A 475 MHz Manycore FPGA Accelerator for RTL Simulation

Sahand Kashani*
sahand.kashani@epfl.ch
EPFL
Lausanne, Switzerland

Mahyar Emami*
mahyar.emami@epfl.ch
EPFL
Lausanne, Switzerland

Keisuke Kamahori†
kamahori@uw.edu
University of Washington
Seattle, USA

Sepehr Pourghannad†
spourghannad@student.ethz.ch
ETH
Zurich, Switzerland

Ritik Raj†
ritik.raj@gatech.edu
Georgia Tech
Atlanta, USA

James R. Larus
james.larus@epfl.ch
EPFL
Lausanne, Switzerland

ABSTRACT

This paper presents the implementation of Manticore: a manycore accelerator for parallel RTL simulation. Manticore packs up to 225 custom soft processors running at 475 MHz on a large FPGA.

Implementing manycore accelerators on FPGAs is challenging as designers must reconcile the conflicting goals of maximizing the number of cores on the chip and clocking them at the highest possible frequency. Designers face two classes of constraints: (1) architectural constraints imposed by a large FPGA's multi-die structure, and (2) physical constraints imposed by the FPGA shell's size and placement. Physical design therefore plays a critical role in the implementation of manycore accelerators.

We present physical design challenges faced during Manticore's implementation on the AMD Alveo U200 card—a large FPGA with a poorly-placed, wide shell that challenges physical implementation.

CCS CONCEPTS

• **Hardware → Reconfigurable logic applications**; **Hardware description languages and compilation**; **Partitioning and floorplanning**; **Simulation and emulation**.

KEYWORDS

Manycore accelerators, FPGA, physical design, high clock frequency

ACM Reference Format:
Sahand Kashani, Mahyar Emami, Keisuke Kamahori, Sepehr Pourghannad, Ritik Raj, and James R. Larus. 2024. A 475 MHz Manycore FPGA Accelerator for RTL Simulation. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '24), March 3–5, 2024, Monterey, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3626202.3637579

1 INTRODUCTION

Manycore architectures are processor arrays with hundreds or thousands of cores on a single device: they exhibit extensive on-chip bandwidth and are effective platforms for communication-intensive parallel workloads. Designers seek to maximize two key features of these architectures to increase the available on-chip bandwidth: (1) the number of cores and (2) their clock frequency.

Implementing a manycore accelerator on an FPGA is challenging as high core counts and high clock frequencies are conflicting goals. The desire for high core counts pushes implementations onto large datacenter-grade devices—devices for which high-speed design is complicated by two factors. First, these devices interface with a host over a PCIe connection. So, FPGAs are split into two regions: a shell that interfaces with the host computer and a user-design region that contains the accelerator logic. Unfortunately, vendor-provided shells have a large footprint and are immovable: their placement results in a non-rectangular user-design region, which makes floorplanning challenging. Second, large FPGAs are multi-die devices with multiple "Super Logic Regions" (SLRs) interconnected via high-delay wires: designs that span multiple SLRs require careful implementation to achieve a high clock frequency.

This paper focuses on the low-level optimizations needed to implement a high core count and high clock frequency manycore accelerator on a large FPGA. We do so in the context of Manticore: an open-source [7, 8] manycore FPGA accelerator purpose-built for RTL simulation with 225 cores clocked at 475 MHz on an AMD Alveo U200 datacenter accelerator card. We make two main contributions: (1) we describe how we tailored Manticore's soft processors and Network-on-Chip (NoC) to the traits of parallel RTL simulation, and (2) we present an uncommon floorplan to work around the U200's multi-die structure and intrusive shell. We provide a top-down description of Manticore's design, from requirements to implementation, and discuss implementation obstacles and their corresponding physical design optimizations. Manticore's original paper [6] provides an extensive application performance analysis.

The rest of this paper is organized as follows: Section 2 briefly introduces parallel RTL simulation. Section 3 continues with an overview of Manticore's design. Section 4 details the microarchitecture of Manticore's 500 MHz core, and Section 5 describes its NoC. Section 6 explains how to initialize a statically-scheduled machine. Section 7 presents floorplanning challenges when scaling Manticore's design, and our proposed solutions to work around them. Section 8 evaluates the QoR of Manticore's physical design. Section 9 discusses related work. Section 10 concludes.

*Both authors contributed equally to this work.
†Work done during EPFL internship.

