assigned

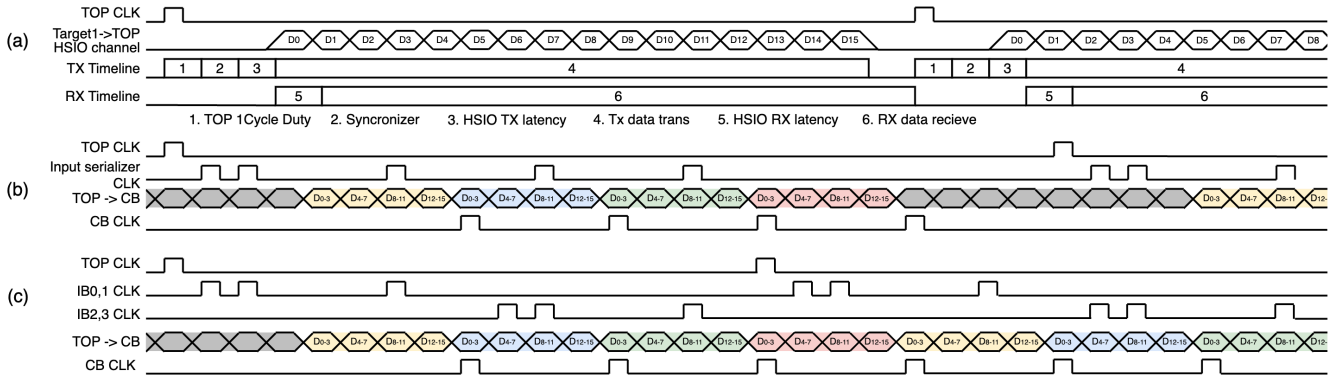

Figure 6: Our FPGA partitioning strategy with timing diagram in inter-FPGA data transfer. (a) Traditional (b) MMM Basic (c) MMM Advanced

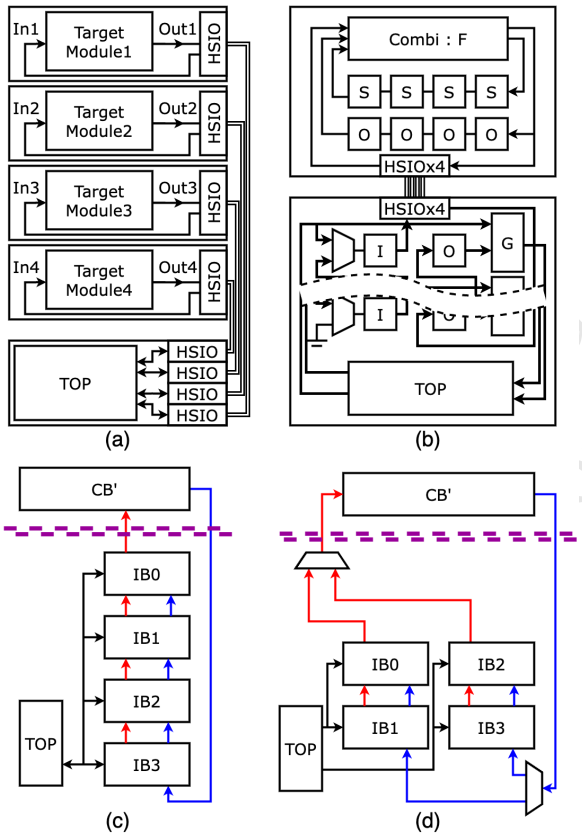

Figure 7: MMM partition. (a) Traditional Partition (b) MMM Basic Partition (c) MMM Basic Partition abstraction (d) MMM Advanced Partition abstraction. The red line represents the input register chain, while the blue line indicates the output register chain.

to each FPGA. Unlike in Figure 3b, both the Compute Block (CB) and the Interface Block (IB) have an output chain register. This design eliminates input/output dependency between the two FPGAs by maintaining synchronized copies of the output register chains on both sides. As a result, the FPGA with four IBs can send data immediately without waiting for data from the FPGA with the CB block, decoupling the input serializer chain from the output deserializer chain. In MMM basic partitioning, the four FPGAs, previously had one target module for each, are replaced with a single FPGA containing the CB module. Four HSIO connections are used between the two FPGAs, providing faster inter-FPGA bandwidth compared to traditional partitioning methods. Figure 6b shows the timing diagram for MMM basic partitioning, where four input register chain data transmissions to the CB module and four output register chain transmissions are placed between TOP CLK rising edges. Although the use of four HSIO interfaces enables 4x faster data transmission, the total amount of data transmitted between TOP CLKs must still match the TDM ratio, and the response data from the CB must also be transmitted to the TOP module to finish one trip of data transaction. This setup introduces two HSIO latencies, making it slower than traditional partitioning, which only incurs one HSIO latency for data transmission.

