ManyRVData Project Report

CachePool: Manycore Cluster of Customizable, Lightweight Scalar-Vector PEs for Irregular Data-Plane Workloads

Report by:
Diyou Shen, Ph.D. Student
Zexin Fu, Ph.D. Student
dishen, zexifu@iis.ee.ethz.ch

Supervised by:
Prof. Luca Benini
Prof. Alessandro Vanelli-Coralli
lbenini, avanelli@iis.ee.ethz.ch

Integrated Systems Laboratory,
ETH Zürich,
Zürich, Switzerland

October 9, 2025

Contents

1 Executive Summary

This document reports on the *ManyRVData* project. The aim of the project is to design and evaluate a scalable, cache-based manycore architecture with scalar-vector capability for the 5G Radio Access Networks (RAN) dataplane. The focus is on efficiently supporting the RLC layer, which requires both high-throughput packet processing and control parameter computation under tight latency constraints. The architecture is designed to maintain high performance and energy efficiency when scaling to hundreds of Processing Elements (PEs), each operating on large, sparse, and irregular data structures typical of dataplane workloads.

The project explores a heterogeneous manycore cluster in which lightweight, in-order scalar RISC-V cores handle control-intensive tasks, while vector engines excel at data-intensive workloads, all connected through a shared, cache-based memory hierarchy. To better match the unpredictable memory access patterns in RLC packet processing, the design abandons scratchpad-only approaches (such as in MemPool) and avoids hardware cache coherence entirely. Instead, it employs a **shared L1 cache** with **configurable private L1 partitions** per PE or per group of PEs, enabling flexibility in bandwidth allocation and minimizing contention while keeping implementation complexity low.

The architecture is designed to address three representative RLC dataplane scenarios. The first is a **single-user peak-rate case**, representing the extreme throughput demand of a single access user and serving as a stress test for per-user performance, latency, and sustained packet processing under continuous load. The second is a **multi-user peak-load case**, where a moderate number of active users generate high aggregate throughput, testing the system's ability to scale across many threads while maintaining low contention and predictable latency. The third is a **multi-user average-load case**, reflecting a large-scale deployment with thousands of concurrent users at more typical per-user rates, emphasizing fairness, resource isolation, and stable long-term operation. In the first year, the project will concentrate on the single-user peak-rate case to validate the core architecture, memory subsystem, and scalar-vector processing efficiency before extending evaluation to multi-user scenarios in the second stage.

The project spans two years, with the first-year activities organized into three main Work Packages (WPs):

- **WP1 – Scalable Cache Design for a Many-core Cluster:** Define the architecture specification and explore design options, followed by the initial design of key building blocks. Carry out preliminary implementation and power–performance–area (PPA) analysis.
- **WP2 – Processing Element: Microarchitecture and ISA Optimization:** Specify the microarchitecture and prototype a non-blocking load/store interface, together with instruction set extensions (vector unit) for dataplane processing workloads and control parameter computation.
- **WP3 – HW-SW Mapping, Runtime Management, and Benchmarking:** Conduct preliminary benchmarking with representative dataplane packet-processing kernels, and develop prototype programming APIs and runtime support for deployment, allocation, and scheduling.

Building on ETH Zürich's experience with the *MemPool* architecture, the *ManyRVData* design transitions from software-managed scratchpads to a hardware-managed, shared cache system, while keeping the per-core architecture simple and scalable. This approach provides high flexibility in resource partitioning and efficient support for sparse, irregular workloads without the area and energy costs of full hardware coherence.
The first year will deliver the following milestones:

- **M1.1 – Preliminary ManyRVData Cluster Design (M12):** Release of open-source hardware and documentation.

- **M1.2 – Preliminary Benchmark Suite and Analysis (M12):** Release of open-source software and documentation, accompanied by a benchmarking report.

The design source code for hardware and software will be made available as open source.

2 Introduction

Wireless baseband dataplane processing is a prime domain for energy-efficient parallel computing: the RAN market is vast, and the throughput and efficiency requirements are escalating. As 5G evolves toward 5.5G and 6G, per-cell computational complexity is projected to grow far beyond what technology scaling alone can deliver, demanding architectural and implementation advances across the stack. This project addresses that challenge with a focus on dataplane tasks in the RLC layer. Service Data Units (SDUs) from higher layers are queued and assembled into Protocol Data Units (PDUs) with sequence headers for retransmission management before being passed to MAC, an interaction that tightly couples performance and reliability constraints.

Viewed as a computing workload, RLC exhibits three defining features. First, scale: modern deployments must sustain on the order of 10 million packets/s overall, with each RLC instance exceeding 10^3 packets/s and thousands of instances active concurrently. Second, latency: the Transition Time Interval (TTI) spans 62.5 µs to 1 ms (typically 500 µs), setting a hard real-time envelope for all sub-tasks. Third, irregularity: per-user tasks are independent but highly variable, the code footprint is small, while data structures (e.g., linked lists) create sparse, non-contiguous accesses over large address ranges. Together, these properties imply hundreds of concurrent threads touching dispersed state with poor spatial locality.

To ground our design choices, the remainder of this section surveys state-of-the-art manycore architectures alongside prevailing cache-coherence strategies and evaluates their feasibility under the RLC workload's stringent throughput, TTI-scale latency, and irregular memory behavior. We then introduce the high-level design of *ManyRVData* and summarize its principal contributions.

2.1 State of the Arts

2.1.1 On-Chip Memory Organization in Manycore Architectures

Modern manycore processors use either hardware-managed caches or software-managed scratchpads as their L1 memory. Scratchpad designs give each core or tile a shared banked memory without coherence. For example, the MemPool architecture is a manycore architecture with hundreds of individually programmable cores sharing a low-latency L1 Scratchpad Memory (SPM) [1] and TeraPool scales further beyond a thousand cores with a shared SPM [2]. Such systems achieve very low and predictable global access latency (e.g. 5 cycles) but require explicit Direct Memory Access (DMA) transfers and data placement by software.

In contrast, some other manycore designs use hardware-managed caches. In general purpose CPUs, recent systems include *AmpereOne* (96–192 Arm cores) with 64 KB L1D and 2 MB private L2 per core plus a 64 MB system-level cache on a mesh interconnect [3]; AMD's 128-core *EPYC Bergamo*, which composes eight 16-core CCDs around shared L3 slices for a large last-level cache [4]; and Intel's *Sierra Forest* Xeon, which tiles E-cores into clusters that share L2/L3 slices over a mesh fabric [5]. These manycore CPU designs target control-flow-heavy code with irregular and discontinuous memory access. Compared to compute-intensive accelerators, cache hierarchies offer better performance for workloads with unpredictable data access and potential for temporal locality and reuse. In GPGPUs, hardware-managed caches are also employed to improve programmability and performance under diverse workloads. NVIDIA's Hopper GPUs, for example, feature a unified per-SM memory block that serves as both L1 cache and shared memory [6]. This physical memory can be partitioned at runtime: one portion operates as a hardware-managed L1 cache, while the other functions as software-managed shared memory. This configurable design allows the architecture to adapt to both regular, contiguous data access patterns (where shared memory excels), and irregular, unpredictable access patterns (where L1 caching provides robustness without manual data management). It is also very inspiring to configure the same block of memory into different management schemes.

In summary, dataflow processors and ASIC-based accelerators tend to favor scratchpads and

static scheduling for their highly predictable access patterns, whereas general-purpose CPUs and GPUs utilize hardware-managed caches to handle control and irregular workloads more efficiently. Inspired by these architectural trade-offs, our project targets the RLC workload, which involves frequent control flow, linked-list traversal, pointer chasing, and synchronization mechanisms such as spin locks. These characteristics result in dynamic and unpredictable memory access patterns that are better served by a cache-based memory hierarchy. Therefore, we adopt a hardware-managed cache design as the L1 memory.

2.1.2 Cache Coherence Schemes

Traditional CPU coherence. Conventional multiprocessors maintain cache-line coherence system-wide via snooping (broadcast on a shared bus/ring) or directory-based protocols. While effective up to tens of cores, both approaches encounter fundamental scaling barriers in manycore, data-intensive settings: snooping saturates interconnect bandwidth and power with broadcast traffic; directories incur metadata that grows with sharers (and indirection latency), plus invalidation/ownership traffic that contends on the NoC under fine-grained sharing and write bursts. These effects manifest as coherence hotspots, rising miss/ack latencies, and energy overheads that compete with useful bandwidth, motivating alternative schemes in emerging designs [7, 8, 9].

Emerging schemes for heterogeneous and manycore systems. To accommodate accelerators and extreme parallelism, several directions relax granularity, timing, or scope. (i) *Coarse-grained (region) coherence* amortizes metadata/traffic by tracking permissions over larger regions, particularly effective for CPU–GPU sharing where data moves in bulk [10, 11]. (ii) *Timestamp-based coherence* (e.g., Tardis) replaces invalidations with logical time, eliminating sharer vectors and broadcasts while retaining performance comparable to directories on large core counts [9, 12]. (iii) *Scoped/domain-limited coherence* confines coherence to domains to cap protocol traffic and metadata. In ARM systems, shareability attributes define Non-shareable, Inner Shareable, and Outer Shareable domains; ACE/CHI interconnects and CMN-class meshes honor these domains for coherence snoop and maintenance behavior, enabling cluster-local coherence while avoiding system-wide broadcast [13, 14, 15]. HSA formalizes a complementary scoped memory model (work-group/agent/system) so software can request visibility at the smallest required scope [16, 17]. At the host–device boundary, CXL-cache and CCIX protocols create link-local coherent domains between CPUs and attached accelerators, maintaining coherence only within the CXL/CCIX fabric rather than the entire platform [18, 19, 20, 21]. These designs exemplify domain scoping: keep strong coherence where it pays (clusters, coherent links), and rely on software/fences when crossing domains.

Software-managed or no-coherence designs. Several research/industrial manycores deliberately omit global coherence, shifting responsibility to software/runtimes. Cerebras' Swarm/SwarmX interconnects all cores and systems for software-managed data movement without traditional caches or global hardware coherence [22]. Many RISC-V accelerator clusters (e.g., MemPool/TeraPool) adopt a shared, multi-banked L1 SPM (Tightly Coupled Data Memory (TCDM)) and rely on DMA and software synchronization rather than cache protocols [1, 23]. Intel's 48-core Single-Chip Cloud Computer (SCC) deliberately omits coherence; coherence is maintained via message passing and explicit cache management [24]. Industrial manycores such as Kalray MPPA organize cores into clusters that share a local multi-bank memory with software-managed coherence and no global coherence domain [25]. In the accelerator space, GPUs historically employ scoped visibility with fences and atomics; recent work [26] explores extending coherence across hierarchical multi-GPU systems without full broadcast. These approaches reduce hardware complexity and traffic, but shift responsibility for correctness and data movement onto the runtime, an attractive trade-off when per-flow state is private and communication is structured.