FPGA '24, March 3–5, 2024, Monterey, CA, USA

ACM ISBN 979-8-4007-0418-5/24/03.
https://doi.org/10.1145/3626202.3637579

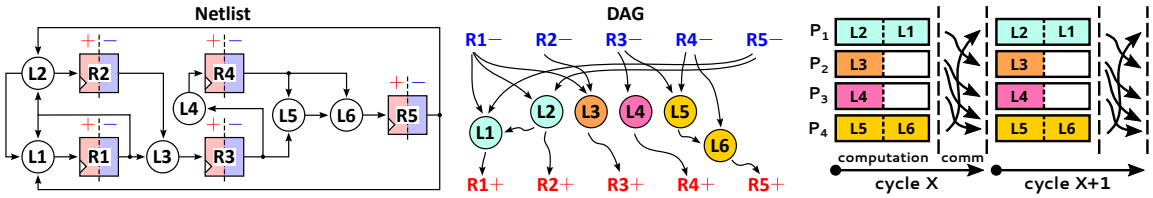

Figure 1: Parallel simulation of an RTL netlist with bulk-synchronous parallelism. The input netlist (left) is split at its registers to form a DAG (center) where inputs and outputs correspond to current (−) and next (+) register values, respectively. Independent paths (colored vertices) in the DAG can be executed in parallel on a processor array (right).

2 PARALLEL RTL SIMULATION

Hardware description languages represent circuits in the form of a netlist: a directed graph whose nodes are circuit cells (gates, registers, memories) and edges are its connecting wires. Figure 1 shows how to use a bulk-synchronous parallel (BSP [22]) execution model to evaluate a netlist graph: First, the netlist graph is made acyclic by splitting the registers into a *current* and *next* value (left). This results in a directed acyclic graph (DAG) where current and next values are denoted by − and +, respectively (center). Second, the netlist DAG is evaluated in data dependence order. A simulated cycle concludes after computing all next register values from current ones. The current values are then updated and the process repeats. A compiler partitions the netlist DAG so independent paths run in parallel on a processor array (right).

Netlist DAGs are very wide and have far more parallelism than the number of cores in general-purpose multicore processors, which motivates using a manycore accelerator for parallel RTL simulation.

3 OVERVIEW

Manticore consists of an array of highly customized cores linked by an interconnect. The top part of Figure 2 shows the physical layout of the U200 FPGA. This FPGA contains three identically-sized SLRs, each containing a grid of clock regions, and is split into a shell (orange) and a user-design region (black). The U200 contains four DRAM banks. Interfacing with a bank requires instantiating a memory controller: an IP with a large footprint that we expect to complicate clocking a dense arrangement of cores at high frequencies. On-chip bandwidth is more critical than off-chip bandwidth for parallel RTL simulation; hence, we restrict ourselves to a single DRAM bank (blue, contained in the shell). This entirely reserves the user-design region for cores, NoC switches, and control structures.

The bottom part of Figure 2 shows an overview of Manticore's design for a small 4-core instance. Manticore is a message-passing machine: cores can access only their private instruction/data memories and explicitly exchange messages to communicate (no shared memory). All cores are identical, apart from one *privileged core* (black), which can access a 16 GiB DRAM through a 256 KiB on-chip cache. Each core stores a partition of the netlist DAG in a private, fixed-latency on-chip instruction memory and sends its computed *next* register values to the consuming cores at the end of each simulation cycle (see Figure 1). The fixed structure of netlist DAGs allows Manticore to *statically schedule* computation and communication globally. This allows us to remove interlocks and forwarding paths from cores and remove buffering from the NoC. The cores are simple *feed-forward* pipelines that *cannot dynamically stall* and are easy to implement at high clock frequencies. Statically scheduling the cores and the NoC is possible because they are built out of fixed-latency components. However, DRAM has non-deterministic latency and is incompatible with static scheduling without making worst-case assumptions about memory latency. Therefore, we separate the user-design region into a *compute* domain containing all cores and the NoC and a *control* domain containing control circuitry. Logic in the control clock domain *gates* the compute clock when a non-deterministic latency operation starts and resumes it upon completion. Hence, from the compiler's point of view, off-chip and on-chip memory accesses have a fixed latency.

4 CORE DESIGN

Designing a core with the fewest stages for a target clock frequency is best. Since FPGA primitives (BRAMs, URAMs, and DSPs) are placed unevenly across FPGA columns, we could minimize each core's pipeline depth based on its placement. However, such a design

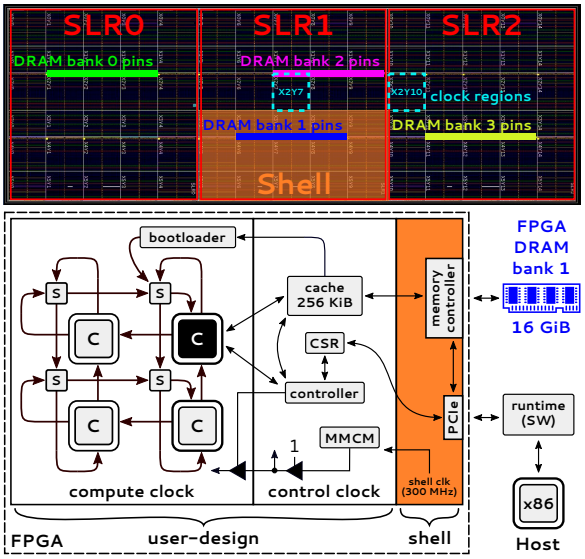

Figure 2: U200 FPGA layout (top) and high-level diagram of Manticore (bottom). All cores (C) and switches (S) are in the compute clock domain. A privileged core (black) accesses DRAM through a cache (in the control clock domain). Operations with non-deterministic latency gate the compute clock.

