TeraPool Project Report

Physical Uplink Shared Channel Processing in 1024 RISC-V Cores Shared-L1 Cluster

Report by:
Yichao Zhang, Ph.D. Student
Marco Bertuletti, Ph.D. Student
yiczhang, mbertuletti@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

June 4, 2024

Contents

1 Executive Summary

This document reports on the TeraPool project. The aim of the project is to scale up manycore general-purpose programmable architectures for embarrassingly parallel applications. The focus is to keep the application data shared between all the computing units and close to them to reduce data movement overheads. This requirement must be fulfilled as the number of cores increases to meet the performance and energy-efficiency demands of growing problem sizes. The project focuses on a manycore shared-L1-memory computing cluster architecture. A large shared L1 minimizes data swapping and transferring overhead from higher-level memory hierarchies. The research starts with the MemPool architecture, a 256-core cluster with a shared 1 MiB L1 memory, towards scaling up to thousands of cores while maintaining physical feasibility. The target workload is 5G New Radio's (and beyond's) baseband receiving physical layer processing, Physical Uplink Shared Channel (PUSCH), whose most compute-intensive steps with tight latency and throughput requirements, motivated the search for providing tera-operations-per-second and massive parallelism with very high energy efficiency.

The project was two years long and addressed the following Work Packages (WPs):

1. WP1-Y1: Exploration of the MemPool architecture and evaluation of scale-up opportunities. This is addressed in the Section 2 and 3 of this document.

2. WP2-Y2: Design of a benchmark suite, centered on 5G New Radio to assess the performance of MemPool-based clusters. The design of the software stack for TeraPool and the benchmark suite developed during the project is reported in sections 5 and 6.

3. WP1-Y2: Finalization of the TeraPool 1024-cores cluster RTL and physical implementation. Details on the core complex extensions added to support 5G New Radio and on the final architecture are in Section 3. A full physical implementation of the cluster in an advanced technology is described in Section 4.

4. WP2-Y2 & WP3-Y2: Assessment of the performance of 5G New Radio on the proposed architecture and further expansion of the RTL to adapt it to multi-cluster parallel execution. This last milestone follows the performance analysis of TeraPool on the selected benchmarks. It is motivated by the desire to further scale up the computing platform with a multi-cluster approach, instead of increasing the dimensions of the single cluster. The extensions to the hardware for multi-cluster execution, consisting of a DMA engine and a high bandwidth link to DRAM memory are discussed in Section 3.

The output of the project is TeraPool, an open-sourced, general-purpose programmable architecture featuring 1024 latency-tolerant 32-bit RISC-V cores that share 4 MiB of L1 memory via an ultra-low latency interconnect (7-11 cycles). A modular Direct Memory Access (DMA) engine efficiently links to high bandwidth memory, such as HBM2E (98% peak bandwidth at 910 GBps). A single TeraPool Cluster achieved a floating-point data symbol compute latency of only 1.61 ms per Transition Time Interval (TTI) for PUSCH on a ultra-high demand use-case. Average power consumption is less than 6 W. Data transfer overhead is less than 6%. This demonstrates that TeraPool is suitable as an accelerator for next-generation Software Defined Radio (SDR) baseband processing (three clusters are needed in the high-demand scenario to ensure the required 5th Generation (5G) PUSCH throughput).

The scientific publications related to the project are listed in Section 8.1, and all design source code for hardware Register Transfer Level (RTL) and software are open-sourced with more information in Section 8.2.

2 Introduction

Wireless baseband processing is one of the key application domains for data-intensive computing: the Radio Access Networks (RAN) market is huge, and the requirements in terms of computational throughput and energy efficiency are staggering. The 5G standard introduced novel features in its air-interface, such as larger bandwidths, higher spectrum frequencies, increased massive multi-user Multiple-Input, Multiple-Output (MIMO), Beam Forming (BF), etc. Focusing on the baseband processing chain, the PUSCH massive-MIMO at the lower physical layer presents the most compute-intensive demands, requiring the processing of high-dimensional signals in just a fraction of milliseconds[1]. This scenario is particularly suited to leveraging the embarrassingly parallel features of modern computing architectures.

To guarantee time-to-market and performance, the industry is moving from architectures where network functions are offloaded to specialized accelerators to an open and disaggregated SDR paradigm [2], building processing pipelines as a chain of programmable components. Off-the-shelf products include programmable [3, 4, 5] or reconfigurable [6] hardware with their in-house software libraries. Their designs target 10 Gbps uplink bandwidth at less than 50 W power consumption [3, 4], on Fast Fourier Transform (FFT) for Orthogonal Frequency Division Multiplexing (OFDM), Matrix Multiplication (MatMul) for BF, Channel Estimation (CHE), and Linear System Inversion (SysInv) for MIMO detection: the big heterogeneous workloads of the 7.X Open Radio Access Networks (O-RAN) functional splits [7]. However, closed proprietary does not allow open research, community-based hardware-software co-design, a propulsive force for SDRs. The RISC-V Instruction Set Architecture (ISA) is strategically significant in this context, as it enables open software and hardware architectures and co-designs, free from the limitations of proprietary instruction sets.

Focusing on programmable solutions, the end of Dennard's scaling made resorting to a single complex processor running at high frequency impractical due to power consumption and heat dissipation limitations [8]. The design focus shifted to multi-/many-core general-purpose systems, which achieve high performance and energy efficiency through parallel processing [9]. A successful architectural pattern is a shared-L1 cluster that eliminates the hardware and software overheads incurred by replicating a private-L1-base many-core cluster for performance purposes [10, 11] (synchronization, inter-cluster data allocation-splitting, and workload distribution among clusters). This architecture is beneficial for baseband receiving processing steps such as OFDM and BF, which are highly dependent and compute-intensive, requiring a large-scale cluster to minimize data swapping and transferring overhead from higher-level memory hierarchies. Increasing the scale of the shared-L1 many-core cluster is highly desirable to exploit the embarrassingly parallel features of SDR processing.

This project focuses on developing a physically feasible, open-source computing cluster architecture at the thousand-core scale, featuring fully shared L1 memory. We introduce TeraPool, a peak 1.89 TOPS cluster, featuring 1024 RISC-V cores with hierarchical low-latency interconnections to a fully shared 4 MiB, 4096-banked L1 Scratchpad Memory (SPM). We focus on the PUSCH lower Physical Layer (PHY) of the uplink receiver at the Next Generation Node B (gNB), as it represents one of the most challenging processing parts of the entire receiving chain. We aim to achieve single-cluster performance for RAN baseband processing on the order of TOPS, further augmenting performance with a multi-cluster architecture supported by efficient hardware and software co-design. The contributions are:

- A physical-aware architecture design for O-RAN workloads, including both fixed-point and floating-point support, based on a three-level physical implementation hierarchy: the Tile, the SubGroup, and the Group; and a detailed latency-throughput evaluation of on-chip Non-Uniform Memory Access (NUMA) latency interconnections.

- The system-level Advanced eXtensible Interface (AXI) interconnection and hierarchical

DMA engine design, and the open-sourced DRAMsys co-simulation with L2-to-Cluster transferring performance analysis.

- A complete physical implementation and Power, Performance and Area (PPA) analysis of each design configuration, using the cutting-edge 12 nm FinFET technology node.
- A synchronization strategy for large-scale many-core systems, and the implementation of a central counter barrier and a k-ary tree synchronization barrier for *TeraPool*, exploiting hardware support to trigger the wakeup of all the Processing Elements (PEs) in the cluster or a fraction of them.
- A comprehensive performance and energy efficiency evaluation on key 5G-SDR kernels, introducing the local memory access parallel implementation of these key kernels to reduce memory-related stalls.

3 TeraPool Architecture

While it was demonstrated that shared-L1 clusters with up to 256 cores can be built [12], their performance is still insufficient for low-latency TTI 5G pipelines. To meet real-life O-RAN requirements, TeraPool needs to "enter into uncharted territories" in terms of core count, aiming at a 4x increase over the largest cluster reported in the literature. TeraPool is a highly flexible and parametric many-core architecture featuring 1024 RISC-V cores. It is designed with hierarchical low-latency interconnections to a fully shared 4 MiB, 4096-banked L1 SPM. We must aggressively leverage physical design awareness, building the cluster hierarchically to ensure high PEs-L1 bandwidth and energy efficiency.

3.1 Processing Elements

3.1.1 Snitch Core

TeraPool's PEs are single-stage, 32 bit RISC-V *Snitch* cores [13]. Its small hardware cost, latency tolerance, and extensibility make Snitch the ideal candidate to implement an application-tunable shared-memory many-core design. Specifically, we use a Snitch version supports the RV32IMAXpulpimg ISA, includes an Integer Processing Unit (IPU), implementing integer multiplication and division, and the Single Instruction Multiple Data (SIMD) *Xpulpimg* extensions [1], friendly to complex fixed-point 16b data types, which are typical of radio signal processing.

The instructions in the RISCV32I base ISA are decoded and executed inside the core, while multi-cycle instructions are offloaded to specialized functional units. The Snitch's Load Store Unit (LSU) contains a scoreboard that keeps track of outstanding memory accesses and allows Snitch to proceed with instructions as long as there are no read-after-write (RAW) hazards. To tolerate multi-cycle L1 access latency, the LSU supports multiple outstanding memory requests, allowing Snitch to issue a series of loads and stores without waiting for the response [12]. Given TeraPool's NUMA interconnection design, Snitch's scoreboard retires loads out-of-order while guaranteeing in-order delivery to the execution units. The number of supported outstanding transactions is parameterizable and can be tuned depending on the maximum memory access latency within the Cluster's L1 memory.

Snitch features an accelerator port to offload complex instructions to pipelined functional units. The architecture view of Snitch core with accelerator port shown in Figure 1. In the following section, we will introduce floating-point support through the implementation of the configurable Floating Point Unit (FPU) module, *fpnew*, via the accelerator port of Snitch, which facilitates the offloading of complex instructions to pipelined functional units.

3.1.2 Floating Point Extension

For enhanced arithmetic precision, we introduced floating-point support, specifically targeting telecommunications workloads with 32 bit and 16 bit data types. Due to the stringent area constraints in a many-core compute cluster, which requires minimizing the compute units' footprint, we decided on the implementation of the *zfinx* and *zhinx* extensions. This solution eliminates the need for a separate floating-point register and floating-point load and store unit in the standard Snitch configuration.

The Floating Point Sub-System (FP-SS) utilizes the same functional-unit port as the IPU. It includes a decoding stage and the configurable FPU module, *fpnew*. Floating point instructions are first decoded in Snitch. Next, they are offloaded to the floating point subsystem, where they encounter a second decoding level, which is used to reorder the operands and properly feed the *fpnew* module.