The target RLC workload involves linked list traversal, branch-heavy control logic, and frequent synchronization using spinlocks and atomics, all within tight TTI-scale deadlines. In this project, we design the L1 memory to be cache-based to handle irregular and sparse memory access. We further simplify the cache coherence design by making all L1 cache partitions shared by default. At the same time, when certain data structures (e.g., local stack, per-user buffers or hot data) are accessed heavily by a single tile and are sensitive to contention, the system allows configuring selected L1 partitions as private to each tile, with software responsible for maintaining cache coherence. Carefully partitioning shared and private regions helps reduce interference while preserving flexibility and programmability.

2.1.3 On-Chip Interconnection

The on-chip interconnection is a critical component in connecting PEs with memory subsystems in a cluster. Existing approaches can be broadly categorized into (1) low-latency crossbars (Xbars) and (2) router-linked meshes.

Xbars provide low-latency access by enabling direct connections between all endpoints, but their scalability is limited by quadratic growth in routing complexity and wiring overhead. Multi-level memory hierarchies can act as coalescers to reduce the number of Xbar endpoints [27], but at the cost of increased latency and data duplication across hierarchy levels. Hierarchical Xbars also help relieve routing pressure, but the effective bandwidth per core decreases as the system scales [28, 29]. Software techniques such as task scheduling [30, 31] and dynamic address reallocation [32] can reduce interconnect pressure, but they cannot fundamentally overcome the bandwidth limitations of Xbar-based designs and impose additional programming complexity.

Meshes, by contrast, offer better scalability, with latency increasing linearly in the number of hops. The *Celerity* manycore system [33] employs a 2D-mesh Nework on Chip (NoC) with a Partitioned Global Address Space (PGAS) memory system to connect 496 cores, but relies heavily on software to tolerate the high communication latency, complicating programmability. The *HammerBlade* architecture [34] uses a half-Ruche NoC [35] to connect cores and cache banks, but still struggles to sustain non-blocking memory accesses at scale, even with support for up to 63 outstanding requests.

More recent designs attempt to combine the low-latency of Xbars with the scalability of meshes through hybrid interconnection schemes. TensTorrent's *Wormhole* architecture [36] uses Xbars locally to connect five PEs and memory banks within a cluster, and a router-linked 2D-torus to connect 80 clusters globally. Similarly, *ET-SoC-1* [37] clusters 32 cores using Xbars and scales further to 1088 cores through a 2D-mesh NoC. The *TeraNoc* architecture [38] follows a comparable approach, replacing the hierarchical Xbars in *TeraPool* with a hybrid Xbar–NoC interconnect to balance latency and scalability.

Overall, interconnection fabrics in manycore systems expose a fundamental trade-off between latency, scalability, and programmability. Xbars deliver low-latency access but face quadratic growth in complexity, meshes provide scalable layouts but incur higher and variable latency, and hybrid designs attempt to balance the two. While software techniques can mitigate some of these challenges, they increase programming complexity and do not fundamentally resolve the bandwidth limitations. This underscores the need for interconnect solutions that sustain high utilization at scale without prohibitive hardware or software overhead.

2.2 Overview and Contribution

For the given access patterns of RLC workload, explicitly managed hierarchies (e.g., scratch-pad memory + DMA) are ill-suited: tiling and double-buffering become inefficient as data dependencies branch unpredictably. A hardware-managed, cache-based hierarchy is required; however, enabling hundreds of PEs to access shared cache banks efficiently is challenging. Using hardware-managed private caches would require costly hardware cache-coherence maintenance.

To address this, we propose a shared L1 cache with a configurable, software-managed private-L1 partition design.

ManyRVData explores a scalable, cache-based manycore architecture built around the following key design elements.

1. *Heterogeneous scalar–vector execution with fine-grain interleaving.*

   The cluster combines lightweight, in-order scalar RISC-V cores for control-intensive tasks with tightly coupled vector engines for data-intensive kernels. Scalar and vector execution are closely integrated, with negligible switching overhead between scalar and vector instructions, making the architecture particularly suitable for workloads that interleave scalar–vector operations at fine granularity.

2. *Scalable L1 cache controller for high in-flight miss handling.*

   An L1 cache controller that scales to handle many in-flight misses by embedding MSHR metadata directly into the cache-line SRAM arrays, eliminating the need for dedicated MSHR registers and allowing the theoretical number of outstanding misses to match the number of cache lines.

   This enables high in-flight cache miss handling, which is beneficial for the target workloads, e.g., sparse linked-list traversal and per-user buffer/reassembly lookups, where many independent misses need to be in flight to hide memory latency and sustain throughput.

3. *Shared L1 with configurable, software-managed private partitions.*

   By default, all L1 cache partitions are shared and accessible by all PEs, connected through *hierarchical interconnects*. Because there are no private caches in that case, the system is inherently coherent, reducing programmer burden and eliminating directory overhead and coherence traffic.

   For data that is heavily accessed by a specific tile and sensitive to latency or bandwidth contention, software can reconfigure selected L1 partitions to be private to that tile; in this mode, cache coherence for these private partitions is maintained entirely by software. This private partition reduces average access latency, increases effective bandwidth for the owner tile, and improves predictability of timing under the tight TTI constraints of the workload.

4. *Scalable hierarchical interconnect design*

   The design target of the interconnect in this project should balance both high performance and scalability. To achieve this, we design a hierarchical interconnect:

   For the core to L1 cache bank interconnect inside of the single tile, we use multiple decomposed crossbars to achieve low latency and physical feasibility.

   For the L1 cache to next level memory interconnect (the L2 refill interconnect), we use crossbar-based interconnect with TCDM protocol instread of AXI protocol. The more relaxed ordering requirement of TCDM protocol help us reduce the hardware complexity of the L2 refill interconnect.

   To further scale the system, there will be another hieratchy of the interconnect across different tiles

Here we summarize the main contributions achieved in the first year of the *ManyRVData* project. Guided by the work package plan, our efforts have spanned microarchitecture design, RTL implementation, memory-system innovation, physical-aware evaluation, and software–hardware co-design for representative dataplane workloads. Our contributions in the first year include:

1. **Scalar–vector PE microarchitecture.** We designed the *Snitch* lightweight in-order scalar core tightly coupled to the *Spatz* vector engine. This integration allows scalar control and vector data phases to interleave at fine granularity with near-zero handoff overhead, enabling efficient execution of workloads with frequent scalar–vector switching.

2. **Tile-level RTL integration.** We implemented a parameterizable tile consisting of multiple Snitch+Spatz core complexes, connected to multiple shared L1 cache banks via a low-latency interconnect. The interconnect supports configurable memory address interleaving to match workload patterns. DRAMSys integration is included to enable cycle-accurate simulation of real-world DRAM behavior.

3. **Novel L1 cache controller.** We developed an L1 cache controller that embeds MSHR metadata directly into the cache-line SRAM, removing the need for a fixed-size MSHR file. This design allows the number of outstanding misses to scale with the number of cache lines, supporting deep hit-under-miss and secondary-miss merging, well suited for sparse, pointer-chasing workloads such as linked-list traversal and per-user buffer/reassembly lookups.

4. **Physical-aware PPA evaluation.** We performed power, performance, and area analysis using physical-aware synthesis in a 12 nm technology node, providing realistic insights into implementation cost and scalability.

5. **Dataplane kernel implementation and evaluation.** We implemented and benchmarked a RLC data management kernel, which includes key features such as linked-list traversal, vectorized data movement, spin-lock synchronization, and memory management runtime, along with vector compute kernels such as `dotp`, `gemv`, and `gemm`. These kernels serve both as performance drivers for architecture evaluation and as a foundation for future runtime and application development.

3 CachePool Architecture

Previous works [39, 40] primarily scale the number of PEs to increase performance, relying on an SPM-based shared L1 memory together with a DMA engine for data movement. While effective at small to medium scales, this approach struggles to sustain the bandwidth demands of RLC workloads and offers limited programming flexibility. To address these challenges, we explore a cache-based memory hierarchy combined with a bandwidth-aware scaling strategy.

CachePool is a heterogeneous many-core architecture built around a fully shared, multi-banked L1 cache connected to multi-channel DRAM/HBM main memory. This design delivers high bandwidth and performance, while maintaining cache coherence in an efficient and lightweight manner.

3.1 Processing Elements

3.1.1 Snitch Scalar Core

CachePool employs single-stage, 32 bit RISC-V *Snitch* cores [41] as scalar and host processors, supporting the RV32I base Instruction Set Architecture (ISA). The cores execute scalar instructions directly, while vector and floating-point instructions are offloaded to the attached Spatz vector core. To prevent read-after-write (RAW) hazards, a dedicated counter tracks in-flight vector memory operations, ensuring that Snitch stalls loads until pending vector stores have completed.

As in MemPool and TeraPool, the Snitch Load Store Unit (LSU) integrates a scoreboard to monitor outstanding memory operations, enabling forward progress as long as no RAW hazards occur. The LSU supports multiple outstanding requests, allowing Snitch to issue a sequence of loads and stores without waiting for responses [39], thereby tolerating long memory latencies. Owing to the Non-Uniform Memory Access (NUMA)-style interconnection and main memory behavior, the scoreboard retires loads out-of-order while guaranteeing in-order delivery to the execution pipeline. The maximum number of outstanding transactions is parameterizable and can be tuned according to the expected memory latency of the cluster's shared L1 cache.

Each Snitch core features an accelerator port to offload vector and floating-point instructions to the Spatz vector core. Figure 1 illustrates the core complex architecture, showing a Snitch core tightly coupled with its Spatz counterpart. Dashed-lined modules indicate optional components that can be disabled at configuration time. The next section introduces the Spatz vector core in more detail, including its optional floating-point support.

3.1.2 Spatz Vector Core

The 32-bit streamlined vector core *Spatz* [42] is tightly coupled with the Snitch scalar core within the core complex, boosting both performance and energy efficiency.

Spatz implements the RVV 1.0 specification through a generic accelerator interface and features a 2 KiB latch-based Vector Register File (VRF) optimized for energy-efficient vector processing. It supports both vector and scalar floating-point operations, coordinated by an additional Floating Point Unit (FPU) sequencer. To minimize area overhead, scalar floating-point instructions reuse the FPUs embedded in the Vector Functional Unit (VFU). For area- or energy-constrained applications, floating-point support can be selectively disabled.

The architectural view of Spatz is shown in Figure 1. A scoreboard within the controller provides dependency tracking and enables vector chaining. Spatz includes a dedicated Vector Load Store Unit (VLSU) equipped with reorder buffers, allowing multiple outstanding memory requests in the NUMA environment. The compact latch-based VRF further reduces area and energy, making Spatz a scalable building block for many-core architectures well-suited.