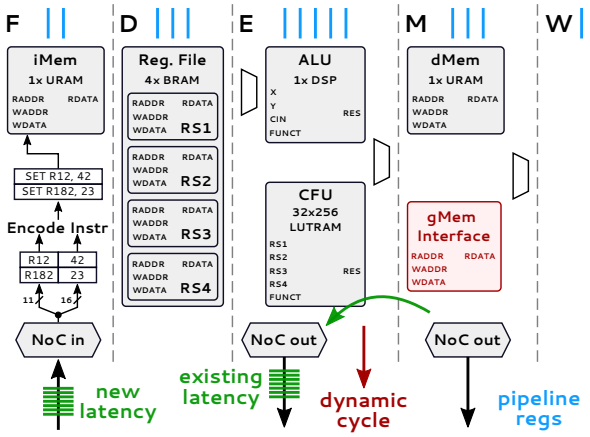

Figure 3: Microarchitecture of the privileged core. The green pipeline registers are enhancements necessary for improved floorplanning and will be presented in Section 7.

would complicate the compiler's task in scheduling. Therefore, we opted for a more regular—and very deep—pipeline that maximizes the clock speed in a worst-case scenario.

Figure 3 shows a block diagram of the privileged core: it consists of the usual five-stage pipeline, but each stage is further pipelined for more place-and-route (P&R) slack. We use all built-in hard registers of the FPGA primitives (BRAMs, URAMs, and DSPs) and complete with fabric registers if needed. This additional pipelining enables individual cores to run at 500 MHz. Although the pipeline is 14 stages deep, the compiler can fill the pipeline slots due to the abundant instruction-level parallelism in netlist DAGs.

Instructions are fetched from a large 4096×64 URAM memory (8× larger than a BRAM). Instructions are 64 bits wide; most of this space is used to index the large register file.

Typically, soft processors have small 32-entry register files. However, Manticore's cores use a large BRAM configured in a 2048×17 structure. The 16 lower bits of each entry contain data, and the 17^{th} bit is an overflow bit to simulate carry chains efficiently. The large register file virtually removes register spilling and is optimized for long-reuse distances in RTL simulation. Instructions read up to four registers per cycle using four write-replicated BRAMs (1W4R).

The execute stage contains a standard ALU and a custom function unit (CFU). The ALU is built from a single DSP configured in "dynamic" mode to support multiplication and other functionality at runtime. We match the processor's datapath width to the native width of the DSP primitive (16 bits) for simplicity. The CFU implements 32 four-input bitwise-parallel logic functions, which the compiler extracts from the netlist DAGs. The CFU uses a shallow 32×256 memory built from LUTRAMs to store logic equations.

RTL memories are stored in each core's local URAM data memory. Local loads execute unconditionally, but local stores are predicated. The global memory interface exists in only the privileged core. Global loads or stores are predicated and privileged: they access off-chip DRAM using a 48-bit address concatenated from three registers and are used for *very* large RTL arrays. Local/global memory accesses and outgoing NoC messages are indistinguishable to the compiler as they all occur in the same pipeline stage. We describe the "dynamic cycle" signal in Section 7.1.

5 NOC DESIGN

The NoC is implemented as a 16-bit *uni-directional* 2D torus with buffer-less switching and dimension-ordered routing: its switches contain only pipeline registers and muxes for a minimal area footprint. It is inspired by the Hoplite NoC [16]. However, Manticore's NoC does not use deflection routing since the NoC is statically scheduled with the cores to avoid runtime contention: switches drop messages if the target link is busy since contention is erroneous behavior.

The left side of Figure 4 depicts a standard uni-directional torus in which wrap-around links (left, bold) have much higher delay than inner links: they need additional pipelining to achieve high clock frequencies [10]. Manticore's NoC instead uses a folded torus so all links have approximately equal length (right side of Figure 4). Having a uniform NoC simplifies the compiler's scheduler.

Overlapping computation and communication improves simulation performance, but requires extra memory to queue messages at the receiver core. To save memory, we use each core's instruction memory and its unused write port (after boot) as this ingress queue: incoming register updates are encoded into a set-immediate instruction and appended to the end of the instruction memory (see Figure 3); the core then executes it like any other instruction when its program counter reaches it.

6 BOOTLOADING

We briefly describe how to initialize a machine with a statically scheduled NoC. Figure 5 shows the NoC ingress path, which is used as part of each core's bootloading process before simulation starts (red), and for register updates during program execution (blue).