MMM Advanced Partition. To further reduce link latency in HSIO, we developed a new partitioning strategy called MMM Advanced Partition. Figure 7c shows an abstracted version of basic partitioning. In this scheme, the TOP FPGA has four IB modules that send the values from the input and output register chains to CB' (a CB module with an output chain register copy) through the HSIO interface. In basic partitioning, the TOP clock can only go high once all chain data is received, adding delay in inter-FPGA communication. In contrast, advanced partitioning resolves this by bundling two IB modules into a set and sending data to CB' in a ping-pong manner using two IB module bundles. Figure 6c illustrates the operating clock for IB modules, where IB0, IB1, and IB2, IB3 alternate sending data through HSIO via MUX selection. The MUX alternates between the IB bundles, selecting the input register chain to accept data. Similarly, CB' output data is received through HSIO by selecting the destination IB bundles. In our design, the target module operates asynchronously with the TOP module, thus no clock domain crossing (CDC) logic is required. By alternating

between IB bundles and sending data in this manner, HSIO latency is hidden, allowing the TOP clock frequency to increase.

3.5 Routing congestion reduction in MMM

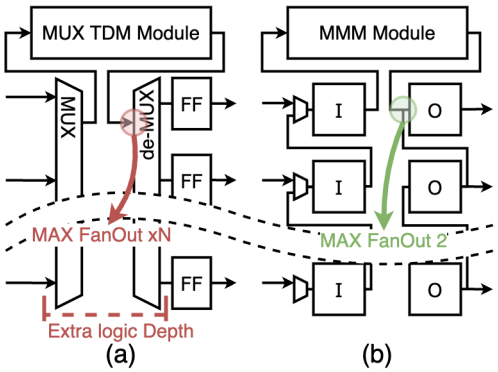

Figure 8: MUX vs. MMM fanout comparison. (a) MUX (b) MMM

In FPGAs, high-fanout nets usually refer to multiple control signals (e.g., set/reset, clock enable, and clock), which are common and difficult to route [8]. In MUX-based TDM, fanout issues worsen due to the increased fanout in the data pipeline. In MMM, routing congestion is drastically reduced by addressing the major causes of (1) fanout, (2) resource usage, and (3) the number of wires. Due to the unavoidable high fanout on control signals, we are going to only discuss fanout in data pipeline. Since we use scan-chain-like shift registers instead of large demultiplexers, we only have maximum fanout of 2 at the output regardless of the size of multiplexing ratio. This is useful in the case where we need to use a higher multiplexing ratio, since large multiplexing ratios will require large 1-to-N demultiplexers with the fanout of N_{fold}. Figure 8 shows the comparison of fanout size in TDM and MMM.

The benefit of high multiplexing ratio, N_{fold}, is the reduction in resource utilization. Since we can reduce resources used in module by $1/N_{fold}$ and the effect of reduction increases as the size of multiplexed module increases. In MUX-based TDM method, applying resource sharing is not easy when there is a data pipeline. If there is a data pipeline, MUXes for ordering data needs to be inserted in each data pipeline stages. However, in MMM, we only need to insert SRLs to hold intermediate values for multiple data pipeline. In Figure 3, chain FF S is implemented with SRLs. Since SRLs can be implemented by using a dedicated LUT unit, no wiring between pipeline stages is necessary and thus wiring burden will be also reduced.

Overall, optimization of these three factors (fanout, resource usage, and the number of wires) reduces the routing congestion in our proposed MMM method. In Section 6, we compare a MUX-based TDM version of NVDLA with our Shift Register-based MMM method in terms of resource utilization, routing congestion, and logic depth.