[1]The Xpulpimg extension includes domain-specific Digital Signal Processing (DSP) instructions like Multiply Accumulate (MAC) and load-post-increment, which are offloaded through the accelerator port to an IPU [14].

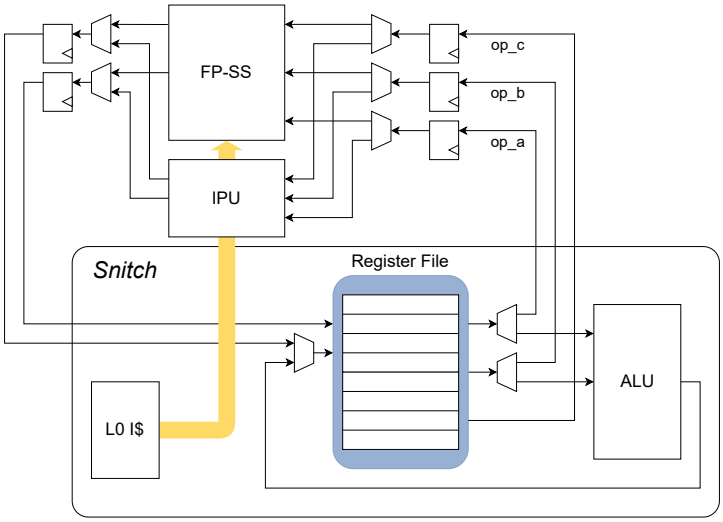


Figure 1: Snitch core: The processing element for TeraPool

To cut the timing path from Snitch to the functional units we included buffer registers on the operand offloading path. Additionally, the *fpnew* module contains a configurable number of pipeline stages on its datapath. The position of the pipeline stages is fixed in RTL, but the design relies on the backend tool retiming features to move them appropriately during the backend phase.

The architecture of *fpnew* and the flow of complex multiplication execution with Sum Dot Product (SDOTP) are depicted in Figure 2.

The SDOTP block within our *fpnew* architecture is designed to support SIMD dot-product extensions. Operands are 32 bit vectors composed of 16 bit sub-words, denoted as $[[31:16],[15:0]]$. For two input vectors, $\mathbf{a} = [a, b]$ and $\mathbf{c} = [c, d]$, the SDOTP block computes the accumulation $x = x + ac + bd$. This operation can be expressed as:

$$x_{\text{out}} = x_{\text{in}} + (a \times c) + (b \times d) \tag{1}$$

Variants of this computation are supported to accommodate diverse requirements, such as different bit-widths for the output x (either 32 bit or 16 bit), and modifications to the sign of the accumulation or the product, all configurable via specific instructions. To extend the functionality from real to complex operations, we instantiate two SDOTP slices. One slice computes the real component, while the other focuses on the imaginary component of the output. By employing operand multiplexing techniques, we provide the slices with inputs $[a, b]$ and $[c, -d]$, as well as $[b, a]$ and $[c, d]$. Consequently, the block outputs two 16-bit results:

$$x = x + (a \times c) - (b \times d) \tag{2}$$
$$y = y + (b \times c) + (a \times d) \tag{3}$$

These outputs are then packed into a single 32 bit word and relayed back to the register file in the Snitch core. The architecture also supports variants to implement operations like complex conjugate multiplication, further enhancing the versatility of the SDOTP block in handling complex arithmetic.

3.2 Tile and Interconnection

3.2.1 Computing Tile

The Cluster is constructed hierarchically to ensure the feasibility of TeraPool's physical design. The basic building block of TeraPool is the *Tile* shown in Figure 3, which is designed for

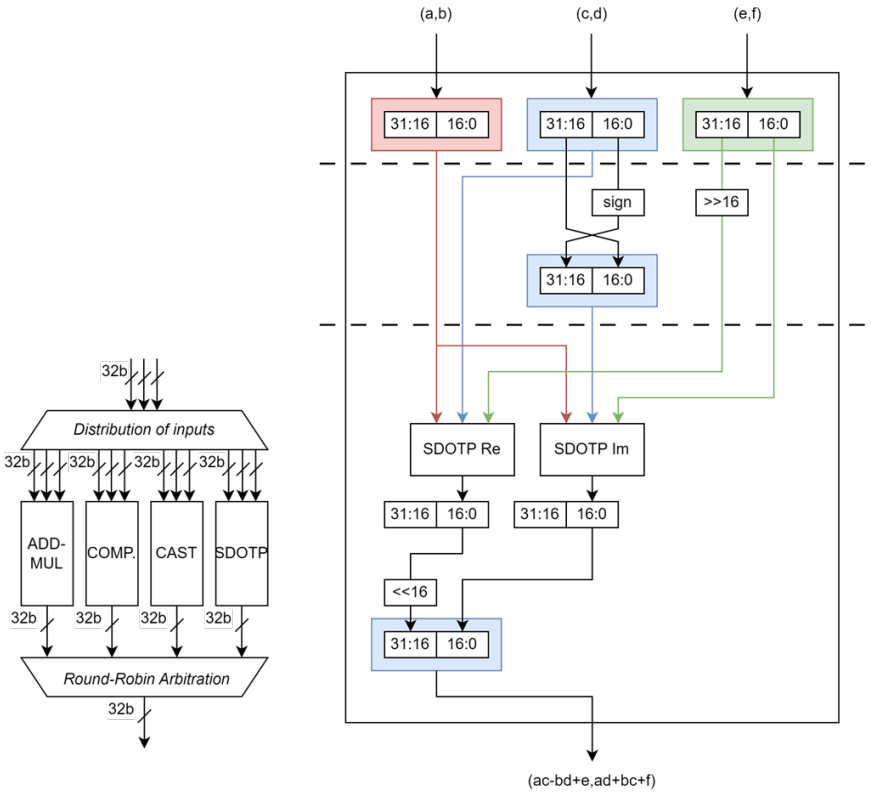


Figure 2: FPnew architecture and complex multiplication

massive replication. It contains 8 Snitch PEs, each equipped with a small private L0 Instruction Cache (I$) of 32 instructions. The PEs share a 4 KiB two-way set-associative L1 I$. The AXI master port, used for L1 I$ refill, and DMA transfers, for data-intensive applications, is shared between all the cores. On the key data memory side, PEs are tightly coupled to 32 1 KiB Static Random-Access Memory (SRAM) banks of L1 data memory. The interconnection architecture for core to local Tile and remote Tile memory access will be described in the following section.

3.2.2 Interconnection Module

Implementing a hierarchical topology is critical for realizing a functional interconnection network between 1024 PEs and 4096 memory banks. To maintain low-latency Tightly Coupled Data Memory (TCDM) access, we have implemented fully combinational routing through logarithmic crossbars and arbitrators at each design level, the detail design for the interconnection node refers to [15]. TeraPool's NUMA architecture makes it necessary to route and arbitrate requests and responses independently. Thus, we have separate networks for handling NUMA accesses: one for requests (address, data write, and control) and another for responses (request ID, data read, and acknowledgment). A round-robin strategy is employed for managing simultaneous requests at the same arbitration switch, ensuring complete arbitration for both request and response paths.

Cores within a Tile are connected to this local portion of the Cluster's shared SPM via a fully connected crossbar, which provides single-cycle access latency. Each Tile in the design is parameterized to support a specific number of remote ports, which handle memory requests targeting the banks of the shared SPM in other Tiles. Requests outgoing from a Tile are routed to the appropriate port via the *remote request interconnect*, and the corresponding responses are directed back to the originating core through the *remote response interconnect*. Incoming

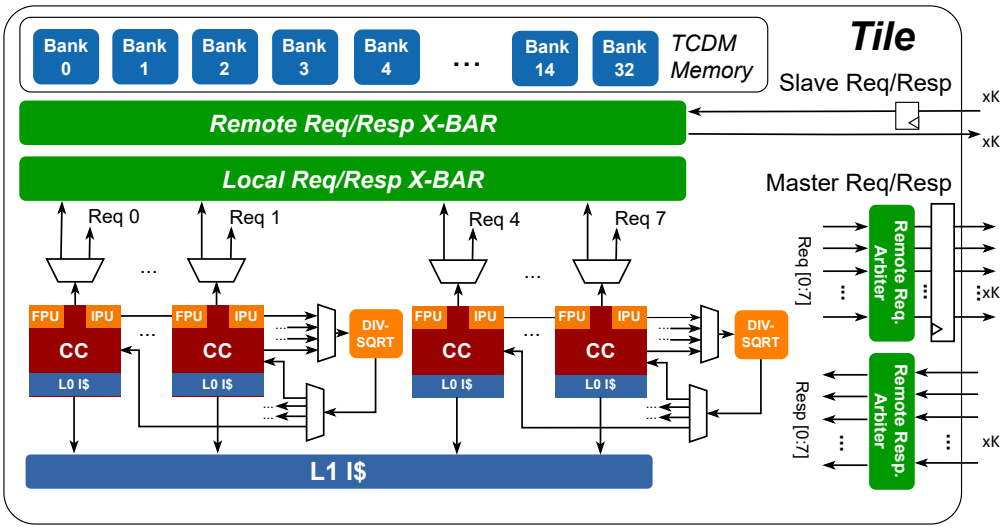


Figure 3: Overview of the TeraPool Tile architecture with ports factor K. The division and square root units (DIVSQRT) are optional, and the number of units can be parameterized and shared across the number of cores.

requests are directly linked to the fully-connected local crossbar within the Tile to access the SRAM banks. This local interconnect is designed with a dedicated master port for each of the K incoming request ports, forming an $(8 + K)\times32$ fully-connected crossbar. To optimize for physical design constraints, an optional buffer register can be introduced on both incoming and outgoing master ports. While this may increase latency, it effectively reduces the critical paths in the physical design, thus enhancing design operating frequency.

3.2.3 Shared Division Square-Root Unit

Considering the high implementation cost of the Division and Square-Root Unit (DIVSQRT), and its limited use in CHE and Minimum Mean Squared Error (MMSE) processing steps, a shared DIVSQRT for each Tile was tested. This required arbitration at the Tile level and hart-id propagation through the DIVSQRT module. The number of shared units per Tile is configurable, the Tile view architecture shown in Figure 3.

3.3 Physically Feasible Hierarchical Interconnection Design

The TeraPool Cluster consists of 128 Tiles and thus contains 1024 Snitch cores and 4096 SRAM banks with a capacity of 1 KiB each, totaling 4 MiB of SPM. The Tiles can be arranged in different hierarchical configurations. Choosing the hierarchies and the placement of the interconnects within, strongly conditions the design feasibility. For example, a single upper hierarchy with 128 Tiles connected by a 128×128 crossbar is clearly unfeasible. Grouping 16 Tiles results in $K = 8$ ports per Tile (one for each Tile-Group) and delivers high inter-Tile interconnect bandwidth, which the Tile interface shown in Figure 4 (a). However, it requires eight 16×16 Fully-Connected (FC) crossbars within the hierarchy level, and leads to unmanageable routing congestion. Moreover, it is impossible to place the 8 Tile-Groups in a grid that results in balanced, short access paths and the Electronic Design Automation (EDA) runtime for the Tile-Group design iteration is unmanageable.