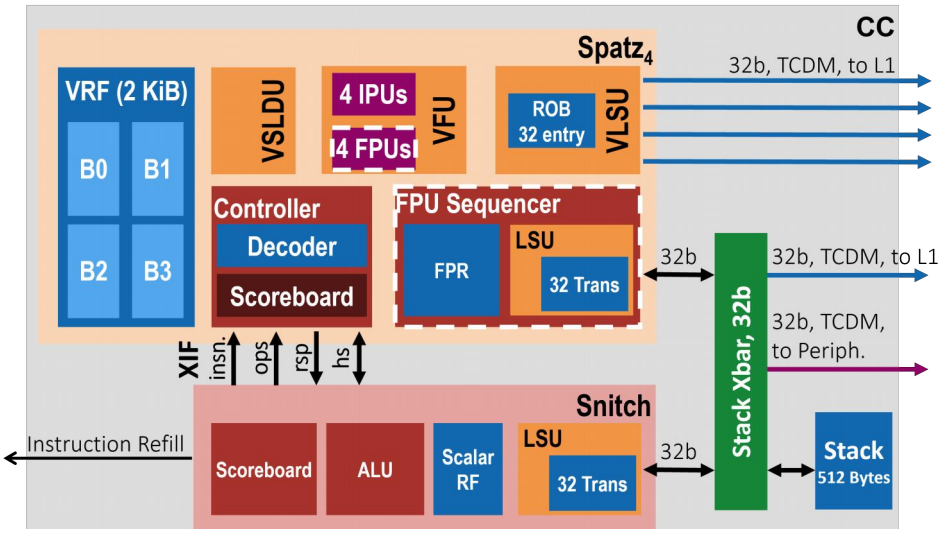


Figure 1: CachePool Core-Complex (CC): a Snitch core tightly coupled with a Spatz vector core. Dashed-lined modules represent optional components that can be disabled in the configuration.

3.2 Tile

3.2.1 Computing Tile

CachePool adopts a hierarchical cluster organization to ensure scalability and physical design feasibility. The fundamental building block is the *Tile*, shown in Figure 2, which is designed for massive replication. Each Tile integrates 4 32-bit Snitch–Spatz CCs, each equipped with a private 512 B stack memory. The number of cores per Tile is configurable through the system configuration files.

All CCs within a Tile share an 8 KiB four-way set-associative L1 Instruction Cache (I$). A single Advanced eXtensible Interface (AXI) master port handles the refill traffic for this instruction cache, shared among all cores.

On the data side, the CCs are tightly coupled to a shared 256 KiB four-banked L1 Data Cache (D$) via runtime-configurable crossbars, described in detail in Section 3.3. Each cache bank is managed by an in-situ cache controller (Section 3.4), which maps to a portion of the global memory address space. Cache refills are served to the cluster through a TCDM-based protocol.

3.2.2 Atomic Unit

Although the single-level shared-cache design eliminates the need for complex cache coherence, memory consistency must still be ensured. To this end, an atomic unit is placed before the L1 D$ controller, providing Atomic Memory Operations (AMOs) in compliance with the RISC-V "A" standard extension.

This unit was designed initially for SPM-based systems, where memory access latency is highly predictable. For CachePool, it was adapted with a handshake-driven Finite State Machine (FSM) to cope with the variable latencies introduced by cache-based memory access.

3.3 Physically Feasible Hierarchical Interconnection Design

3.3.1 Interconnection Protocol Selection

The choice of interconnection protocol is critical in large clusters, as it directly impacts both physical feasibility and performance. The AXI protocol, developed by AMBA, is widely adopted in modern System-on-Chip (SoC) designs and is therefore a natural candidate for off-cluster

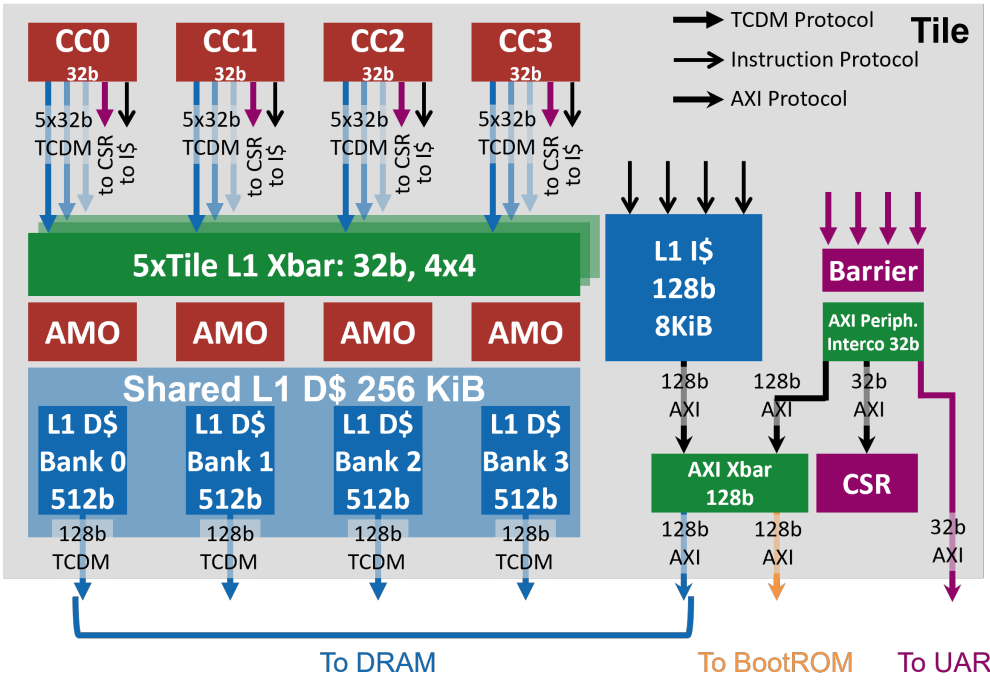


Figure 2: Overview of the CachePool Tile architecture with 256KiB L1 Cache. The remote interconnection between tiles is not shown.

communication, where compatibility with existing components is important. However, its multi-channel design significantly increases wiring complexity, which limits scalability. Moreover, the in-order processing constraint within each AXI ID requires support for a large number of IDs to sustain high parallelism, inflating the area of interconnection modules such as crossbars. For these reasons, AXI is not the preferred choice for inter-cluster communication.

By contrast, the TCDM protocol [43] is tailored for low-latency access between cores and memory. Although TCDM lacks a universal standard, making it unsuitable for off-cluster integration, it provides an efficient and scalable solution within a cluster. In the CachePool design, the adopted TCDM variant features a request channel and a response channel, with transaction IDs enabling in-order delivery at the receivers while tolerating out-of-order completions. Burst support is additionally incorporated into the L2 refill interconnection to handle longer cacheline transfers efficiently.

3.3.2 L1 Cache Interconnection

A hierarchical topology is essential to implement a physically feasible interconnection network between PEs and the fully shared L1 D$. To sustain low-latency TCDM access, the design uses fully combinational routing with logarithmic crossbars and arbiters at each hierarchy level. Accordingly, this interconnection is realized using the TCDM protocol described in the previous subsection.

The L1 crossbar is decomposed into five independent 4×4 32-bit crossbars for future scaling and physical implementation consideration, alleviating routing pressure and improving layout feasibility. Bank interleaving of the L1 D$ is integrated directly into the crossbar. Instead of embedding complex address-remapping logic inside the cache controller, a lightweight address-scrambling mechanism is implemented within the crossbar and configured via Control Status Registers (CSRs). This idea is inspired by the DAS design [44], which introduced software-controlled address remapping in the TeraPool architecture. Our implementation is simplified, since the CachePool cache controller can directly access the entire memory address space.

3.3.3 L2 Refill Interconnection

A refill crossbar connects the cache refill interface to the off-chip DRAM channels at the cluster level. Similarly, this crossbar is implemented using TCDM protocol for area and performance considerations. To bridge the mismatch between the cacheline width (512 bit) and the interconnection width (128 bit), burst request support is added to the TCDM interconnect. Unlike the previous TCDM-Burst work [45], which targeted bandwidth-limited systems with constrained physical resources, CachePool provides sufficient memory bandwidth to sustain core consumption. Here, the wider cacheline acts similarly to a "next-line prefetcher," primarily serving to hide DRAM access latency. As a result, burst responses in CachePool are forwarded serially rather than grouped.

The transactions are converted to AXI protocol before leaving the cluster, for better compatibility at SoC level.

The refill crossbar implements a select-lock mechanism to enforce ordering within a burst. This mechanism guarantees in-order delivery of burst responses, significantly reducing the cache controller's logic complexity.

In the current single-tile implementation, the refill crossbar only routes traffic from the cache controller to the DRAM channels through an all-to-all topology (Figure 3a). In future multi-tile designs, traffic from multiple tiles must be multiplexed, making a hierarchical interconnection essential. Figure 3b illustrates a possible four-tile organization, where a two-level structure, similar to the L1 crossbar, minimizes routing overhead. Looking ahead, the refill crossbar could also be integrated into each Tile to simplify cluster-level routing further.

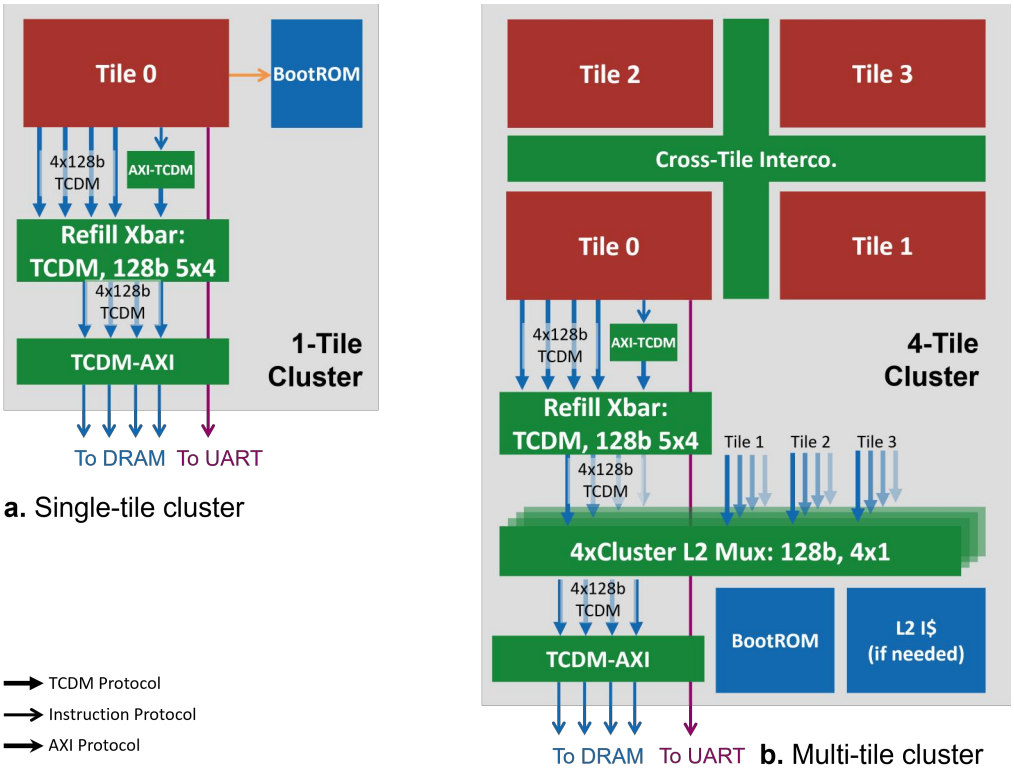


Figure 3: Overview of the CachePool cluster design. (a) Single-tile implementation with direct crossbar refill. (b) A possible four-tile extension with a hierarchical crossbar is needed for physical feasibility.