4 FPGA Verification on Commercial Product

To improve the performance of the machine learning accelerator, both adding more neural engines and making bus connection to more SoC components makes FPGA verification more challenging. In our Company's upcoming commercial chip2, the neural engine's performance has increased four-fold compared to the previous commercial chip1 design, and the number of neural engines has doubled. With four dies combined into a single chiplet, this has led to a 32× increase in performance and 8× increase in logic area compared to chip1. In the chip1 project, we used eight U250 boards and Synopsis HAPS-100 (HAPS) for verification, but due to the increased design size and complexity, these systems could not be used for the new chip2. To facilitate the development and verification process of chip2, we introduced ZeBu4 and ZeBu5. However, due to the large resource usage, verifying all four dies was not feasible, and even single-die verification could only be accessed by a single user at a time. Given the need for a simulation and verification system for ASIC prototyping, for developing a software environment, and for modifying and verifying machine learning models to run on our company's hardware, multiple teams require access to the simulation system. This situation prompted us to develop a method to either provide on-premise simulators for individual users or reduce resource usage on the ZeBu servers.

4.1 Commercial chip architecture

In our machine learning accelerator design, computing core which is called, Neural Engine, is used as target module for applying MMM. The logic compartments that process data are called as, Tensor Processor and Vector Processor. In most cases, computing elements or computing array which has repeating logic pattern is used as CB logic. Inside our company's Neural Engine, there are 4MB Scratch Pad, Vector Processor, Tensor Processor (4 TFLOPS, 16 TOPS), Load/Store Unit, DMA, Task Queue, and Sync Control unit. We applied MMM to Tensor Processor and Vector Processor, these processors contain repetitive structure. Tensor Processor is composed of 128 MAC (Multiply–accumulate) and Vector Processor has 16 MAC. To trade-off between performance and resource utilization reduction, we set the TDM ratio of 16 for these modules.

4.2 FPGA Implementation

To verify the functionality of neural engine, we implemented it on FPGAs. The initial version (chip1 v0) of the neural engine was mapped to the FPGA, as development progressed, the number of neural engine calculators increased (chip1 v1), making it impossible to fit onto the FPGA. We modified the code to reduce FPGA LUT usage and manually mapped the arithmetic operations to DSPs. In the process, a team of four engineers worked for about six months, achieving a 20% reduction in LUT usage. But as the engine grew larger (chip2 v0), the DSP mapping method was no longer effective due to an increased complexity in our processor design. To address this issue, we invented MMM, and a single engineer was able to achieve a 42% reduction in LUT usage with just a few hours of compiler run.

4.3 ZEBU Implemetation

When emulating our SoC design on the ZeBu system, resource optimization was essential due to the large size of the multi-die chiplet architecture, which limited the multi-tenancy capabilities of the ZeBu server [20]. The emulation environment was critical for everything from architectural exploration and firmware development to verification during the project's initial stages. Consequently, we extended MMM's applicability to ZeBu to improve resource utilization in server-grade emulation systems as well.

Contrary to our initial assumption that the MMM-applied netlist (.v) could run directly on ZeBu, several modifications were required. Since ZeBu's emulation system uses a single clock source, clock gating was not feasible which is necessary for MMM. To resolve this, we adjusted Flip-Flop (FF) operations by using enable signals instead of generating new clocks with Integrated Clock Gating cell (ICG). Additionally, when exporting a netlist from AMD's design tool, design elements with FF (FDRE, FDCE, FDPE, FDCE) using the 'IS_INVERTED' property were generated. We modified these elements with FF with inverter attached, as these designs are specific to AMD FPGA products. To further develop and integrate our MMM method into Synopsys's simulation toolchain (e.g., ZeBu), we collaborated with Synopsys to create an automated flow that seamlessly applies our method to their system. We believe this partnership will improve the emulation environment's efficiency by optimizing resource utilization and reducing simulation runtime. This, in turn, enhances system sharing, allowing multiple users to work concurrently and ultimately accelerating the time-to-market process.

5 NVIDIA Deep Learning Accelerator (NVDLA)

For the comparison of our MMM method to other accelerated hardware simulators, we deployed NVDLA [12], an open-source architecture, onto FPGA. We first analyzed the NVDLA architecture and carefully selected the modules to apply MMM for optimal resource utilization on FPGA while maintaining performance for simulation. Additionally, NVDLA was chosen for our experiments because it is a widely known open-source hardware with many related studies, making it ideal for comparison.

5.1 NVDLA architecture

NVDLA has a modular architecture and thus can be easily configured, scaled, and designed toward target applications. Figure 9 shows the abstracted version of NVDLA architecture deployed for hardware simulation on FPGA.