To make such a huge cluster physically feasible, we propose the following design strategies for the TeraPool architecture:

1. The topmost hierarchy, the Group, is replicated 4 times and arranged in a 2×2 grid to both shorten and balance the diagonal access paths between the Groups.

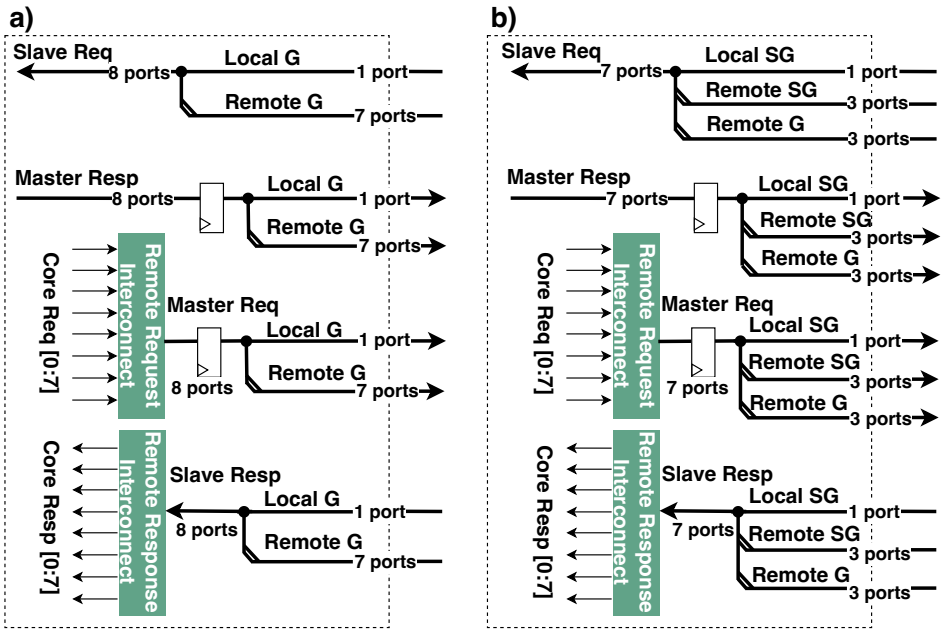


Figure 4: Tile's interfaces in different design configurations. Left side *a)* is Tile's interface of grouping 16 Tiles with 8 Groups, which delivers high inter-Tile interconnect bandwidth but is not physically feasible; Right side *b)* is for $TeraPool_{1\text{-}3\text{-}5\text{-}X}$ Tile's interface.

2. To keep the EDA tool's runtime manageable, a Group is divided into multiple finely-tuned physical implementation hierarchies. That is, each Group consists of 4 SubGroups, and each SubGroup contains 8 Tiles.

3. Targeting high inter-Tile bandwidth, we forward Tiles' remote Group requests directly to the Group level, where we implement 32×32 FC crossbars for each target Group.

We provide a summary of our physically feasible interconnection design, which hierarchically connects all 1024 cores with 4096 banks while maintaining low latency and high bandwidth. The cluster is also represented in Figure 5.

1. **Tile Design**: Each Tile consists of 8 Snitch cores and 32 1 KiB SPM banks, with cores accessing local Tile banks with only 1 cycle zero-load latency. Each Tile has 7 ports for remote connections. The Tile's interface is depicted in 4 (b).

   - Port[0] for local SubGroup[0].
   - Ports[1–3] for remote SubGroups[1–3] within its own Group[0].
   - Ports[4–7] for remote Groups[1–3].

2. **SubGroup Design**: Each SubGroup consists of 8 Tiles. Additionally, in a SubGroup, we have interconnects to connect to Tiles in other SubGroups of the same Group:

   - The 8×8 *local SubGroup Crossbar* connects all Tiles within the same SubGroup, in 3 cycles zero-load latency. It is accessed with the Tile' port[0].
   - Three 8×8 *Remote SubGroup Crossbars* connect SubGroup-Tiles' ports[1–3] to corresponding SubGroups within the same Group, with 5 cycles zero-load latency.

3. **Group Design**: Each Group consists of 4 8 SubGroups. In a Group, we also place three 32×32 *Remote Group Crossbars* to connect its Tiles with the Tiles of other Groups. Group-Tiles' ports[4–6] are connected to each one of these crossbars. The zero-load latency for remote Group access is hardware parameterizable (7 to 11 cycles).

4. **Conflict Situation**: In our hierarchical interconnection design, conflicts occur when cores in the same Tile attempt to access the same SubGroup/Group in the same cycle (port conflict) or cores access the same banks in the same cycle (bank conflict).

To establish local connections among Tiles in a SubGroup, instantiate a local-access 8×8 FC crossbar. To connect the Tiles of a SubGroup with the Tile of the other three Subgroups we also instantiate other three 8×8 FC crossbars for remote access. Dividing the interconnect between tiles in a Group over the four SubGroups instead of having a larger 32×32 FC crossbar improves the routing.

The hierarchy levels provide flexibility in placing pipeline registers and breaking long remote access paths. Within the Group, registers are placed at each hierarchy boundary on master ports. The latency between Groups is a hardware-parameter that trades off the target operating frequency and latency. We call those parametrizations **TeraPool$_{1\text{-}3\text{-}5\text{-}X}$** architecture, where the subscripts indicate the zero-load cycle latency for core access at each hierarchy level, i.e., Tile, SubGroup, Group, Cluster. The **TeraPool$_{1\text{-}3\text{-}5\text{-}5}$** design has no extra registers between the two Groups. A register is added on both request and response paths between Groups at the Cluster level in the **TeraPool$_{1\text{-}3\text{-}5\text{-}7}$** design, increasing the round-trip latency by 2 cycles. In **TeraPool$_{1\text{-}3\text{-}5\text{-}9}$** and **TeraPool$_{1\text{-}3\text{-}5\text{-}11}$**, additional buffer registers are added respectively to slave or both master-slave ports of the Group hierarchy level. The maximum access latency increases by 4 and 6 cycles. As an example, the full architecture overview of TeraPool$_{1\text{-}3\text{-}5\text{-}7}$ is shown in Figure 5. These configurations create a trade-off between achievable frequency and L1 worst-case latency.

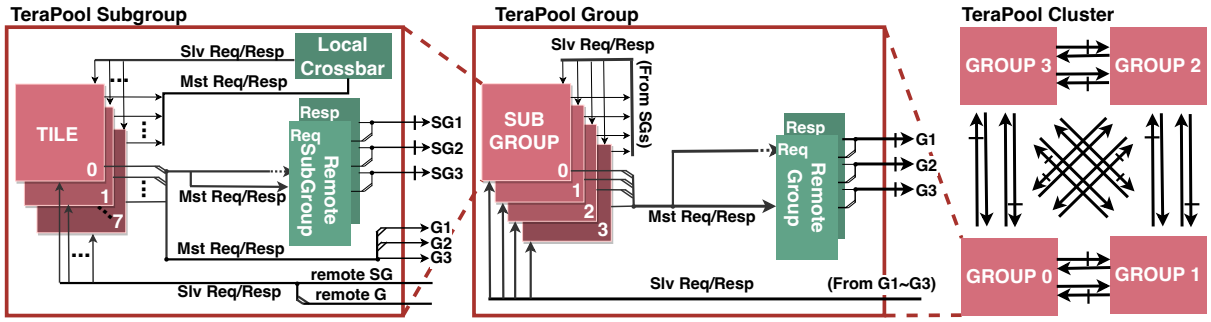


Figure 5: Bottom-up architecture overview of TeraPool$_{1\text{-}3\text{-}5\text{-}7}$.

3.4 System level design

In this section, we introduce the system-level design of TeraPool. This is represented in Figure 6. On the top left, we represent the cluster, which was discussed in the previous paragraphs. We further include the AXI connection to L2 memory, and to the cluster peripherals, the cluster Control Status Registers (CSRs), and the DMA.

To achieve system connectivity, we use AXI ports to access the L2 (or main memory) and peripherals from the TeraPool Cluster. TeraPool requires a few CSRs for its runtime, which enables core wake-up and holds system parameters, such as core count and I$ configuration. With the Tile's I$ optimized and the cores granted access to a shared L1 data memory, it is crucial to ensure that the I$ can be refilled and that data can be transferred to and from the system memory efficiently. Providing each of the 128 Tiles with a private high-bandwidth connection to the system presents a significant challenge in a design already dominated by routing constraints.

This section addresses the challenge of connecting all 128 Tiles' AXI ports to the system bus while minimizing area and routing overhead. Specifically, we aim to extend the I$ hierarchy to continuously support the cores' instruction stream from L2 or external memory. Additionally,

we must design a specialized DMA that facilitates data transfers between TeraPool's distributed L1 SPM and L2 or external memory, ensuring minimal overhead and preventing congestion on the cores' interconnects.

3.4.1 AXI Interconnection

The instruction and data paths are latency-tolerant and exploit spatial locality by operating on contiguous memory chunks. Consequently, we utilize the open and standard AXI protocol to connect the Tiles to the system-level components. This protocol is burst-based, supports multiple initiators and targets, and features decoupled read and write channels, making it suitable for our requirements. Each Tile has an AXI master port that is shared among all cores and the I$ refill. The AXI master grants access to higher-level/main memory, peripherals, and CSRs in the system-level design. However, routing a private AXI bus for each Tile to the cluster boundary is not physically feasible. To address this challenge, we have designed the AXI interconnect using a hierarchical approach, as illustrated in 6.

In this configuration, each Tile and DMA acts as a leaf node in a configurable AXI tree. At each interconnect level, neighboring child nodes converge into a single AXI bus, resulting in each SubGroup having one 512 bit AXI master. At the top level, we maintain a configurable number of master ports to facilitate high bandwidth, effectively creating multiple AXI trees, each including a subset of the Tiles and DMAs. By default, the TeraPool Cluster is equipped with 16 512 bit AXI masters, which are aggregated from the SubGroups' AXI masters.