3.3.4 Cross-Tile Interconnection

As the cluster scales beyond a single Tile, a cross-tile interconnection is required to enable data access across neighboring Tiles. In this case, additional outgoing ports are added to the L1 cache crossbar, which forward requests to the cross-tile interconnect. The cross-tile fabric is implemented under the TCDM protocol and forwards transactions to their destination Tiles.

For a small number of Tiles, this can be realized with an all-to-all crossbar. As the system scales further, however, a NoC-based design becomes more appropriate to sustain performance while preserving physical feasibility. In the current first-phase implementation, CachePool targets a single-Tile cluster; therefore, no cross-tile interconnection has been included yet.

3.4 Memory Subsystem

3.4.1 Cache Hierarchy and Organization

CachePool adopts a fully shared L1 cache design without an intermediate L2 level. This choice eliminates the need for coherence management and avoids data duplication across cache levels. However, it increases interconnection complexity, since direct communication paths between Tiles are required (Section 3.3).

Alternatively, we are exploring an architecture with small private write-through caches at each PE, backed by a shared L2. This organization simplifies coherence, since a lightweight snoopy-style invalidation network would suffice. A comparative evaluation of these two approaches will be carried out once the write-through L1 design is completed.

3.4.2 InSitu Cache Controller

The shared L1 cache is managed by the *InSitu Cache* controller [46], which tolerates long-latency accesses directly to DRAM. InSitu Cache is a non-blocking, high-performance design that avoids the need for additional SRAM structures such as write buffers or Miss Handling Architectures (MHAs).

Its efficiency comes from reusing otherwise wasted cache resources for miss handling:

- Unified in-cacheline miss handling for both read and write misses.
- A Dynamic Subentry Expansion (DSE) mechanism that allocates multiple cache lines to a single miss entry, extending miss handling capacity under high-latency conditions.
- A Dynamic Subset Least Recent Used (DS-LRU) replacement policy that reclaims wasted cache lines for miss handling, while keeping the critical path short enough for high-frequency implementation.

Further details on the InSitu Cache controller will be provided once the related work is published.

3.4.3 DRAMSys Co-Simulation

A time-efficient yet cycle-accurate Dynamic Random-Access Memory (DRAM) subsystem is essential for TeraPool cluster co-simulation to evaluate data transfer performance alongside high-level memory hierarchies. *DRAMSys5.0* [47], an open-source and flexible framework for DRAM design space exploration, meets these requirements.[1] Developed by the Microelectronic Systems Design Research Group at RPTU Kaiserslautern-Landau, *DRAMSys5.0* is based on SystemC TLM-2.0 and supports cycle-accurate modeling of multiple memory technologies, including DDR3/4, LPDDR4, Wide I/O 1/2, GDDR5/5X/6, and HBM1/2.

[1]The DRAMSys5.0 repository is available at `https://github.com/tukl-msd/DRAMSys`, released under the BSD 3-Clause License.

During the TeraPool project, an Register Transfer Level (RTL) interface to *DRAMSys5.0*[2] was developed and connected to the cluster via the AXI interconnect. The *DRAMSys5.0* subsystem was compiled into a shared library (`.so` format) and linked into the ModelSim environment,[3]. In CachePool, we reuse the same infrastructure with a default DDR4 configuration featuring four memory channels.

[2] `https://github.com/pulp-platform/dram_rtl_sim`

[3] ModelSim supports behavioral, RTL, and gate-level simulation with a platform-independent compile flow: `https://eda.sw.siemens.com/en-US/ic/modelsim/`.

4 Physical Implementation

This section evaluates the physical feasibility and area efficiency of different CachePool single-tile cluster configurations, with particular focus on the impact of varying cacheline widths.

4.1 Methodology

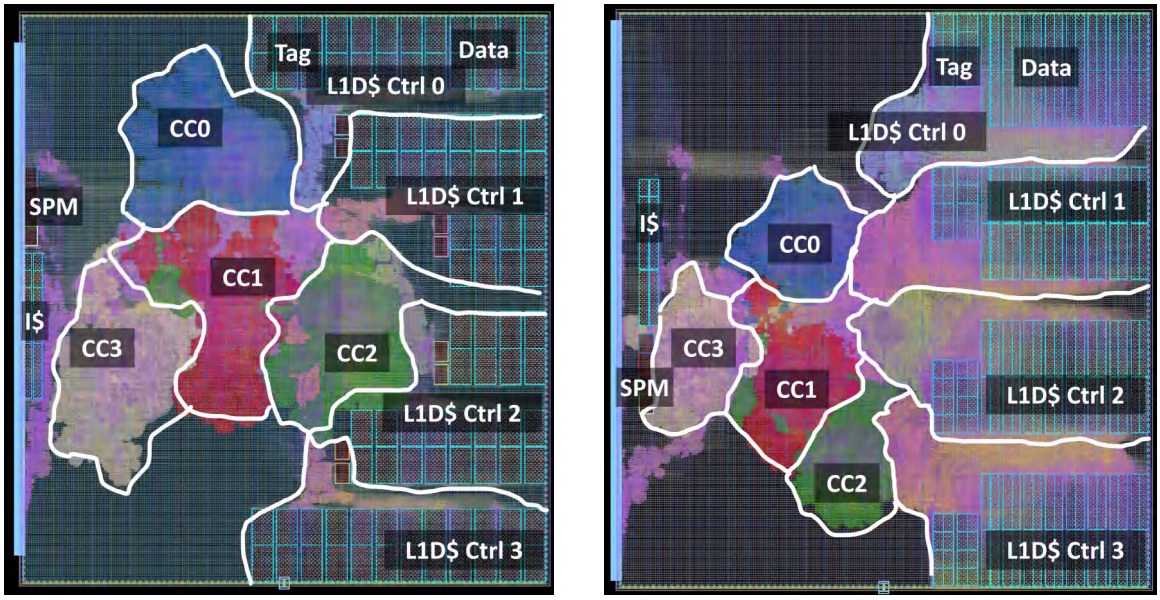
The CachePool cluster is implemented in GlobalFoundries' 12LPPLUS FinFET technology. Once the design flow is fully validated, we plan to migrate to TSMC's 7 nm process. For the single-tile cluster, a hierarchical implementation is not required; however, such a flow will become essential as the design scales to multi-tile systems. All synthesis, placement, and routing are carried out using Synopsys Fusion Compiler 2023.12-SP4 in a flattened flow. Future scaled-up clusters will adopt a top-down hierarchical approach. The current design targets 1 GHz at worst-worst PVT corner for the 12LPPLUS process.

4.2 Floorplan

Since the current technology node is not the final target, the floorplan is not tuned for optimal density. Instead, the backend results primarily serve to assess physical feasibility and provide a reference for logic area estimation.

Figures **??** and **??** illustrate the placed-and-routed layouts of the single-tile CachePool cluster for cacheline widths of 128 b and 512 b, respectively. Both configurations integrate a four-banked 256 KiB L1 D$.

Due to the unavailability of 128×512 b SRAM macros, the 512 b configuration is emulated using four 128×128 b SRAM instances. This workaround results in a larger area footprint than a dedicated macro would require. A more detailed breakdown is presented in the next section.



128 b cacheline configuration with FPUs. 512 b cacheline configuration without FPUs.

Figure 4: Placed-and-routed layouts of the CachePool single-tile cluster for two configurations: (left) 128 b cacheline with FPU, (right) 512 b cacheline without FPU.

4.3 Area Breakdown

The Gate Equivalent (GE), defined as the area of a two-input NAND gate, is a widely used metric to represent logic area in physical design independently of the technology node. The main area breakdown is derived from post-synthesis data, since the floorplan used in Figure 5 is not fully optimized. We also report a breakdown from post-layout results in Figure 6, but these numbers are more sensitive to floorplan variations and should be interpreted with caution.

The dominant contributor to the total area is the L1 memory subsystem, which accounts for roughly 60% of the total in the 128 b cacheline configuration with FPU support. Increasing the cacheline size leads to a significant growth in both the data banks and the controller. The data bank overhead arises from the lack of native 128×512 SRAM macros, which forces us to emulate them using four 128×128 SRAMs. This results in poor area efficiency. Two possible remedies are under consideration:

- *Multi-muxed SRAMs:* A 128×512 configuration can be realized by multiplexing narrower SRAM macros. While still suboptimal, this approach improves efficiency compared to the current solution.
- *Bandwidth-matched SRAMs:* The effective core-to-cache bandwidth is only 128 b. Wider cachelines are mainly used for pseudo prefetching and bandwidth matching with DRAM. Therefore, using a 512×128 SRAM macro could achieve full core utilization without performance loss, while improving area efficiency.

The PEs together occupy about 30% of the total area, with approximately 900 kGE attributable to the FPUs.

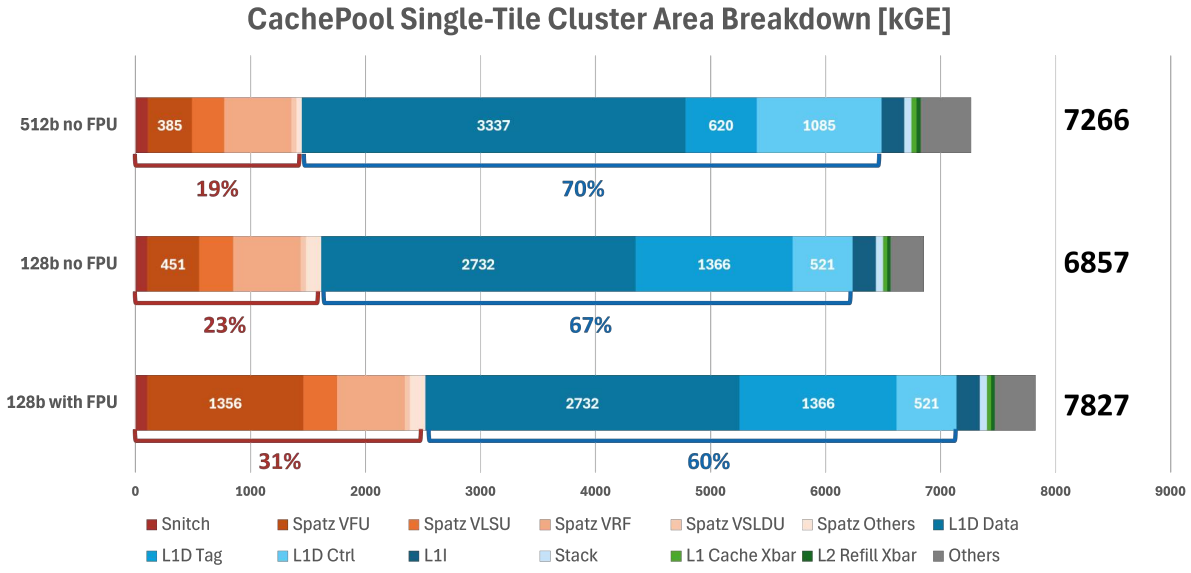


Figure 5: Post-synthesis area breakdown of the CachePool single-tile cluster. The 128 b configuration without FPU is estimated by subtracting the FPU area.