NVDLA primarily targets small IoT and embedded devices. However, the full NVDLA configuration is still too large to fit on the U250 FPGA board as is. By applying MMM method, we successfully reduced design size of the NVDLA. In Section 6, we report the amount of resource reduction we can achieve with MMM on NVDLA. Also, instead of using a microprocessor for driving NVDLA core, we implemented our own state machine based test pattern injector since it will take less resources then a soft-core microprocessor leaving more space to NVDLA to be implemented. NVDLA consists of five main modules: SDP (Single Data Point Processor), PDP (Planar Data Processor), CDP (Cross-channel Data Processor), RUBIK (Data Reshape Engine), and the Convolution Pipeline, which

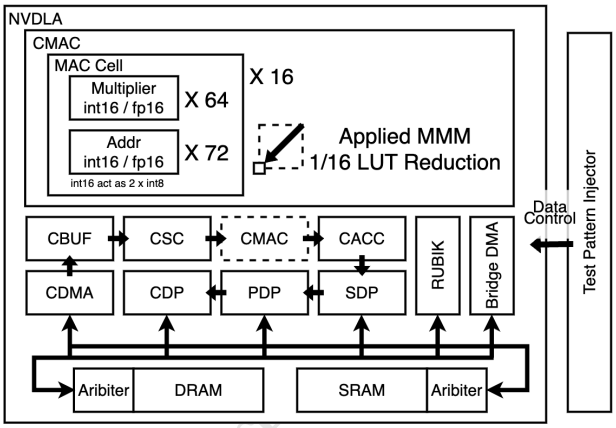

Figure 9: NVDLA architecture deployed on FPGA for simulation acceleration.

includes components such as DMA, Buffer, Convolution Sequence Controller (CSC), Convolution MAC array (CMAC), and Convolution Accumulator (CACC). We implemented resource sharing in the 'MAC cell' in CMAC. Since the MAC cell repeats 16 times, it was an optimal candidate for applying MMM. The implemented configuration of NVDLA includes 2048 Int8 MACs, 1024 INT16 MACs and FP16 MACs, and a 512KiB buffer, corresponding to the largest 'nv_full' configuration of NVDLA.

6 Experiments

This paper applies MMM for two machine learning accelerator designs, commercial chip2's Neural Engine (NE) and NVDLA. For NE and its SoC emulation we have tested our design in two environments. First, we utilized ZEBU5, which consists of 144 AMD VU19P FPGAs. In another experiment which represents an on-premise FPGA simulation use case, we partitioned the NE and assigned each partition onto two AMD U250 FPGA boards. For the NVDLA experiment, we only used one U250 board. To connect the FPGAs, we utilized AMD Aurora IP, leveraging the Quad Small Form factor Pluggable (QSFP) ports on the boards. In all on-premise FPGA settings, the test system has Intel's i9-12900 CPU with 32 GBs DRAM and FPGA cards are attached to the PCIe 3.0 x16 slot. Figure 10 shows our FPGA-accelerated system running NE. The host system runs Synopsys VCS simulator and this simulator is connected to the FPGA through the Programming Language Interface (PLI) of VCS to invoke C++ functions in the FPGA driver. This driver is developed by using AMD's XDMA module which is published as an open-source Linux driver [19].

Resource Utilization Reduction. Using MMM significantly reduces resource utilization on FPGA. Figure 11 illustrates the LUT usage for each element in the original, MMM-applied, and DSP-mapped versions of NE, Additionally, it shows the LUT capacity limit for the FPGA models VU19P and U250. The DSP-mapped version required four engineers to work for six months, achieving a 20% reduction. In comparison, MMM achieved a 71% reduction in just 12 hours of compiler run. The same MMM NE was implemented on both the ZeBu system and a two-FPGA simulator. We

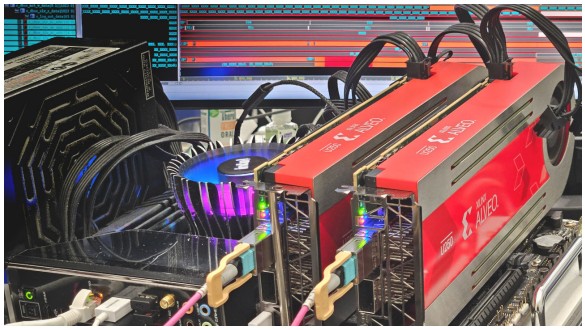

Figure 10: NE simulation system with two FPGAs.