3.4.2 DMA Engine

To efficiently move data to and from L1 memory, we utilize a DMA engine that leverages the AXI interconnect and accesses all 4096 SPM banks. Given TeraPool's distributed L1 SPM and large scale, a traditional DMA engine is not viable. Although TeraPool's L1 memory is shared, allowing a single DMA to access the entire memory simplifies programming but connecting one DMA to all 4096 banks would require excessive routing. A potential solution is to repurpose the L1 interconnect; however, it is not suited for wide data transfers and could significantly congest the interconnect, thereby slowing down core transactions. In contrast, the AXI bus that connects all tiles is ideally suited for DMA-type transfers.

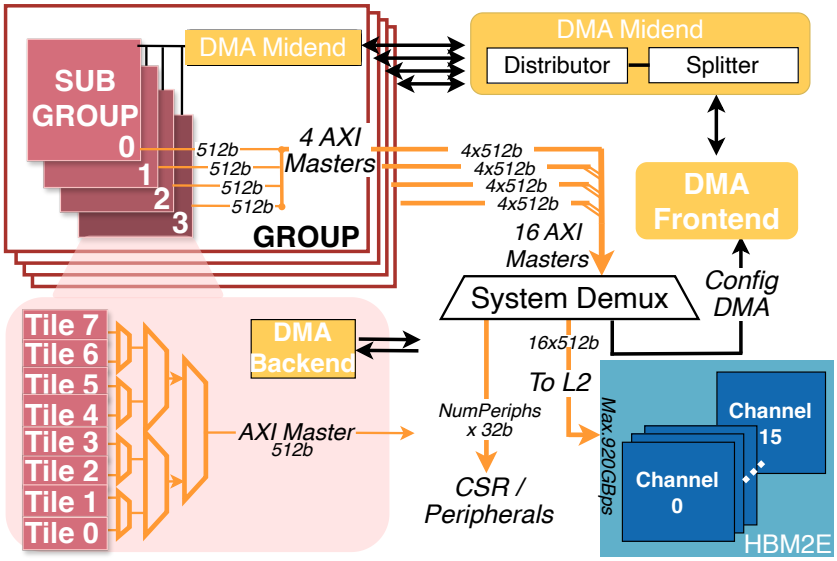


Figure 6: A modular DMA engine distributed for TeraPool Cluster.

Using a modular DMA engine [16], we have designed a distributed DMA for TeraPool as illustrated in Figure 6. This system comprises a configurable number of data movers, referred to as backends (e.g., one per eight tiles), which manage their respective tiles' memory regions. Each backend connects to the tiles' AXI port on one side and to a fully connected local crossbar on the other. A single configuration frontend oversees all backends. Programmers can initiate DMA transfers across the full TeraPool cluster; the DMA control then divides the transfer into multiple smaller segments by the midend, and coordinates the actions of the data movers. We implement two crucial modules, a splitter and a distributor, to facilitate this process. The splitter divides a transfer at the L1 memory's address boundary into several serial requests, considering the interleaved addressing scheme. The distributor then allocates these requests across multiple data movers requiring access to different L1 memory regions. This design optimally utilizes the hierarchical AXI interconnect to minimize additional routing and connects the DMAs to the tiles' internal, fully connected crossbars, where bandwidth is high. The impact on core operations is minimized.

In the TeraPool architecture, data is interleaved across L1 memory banks to optimize access efficiency. Specifically, within each SubGroup, which contains 256 SPM banks, data is arranged such that 256 words per row are interleaved, with each bank holding one 32 bit word. Each SubGroup possesses a dedicated AXI port that manages the transfer of consecutive L1 memory addresses to and from higher memory hierarchies. As a result, the maximum transfer capacity controlled by the DMA via each AXI port amounts to 256 continuous words per AXI burst.

To better understand the transfer mechanics, consider the following example which details the hardware parameters related to the AXI Port Width and Masters: Each SubGroup is equipped with one 512-bit AXI Master port, and the entire TeraPool Cluster comprises 16 AXI Masters. Regarding the AXI burst length:

1. The AXI burst length is directly linked to the number of DMA backends allocated to each SubGroup. The formula for determining the AXI burst length is:

$$\text{AXIBurstLen} = \frac{\text{MaxTransferBitsPerReq}}{\text{Nr.DMABackendPerSG} \times \text{AXIWidth}}$$

2. For example, with one DMA Backend assigned per SubGroup, this configuration results in 16 bursts per AXI request.

$$\text{AXIBurstLen} = \frac{256 \times 32}{1 \times 512} = 16$$

3.4.3 DRAMsys Co-Simulation

To facilitate the co-simulation of data transfer performance with high-level memory hierarchies, a time-efficient, cycle-accurate Dynamic Random-Access Memory (DRAM) subsystem is essential for the TeraPool Cluster co-simulation. The *DRAMSys5.0* [17], an open-source, flexible framework for DRAM subsystem design space exploration meets our requirements [2]. Developed by the Microelectronic Systems Design Research Group at RPTU Kaiserslautern-Landau, *DRAMSys5.0* is based on SystemC TLM-2.0 and enables cycle-accurate simulation of various memory technologies, including DDR3/4, LPDDR4, Wide I/O 1/2, GDDR5/5X/6, and HBM1/2. For our implementation, we engineered an RTL interface for the *DRAMSys5.0* subsystem, which we integrated with the TeraPool Cluster via the AXI interconnect.

Integrating *DRAMSys5.0*, developed using object-oriented C++ within the SystemC framework, necessitates the conversion of the Cluster RTL into a SystemC-based model to enable the co-simulation. Initially, we utilized Verilator [3], a hardware compiler and simulator, to convert

[2]The DRAMSys5.0 open-sourced code repository: `https://github.com/tukl-msd/DRAMSys`, released under the BSD 3-Clause License from their information.

[3]The free Verilog/SystemVerilog simulator software, more information: `https://www.veripool.org/verilator/`

the TeraPool's SystemVerilog RTL sources into an optimized SystemC model. However, we encountered significant development runtime challenges: each hardware tuning process in the TeraPool's RTL necessitated recompiling the entire TeraPool Cluster into a SystemC model. This recompilation process is extremely time-consuming and has become a major bottleneck in our development efficiency.

To mitigate the challenge of lengthy recompilation times and enhance our design process efficiency, we transition the *DRAMSys5.0* subsystem model into a *Dynamic Link Library* (*.so* file format), which is subsequently integrated into the ModelSim [4] library list. The DRAM controller remains configurable via a configuration (*JSON*) file, facilitating adjustments to system settings such as bank address mapping, PHY delay, and controller strategy during the simulation setup phase. These modifications are depicted in the Figure 7.

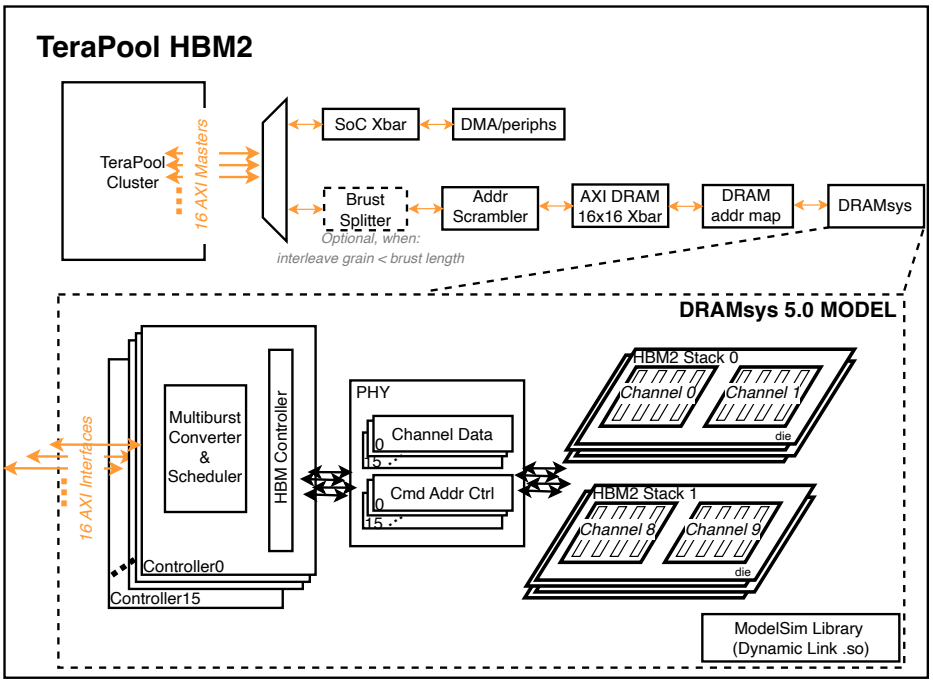


Figure 7: DRAMsys5.0 subsystem two stacks of HBM2 implementation with TeraPool Cluster.

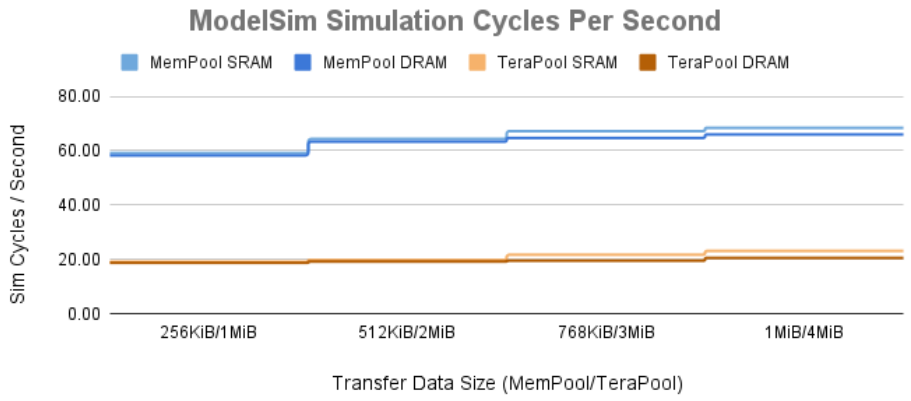


Figure 8: Simulation speed of SRAM and DRAM comparing for both MemPool and TeraPool Cluster by ModelSim.

[4]ModelSim simulates behavioral, RTL, and gate-level code - delivering increased design quality and debug productivity with platform-independent compile. See more: `https://eda.sw.siemens.com/en-US/ic/modelsim/`

We conducted a runtime analysis for real-time simulations on both MemPool and TeraPool architectures, comparing them with their respective SRAM simulations. The results, as depicted in Figure 8, demonstrate that the DRAM co-simulation speeds for both MemPool and TeraPool are closely comparable to those of the SRAM simulations, with data transfers involving the entire Cluster L1 size being only 5% slower.

We have open-sourced the support for system-level hardware-DRAM co-simulations at the RTL level, utilizing *DRAMSys5.0* for simulating DRAM and controller models. This resource is available under the Solderpad Hardware License 0.51 and the original *DRAMSys5.0* licenses [5].