Overall, the total cluster area ranges from roughly 6.8 MGE (128 b cacheline without FPU) to 7.8 MGE (128 b cacheline with FPU), with the 512 b cacheline variant occupying about 7.3 MGE. This confirms that memory dominates the area budget, while FPUs contribute a significant but optional overhead. Interconnection logic and miscellaneous components remain below 5%, indicating they are not a limiting factor at this scale. Looking ahead, scaling CachePool to many tiles will be primarily constrained by SRAM efficiency rather than logic complexity, making optimized memory macros a key requirement for future implementations.

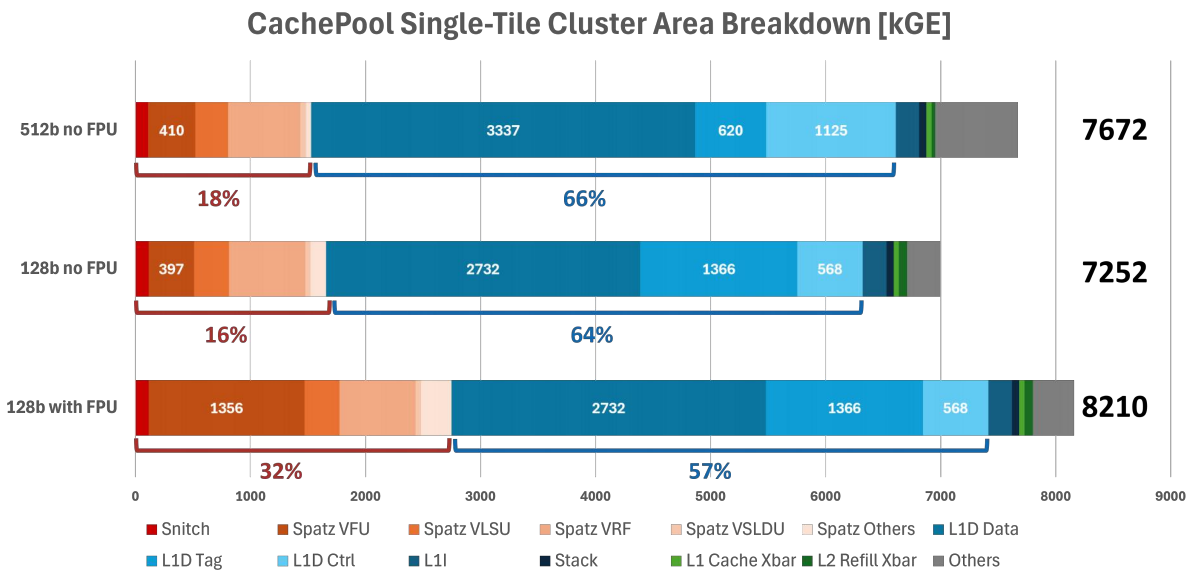


Figure 6: Post-route area breakdown of the CachePool single-tile cluster. The 128 b configuration without FPU is estimated by subtracting the FPU area.

5 Vector Kernels

This section describes the software implementation of key micro-kernels used in the ManyRVData dataplane. At the current stage, the algorithm is decomposed into multiple standalone micro-kernels to simplify implementation and enable cross-comparison. In the second half of the project, these kernels will be chained and evaluated to represent the full control algorithm.

The main micro-kernels are:

- **Dot Product (DOTP)**: Computes the dot product between two vectors, producing a scalar result.
- **Matrix-Vector Multiplication (MVM)**: Computes the product of a matrix and a vector, producing a vector result.
- **Matrix Multiplication (MatMul)**: Computes the product of two matrices, producing a matrix result.
- **Shannon–Hartley Channel Capacity Formula (Shannon)**: Evaluates $\vec{y} = \log(1 + \vec{x})$, producing a vector result.
- **Vector Sorting**: Sorting an input vector from largest to smallest.

5.1 DOTP

This kernel is implemented using vector instructions and distributed across vector cores according to their core IDs. Two input vectors are loaded into the vector core and processed using vector-vector multiply-accumulate (`vmacc`) operations. The partial results are first reduced within each vector core, followed by a final scalar reduction across cores.

5.2 MVM

This kernel is implemented using vectorized column-wise scalar–vector multiplications. Each column of the input matrix is loaded via vector load instructions and multiplied by the cor-

responding scalar element from the input vector, with results accumulated across the column. Each core contributes a portion of the final output vector.

5.3 MatMul

This kernel is based on the Gustavson algorithm, which streams scalar elements from matrix A and performs multiply-accumulate operations with columns of matrix B. The approach overlaps vector load latencies with computation, enabling sustained throughput, and distributes workloads across cores with minimal interference.

In practice, each row of matrix A is streamed element by element, and the corresponding scalar values are broadcast across the vector cores. Each scalar multiplies an entire column of matrix B, which is held in vector registers, followed by an accumulation into the partial results of matrix C. This streaming style reduces memory traffic by reusing data from matrix B across multiple iterations, while keeping the vector units busy.

The Gustavson algorithm is particularly attractive for the CachePool architecture: It exposes a high degree of parallelism across both rows of matrix A and columns of matrix B, maps naturally to vector–scalar and vector–vector multiply-accumulate instructions, and avoids inter-core synchronization during the main loop. The resulting workload distribution ensures balanced utilization of compute and memory resources, making this kernel compute-bound rather than memory-bound at large problem sizes.

5.4 Shannon (Not Implemented Yet)

The Shannon kernel is motivated by the Shannon–Hartley channel capacity formula, which characterizes the maximum achievable data rate of a communication channel as

$$C = B \cdot \log_2\left(1 + \frac{S}{N}\right),$$

where B is the channel bandwidth, S is the signal power, and N is the noise power. The logarithmic term $\log(1 + S/N)$ appears frequently in physical-layer and baseband algorithms, making it an important target for acceleration.

Within the control algorithm, this micro-kernel is applied to compute the subcarrier gain for each user. Here, the bandwidth factor can be omitted, the signal-to-noise ratio is represented as a vector input, and the logarithm is evaluated in the natural base. The kernel therefore simplifies to

$$\vec{G} = \log(1 + \vec{S}),$$

where $\vec{G}$ denotes the per-subcarrier gains and $\vec{S}$ is the vector of signal-to-noise ratios produced by a previous stage.

A practical implementation will approximate the natural logarithm using polynomial expansions, such as a Taylor series or minimax approximation, to strike a balance between accuracy and computational efficiency. These approximations can be mapped efficiently to vector instructions, enabling parallel evaluation across all input elements.

5.5 Vector Sorting (Not Implemented Yet)

The vector sorting kernel targets a straightforward ordering operation, producing a vector rearranged from maximum to minimum. Unlike indirect sorting, which maintains a separate index vector, this kernel focuses on directly reordering the data elements.

A key challenge arises from the lack of dedicated sorting instructions in the RVV 1.0 ISA. Consequently, sorting must be constructed from existing primitives such as masking, gathering, and indexed memory operations. On the Spatz core, masking and gather functionality are still

under development, and scatter operations are not yet available. As a potential workaround, indexed load and store instructions could be used to emulate scatter–gather behavior.

Alternatively, the scalar Snitch core could be employed to execute the sorting algorithm, trading off vector parallelism for reduced implementation complexity.

The detailed design and evaluation of both the vector sorting and Shannon kernels will be carried out in the next phase of the project.

6 RLC Kernel Implementation

This section presents the software implementation of the RLC kernels used in the ManyRVData dataplane. We focus on downlink AM-mode processing at gNodeB side under high-load traffic models, aligned with the Layer-2 DP scenarios and timing target provided by the partner specifications. In our DP setting, RLC entities operate under tight TTI constraints (typ. 0.5 ms), must sustain multiple million packets per second throughput, and manipulate irregular data structures (linked lists), yielding sparse and non-contiguous memory accesses. These properties favor a cache-based, hardware-managed memory hierarchy rather than software-managed DMA and scratchpad memory. *ManyRVData* therefore targets a shared-cache cluster. By default, L1 cache banks are shared across all cores, with an optional per-tile private L1 partition for stronger locality or isolation.

6.1 Workload model and constraints

We adopt three traffic scenarios for evaluation (single-user peak, multi-user peak, multi-user average). They determine per-slot packet counts per RLC entity, and consequently the concurrency and memory pressure seen by the kernels. The partner document defines packet sizes, per-second slot counts, and closed-form relationships between throughput and packet rate, which we reuse in our test harness.

Table 1: RLC evaluation scenarios (DL/UL). Values shown are from the partner's L2DP description and serve as our default harness configuration.

Scenario	Users	DL Gbps	DL pkt B	DL pps	DL slots/s	DL pkts/slot	UL Gbps	UL pkts/slot
Single-user peak	1	20	1350	1,851,851	1600	1157	1	1953
Multi-user peak	48	50	800	7,812,500	1600	4882	8	15625
Multi-user average	4800	20	800	3,125,000	1600	1963	4	7813

Notes. Per-user rates derive from the total divided by the number of users; slot counts per second are fixed by numerology; TB sizes for assembly come from the control/scheduler stage and are provided to the RLC assembly task at runtime.

Protocol and configuration. All entities run in Acknowledged Mode (AM) with 18-bit sequence numbers (SN); `pollPDU`=32 and `pollByte`=25,000. In our testcases, gNB receives only *positive* ACK STATUS PDUs; retransmission and timers are disabled.

6.2 Software Architecture and Implementation

6.2.1 Workflow of the RLC Kernel

The RLC kernel can be abstracted as a producer–consumer program: producers generate and enqueue work items; the consumer processes them and maintains protocol state. The end-to-end workflow and how the two sides interact are shown in Fig. 7.