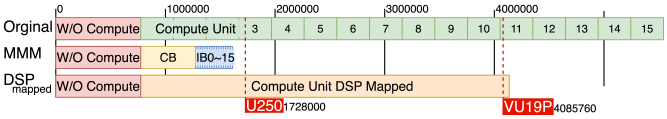

Figure 11: LUT reduction result for NE.

used Zebu5 for the emulation of our entire SoC. Without LUT reduction, emulation was impossible with 204 AMD VU19P FPGAs. However, after applying MMM, we were able to implement it using 144 FPGAs (Zebu5 3U). For NVDLA case, a reduction ratio is 66.48% 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 component. The only other work that deployed 'NVDLA full' is on Amazon EC2 FPGA cloud, but it removed the convolution engines for INT16 and FP16 [6]. Table 1 shows the extent of resource reduction achieved by our deployed simulation accelerators.

Table 1: Resource utilization of NVDLA for MMM vs. vanilla NVDLA.

	LUTs	FFs	DSPs
Full_vanilla	3.31M	6.17M	459
Full_MMM	1.11M	1.09M	402
Reduction (%)	66	82	12

Routing Congestion Reduction. MMM not only reduces resource use but also achieves routing congestion reduction. Through experiments using CMAC, we obtained the following metrics: LUT count, NET count, Sum of FanOut (sum of FO), highest FanOut (highest FO), logic depth, and the logic depth histogram. Additionally, we compared the results for ORG16 (CMAC contains the original 16 MAC Cells), MUX16, and MMM16. Table 2 shows that MMM16 achieves reductions of 17% in LUT count and 44% in NET count. In terms of FF count, MMM16 shows a slight increase compared to MUX16 due to the shift-register-based implementation, which inherently increases FF usage. The sum of FanOut also decreased similarly, while the highest FanOut remained the same as ORG16. Regarding logic depth, the number of nets at maximum depth is indicated in parentheses. Compared to ORG16, MMM16 increased

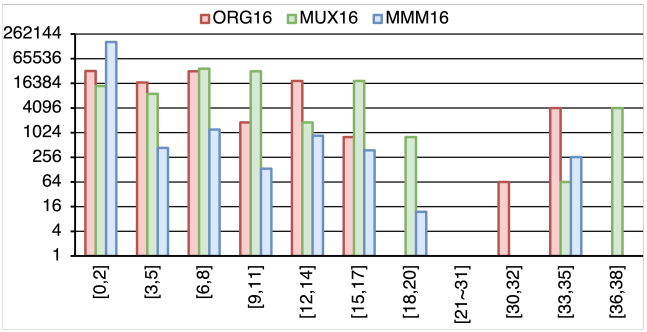

Figure 12: NVDLA CMAC logic depth histogram.

the maximum logic depth by 1 but reduced the net count from 2048 to 128 at this depth. In contrast, MUX16 maintained the same net count but increased logic depth by 3. Figure 12 shows the histogram of logic depth for ORG16, MUX16, and MMM16. The x-axis represents logic depth ranges in specific histogram bins (e.g., [0,2] covers depths from 0 to 2), and the y-axis indicates the number of nets in each bin. In MMM16, the number of nets in each bin decreased, except for the 0 to 2 range, suggesting that high logic depth nets were reduced due to shift registers inserted between combinational logics. The MUX16 shows increased logic depth with more nets distributed in higher logic depth bins.

Logic depth significantly impacts criticality during static timing analysis (STA), while the number of wires affects routing congestion. In this context, MMM demonstrates superior results compared to other methods.

Table 2: Resource reduction of NVDLA CMAC.

	1 Mac Cell	ORG16	MUX16	MMM16
LUT	45k	728k	123k	102k
NET	62k	993k	206k	116k
FF	3,311	52,976	55,824	56,767
sum of FO	345k	5515k	1022k	787k
highest FO	14k	14k	26k	14k
logic depth	34(128)	34(2048)	37(2048)	35(128)

Simulation Speedup in MMM Simulation System. The main purpose of our MMM method is offloading workload of compute-intensive modules that take a lot of time to simulate to an FPGA card so they can be run on actual hardware. In this section, we are going to talk about how fast VCS version of simulation can be done on CPU and how fast our CPU-FPGA hybrid system can finish the same simulation task. For NE case, simulation took 372.8 seconds in CPU on VCS simulator and took 0.6 seconds on our CPU-FPGA hybrid simulator, showing our proposed method has 621.3× speedup in simulation time. The workload we run in this test is our own matrix multiplication test pattern for chip functionality verification. For NVDLA, simulation on the CPU with the VCS simulator took 0.88 hours, while the FPGA-only simulation system completed in 0.867 seconds, demonstrating a 3653× speedup for the AlexNet experiment.