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

4 Physical implementation

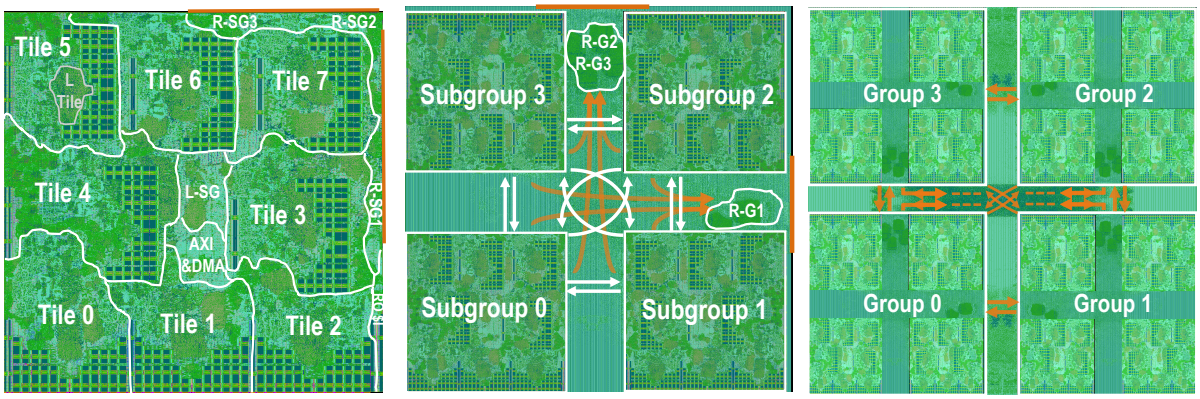
This section explores the physical feasibility and efficiency of multiple TeraPool Cluster configurations, including system-level hierarchical AXI and DMA support.

4.1 Methodology

We describe the implementation methodology and discuss TeraPool's PPA at the post-place-and-route stage. We implement the TeraPool Cluster using GlobalFoundries' 12LPPLUS FinFET technology. TeraPool's large scale and complex interconnection structure require a hierarchical implementation flow. We use Synopsys' Fusion Compiler 2022.03 to synthesize, place, and route each hierarchy level and assemble blocks in a bottom-up fashion. To keep the EDA tools' runtime manageable, the abstract design view of lower hierarchical blocks, which contains the required interface logic and register timing information of the block, is created during bottom-up assembling. Keeping the hardware-parameterizable latency for remote Group access between 7 and 11 cycles, the achieved operating frequencies range from 730 MHz to 920 MHz in typical operating conditions (TT/0.80 V/25 °C). We determine the power consumption of TeraPool under typical operating conditions (TT/0.80 V/25 °C) using Synopsys' PrimeTime 2022.03 and referring to switching activities obtained from a post-layout gate-level simulation with back-annotated delay information.

4.2 Floorplan

Routing congestion presents a fundamental challenge for the large-scale, tightly interconnected TeraPool design implementation [18]. To fully leverage the available Back-End-of-Line (BEOL) resources, we implement TeraPool's Subgroup as the initial hierarchy, flattening the Tile-level design. This approach allows routing the interconnect through Tiles without manually placing ports. Additionally, buffer registers can be flexibly added at the hierarchy boundary to break long-distance interconnect paths. By tuning their placement for optimal timing, we can further reduce the overall implementation footprint.



TeraPool's Subgroup. TeraPool's Group. TeraPool's Cluster.

Figure 9: Placed-and-routed layout annotated view of each TeraPool hierarchical instance.

Fig. 9 left presents an annotated die shot of the fully placed and routed TeraPool's Subgroup. The Subgroup is implemented as a $1.52mm \times 1.52mm$ block, resulting in a satisfactory utilization of 58%. The interface ports for remote-Subgroups and remote-Groups are placed on the northeast boundaries. Each Subgroup comprises 8 Tiles, we put 3 of them on top, 2 in the middle, and other 3 on the bottom. The SPM macros of each Tile are grouped and placed in U-shape, to enclose the local crossbar. This further minimizes overall distance and avoids excessive stacking of macros [19]. The center and edges of the Subgroup are left free to place interconnect cells, and

the macros U-shape is rotated on the bottom side to better utilize the design area. Fig. 9 middle and right show TeraPool's Group and Cluster layouts with annotated interconnects. Subgroup and Group blocks are arranged in a point-symmetric grid, with channels in between to place and route the interconnects. To further improve area utilization, we place interface ports behind the Subgroup blocks and shrink the channel width until BEOL resources become limited.

4.3 Power, Performance and Area

Table 1 presents the post-place-and-route PPA results for the most promising TeraPool configurations and provides a comparison with the reference MemPool design [12]. TeraPool achieves energy efficiency greater or equal than a 4× smaller, 256 cores shared-1 MiB MemPool cluster, thanks to its low-power interconnect and a hierarchical interconnection architecture that allows adding TCDM buffer registers on the critical paths. The design therefore breaks the TOPS wall, achieving peak performance comprised between 1.50 TOPS and 1.89 TOPS, depending on the configuration.

Table 1: Post-Place and Route (PnR) peak-performance & area results.

	MemPool$_{256}$[(a)]	**TeraPool$_{1024}$**[(a)]		
Hier. Access Latency [cycle]	1-3-5	1-3-5-7	1-3-5-9	1-3-5-11
Area [mm^2]	10.0	68.9	68.9	68.9
Area Per Core [mm^2/core]	0.039	0.067	0.067	0.067
Logic Gate [MGE]	44	176	176	176
Logic Gate Per Core [MGE/core]	0.17	0.17	0.17	0.17
TCDM Buffer Register in Group	✓	✓	✓✓	✓✓✓
TCDM Buffer Register in Cluster	-	✓	✓	✓
Operating Frequency *(Worst)* [MHz]	728	530	637	740
Operating Frequency *(Typ.)* [MHz]	915	730	880	924
Logic Delay [FO4]	162.4	204.5	169.8	161.2
Max Throughput [req/core/cycle]	0.33	0.23	0.24	0.25
Avg. Latency *(Zero-load)* [cycle]	4.7	6.4	7.9	9.3
Peak Performance *(Typ.)* [TOPS]	0.47	1.50	1.80	1.89
Area Efficiency [GOPS/mm^2]	47.0	21.8	26.1	27.4

[(a)] Subscript indicates core count; implemented in GF12LP+ technology.

When the configured latency exceeds 11 cycles, the operating frequency is no longer constrained by the remote Group access path but by the Snitch core: the critical path, consisting of only 63 logic levels, starts at a register after the instruction cache, passes through Snitch and a request interconnection, and arrives at the clock gating of an SRAM bank. We normalized the critical paths delay using Fan-Out-of-4 (FO4), a technology-independent metric that refers to the delay of an inverter driving four identical inverters. Although a large-scale physical design leads to longer distance paths, TeraPool's low-latency scheme still keeps a small FO4 delay. For fair comparison, we keep 55 % area utilization on the base hierarchy of all the implementations. On the top hierarchy, TeraPool loses area efficiency due to the large 32×32 interconnection modules in each Group, requiring more routing area, and the difficult routing of feedthrough connections across Groups, caused by the blocking SubGroups placement. These challenges impose physical constraints on the further scaling of shared-L1-memory architectures beyond the 1000-cores milepost. We nevertheless believe that exploring 3D-IC solutions will be beneficial for future research. Given that the target operating frequency of TeraPool$_{1\text{-}3\text{-}5\text{-}5}$ is lower than 500MHz, and both TeraPool$_{1\text{-}3\text{-}5\text{-}13}$ and TeraPool$_{1\text{-}3\text{-}5\text{-}15}$ exhibit similar frequencies as TeraPool$_{1\text{-}3\text{-}5\text{-}11}$, the results analysis will not focus on these design configurations.

4.3.1 Area Breakdown

The Gate Equivalent (GE), defined as the area of a 2-input NAND gate, is a widely-used metric to represent the logic area in physical design, independently of the specific technology node. Figure 10 shows the hierarchical area distribution of TeraPool's Cluster. Our hierarchical interconnection contributes minimal area overhead, accounting for only 8.5% of the total design area. Within a Tile, the largest component is the SRAM, followed by the Snitch cores, split into the cores themselves and their IPU accelerators, then the instruction cache.

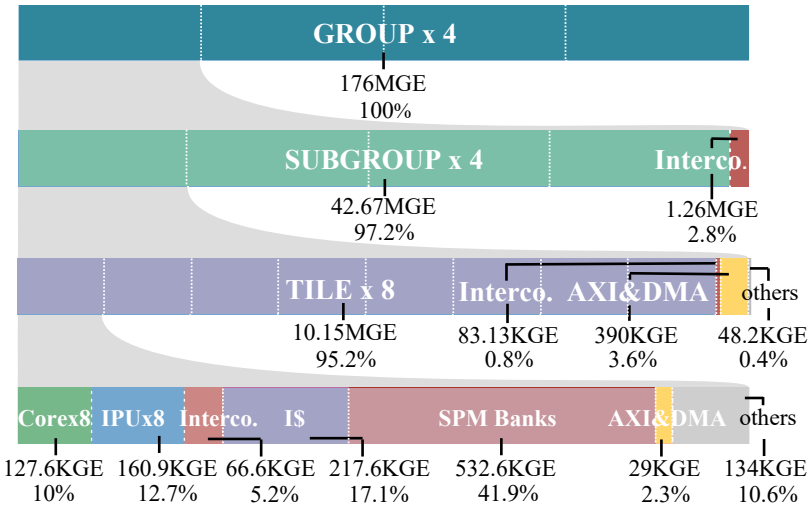


Figure 10: Hierarchical area breakdown of TeraPool Cluster with annotations showing the *percentage* of the immediate parent component.

4.3.2 Energy Consumption Breakdown

Given the power simulation methodology discussed, Figure 11 presents the energy breakdown (pJ/instruction/core) for TeraPool$_{1\text{-}3\text{-}5\text{-}7}$ when executing memory accesses and arithmetic instructions.

To illustrate the energy consumption of memory accesses in the hierarchical interconnection design, we show the energy consumption for a core that accesses memory banks in the local Tile, remote Tiles, remote Subgroups, and remote Groups. During memory accesses, the interconnect portion dominates energy consumption, accounting for 36% to 57% of the total, as access latency increases. The Snitch and SRAM banks consume only 2.57pJ and 1.33pJ, respectively. For arithmetic instructions, Snitch consumes 4.43pJ for Add (add), 7.96pJ for Multiply (mul), and 8.47pJ for Post-increment Multiply-accumulate (p.mac) operations. During the execution of arithmetic instructions, the interconnects consume 2.68pJ due to internal power from clock propagation and cell leakage. Thanks to SRAM clock gating, the SRAM banks' energy consumption is reduced by 75% and becomes negligible when they are not accessed.