The bootloader starts with a soft reset that brings all cores into a "dynamic" state: they snoop the NoC for instructions and push NOPs through their pipelines. The bootloader then reads the program binary from FPGA DRAM through the cache and injects its contents into the NoC at the privileged core's NoC switch (see Figure 2). The program binary contains multiple fields, which a finite state machine (FSM) in each core intercepts and uses to configure the processor. The first field denotes the total number of instructions that the core will receive. Each instruction is received as four 16-bit chunks, which the FSM assembles into a single 64-bit instruction and writes to the instruction memory. Once all instructions are

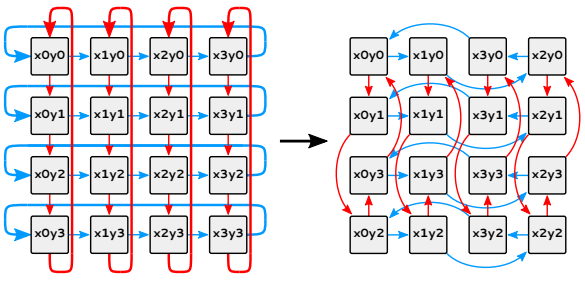

Figure 4: Unfolded vs folded torus layout.

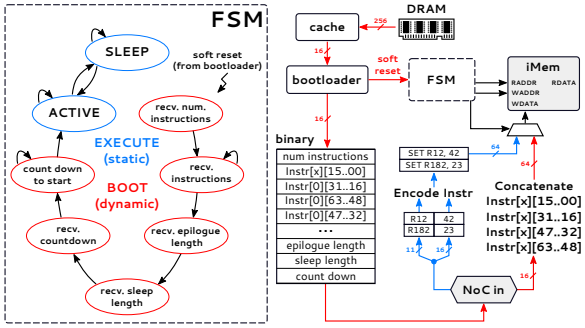

Figure 5: Core NoC ingress behavior at boot and execution.

received, the FSM then receives three counters. The first counter is the epilogue length, which is the number of messages the core is expected to receive from other cores during program execution. The second counter is the sleep length and corresponds to the number of clock cycles each core must sleep after the computation phase. The final counter is a per-core countdown timer that is necessary to start all cores *at the same time* such that they execute the program in strict lock-step. The deterministic execution of the NoC allows these messages to arrive at exactly the right time. Each core counts down and starts "static" program execution once this counter reaches 0.

7 FLOORPLANNING

Floorplanning has a large effect on quality-of-results (QoR) [13]. It is performed using *constraints*, which in Vivado are defined through a PBlock or a resource-specific constraint. We first describe the necessary constraints for Manticore's implementation on any FPGA. We continue by analyzing Vivado's P&R performance without device-specific constraints to understand the architectural challenges in the U200's design. We then study how a regular, grid-structured floorplan performs. Finally, we present a minor modification to the core presented in Section 4 which, when combined with multiple design optimizations, leads to an unconventional floorplan with better f_{max} on large Manticore configurations.

Our general approach during floorplanning is that it is impractical to manually place individual cells in the design. We instead generate Tcl scripts that constrain the design using information about the target FPGA. Manticore is developed in Chisel [5], a DSL for hardware generators embedded in the Scala programming language. Manticore's hardware generator is therefore aware of cores' names and those of their internal structures. We add a generic Scala floorplanning class after Chisel's final Verilog emission point, which we then subclass to implement various floorplanning strategies.

7.1 Generic Constraints

Manticore relies on clock gating to enable static scheduling. The privileged core emits a "dynamic cycle" signal early in its pipeline before an operation with non-deterministic latency's effect becomes apparent (see memory stage in Figure 3). This signal exits the core and travels to the controller and to the cache in the control clock domain (see Figure 2), where the controller then decides whether to gate the clock on the next cycle. The privileged core, controller, and cache must be close to each other for this operation to pass timing at 500 MHz. We therefore create a PBlock constraint to co-locate them in the same clock region as the clock buffer.

The clock generator and its control path are carefully designed to permit scalability. The compute and control clock buffers are cascaded and are sourced from the same clock generator (the MMCM in Figure 2), which we configure to generate two frequency-matched and phase-aligned signals using a source clock from the shell. Some skew is inevitable between the compute and control clock domains, and so it should be carefully controlled. Vivado typically routes each clock to the "clock root" before fanning it out to all sinks. The clock root is located in the clock region at the center of the bounding box that contains all logic driven by the clock. This introduces a large skew between the compute and the control clocks for large core arrays as the control clock is local to a small part of the circuit close to the shell, whereas the bounding box of the compute clock spans across the entire FPGA. To avoid this, we manually choose nearby clock regions as clock roots for the clocks.

7.2 Automatic Floorplanning

Figure 6 reports Vivado's P&R performance without device-specific constraints. Designs up to 12×12 cores run near 500 MHz. Performance decreases sharply past this point as 15×15 designs run at 395 MHz, and 16×16 designs drop to 180 MHz. This decrease is explained by (1) the shell's placement and (2) the increased wire delay due to SLR crossings: With fewer than 160 cores (maximum capacity of one SLR, limited by its URAM capacity), Vivado fits Manticore entirely in SLR2, unperturbed by the shell, and finds a high-quality folded torus floorplan automatically. Beyond 160 cores, the design is forced to spread around the shell. Vivado cannot find a good floorplan for a rectangular core array in a non-rectangular user-design region, making timing closure difficult.