Partitioning Frequency. To evaluate our partitioning strategy, we applied ZeBu5's auto partitioning and compared it to our

method. While ZeBu log and other FPGA-accelerated simulators report FPGA implementation frequency, the actual speed is slower due to inter-FPGA communication. We define this reduced speed as 'Effective Frequency'. Additionally, MMM achieves faster computation of repetitive logic pattern through MMM folding, this clock frequency within MMM modules is marked as the 'Fastest Frequency'. Table 3 shows the results from various FPGA-based simulators.

Table 3: Effective frequency of the commercial chip2 Neural Engine using different partitioning methods on various FPGA-based devices.

System	Partition	Folding (MMM)	Impl. Freq(Hz)	Fastest Freq(Hz)	Effective Freq(Hz)	FPGA	LUT (% 1FPGA)
ZEBU	Auto	1:1	3M	-	57.8k	VU19Px5	5.5M(134%)
ZEBU	Auto	1:16	3M	295k	16.4k	VU19Px2	1.6M(39%)
ZEBU	NO	1:16	3M	285k	11.4k	VU19P	1.6M(39%)
FPGA	Tradition	1:1	60M	-	318k	U250x5	5.5M(318%)
FPGA	Basic	1:16	60M	1.75M	97.4k	U250x2	1.6M(92%)
FPGA	Advance	1:16	60M	7.14M	397k	U250x2	1.6M(92%)
HAPS	NO	1:16	100M	100M	5.6M	VU19P	1.6M(39%)
FPGA	Advance	1:8	60M	7.2M	720k	U250x3	2.1M(121%)
HAPS	NO	1:8	100M	100M	9.9M	VU19P	2.1M(51%)
FPGA	Advance	1:4	60M	3.6N	590k	U250x5	3.1M(179%)
HAPS	NO	1:4	routing	fail	-	VU19P	3.1M(76%)

With MMM-applied cores, ZeBu achieves an effective frequency of 11.4 kHz using a single FPGA, and runs even faster with autopartitioning, reaching 16.4 kHz on two FPGAs. With MMM and its partitioning strategy, the effective frequency reaches 397 kHz using only two FPGAs, surpassing the 318 kHz performance of traditional partitioning with fewer FPGAs. For MMM's single FPGA performance (without partitioning), ZeBu and HAPS achieved fastest frequencies of 285 kHz and 100 MHz, and effective frequencies of 11.4 kHz and 5.6 MHz, respectively. ZeBu's design focuses on multi-FPGA simulation with additional software and hardware logic for debugging, leading to lower speeds compared to HAPS. However, MMM on ZeBu can reduce FPGA usage by 2/5 or even 1/5, freeing resources for other users.

Comparison with Other FPGA-accelerated Simulators. We have compared our work with other FPGA-accelerated simulators in the literature that can either perform software-hardware co-simulation or automatic resource reduction with compiler support, with respect to supported source design format, deployable hardware, simulation features, resource utilization reduction, and routing congestion ease effect. FireSim [9] is an FPGA-accelerated simulation platform offers scalable, cycle-exact microarchitectural simulation for chip-scale (or already silicon-proven) hardware design and supports automatic design partitioning [18] and resource optimization [10] for FPGAs. However, its current resource optimization is limited to multi-ported memories, with additional resource-reducing optimizations still under development.