The light-colored segments in the plot, following the Snitch and Interconnect, represent the energy consumption of optimization cells introduced by the EDA tools to achieve the frequency target. These segments have slight changes with the varying target design frequencies. Figure 12 illustrates a marginal increase in energy consumption as the operating frequency target rises for TeraPool Clusters simulated at their typical operating frequencies. The energy of remote Group memory access increases only 2.1 pJ when the typical operating frequency rises from 730 MHz to 920 MHz. Snitch dominates the energy consumption of arithmetic operations. As the target operating frequency approaches the gigahertz range, low-threshold optimization cells, which have minimal delay but higher leakage, are introduced in Snitch for timing optimization.

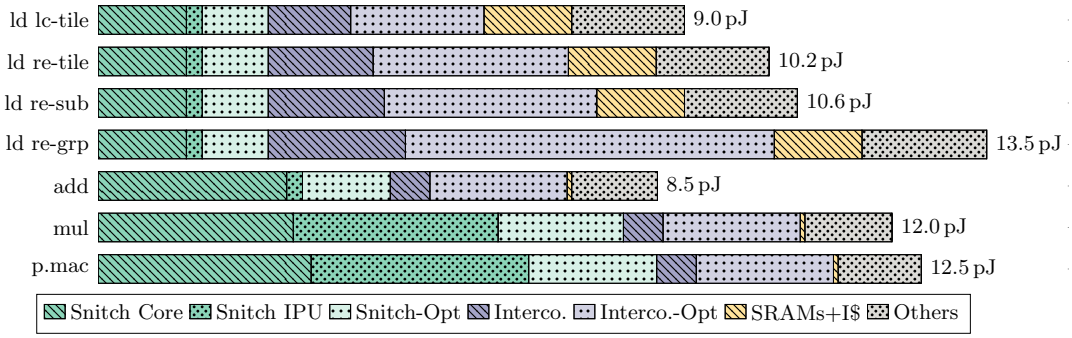


Figure 11: Breakdown of the TeraPool$_{1\text{-}3\text{-}5\text{-}7}$ Cluster's energy consumption per elementary operation in the instruction.

This leads to an average 20% increase in the energy consumption of arithmetic operations for TeraPool Clusters operating at 730 MHz to 920 MHz in typical conditions (TT/0.80 V/25 °C).

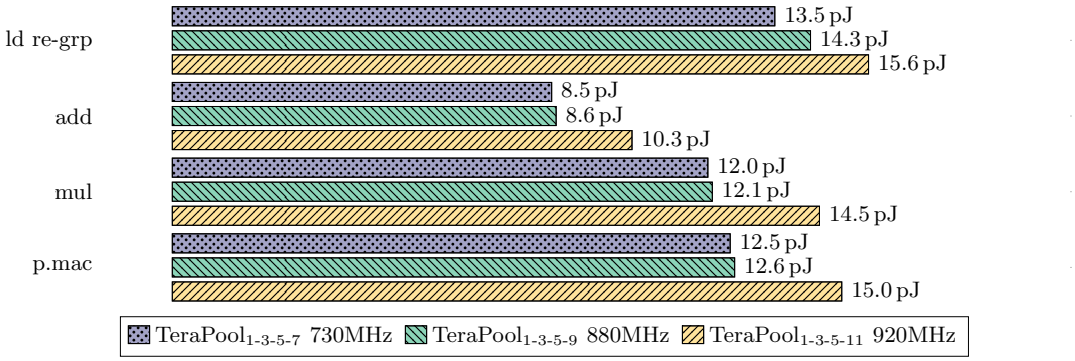


Figure 12: Energy consumption per elementary operation of TeraPool Clusters with annotations showing their typical operating frequency.

Table 2: TeraPool$_{1\text{-}3\text{-}5\text{-}9}$ Cluster Post-PnR peak-performance & area results.

	INT	**INT-FP**	**INT-FP-sDIVSQRT**
Cluster Area [mm^2]	68.9	81.8	81.8
SubGroup Area [mm^2]	2.30	3.01	3.01
Area Per Core [mm^2/core]	0.067	0.080	0.080
Logic Gate [MGE]	176	224	232.32
Logic Gate Per Core [MGE/core]	0.17	0.22	0.23
Operating Frequency *(Worst)* [MHz]	637	634	634
Operating Frequency *(Typ.)* [MHz]	880	875	875
Peak Performance *(Typ.)* [TOPS]	1.80	1.79	1.79
Peak Performance *(Typ.)* [TCFLOPS]	/	1.79	1.79
Area Efficiency [GOPS/mm^2]	26.1	21.9	21.9

4.4 FPU and Shared DIVSQRT

For the FPU-supported TeraPool design, implemented using the same GF12 technology node, we applied register retiming exclusively to the FP-SS as it is designed as a processing pipeline. The Cluster design is physically feasible with floating-point support, demonstrating a 19% increase in physical area and a 28% increase in logical area, along with a 1.52× increase in EDA design runtime.5 The critical paths, similar to those in the Integer TeraPool design, remain consistent,

with paths not originating from the FPU. However, the design frequency experienced a slight decrease due to the increased physical length of timing paths.

Design changes are confined to the Snitch Core, hence the Group/Cluster level architecture remains as in its integer counterpart. We compare the Subgroup level design performance and area incorporating all changes associated with the FPU and FPU with DIVSQRT. The SubGroup physical implementation results are shown in Table 2.

5 TeraPool Programming Model

A streamlined and efficient programming model is essential for our highly parallel computing platform. With programmability as a primary design goal, TeraPool has been developed to facilitate ease of use. TeraPool's shared memory architecture simplifies programming compared to systems with isolated memory regions, such as multi-cluster or systolic architectures. This shared memory enables all-to-all communication. Additionally, the independent and individually programmable cores in TeraPool circumvent common issues found in other architectures, such as lock-step execution, branch divergence, or thread warping. *TeraPool* has a fork-join (the abstraction at the base of Open-MP) programming model: sequential execution forks to a parallel section, where PEs access concurrently the shared memory. Barriers are used to synchronize and switch back to the sequential execution.

5.1 Compiler Support and Runtimes

We have enhanced the GNU GCC and LLVM toolchains to support TeraPool's ISA extensions and optimize instruction scheduling. Both toolchains now recognize TeraPool's architectural latencies and are engineered to schedule instructions strategically to mitigate RAW hazards. This includes arranging load operations to occur significantly before their use to hide the L1 latency effectively. Similarly, pipelined instructions offloaded to the Snitch accelerator are carefully scheduled. Such compiler-level instruction scheduling masks the non-ideal latency of the L1 interconnect and Snitch's accelerator, presenting TeraPool to the programmer as an idealized cluster with single-cycle latency.

TeraPool supports diverse programming models including bare-metal C, OpenMP, and Halide:

1. **Bare-metal C:** Programs all cores with identical code, allowing for branch creation or data assignment using unique core IDs. Programmers control core operations through manual stack allocation and runtime initialization, gaining access to runtime functions like memory allocator and barrier. This model is suitable for highly optimized, low-level application development.

2. **OpenMP:** Implements a shared-memory fork-join model in C. A master core manages execution, capable of engaging other cores via wake-up triggers and synchronization using RISC-V atomics. It supports static and dynamic loop scheduling, parallel sections, and synchronization directives (master, critical, atomic, barrier, and reductions). While OpenMP simplifies workload parallelization, it introduces additional runtime overhead.

3. **Halide:** A Domain-Specific Language (DSL) for image processing and machine learning that decouples application function from scheduling—like tiling and parallelizing—to enhance cross-platform code sharing and architectural optimization. We have enhanced Halide for TeraPool with RISC-V support in its LLVM backend, incorporating features such as fork/join functions for parallel execution and dynamic memory management for buffer creation, thus utilizing the LLVM backend's native scheduling capabilities.

5.2 Fast shared-memory barrier synchronization

The synchronization of independent threads is a fundamental aspect of the programming model of TeraPool. In this section, we describe the implementation of barrier primitives, more details can be found in [20].

The synchronization barrier algorithm can be divided into three phases: an arrival, a notification, and a re-initialization phase. For the arrival phase, we adopt a k-ary tree. The N_{PE} cores of the cluster are divided into N_{PE}/k groups of k PEs. In each group, synchronization

occurs in the form of a central counter barrier [21]. Each PE arriving at the barrier updates a shared counter via an atomic *fetch&add* operation and goes into a wait-for-interrupt (WFI) sleeping state. The last PE reaching the synchronization step, fetches a counter value equal to $k-1$ and continues with the next steps, where it is further synchronized with the other $N_{PE}/k-1$ PEs that survived the first step. The last step counts k PEs, and the very last PE arriving wakes up all the PEs in the cluster. The arrival tree works best when the $log_k(N_{PE})$ is an integer, but it is also adapted to the case where k is any power of $2 < N_{PE}$, by synchronizing a number of PEs different from the radix of the tree in the first step of the barrier. Varying k, we encounter two extremes, represented in Fig. 13 (a-b): the left shows a radix-2 logarithmic tree barrier, where each step only synchronizes pairs of PEs, the right illustrates the central-counter barrier. The re-initialization phase is implemented concurrently with the arrival phase, as each PE arriving last in a synchronization step also resets the shared barrier counter before proceeding to the next step.

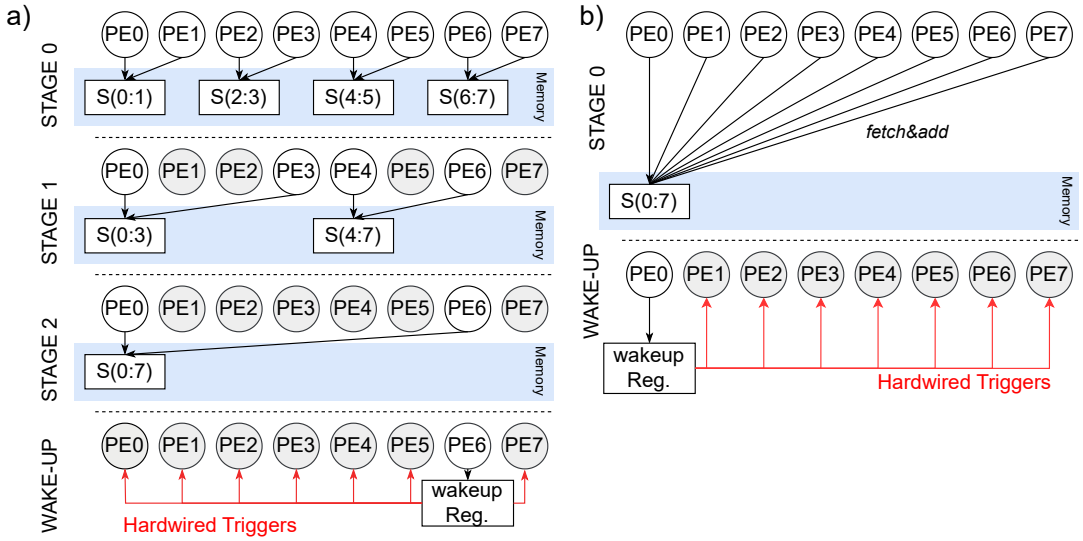


Figure 13: a) Binary tree for the arrival phase of the barrier. Couples of PEs synchronize by atomically accessing shared synchronization variables. b) Central counter barrier.