Producers repeatedly pull the next PDCP SDU descriptor from a shared cursor (under a small lock), request a `Node` from the fixed-size pool (`mm_alloc()` under allocator lock), fill its metadata (`data`, `tgt`, `len`), and append it to the tail of the shared *to-send* list (`list_push_back()` under list lock). If allocation fails or no new SDU is available, the producer enters the *Wait* state (short backoff) and retries.

A single *consumer* thread loops on the *to-send* list: it removes the head node (`list_pop_front()` under lock); if the list is empty, it briefly *Wait*s. Otherwise, it performs the data move (vectorized memcpy standing in for package assembly) from source memory address to the target address, then updates RLC status: increments `pduWithoutPoll` and `byteWithoutPoll`, advances the

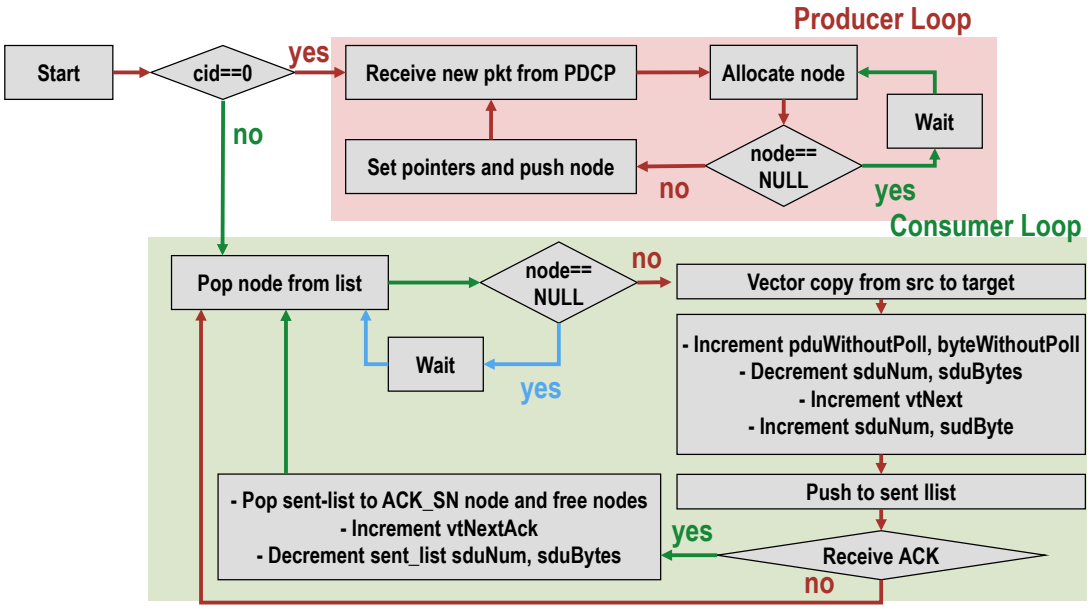


Figure 7: High-level producer–consumer architecture for RLC kernel

next sequence number `vtNext`, and accounts the package leaving/entering queues by decrementing `sduNum`/`sduBytes` on the *to-send* list and incrementing the same counters on the *sent* list. The processed node is appended to the *sent* list under lock. Upon reception of a STATUS with `ACK_SN`, the consumer pops acknowledged nodes from the head of the *sent* list up to `ACK_SN`, advances `vtNextAck`, decrements the *sent*-list counters accordingly, and returns each acknowledged node to the pool via `mm_free()`.

All shared structures (PDCP cursor, allocator, both linked lists) are protected by compact spinlocks to ensure atomic push/pop and allocation; a one-time cluster barrier precedes the workflow so producers and consumer start from a consistent state. Back-pressure naturally arises at two points shown as *Wait*: empty queues (consumer idle) and pool exhaustion (producers back off). The loop continues until producers exhaust PDCP SDUs and the consumer drains and acknowledges all in-flight nodes.

6.2.2 Producer–Consumer Roles

Producers. Producer threads simulate packet ingress (e.g., PDCP SDUs) and enqueue work items into a shared *to-send* queue implemented as a doubly linked list. Each producer repeatedly: (i) obtains the next PDCP packet descriptor from a shared cursor, (ii) allocates a `Node` from a fixed-size memory pool, (iii) fills its metadata (payload pointer, size), and (iv) appends it to the tail of the *to-send* list under a lock. This models concurrent ingress sources feeding the RLC layer.

Consumer. A single consumer thread represents the RLC transmitter. It repeatedly: (i) pops a `Node` from the head of the *to-send* list under a lock, (ii) performs data movement (copy/assemble) to the target buffer (standing in for AMD PDU assembly), (iii) updates RLC state (e.g., sequence counters; bytes/PDUs without poll), and (iv) pushes the `Node` into a *sent* list (tracking in-flight PDUs) under a lock. The consumer also simulates ACK processing: once an in-flight threshold is reached, it dequeues a batch from the *sent* list (front), advances the transmit window, and returns those `Nodes` to the memory pool.

6.2.3 Concurrency and Synchronization

Spinlocks. Shared resources (*to-send* list, *sent* list, memory pool, and the PDCP package cursor) are protected by simple spinlocks implemented with atomic test-and-set. There is also an optional backoff delay added to reduce contention. A hardware barrier synchronizes cores once after initialization to ensure a consistent start.

6.2.4 Data Structures and Memory Management

Double linked lists. Work queues are linked lists (head/tail, counters) of nodes. `list_push_back()` appends in $O(1)$ time (updates tail), `list_pop_front()` removes in $O(1)$ time (updates head).

Node. Each `Node` holds `prev/next` pointers, a pointer to the source payload, a pointer to the destination buffer, and the payload length.

Fixed-size pool allocator. A global memory context manages a statically allocated, cacheline-aligned buffer partitioned into fixed-size *pages* (one page per `Node`). Allocation uses a lock-protected bump pointer; when the pool is exhausted, freed pages are recycled from a lock-protected free-list. This yields deterministic, fragmentation-free allocation without a general-purpose heap.

6.2.5 Task Scheduling and Thread Management

For now, we use a static, bare-metal mapping: each core executes one function (producer or consumer). Coordination is via shared queues and spinlocks. Back-pressure arises naturally from the finite memory pool and queue occupancy.

7 Benchmarking and Performance Analysis

This section evaluates the performance of the CachePool single-tile cluster through cycle-accurate simulation using QuestaSim, with *DRAMSys* serving as the main memory model. We investigate the impact of different cacheline widths (128 b, 256 b, and 512 b) under both hot- and cold-cache conditions. The simulations are carried out on the configurations with FPUs.

7.1 Vector Kernel Benchmarking

To better illustrate the performance characteristics of the design, hardware utilization is analyzed through both cycle-level measurements and the roofline model.

Figure 9 reports the execution cycles of representative vector kernels. Solid bars correspond to hot-cache runs, while the shaded extensions capture the additional cycles incurred under cold-cache conditions. In general, longer cachelines improve performance, particularly in the cold-cache case.

For the most memory-bound kernel (dot product), the 512 b configuration achieves over 20% FPU utilization even under cold-cache conditions, highlighting the cluster's ability to tolerate memory latency and exploit available off-chip bandwidth. As data reuse increases, cache misses have a smaller performance impact. For compute-bound kernels, such as matrix multiplication, the cluster reaches over 90% FPU utilization in both hot- and cold-cache scenarios, demonstrating that computation resources can be fully exploited when memory bandwidth is not the limiting factor.

In addition to the execution cycles and utilization data shown in Figure 9, Figure 8 presents roofline plots that relate achieved performance to the theoretical compute and memory bandwidth limits. A consistent trend can be observed: in hot-cache scenarios, the cluster performance approaches the roof, indicating that there are no significant bottlenecks within the core microarchitecture or the core-to-L1 interconnect for typical vector kernels. In cold-cache runs, performance progressively converges toward the hot-cache results as the arithmetic intensity increases. Moreover, the performance gap between hot and cold caches narrows with larger cacheline widths, owing to the prefetching benefits of longer cachelines that improve hit rate and bandwidth utilization.

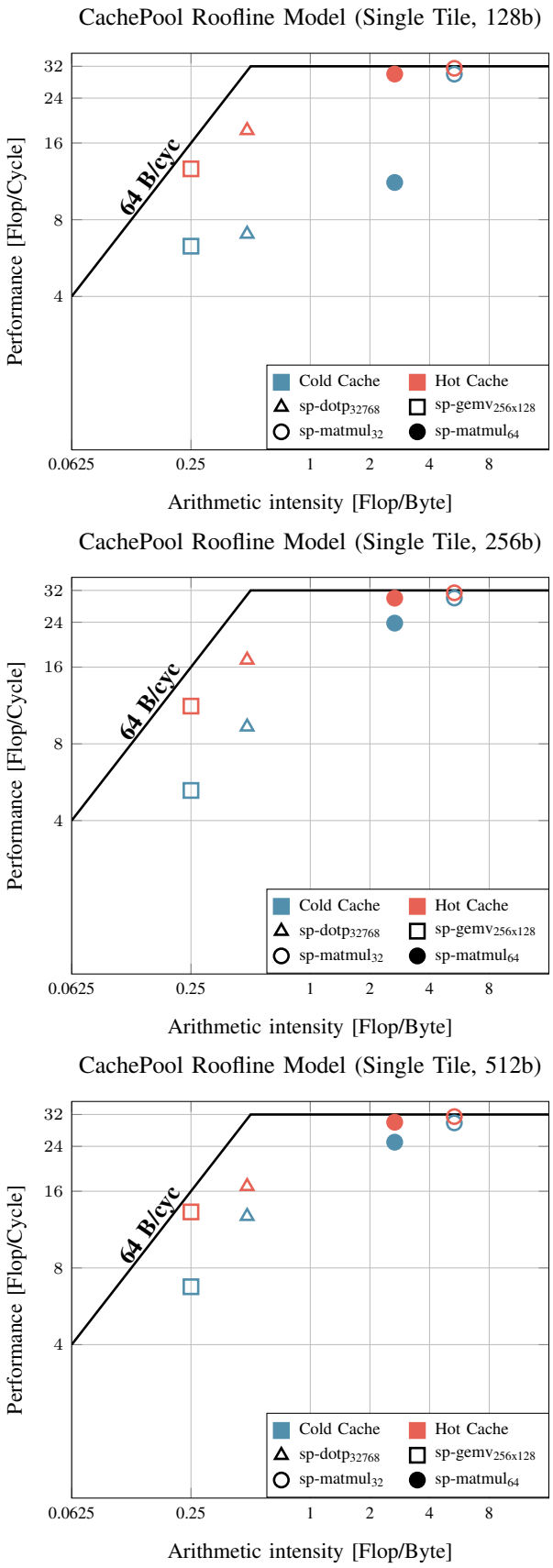


Figure 8: Roofline plots of the single-tile CachePool cluster with 128-, 256-, and 512-bit cachelines. Each plot shows the achieved performance of representative vector kernels under hot- and cold-cache conditions relative to the theoretical compute and bandwidth roofs.