Twinstar [3] is a ASIC design simulator for IBM's Bluegene/Q computing node and it supports auto partitioning of the target design. However, it can only work on the dedicated FPGA cluster server they created which can be configured to have 28 to 60 FPGAs. Also, to minimize latency of simulation system with inter-FPGA connection, they utilized TDM strategy with LVDS-based communication, but they did not implemented any routing optimization technique or resource reduction strategy as we do. Plus, their operating frequency is limited by inter-FPGA communication delay. RAMP Gold [17] is an architecture simulator which aimed to reduce the resource utilization of the soft-core processors deployed on FPGAs by removing forwarding-path multiplexers in the processors. By using this method, they could reduce 26–32% of combined area for both LUTs and FFs in their design and frequency is also increased by 18–58%. Also, they utilized the TDM method to virtually increase the bandwidth of the BRAM resources. To address high routing delays in FPGA primitives like DSPs and BRAMs, which run at hundreds of MHz but suffer from fixed on-die locations, they implemented a location-aware routing strategy that prioritizes resources to these components. However, their performance and resource reduction ratio is lower than ours.

DEEP [14] is a FPGA-based many-core emulation system for chip verification. By using iterative emulation technique, they tried resource-sharing technique on FPGA by running sub-modules of chip design iteratively module by module on one or few FPGA boards. This is similar to our approach since we also run multiple data flow path iteratively on the module level for FPGA resource utilization reduction. Unfortunately, they did not report the resource utilization reduction they could achieve through this method. Like our compiler design, their simulation system also requires separate synthesis procedure for FPGA deployment and thus adding extra processing time in compiler.

In summary, none of the previous works feature a module-level resource utilization technique with up to 96.88% of LUT saving like ours, with full compiler support for applying resource-sharing to large ASIC designs. Additionally, our novel latency-hiding and inter-FPGA communication optimization techniques, combined with routing congestion reduction, demonstrate strong performance compared to prior works. With these features and compiler support, our proposed simulation system can be deployed on any FPGA-based device available on the market. Table 4 shows the comparison of our proposed FPGA-accelerated simulator with aforementioned SoTA simulators in the literature.

7 Conclusion

In this work, we developed a resource-optimized FPGA simulation system with an automatic compiler. Using the MMM, designs with resource-intensive implementations can reduce resource utilization on FPGA-accelerated system by up to 93.75%, depending on the size of the time-multiplexed target module. To further enhance system performance, we implemented a latency-hiding technique for inter-FPGA communication, optimizing interface communication to reduce latency overhead. However, CPU-FPGA communication remains as a bottleneck, even though it provides faster overall simulation speed compared to CPU-only simulations. Additionally, inserting MMM-related logic into the target module adds extra computation time during synthesis, leaving room for optimization in our compiler design. In future work, we plan to optimize CPU-FPGA communication to improve system performance and develop a Vivado-independent MMM compiler, as the current bottleneck

Table 4: SoTA comparison with other FPGA-accelerated simulator in literature.

	Source support	Resource reduction (LUT)	Routing optimization	Implementation frequency	Deployable hardware	Auto-partitioning	Automatic compiler	Abstraction level
This Work	Verilog, VHDL, HLS, Chisel	upto 93.75% (MMM16)	Yes	100 MHz	Any FPGAs Zebu, Palladium	No	Yes	Hybrid
Firesim [9]	Chisel, RTL as blackbox	upto 26%	No	3.42 - 6.6 MHz	Any FPGAs AWS F1 instance	Yes	Yes	Hybrid
Twinstar [3]	Verilog, VHDL	Not available	No	1 - 4 MHz	Dedicated FPGA cluster	Yes	Yes	No
RAMP Gold [17]	Verilog, HLS	upto 32%	Yes	90 MHz	Xilinx FPGA boards	No	Yes	Hybrid
DEEP [14]	Verilog, VHDL	iterative run of sub modules	No	100 MHz 80k cycle/s	Any FPGAs (emulation) Any workstation (simulation)	No	Yes	Instruction level

largely stems from Vivado-related commands executed by our compiler.

References

[1] AMD. 2015 (accessed October 01, 2024). DPU with Enhanced Usage of DSP. [url] https://docs.amd.com/r/3.3-English/pg338-dpu/DPU-with-Enhanced-Usage-of-DSP.

[2] AMD. 2024 (accessed October 01, 2024). DSP48 Slice Switching Characteristics. [url] https://docs.amd.com/r/en-US/ds923-virtex-ultrascale-plus/DSP48-Slice-Switching-Characteristics.

[3] Sameh Asaad, Ralph Bellofatto, Bernard Brezzo, Chuck Haymes, Mohit Kapur, Benjamin Parker, Thomas Roewer, Proshanta Saha, Todd Takken, and José Tierno. 2012. A cycle-accurate, cycle-reproducible multi-FPGA system for accelerating multi-core processor simulation. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. 153–162.