The notification phase leverages hardware support in the form of a centralized wakeup handling unit. The last PE arriving at the barrier and fetching from the shared barrier counter variable writes in a cluster shared register. The address of this register is in the cluster global address space and can be accessed by any core through the hierarchical AXI interconnections. The written value is detected by the wakeup handling logic that sends a wakeup signal to each individual PE, triggering N_{PE} hardwired wakeup lines. A software implementation of the wakeup mechanism is excluded. It would fall into the single master barrier class [22], whose cost scales linearly with N_{PE} and is unsuitable for synchronizing more than a few tens of PEs.

We support synchronizing a subset of PEs in the cluster modifying the wakeup handling unit by adding other memory-addressable shared registers. The core wakeup register is a 32-bit register that can be used to either trigger a wakeup signal to all the PEs in the cluster, when it is set to all ones, or to a single PE, by specifying its ID. One 8-bit register is used to selectively wake up Groups, and a register per Group is added to wake up Tiles in a Group selectively. A bitmask is used to determine the Groups or the Tiles to wake up. Depending on the bitmask written by a PE in one of the synchronization registers, the wakeup logic, asserts a subset of or all the wakeup triggers hardwired to the cores in the cluster, to trigger a wakeup signal.

The implemented barriers can be called from the function body through a custom software Application Programmable Interface (API). The radix of the barrier can be tuned through a single parameter, to ease trials and selection of the best synchronization option.

6 5G-PUSCH implementation

This section reviews the key kernels in PUSCH processing. PUSCH transmission is based on Orthogonal Frequency Division Multiple Access (OFDMA) [23]. User Equipments (UEs) transmitting in the same time resources are multiplexed in frequency. A frequency multiplexed transmission is called OFDM symbol and consists of N_{SC} orthogonal sub-carriers. The time sequence of N_{symb} OFDM symbols is called a TTI. The processing of OFDM symbols consists of four main steps:

- OFDM symbols are received by a set of N_{RX} antennas, meaning that each antenna receives a frequency multiplexed transmission. The first step of PUSCH consists of OFDM demodulation. In this workload, N_{RX} independent antenna streams, consisting of N_{SC} orthogonal sub-carrier samples, undergo a FFT.
- The OFDM demodulation stage of the lower PHY is followed by a digital BF stage. The BF process is a linear combination of the antennas' streams, by known coefficients, producing N_B streams of N_{SC} samples.
- The following step depends on whether we consider a data symbol or a Demodulation and Reference Symbol (DMRS) symbol. DMRS symbols are pilot symbols used to estimate the channel over which the transmission. The output of the process is the estimated channel matrix H. A different channel matrix is estimated for each subcarrier.
- To extract data symbols we execute a MIMO-MMSE processing pipeline, implementing $y = (H^H H + \sigma^2 I)^{-1} H^H x$ for each subcarrier, where x is the received vector and y is the estimated symbols vector. The inversion problem dimensions depend on the number of UE, N_L transmitting on the same subcarrier.

Figure 14 highlights the data dependencies between the PUSCH processing steps. In this regard CHE represents a critical block, as its output feeds the MMSE processing for all the data symbols in a TTI.

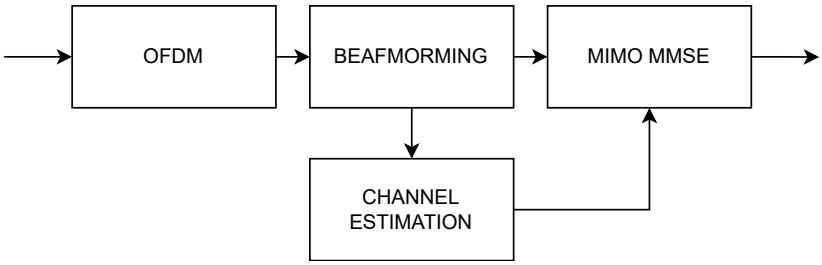


Figure 14: PUSCH processing chain steps: OFDM demodulation, BF, MIMO, channel estimation (CHE).

In our implementation of PUSCH lower PHY receiver, we work under the hypothesis of full load: we assume that all the OFDM resources are allocated to the PUSCH transmission, and that each subcarrier always has four UE simultaneously transmitting, where table 3 summarizes the PUSCH parameters and the values used for our experiments.

6.1 PUSCH kernels implementation

We implemented the kernels for 5G-PUSCH lower PHY processing on TeraPool. We propose two implementations with different numerical precisions and final performance: a fixed-point implementation and a floating-point implementation. Complex data types occupy 32 bit in both cases, 16 bit for real and imaginary parts each. The implementation follows the guidelines reported in [24], to reduce the effects of contentions in the shared memory interconnects.

Table 3: PUSCH parameters

Symbol	Description	Value
N_{SC}	Number of sub-carriers	4096
N_{RX}	Number of antennas	64
N_B	Number of beams	32
N_L	Number of layers	4
N_{symb}	Number of symbols in one TTI	14
$N_{data-symb}$	Number of data-symbols	12
$N_{pilot-symb}$	Number of pilot-symbols	2

- OFDM demodulation consists of an FFT. The kernel is implemented using the decimation in frequency Cooley-Turkey algorithm. FFT from different antennas are assigned to different groups of cores. The execution of each antenna FFT-stream is fine-grain parallelized over all the cores in a group.
- The beamforming stage is implemented as a complex matrix multiplication $C = A \times B$, where B is a matrix of known coefficients and A is the output of the FFT streams for all the antennas. The OFDM and the beamforming problems both entirely fit in the L1 memory of the cluster. Between OFDM and beamforming, there is a strong data dependency, and synchronization is needed. OFDM and BF are parallelized over antenna streams.
- For DMRS symbols we implement CHE as an element-by-element division between received symbols and pilots. Computations are parallelized over sub-carriers, and independent CHE problems are assigned to each core.
- For data symbols, addressing MMSE, we implement Hermitian computation, Matrix-Vector Multiplication (MVM), and matrix inversion with the Cholesky method. The workload is parallelized over subcarriers and independent MMSE problems are assigned to each core.

Whenever possible, in the implementation of the processing chain, we keep data in L1, treating it as a low-latency buffer over the pipeline computation. In particular, the output of OFDM is not transferred to L2 but stays in TeraPool's large-scale L1 SPM for digital BF. Furthermore, results of CHE is kept in L1 for MMSE on adjacent symbols of a TTI. The table 4 reports the computational complexity of PUSCH computing steps given the implemented kernels and the selected use-case parameters.

Table 4: PUSCH kernels and computational complexity

Steps	Key kernel	Complex MACs	Percentage on total
OFDM dem.	Fast Fourier transform	$N_{symb} \times N_R \times N_{SC} \times \log(N_{SC})$	26.7%
BF	Matrix-matrix multiplication	$N_{symb} \times N_{SC} \times N_R \times N_B$	71.1%
MIMO	Cholesky decomposition	$N_{data-symb} \times N_{SC} \times (N_L^3/3 + 2N_L^2)$	1.6%
CHE	Element-wise division	$N_{pilot-symb} \times N_{SC} \times N_B \times N_L$	0.6%

6.2 Implementation results

We report execution cycles, instructions-per-cycle (IPC) and latency for each key PUSCH kernel in Table 5. For the analysis we use the $\text{TeraPool}_{\text{1-3-5-9}}$ configuration.

In table 6, we report an estimate of the overall execution time for PUSCH on TeraPool, including memory transfer latency across the DRAM memory stack, under the hypothesis of maximum load for the base station. Power simulations of the $\text{TeraPool}_{\text{1-3-5-9}}$ in the typical corner (0.80V, 25C) are also reported in the table and reveal the high energy efficiency of

Table 5: Latency of PUSCH kernels $\text{TeraPool}_{1\text{-}3\text{-}5\text{-}9}$.

		Cycles(×1000)	**IPC**	**Latency(µs)/symb**	**Latency(ms)/TTI**
FFT	fixed	32.80	0.73	41.00	0.574
	floating	44.63	0.72	55.78	0.781
BF	fixed	79.51	0.84	99.39	1.390
	floating	22.12	0.61	27.75	0.388
CHE	fixed	9.62	0.63	12.03	0.024
	floating	9.18	0.42	11.48	0.023
HERMITIAN	fixed	33.13	0.62	41.42	0.497
	floating	29.26	0.49	36.58	0.439
MVM	fixed	8.14	0.55	10.18	0.122
	floating	7.38	0.36	9.22	0.111
INVERSION	fixed	6.65	0.33	8.31	0.100
	floating	4.38	0.43	5.47	0.066

the architecture for the selected workload (power computations do not include the energy of transfers).

Table 6: Latency of PUSCH kernels vs latency of transfers in $\text{TeraPool}_{1\text{-}3\text{-}5\text{-}9}$.

		Computation(ms)	**Transfer(ms)**	**Power(Watt)**
FFT	fixed	0.574	0.046	6.090
	floating	0.781	0.046	5.630
BF	fixed	1.390	0.006	6.280
	floating	0.388	0.006	7.220
CHE	fixed	0.024	0.008	5.400
	floating	0.023	0.008	5.480
MMSE	fixed	0.719	0.048	4.950
	floating	0.616	0.048	4.340
Total Latency & Average Power	fixed	DMRS:0.305 DATA:2.404	0.015 0.093	5.840
	floating	DMRS:0.190 DATA:1.610	0.015 0.093	5.500

The results highlight a negligible transfer overhead and a computation latency of the floating point pipeline <2ms for the execution of PUSCH in a high load use-case. Specifications require the data of a TTI to be consumed in 0.5ms, however the PUSCH is in general alternated with other channels, therefore, as a rule of thumb PUSCH processing can stretch to up to 2.5ms. Our conclusion is that TeraPool is a good candidate for the acceleration of signal processing intensive parts of PUSCH lower PHY and that three clusters would be necessary to guarantee the required throughput.