7.3 Guided Floorplanning

7.3.1 Regular Grid Floorplan. Given Manticore's regular design, it is natural to believe that a grid-structured floorplan would produce the densest and highest-speed implementation. Since Vivado cannot find a high-quality, non-rectangular implementation itself, we must

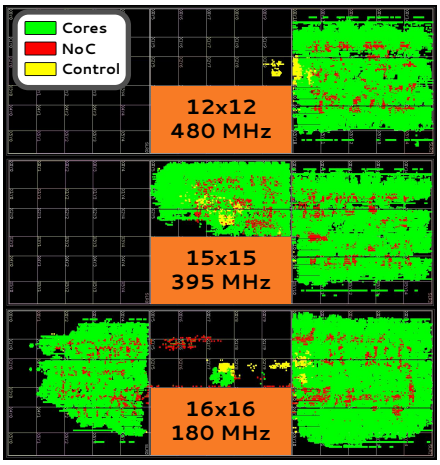

Figure 6: Automatic floorplan QoR.

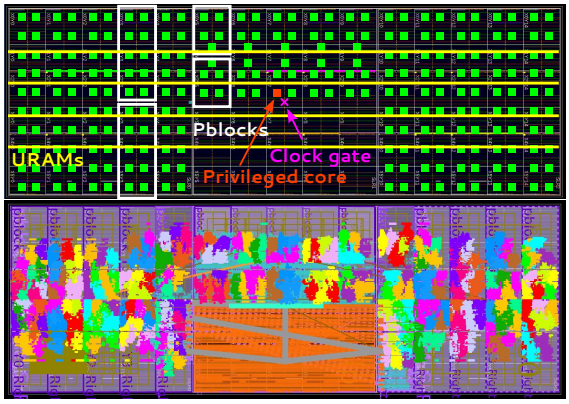

Figure 7: Grid floorplan of a 25×10 Manticore accelerator (top). PBlocks (white) contain either 5 cores (green) in SLR1 or 10 cores in SLR0/SLR2. Each color in the implementation (bottom) represents a core.

guide it with PBlocks. We do not place individual cores to avoid over-constraining the design. We instead create 30 PBlocks that are large enough to contain 5–10 cores each depending on the SLR in which they are located, and let Vivado handle the rest. Figure 7 shows this floorplan (top) and its implementation (bottom).

While the design is regular, cores cluster around the center columns of the device despite the PBlocks spanning the entire width of each SLR. The reason is that URAMs, which all cores use, exist in only the center of the FPGA. Manticore's cores and switches—though regularly placed—are tightly packed in SLR1 and timing closes at 400 MHz due to congestion.

Besides congestion in SLR1, there is the issue of increased wiring delay for SLR crossings. UltraScale+ devices contain pairs of special hard registers (LAGUNA registers) to cross SLR boundaries at high clock frequencies (400+ MHz). Each TX-RX path between LAGUNA registers is a direct link between *two* registers, so the user design must have two back-to-back registers to be able to leverage them. However, links between Manticore's switches are *one* hop from each other and are incompatible with LAGUNA registers. One solution would be to pipeline the links between switches such that each contains at least two registers. However, this doubles the latency of all NoC traffic and degrades application performance.

7.3.2 Split Core-Switch Floorplan. A minor change to Manticore's design yields higher-performing implementations using only five PBlocks, and without changing the latency of switch hops.

The key idea is that switches must be close to each other in a folded torus arrangement, but cores do not: links between NoC switches cannot cross an SLR boundary at high clock frequencies, and so all switches should be placed in a single SLR. We reserve SLR1 for this purpose: since switches use only CLB resources (which are abundant), they will not cluster around the URAM columns, freeing up the cramped space around the shell and enabling short connections to neighboring switches. We then partition cores equally between SLR0 and SLR2. One exception is the privileged core, which we leave in SLR1 as it must be close to the MMCM and controller. Our five PBlocks therefore are: (1) SLR0 for half of the cores, (2) SLR2 for the other half of the cores, (3) SLR1 for all switches and the privileged core, (4) clock region X2Y7 for the clock root, and (5) clock region X1Y7 for the privileged core, cache, and controller. Figure 8 gives a high-level view of this split core-switch floorplan.

Implementing this floorplan requires separating a core's placement from that of its NoC switch. The green registers in Figure 3 show how we modify each core. We pull a core's NoC egress path as early as possible in its pipeline (in the execute stage). We then push the 7 pipeline registers that were previously inside the core to be outside it. An outgoing NoC packet now has 7 cycles of latency to traverse SLR0/SLR2 to SLR1 and reach its switch. This modification is invisible to the compiler as we have simply moved existing registers around. The NoC ingress path also must be pipelined. This introduces new registers, and so the compiler's scheduler must be modified to account for the extra 7 cycles of latency.