[4] Ümit V Çatalyürek and Cevdet Aykanat. 2011. PaToH (Partitioning Tool for Hypergraphs).

[5] Ming-Hung Chen, Yao-Wen Chang, and Jun-Jie Wang. 2021. Performance-driven simultaneous partitioning and routing for multi-fpga systems. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1129–1134.

[6] Farzad Farshchi, Qijing Huang, and Heechul Yun. 2019. Integrating NVIDIA deep learning accelerator (NVDLA) with RISC-V SoC on FireSim. In *2019 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*. IEEE, 21–25.

[7] S Hadjis, A Canis, JH Anderson, J Choi, K Nam, S Brown, and T Czajkowski. 2012. Impact of FPGA architecture on resource sharing in high-level synthesis, ACM. In *SIGDA International Symposium on Field Programmable Gate Arrays, Monterey, CA*, Vol. 10. 2145694–2145712.

[8] Xun Jiang, Jiarui Wang, Jing Mai, Zhixiong Di, and Yibo Lin. 2024. A Robust FPGA Router With Optimization of High-Fanout Nets and Intra-CLB Connections. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024).

[9] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. 2018. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 29–42.

[10] Albert Magyar, David Biancolin, John Koenig, Sanjit Seshia, Jonathan Bachrach, and Krste Asanović. 2019. Golden Gate: Bridging the resource-efficiency gap between ASICs and FPGA prototypes. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[11] Rakhi Nangia and Neeraj Kr Shukla. 2018. Resource utilization optimization with design alternatives in FPGA based arithmetic logic unit architectures. *Procedia computer science* 132 (2018), 843–848.

[12] Nvidia. 2018 (accessed September 20, 2024). NVIDIA Deep Learning Accelerator (NVDLA) Open Source Project. [url] https://nvdla.org.

[13] Chak-Wa Pui, Gang Wu, Freddy YC Mang, and Evangeline FY Young. 2019. An analytical approach for time-division multiplexing optimization in multi-FPGA based systems. In *2019 ACM/IEEE International Workshop on System Level Interconnect Prediction (SLIP)*. IEEE, 1–8.

[14] Juergen Ributzka, Yuhei Hayashi, Fei Chen, and Guang R Gao. 2011. Deep: an iterative fpga-based many-core emulation system for chip verification and architecture research. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. 115–118.

[15] Long Sun, Longkun Guo, and Peihuang Huang. 2021. System-level FPGA routing for logic verification with time-division multiplexing. In *Parallel and Distributed Computing, Applications and Technologies: 21st International Conference, PDCAT 2020, Shenzhen, China, December 28–30, 2020, Proceedings 21*. Springer, 210–218.

[16] Welson Sun, Michael J Wirthlin, and Stephen Neuendorffer. 2007. FPGA pipeline synthesis design exploration using module selection and resource sharing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26, 2 (2007), 254–265.

[17] Zhangxi Tan, Andrew Waterman, Rimas Avizienis, Yunsup Lee, Henry Cook, David Patterson, and Krste Asanović. 2010. RAMP gold: an FPGA-based architecture simulator for multiprocessors. In *Proceedings of the 47th Design Automation Conference*. 463–468.

[18] Joonho Whangbo, Edwin Lim, Chengyi Lux Zhang, Kevin Anderson, Abraham Gonzalez, Raghav Gupta, Nivedha Krishnakumar, Sagar Karandikar, Borivoje Nikolić, Yakun Sophia Shao, et al. 2024. FireAxe: Partitioned FPGA-Accelerated Simulation of Large-Scale RTL Designs. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 501–515.

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

[20] Chang-Hyo Yu. 2024. Software-before-RTL-freeze Verification for a Hyperscale Chiplet-based SoC. In *2024 SNUG Silicon Valley*. Synopsis, 1–27.

[21] Dan Zheng, Xinshi Zang, and Martin DF Wong. 2021. Topopart: a multi-level topology-driven partitioning framework for multi-fpga systems. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–8.

[22] Peng Zou, Zhifeng Lin, Xiao Shi, Yingjie Wu, Jianli Chen, Jun Yu, and Yao-Wen Chang. 2020. Time-division multiplexing based system-level FPGA routing for logic verification. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

Received 1 October 2024; revised XX Dec 2024; accepted XX Jan 2024