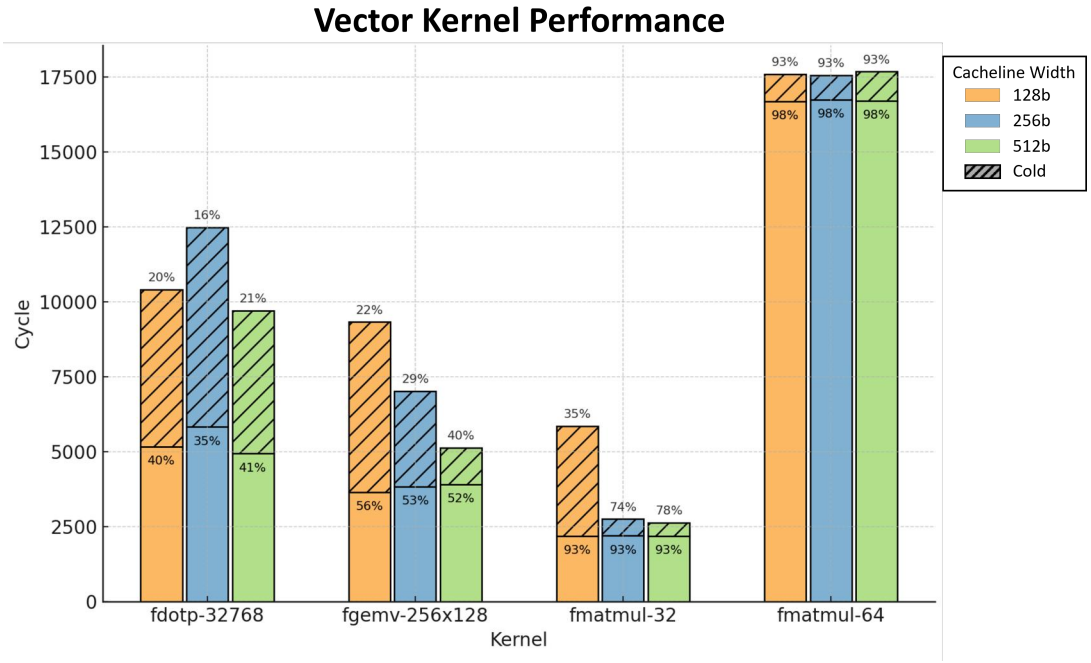


Figure 9: Vector kernel benchmark results of the CachePool cluster with different cacheline widths. Solid bars represent hot-cache execution, shaded extensions show additional cycles for cold-cache execution.

7.2 RLC Kernel Benchmarking

The performance results presented in Table 2 and Table 3 offer an initial view of the RLC data processing kernel performance on the current single tile implementation and how cacheline width impacts them. In this evaluation, we assume the clock frequency is 1 GHz.

Table 2: Data Movement Kernel for One 1350-Byte RLC Package

Cacheline Width	**Cycle/Pkg**	**Bandwidth (Gbps)**
128-bit	330	30.48
256-bit	259	38.84
512-bit	243	41.39

In Table 2, which evaluates the Data Movement Kernel responsible for loading, assembling, and storing a single 1350-byte RLC packet using vector instructions, wider cachelines clearly lead to better memory bandwidth utilization. The 512-bit cacheline configuration achieves the highest throughput (41.39 Gbps), indicating that vector memory operations benefit significantly from longer cachelines due to longer bursts and data prefetching.

Table 3: Full RLC Data Management Kernel

Cacheline Width	Cycle/Pkg	BW (Gbps)	Avg L1$ Miss Rate	Avg Miss Stall Cycle
128-bit	1552	6.48	59.8%	132 667
256-bit	620	16.22	44.4%	2 448
512-bit	1051	9.57	43.9%	47 848

However, this trend does not hold in Table 3, which measures the Full RLC Data Management Kernel, a more comprehensive workload including scalar control logic, linked list traversal, and producer-consumer synchronization. The results reveal that the 256-bit cacheline configuration offers the best performance, achieving a throughput of 16.22 Gbps with a cycle-per-package count of only 620. Interestingly, while both the 256-bit and 512-bit configurations show similar average L1 miss rates (44.4% vs. 43.9%), the 512-bit configuration suffers a dramatic increase in average miss stall cycles per cache bank. This stark contrast highlights the hidden cost of wider cachelines.

A deeper analysis suggests that the 512-bit configuration, due to its reduced number of cachelines under fixed total cache capacity, experiences more conflict misses and evictions. These evictions introduce higher memory channel pressure, blocking subsequent vector loads and degrading overall performance. In contrast, the 256-bit configuration strikes a balance between spatial locality and miss penalty, avoiding excessive evictions while maintaining high bandwidth utilization.

To mitigate such issues in future iterations, we plan to add hardware support for non-cacheable memory access, specifically targeting stream-like RLC packet data. This approach aims to prevent cache pollution by stream data and alleviate contention-induced stalls. Overall, our findings emphasize that wider cachelines do not necessarily translate to higher performance and must be carefully tuned based on workload characteristics.

It is important to emphasize that these results represent a preliminary evaluation. Both the hardware and software stacks are still evolving. While these measurements serve as useful early indicators, further profiling is required to fully understand the bottlenecks behind each configuration. We plan to analyze pipeline stalls, cache behavior, and load/store throughput to guide architectural refinements.

Based on the current observations, we estimate that a 2-tile configuration should be capable of handling the single-user peak rate test case. For the more demanding multi-user average scenario, 4 to 6 tiles are likely sufficient. As of now, only the single-user test case has been

implemented and evaluated. Future work will include the implementation and performance analysis of both the multi-user average and peak workloads to validate system scalability and efficiency.

8 Next Year Plan

Building on the progress made in the first year, we outline the following technical objectives and development tasks for the second year of the ManyRVData project.

8.1 Data Cache Optimization

- **Enhance Area Efficiency:** Optimize the internal architecture of the cache controller and the dimensions of data SRAM macros to improve the area efficiency. Improvements will focus on data RAM utilization and buffer sizing efficiency.
- **Reduce Load-to-Use Latency:** Refine the cache coalescer logic and minimize interconnect latency to reduce memory access latency, especially under contention.
- **Uncached Access Support:** Enable uncached request support to allow better handling of streaming data and reduce interference with frequently reused cache lines.
- **Prefetching Scheme Exploration:** Extend beyond the current next-3-line prefetching (enabled by a 512-bit cacheline design) to investigate more adaptive or pattern-based prefetching strategies.

8.2 Data Cache Partition Exploration

Design and implement configurable tile-private L1 cache partitioning to support high-bandwidth, low-latency local access in multi-tile systems. We plan to explore the following two designs, evaluate their performance and PPA impact:

- **Bank-level Partitioning:** A scheme where the whole cache bank can be configured as a private partition. This scheme is simpler to implement, but has a coarser partition granularity, and the bandwidth for both the shared and private partitions will be reduced.
- **Set-index-level Partitioning:** A finer-grain approach that enables higher concurrency by partitioning the cache based on the set index. This scheme is more complex, but has a finer partition granularity and will not reduce the cache access bandwidth for all the partitions.

Software runtime support will be explored to help manage the partitions, especially the cache coherence between the private partitions.

8.3 Processing Element Microarchitecture Optimization

Further relax RAW hazard checks between the scalar and vector cores within each PE to enhance concurrent scalar-vector execution. This aims to improve dual-issue utilization and is particularly relevant for workloads with frequent scalar-vector interleaving.

8.4 Multi-Tile Scaling and Interconnect Exploration

Scale the architecture to support 4–8 tiles to accommodate multi-user, high-throughput workloads. Investigate and compare different inter-tile interconnects (e.g., hierarchical crossbar, mesh-based network-on-chip) in terms of performance, latency, PPA, and software programmability.

8.5 Advanced Backend Implementation and PPA Evaluation

Shift the backend implementation from the current 12nm technology node to a more advanced node. Do the complete back-end flow, including logic synthesis, floorplanning, placement, and routing, have a detailed PPA analysis and design iteration.

8.6 Software Kernel Finalization and Evaluation

- **RLC Data Management Kernel:** Extend current implementation to support multiple users and support the multi-tile design, as well as the runtime support for the configurable cache partition schemes.
- **Control Algorithm Kernels:** Implement and benchmark additional kernels with representative control logic patterns, including Shannon-Hartley Channel Capacity Calculation and Vector Sorting. These kernels will validate the efficiency of mixed scalar-vector execution under real-time DP constraints.

8.7 Exploration of Private L1 Cache

A parallel effort has been initiated to investigate the feasibility and performance impact of introducing a small private L1 Write-Through (WT)-cache for each core, hierarchically connected to the existing shared L2 cache subsystem. In this configuration, only lightweight coherence mechanisms are required, mainly invalidations triggered when an entry is updated in the L2 cache, thereby reducing protocol complexity compared to coherent designs for write-back L1 caches.

Building upon this concept, a lazy write-through policy is being explored. Unlike a conventional write-through cache that propagates every store immediately to the next level, the lazy write-through scheme temporarily retains dirty data in the L1 and propagates updates to the L2 cache either periodically or upon specific requests. This approach aims to balance the simplicity of write-through with the bandwidth efficiency of write-back, potentially reducing redundant writes and improving overall performance.

The effectiveness of this design will be evaluated against the current cluster architecture to identify the most efficient memory hierarchy for future system

References

[1] Samuel Riedel et al. "MemPool: A Scalable Manycore Architecture with a Low-Latency Shared L1 Memory". In: *IEEE Transactions on Computers* 72.12 (2023), pp. 3561–3575. DOI: 10.1109/TC.2023.3307796. URL: https://arxiv.org/abs/2303.17742.

[2] Yichao Zhang et al. "TeraPool: A Physical Design Aware, 1024 RISC-V Cores Shared-L1-Memory Scaled-up Cluster Design with High Bandwidth Main Memory Link". In: *IEEE Transactions on Computers* (2025), pp. 1–14. DOI: 10.1109/TC.2025.3603692.

[3] Ampere Computing. *AmpereOne Family Product Brief*. May 2023. URL: https://amperecomputing.com/briefs/ampereone-family-product-brief (visited on 09/15/2025).

[4] *AMD EPYC® 9004 Series Processors: Architecture Overview*. Tech. rep. Advanced Micro Devices, Inc. (AMD), 2024. URL: https://www.amd.com/system/files/documents/epyc-9004-series-processors-architecture-overview.pdf (visited on 09/15/2025).

[5] Intel Corporation. *The Next-Generation Efficient-core Xeon Processor (Sierra Forest)*. Hot Chips 2023 presentation. 2023. URL: https://www.intel.com/content/www/us/en/events/hot-chips-2023/briefings.html?presentation=787431 (visited on 09/15/2025).