Notice that cores and switches are only loosely-constrained, i.e., assigned to PBlocks that span entire SLRs. Vivado automatically finds a high-quality torus floorplan for switches. As for the cores, it spreads them *arbitrarily* in SLR0 and SLR2 as 7 cycles are more than sufficient to reach the switches in SLR1, irrespective of their placement in SLR0/SLR2. We now present additional design changes needed to get the most out of the split floorplan described above.

Shift register inference. Vivado—by default—transforms back-to-back registers into shift registers to improve density. This is beneficial inside a core, but hurts performance outside on the newly-pipelined core↔switch paths: a shift register is contained in a single LUTRAM instead of being a physical chain of registers, and so cores cannot be physically far from their switches. Shift register inference is done early during synthesis, and so we had to explicitly instruct Vivado not to infer shift registers on the core↔switch paths.

SLR crossings. Since core↔switch paths are now pipelined with 7 registers, there exists a pair of back-to-back registers which we can map to LAGUNA tiles in opposing SLRs. By default Vivado does not automatically map registers to LAGUNA cells: candidate registers must be marked with a `USER_SLL_REG` *hint* to be considered. However, we found that simply marking candidate registers with this hint was insufficient for Vivado to map them to LAGUNA registers: we had to explicitly constrain registers 1–4 to SLR0 or SLR2

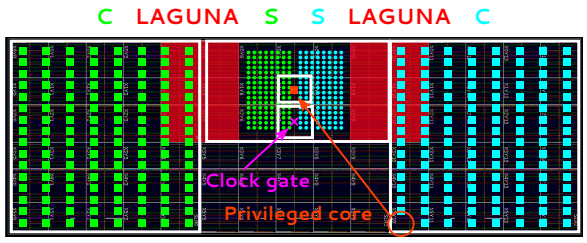

Figure 8: Split core-switch floorplan. We split cores (C) equally between SLR0 (green) and SLR2 (blue), but leave the privileged core in SLR1. All switches (S) are pinned to SLR1. Core ↔ switch links are pipelined with 7 cycles of latency. We map one pair of adjacent registers from these pipelined links to LAGUNA registers for efficient SLR crossing.

and registers 5–7 to SLR1. Once further constrained, Vivado was then able to map them to the hard LAGUNA registers, as intended.

Reset trees. The host can reset cores through the controller. The high fanout reset logic requires extra care not to limit clock speed. The controller drives two, 3-stage pipelined wires from SLR1 to SLR0/SLR2, appropriately marked to be mapped to LAGUNA registers. Once in SLR0/SLR2, a radix-4 tree propagates the reset signal to all cores. We disable shift register inference to preserve the intended tree structure. Switches do not have a reset signal: To preserve correctness, the controller keeps the reset signal active for a design-size-specific number of cycles. Cores that are under reset emit empty messages to the NoC, eventually clearing all NoC state.

Relative placement of memories. We often encountered situations where timing failed due to Vivado placing logically-related BRAMs/URAMs at a distance from each other. For example, the four BRAMs that make up a core's register file would occasionally be placed in *different* BRAM columns, which are far apart. Similarly, the four URAMs that compose the 256 KiB cache would be placed in different URAM columns, significantly degrading timing. We work around this issue by using *relatively-placed macros* for all register files and the cache. The resulting implementation yields more consistent P&R results, making it easier to reason about the influence of incremental design changes on Manticore's clock frequency.

8 EVALUATION

We evaluated Manticore configurations with Vivado 2022.1 and used multiple implementation strategies for each design to improve timing closure. We use the default implementation strategy alongside the "Performance_NetDelay_High" one, which we found sometimes more aggressively works around the unavoidable (though reduced) clock skew between the compute and control clocks.

Figure 9 summarizes our proposed split core-switch floorplan's QoR for increasing Manticore configurations. Manticore's f_{max} is consistently above 450 MHz, even in a 16×16 instance (up from 180 MHz without optimizations). Clock skew is a large source of QoR variability and results in larger instances occasionally achieving higher f_{max} than smaller ones (e.g., 12×12 and 9×9). Design effort increases considerably at high clock speeds: more detailed floorplanning would likely bring performance up to 500 MHz.

Manticore is a memory-heavy design: breaking down per-SLR resource utilization reveals up to ≈ 80% (75%) URAM (BRAM) utilization in SLR0/SLR2 for a 16×16 design. We also report the fraction of total "Super Long Lines" (SLLs) used to cross SLR boundaries. SLLs are a new limiting resource (≥ 67% utilization in SLR2) in our proposed floorplan as all cores' NoC datapaths converge towards their switches in SLR1. Our choice of leaving SLR1 free of cores sacrifices design capacity in exchange for high clock frequencies.

Figure 10 shows the floorplan of our highest-performing large Manticore implementation (15×15, 475 MHz).

9 RELATED WORK

Prior work presented a new type of coarse-grained FPGA and its corresponding CAD flow for RTL emulation [9]. Manticore is not an FPGA: it is an array of processors and is programmed entirely with software (there is no FPGA P&R algorithm or a "bitstream").

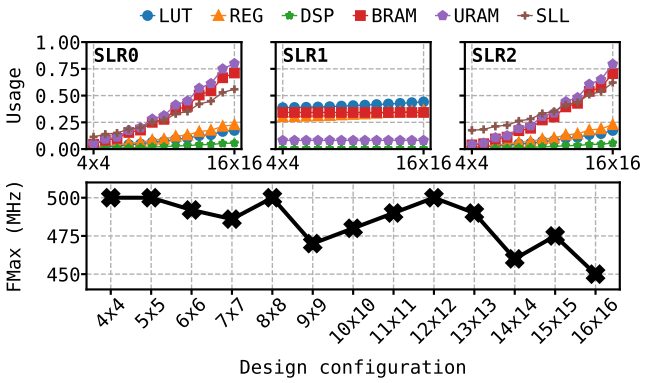

Figure 9: Split core-switch floorplan QoR analysis. We break down resource utilization per-SLR. SLR1 utilization includes shell resources.

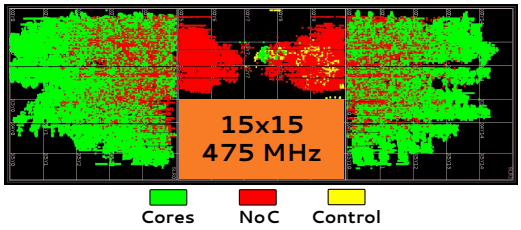

Figure 10: Split core-switch floorplan P&R results.

Most recent work on manycore accelerators for FPGAs target RISC-V processors. These works are either manycore frameworks [14, 15, 17, 18, 23] or optimized manycore implementations [1–4, 10–12, 19–21]. The frameworks provide customizable overlays to create homogeneous processor arrays (or heterogenous arrays of processors and accelerators) and demonstrate new architectural ideas. They propose systems clocked between 20–120 MHz and do not focus on low-level implementation details as it is not their goal. The specialized systems achieve higher-clocked implementations (94–300 MHz) or demonstrate more efficient use of scarce FPGA resources (e.g., SLLs in multi-die FPGAs). While they explain some physical implementation details, explaining low-level details is not their goal as they too must focus on the application of their accelerator. Manticore does not use general-purpose RISC-V processors: its architecture is tailored to the characteristics of RTL simulation and its physical implementation is optimized for the U200 FPGA.

10 CONCLUSION

We described the design and implementation of Manticore: an FPGA accelerator for parallel RTL simulation. Manticore packs 225 cores clocked at 475 MHz on a large datacenter FPGA. We presented Manticore with an emphasis on physical design obstacles incurred when implementing manycore accelerators on multi-die FPGAs with imposing shells. Manticore's unconventional split core-switch floorplan contrasts with traditional regular grid floorplans used in devices with less obstructive shells and allows us to achieve both a high core-count and high frequency implementation.

REFERENCES

[1] Riadh Ben Abdelhamid and Yoshiki Yamaguchi. 2022. Packed SIMD Vectorization of the DRAGON2-CB. In *Proceedings of the 15th IEEE Internation Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-2022)*. Penang, Malaysia, 85–92. https://doi.org/10.1109/MCSoC57363.2022.00023

[2] Riadh Ben Abdelhamid, Yoshiki Yamaguchi, and Taisuke Boku. 2021. A Highly-Efficient and Tightly-Connected Many-Core Overlay Architecture. *IEEE Access* 9 (4 2021), 65277–65292. https://doi.org/10.1109/ACCESS.2021.3074171

[3] Riadh Ben Abdelhamid, Yoshiki Yamaguchi, and Taisuke Boku. 2023. A Scalable Many-core Overlay Architecture on an HBM2-enabled Multi-Die FPGA. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 16, 1 (1 2023), 15:1–15:33. https://doi.org/10.1145/3547657

[4] Chethan Kumar H. B, Prashant Ravi, Gourav Modi, and Nachiket Kapre. 2017. 120-core microAptiv MIPS Overlay for the Terasic DE5-NET FPGA board. In *Proceedings of the 25th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, CA, USA, 141–146. https://doi.org/10.1145/3020078.3021751

[5] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Design Automation Conference*. San Francisco, California, 1216––1225. https://doi.org/10.1145/2228360.2228584

[6] Mahyar Emami, Sahand Kashani, Keisuke Kamahori, Mohammad Sepehr Pourghannad, Ritik Raj, and James R. Larus. 2023. Manticore: Hardare-Accelerated RTL Simulation with Static Bulk-Synchronous Parallelism. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems*. Vancouver, BC, CA, 219–237. https://doi.org/10.1145/3623278.3624750

[7] Mahyar Emami, Sahand Kashani, Keisuke Kamahori, Mohammad Sepehr Pourghannad, Ritik Raj, and James R. Larus. 2023. Manticore parallel RTL simulator. https://github.com/ManticoreRTL.