[6] NVIDIA. *NVIDIA Grace Hopper Superchip Architecture In-Depth*. Nov. 2022. URL: https://developer.nvidia.com/blog/nvidia-grace-hopper-superchip-architecture-in-depth/ (visited on 09/15/2025).

[7] Daniel Sorin, Mark Hill, and David Wood. *A primer on memory consistency and cache coherence*. Morgan & Claypool Publishers, 2011.

[8] Milo MK Martin, Mark D Hill, and David A Wood. "Token coherence: Decoupling performance and correctness". In: vol. 31. 2. ACM New York, NY, USA, 2003, pp. 182–193.

[9] Xiangyao Yu and Srinivas Devadas. "Tardis: Time traveling coherence algorithm for distributed shared memory". In: *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE. 2015, pp. 227–240.

[10] Xiaowei Ren et al. "Hmg: Extending cache coherence protocols across modern hierarchical multi-gpu systems". In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2020, pp. 582–595.

[11] Johnathan Alsop, Matthew Sinclair, and Sarita Adve. "Spandex: A flexible interface for efficient heterogeneous coherence". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2018, pp. 261–274.

[12] Xiangyao Yu et al. "Tardis 2.0: Optimized time traveling coherence for relaxed consistency models". In: (2016), pp. 261–274.

[13] Arm Ltd. *Memory attributes*. Available at: https://developer.arm.com/documentation/100941/0101/Memory-attributes. 2023. URL: https://developer.arm.com/documentation/100941/0101/Memory-attributes.

[14] Arm Ltd. *AMBA AXI and ACE Protocol Specification*. AXI Coherency Extensions (ACE). 2013. URL: https://developer.arm.com/documentation/ihi0022/latest.

[15] Arm Ltd. *CoreLink CMN-600 Coherent Mesh Network - Technical Reference Manual*. Document Number: 100614. 2020. URL: https://developer.arm.com/documentation/100180/latest/.

[16] HSA Foundation. *HSA System Architecture Specification v1.2*. Available at: https://hsafoundation.com/wp-content/uploads/2021/02/HSA-SysArch-1.2.pdf. 2020. URL: https://hsafoundation.com/wp-content/uploads/2021/02/HSA-SysArch-1.2.pdf.

[17] Marc S Orr et al. "Synchronization using remote-scope promotion". In: *ACM SIGARCH Computer Architecture News* 43.1 (2015), pp. 73–86.

[18] Compute Express Link Consortium. *What is CXL?* https://www.computeexpresslink.org/about-cxl. Available at: https://www.computeexpresslink.org/about-cxl. 2023.

[19] Rambus Inc. *Compute Express Link (CXL): All you need to know.* https://www.rambus.com/blogs/compute-express-link/. Available at: https://www.rambus.com/blogs/compute-express-link/. 2024.

[20] Arm Ltd. *Accelerating Cloud Workloads with CCIX.* https://www.arm.com/-/media/global/events/techcon/2017/pdf/303_armtechcon2017_acceleratingcloudworkloads.pdf. Presented at Arm TechCon 2017. 2017.

[21] Cadence Design Systems. *CCIX Verification Solution from Cadence.* https://community.cadence.com/cadence_blogs_8/b/ip/posts/ccix-verification-solution. Available at: https://community.cadence.com/cadence_blogs_8/b/ip/posts/ccix-verification-solution. 2019.

[22] Rebecca Lewington. *An AI Chip With Unprecedented Performance To Do the Unimaginable.* Aug. 2021. URL: https://www.cerebras.ai/blog/an-ai-chip-with-unprecedented-performance-to-do-the-unimaginable (visited on 09/15/2025).

[23] Yichao Zhang et al. "TeraPool-SDR: An 1.89 TOPS 1024 RV-Cores 4 MiB Shared-L1 Cluster for Next-Generation Open-Source Software-Defined Radios". In: *Proceedings of GLSVLSI 2024*. 2024. DOI: 10.1145/3649476.3658735. URL: https://arxiv.org/abs/2405.04988.

[24] Intel Labs. *The SCC Platform Overview.* Tech. rep. Intel Corporation, May 2010. URL: https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/intel-labs-single-chip-platform-overview-paper.pdf (visited on 09/15/2025).

[25] Kalray. *COOLIDGE™ MPPA® DPU Whitepaper.* Tech. rep. Kalray, 2022. URL: https://www.kalrayinc.com/wp-content/uploads/2023/10/WP_Kalray_MPPA_DPU_Coolidge_june2022.pdf (visited on 09/15/2025).

[26] Xiaowei Ren et al. "HMG: Extending Cache Coherence Protocols Across Modern Hierarchical Multi-GPU Systems". In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2020, pp. 582–595. DOI: 10.1109/HPCA47549.2020.00056. URL: https://ericrxw.github.io/xiaoweiren/docs/HPCA2020/HMG-HPCA2020.pdf.

[27] NVIDIA. *NVIDIA H100 Tensor Core GPU Architecture.* en-us. URL: https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c. URL: https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c (visited on 09/30/2024).

[28] Samuel Riedel et al. "MemPool: A Scalable Manycore Architecture with a Low-Latency Shared L1 Memory". In: *IEEE Transactions on Computers* 72.12 (2023), pp. 3561–3575. DOI: 10.1109/TC.2023.3307796. URL: https://arxiv.org/abs/2303.17742.

[29] Yichao Zhang et al. "TeraPool: A Physical Design Aware, 1024 RISC-V Cores Shared-L1-Memory Scaled-up Cluster Design with High Bandwidth Main Memory Link". In: *IEEE Transactions on Computers* (2025), pp. 1–14. DOI: 10.1109/TC.2025.3603692.

[30] Taehyun Kim et al. "Scalable Bandwidth Shaping Scheme via Adaptively Managed Parallel Heaps in Manycore-Based Network Processors". In: *ACM Trans. Des. Autom. Electron. Syst.* 22.4 (2017). ISSN: 1084-4309. DOI: 10.1145/3065926. URL: https://doi.org/10.1145/3065926.

[31] Yang Hu and Tao Li. "Enabling Efficient Network Service Function Chain Deployment on Heterogeneous Server Platform". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2018, pp. 27–39. DOI: 10.1109/HPCA.2018.00013.

[32] Bowen Wang et al. *A Dynamic Allocation Scheme for Adaptive Shared-Memory Mapping on Kilo-core RV Clusters for Attention-Based Model Deployment*. 2025. arXiv: 2508.01180 [cs.AR]. URL: https://arxiv.org/abs/2508.01180.

[33] Scott Davidson et al. "The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips". In: *IEEE Micro* 38.2 (2018), pp. 30–41. DOI: 10.1109/MM.2018.022071133.

[34] Dai Cheol Jung et al. "Scalable, Programmable and Dense: The HammerBlade Open-Source RISC-V Manycore". In: *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 2024, pp. 770–784. DOI: 10.1109/ISCA59077.2024.00061.

[35] Dai Cheol Jung and Michael Taylor. "Evaluating Ruche Networks: Physically Scalable, Cost-Effective, Bandwidth-Flexible NoCs". In: *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. ISCA '25. Association for Computing Machinery, 2025, pp. 1035–1048. ISBN: 9798400712616. DOI: 10.1145/3695053.3731010. URL: https://doi.org/10.1145/3695053.3731010.

[36] Jasmina Vasiljevic et al. "Compute Substrate for Software 2.0". In: *IEEE Micro* 41.2 (2021), pp. 50–55. DOI: 10.1109/MM.2021.3061912.

[37] David R. Ditzel and the Esperanto team. "Accelerating ML Recommendation With Over 1,000 RISC-V/Tensor Processors on Esperanto's ET-SoC-1 Chip". In: *IEEE Micro* 42.3 (2022), pp. 31–38. DOI: 10.1109/MM.2022.3140674.

[38] Yichao Zhang et al. *TeraNoC: A Multi-Channel 32-bit Fine-Grained, Hybrid Mesh-Crossbar NoC for Efficient Scale-up of 1000+ Core Shared-L1-Memory Clusters*. 2025. arXiv: 2508.02446 [cs.DC]. URL: https://arxiv.org/abs/2508.02446.

[39] Samuel Riedel et al. "MemPool: A Scalable Manycore Architecture With a Low-Latency Shared L1 Memory". In: *IEEE Transactions on Computers* 72.12 (2023), pp. 3561–3575. DOI: 10.1109/TC.2023.3307796.

[40] Yichao Zhang et al. "TeraPool-SDR: An 1.89TOPS 1024 RV-Cores 4MiB Shared-L1 Cluster for Next-Generation Open-Source Software-Defined Radios". In: *Proceedings of the Great Lakes Symposium on VLSI 2024*. GLSVLSI '24. Clearwater, FL, USA: Association for Computing Machinery, 2024, pp. 86–91. ISBN: 9798400706059. DOI: 10.1145/3649476.3658735. URL: https://doi.org/10.1145/3649476.3658735.

[41] Florian Zaruba et al. "Snitch: A Tiny Pseudo Dual-Issue Processor for Area and Energy Efficient Execution of Floating-Point Intensive Workloads". In: *IEEE Transactions on Computers* 70.11 (Nov. 2021), pp. 1845–1860. ISSN: 1557-9956. DOI: 10.1109/TC.2020.3027900.

[42] Matteo Perotti et al. "Spatz: Clustering Compact RISC-V-Based Vector Units to Maximize Computing Efficiency". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 44.7 (2025), pp. 2488–2502. DOI: 10.1109/TCAD.2025.3528349.

[43] Abbas Rahimi et al. "A fully-synthesizable single-cycle interconnection network for Shared-L1 processor clusters". In: *2011 Design, Automation and Test in Europe*. 2011, pp. 1–6. DOI: 10.1109/DATE.2011.5763085.

[44] Bowen Wang et al. *A Dynamic Allocation Scheme for Adaptive Shared-Memory Mapping on Kilo-core RV Clusters for Attention-Based Model Deployment*. 2025. arXiv: 2508.01180 [cs.AR]. URL: https://arxiv.org/abs/2508.01180.

[45] Diyou Shen et al. “TCDM Burst Access: Breaking the Bandwidth Barrier in Shared-L1 RVV Clusters Beyond 1000 FPUs”. In: *2025 Design, Automation and Test in Europe Conference (DATE)*. 2025, pp. 1–7. DOI: 10.23919/DATE64628.2025.10992996.

[46] Chi Zhang. *Pulp-Platform-Insitu-Cache*. URL: https://github.com/pulp-platform/Insitu-Cache.

[47] Lukas Steiner, Matthias Jung, Felipe S. Prado, et al. “DRAMSys4.0: An Open-Source Simulation Framework for In-depth DRAM Analyses”. In: *International Journal of Parallel Programming* 50 (April 2022 2022), pp. 217–242. DOI: 10.1007/s10766-022-00727-4. URL: https://doi.org/10.1007/s10766-022-00727-4.