[8] Mahyar Emami, Sahand Kashani, Keisuke Kamahori, Mohammad Sepehr Pourghannad, Ritik Raj, and James R. Larus. 2023. Manticore parallel RTL simulator (artifact). https://zenodo.org/doi/10.5281/zenodo.10363279.

[9] David Grant, Chris C. Wang, and Guy G. Lemieux. 2011. A CAD framework for Malibu: an FPGA with time-multiplexed coarse-grained elements. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, CA, USA, 123–132. https://doi.org/10.1145/1950413.1950441

[10] Jan Gray. 2016. GRVI Phalanx: A Massively Parallel RISC-V FPGA Accelerator. In *Proceedings of the 24th IEEE International Symposium on Field-Programmable Custom Computing Machines*. Washington DC, USA, 17–20. https://doi.org/10.1109/FCCM.2016.12

[11] Jan Gray. 2017. GRVI Phalanx: A Massively Parallel RISC-V FPGA Accelerator Framework; A 1680-core, 26 MB SRAM Parallel Processor Overlay on Xilinx UltraScale+ VU9P. In *First Workshop on Computer Architecture Research with RISC-V (CARRV 2017)*. Boston, MA, USA. https://carrv.github.io/2017/papers/gray-phalanx-carrv2017.pdf

[12] Jan Gray. 2019. 2GRVI Phalanx: A 1332-Core RISC-V RV64I Processor Cluster Array with an HBM2 High Bandwidth Memory System, and an OpenCL-like Programming Model, in a Xilinx VU37P FPGA [WIP Report]. In *Fifth International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC'19)*. Denver, CO, USA. https://h2rc.cse.sc.edu/2019/papers/lightning_2_1_Gray.pdf

[13] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. 2021. AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs. In *Proceedings of the 29th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Virtual Event, 81–92. https://doi.org/10.1145/3431920.3439289

[14] Ahmed Kamaleldin and Diana Göhringer. 2022. AGILER: An Adaptive Heterogeneous Tile-Based Many-Core Architecture for RISC-V Processors. *IEEE Access* 10 (4 2022), 43895–43913. https://doi.org/10.1109/ACCESS.2022.3168686

[15] Ahmed Kamaleldin, Salma Hesham, and Diana Göhringer. 2020. Towards a Modular RISC-V Based Many-Core Architecture for FPGA Accelerators. *IEEE Access* 8 (8 2020), 148812–148826. https://doi.org/10.1109/ACCESS.2020.3015706

[16] Nachiket Kapre and Jan Gray. 2017. Hoplite: A Deflection-Routed Directional Torus NoC for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 10, 2 (3 2017), 14:1–14:24. https://doi.org/10.1145/3027486

[17] Andreas Kurth, Björn Forsberg, and Luca Benini. 2022. HEROv2: Full-Stack Open-Source Research Platform for Heterogeneous Computing. *IEEE Transactions on Parallel and Distributed Systems* 33, 10 (12 2022), 4368–4382. https://doi.org/10.1109/TPDS.2022.3189390

[18] Andreas Kurth, Pirmin Vogel, Alessandro Capotondi, Andrea Marongiu, and Luca Benini. 2017. HERO: Heterogeneous Embedded Research Platform for Exploring RISC-V Manycore Accelerators on FPGA. *arXiv* (12 2017). https://doi.org/10.48550/arXiv.1712.06497

[19] Haruka Mori and Kenji Kise. 2014. Design and Performance Evaluation of a Manycore Processor for Large FPGA. In *Proceedings of the 7th IEEE Internation Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-2014)*. Aizu-Wakamatsu, Japan, 207–214. https://doi.org/10.1109/MCSoC.2014.37

[20] Matthew Naylor, Simon W. Moore, and David B. Thomas. 2019. Tinsel: A Manythread Overlay for FPGA Clusters. In *Proceedings of the 29th International Conference on Field Programmable Logic and Applications*. Barcelona, Spain, 375–383. https://doi.org/10.1109/FPL.2019.00066

[21] Blaise Tine, Varun Saxena, Santosh Srivatsan, Joshua R. Simpson, Fadi Alzammar, Liam Cooper, and Hyesoon Kim. 2023. Skybox: Open-Source Graphic Rendering on Programmable RISC-V GPUs. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems*. Vancouver, BC, Canada, 616–630. https://doi.org/10.1145/3582016.3582024

[22] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (8 1990), 103–111. https://doi.org/10.1145/79173.79181

[23] Joseph Zuckerman, Paolo Mantovani, Davide Giri, and Luca P. Carloni. 2022. Enabling Heterogeneous, Multicore SoC Research with RISC-V and ESP. *arXiv* (6 2022). https://doi.org/10.48550/arXiv.2206.01901