7 Conclusion

In this report, the TeraPool architecture was presented. TeraPool is a physical implementation of the parallel cluster with shared scratchpad memory for 1024 cores and 4MiB of L1. As such, it configures as a difficult backend design problem, that was solved via hierarchical interconnects with an AXI-based protocol. The hierarchy configures a NUMA system. The large shared L1 of TeraPool and its 1.89TOPS peak performance are a precious resource to tackle compute-intensive problems with large dimensions. The design is completed with the addition of a top AXI tree and DMA engine to communicate with L2. We tested the cluster L2 link using an HBM-2E memory as L2. The provided interconnect is therefore suitable for ultra-high bandwidth transfers.

TeraPool has a single-program multiple-data execution model, but the cores have individual instruction streams. The SPMD approach can therefore be scaled to small groups of cores up to cluster scale (1024 cores), depending on the problem dimension. The programmer can use one of the three available runtime (bare-metal, Open-MP, and Halide) to implement the SPMD execution for any parallel kernel. At the end of the execution, the cores are synchronized using synchronization barrier calls.

TeraPool is a suitable acceleration platform for the execution of signal processing intensive parts of 5G-PUSCH. The reasons for this are three. First, PUSCH inputs and outputs have large dimensions, but the transfer overhead can be kept low, using the large L1 of TeraPool as a compute buffer. Second, baseband processing is considered power hungry, however the average compute power for PUSCH on Terapool is less than 6W. Third, the PUSCH processing latency per TTI is compliant with the specifications in a high load use-case. A single TeraPool cluster achieved only 1.61 ms floating-point data symbol compute latency per TTI for PUSCH with only 6% data transfer overhead, demonstrating its suitability as an accelerator for PUSCH with only three clusters to ensure the required throughput.

All the results in this report are reproducible. RTL-code and C-code for TeraPool and the implemented PUSCH kernels are open-sourced on GitHub (`https://github.com/pulp-platform/mempool`).

8 Open Source Information

8.1 Publications

Under this section, we collect additional open-source publications on TeraPool:

- S. Riedel, M. Cavalcante, R. Andri and L. Benini, "MemPool: A Scalable Manycore Architecture With a Low-Latency Shared L1 Memory," in IEEE Transactions on Computers, vol. 72, no. 12, pp. 3561-3575, Dec. 2023, doi: 10.1109/TC.2023.3307796.
- M. Bertuletti, Y. Zhang, A. Vanelli-Coralli and L. Benini, "Efficient Parallelization of 5G-PUSCH on a Scalable RISC-V Many-Core Processor," 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE), Antwerp, Belgium, 2023, pp. 1-6, doi: 10.23919/DATE56975.2023.10137247.
- M. Bertuletti, S. Riedel, Y. Zhang, A. Vanelli-Coralli and L. Benini, "Fast Shared-Memory Barrier Synchronization for a 1024-Cores RISC-V Many-Core Cluster," in Embedded Computer Systems: Architectures, Modeling, and Simulation: 23rd International Conference, SAMOS 2023, Samos, Greece, July 2–6, 2023, Proceedings. Springer-Verlag, Berlin, Heidelberg, 241–254. https://doi.org/10.1007/978-3-031-46077-7_16.
- Y. Zhang, M. Bertuletti, S. Riedel, M. Cavalcante, A. Vanelli-Coralli and L. Benini, "TeraPool-SDR: An 1.89TOPS 1024 RV-Cores 4MiB Shared-L1 Cluster for Next-Generation Open-Source Software-Defined Radios," In proceeding of 2024 Great Lakes Symposium on VLSI (GLSVLSI), Clearwater, Tampa, USA, 2024, https://arxiv.org/abs/2405.04988.

8.2 Software and hardware source code

We have made the source code for MemPool/TeraPool architecture and PUSCH kernels work publicly available, reflecting our commitment to open-source development and collaboration within the research community. Our design is open-sourced under a liberal license.

- **Software Implementation:** All our implemented key kernels can be found on GitHub at the following URL: `https://github.com/pulp-platform/mempool/tree/mbertuletti/mimo_receiver`. The bare-metal C code for the PUSCH kernels is located in the *software/kernels* repository's directory.
- **Open-Sourced Hardware Architecture:** The hardware architecture, which forms the backbone of our project, is also open-sourced and accessible at `https://github.com/pulp-platform/mempool`. The repository's *hardware/src* folder contains all the design RTL source code, showcasing our approach to hardware design and scalability.
- **Documentation and Building the Environment:** For a comprehensive understanding of our source code and hardware architecture and instructions for building and deploying the design environment, we encourage interested parties to refer to the README file located in the root of our GitHub repository. This document provides detailed guidelines and necessary steps to replicate our environment, ensuring ease of use and accessibility for researchers and developers alike.

Acknowledgments

Huawei Technologies Sweden AB supported this work.

References

[1] Xingqin Lin et al. "5G New Radio: Unveiling the Essentials of the Next Generation Wireless Access Technology". In: *IEEE Communications Standards Magazine* 3.3 (2019), pp. 30–37. DOI: 10.1109/MCOMSTD.001.1800036.

[2] J. Mitola. "The software radio architecture". In: *IEEE Communications Magazine* 33.5 (1995), pp. 26–38. DOI: 10.1109/35.393001.

[3] EdgeQ. *5G meets AI*. https://www.edgeq.io/technology/. Accessed: 11/13/2023. 2023.

[4] Qualcomm. *How we won the acceleration architecture debate*. https://www.qualcomm.com/news/onq/2023/03/how-we-won-the-acceleration-architecture-debate. Accessed: 11/13/2023. 2023.

[5] Marvell. *Data Processing Units (DPU) Empowering 5G carrier, enterprise and AI cloud data infrastructure*. https://www.marvell.com/products/data-processing-units.html. Accessed: 11/13/2023. 2023.

[6] Xilinx. *Breakthrough Adaptive Radio Platform for Mass 5G Deployments*. https://www.xilinx.com/products/silicon-devices/soc/rfsoc/zynq-ultrascale-plus-rfsoc-dfe.html. Accessed: 11/13/2023. 2023.

[7] Line M. P. Larsen, Aleksandra Checko, and Henrik L. Christiansen. "A Survey of the Functional Splits Proposed for 5G Mobile Crosshaul Networks". In: *IEEE Communications Surveys & Tutorials* 21.1 (2019), pp. 146–172. DOI: 10.1109/COMST.2018.2868805.

[8] M. Horowitz. "Computing's Energy Problem (and What We Can Do About It)". In: *IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 2014, pp. 10–14. DOI: 10.1109/ISSCC.2014.6757323.

[9] 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.

[10] Dave Ditzel, Roger Espasa, and et al. "Accelerating ML recommendation with over a thousand RISC-V/Tensor processors on Esperanto's ET-SoC-1 Chip". In: *2021 IEEE Hot Chips 33 Symp.* Palo Alto, California: IEEE, Aug. 2021, pp. 209–220. ISBN: 9781665413978. DOI: 10.1109/HCS52781.2021.9566904.

[11] R. Ginosar et al. "Ramon Space RC64-based AI/ML Inference Engine". In: *European Workshop on On-Board Data Processing (OBDP)*. Online: Zenedo, June 2021, pp. 1–33.

[12] 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.

[13] 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.

[14] Sergio Mazzola et al. "ISA extensions in the Snitch processor for signal processing". In: *Master's thesis, Politecnico di Torino*. Apr. 2021.

[15] 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.

[16] T. Benz et al. "A High-Performance, Energy-Efficient Modular DMA Engine Architecture". In: *IEEE Transactions on Computers* 73.01 (Jan. 2024), pp. 263–277. ISSN: 1557-9956. DOI: 10.1109/TC.2023.3329930.

[17] 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.

[18] Anthony Agnesina et al. "Hier-3D: A Hierarchical Physical Design Methodology for Face-to-Face-Bonded 3D ICs". In: *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*. ISLPED '22. Boston, MA, USA: Association for Computing Machinery, 2022. ISBN: 9781450393546. DOI: 10.1145/3531437.3539702. URL: https://doi.org/10.1145/3531437.3539702.

[19] Gianna Paulin et al. "Soft Tiles: Capturing Physical Implementation Flexibility for Tightly-Coupled Parallel Processing Clusters". In: *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2022, pp. 44–49. DOI: 10.1109/ISVLSI54635.2022.00021.

[20] Marco Bertuletti et al. "Fast Shared-Memory Barrier Synchronization for a 1024-Cores RISC-V Many-Core Cluster". In: *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS XXIII)*. 2023. eprint: 2307.10248. URL: https://doi.org/10.48550/arXiv.2307.10248.

[21] Torsten Hoefler et al. "A Survey of Barrier Algorithms for Coarse Grained Supercomputers." In: *Chemnitzer Informatik Berichte* (Jan. 2004).

[22] Aboul-Karim Mohamed El Maarouf et al. "Combining reduction with synchronization barrier on multi-core processors". In: *Concurrency and Computation: Practice and Experience* 35.1 (2023), e7402. DOI: 10.1002/cpe.7402. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.7402.

[23] 3GPP. *Physical Channels and Modulation*. Technical Specification (TS) 38.211. Release 17. 3rd Generation Partnership Project (3GPP), 2017.

[24] Marco Bertuletti et al. "Efficient Parallelization of 5G-PUSCH on a Scalable RISC-V Many-Core Processor". In: *2023 Design, Automation, and Test in Europe Conference and Exhibition*. Antwerp, Belgium: IEEE, Apr. 2023, pp. 396–401. DOI: 10.23919/DATE56975.2023.10137247.

A FPU instructions

TeraPool supports the *zfinx* and *zhinx* standard RISC-V extensions. Additionally, it implements also the following *smallfloat* extensions with the same rationale: the floating point unit reads and writes from the integer register file. A complete list of the smallfloat extensions and a detailed explanation of the acronyms used below can be found at `https://iis-git.ee.ethz.ch/smach/smallFloat-spec`.

31-25	24-20	19-15	14-12	11-7	6-0	
funct7	rs2	rs1	funct3	rd	opcode	R-type

31-27	26-25	24-20	19-15	14 13-12	11-7	6-0	
rs3	fc2	rs2	rs1	rm*	rd	opcode	R4-type

31-20	19-15	14 13-12	11-7	6-0	
imm[11:0]	rs1	funct3	rd	opcode	I-type

31-25	24-20	19-15	14 13-12	11-7	6-0	
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type

31-30	29-25	24-20	19-15	14	13-12	11-7	6-0	
f2	vecfltop	rs2	rs1	R	vfmt	rd	opcode	RVF-type

RV32Xf16 Half-Precision Floating-Point Extension, bit[26,25]=10 (binary16)

rs3	10	rs2	rs1	rm*	rd	1000011	FMADD.H	$(rs1 * rs2) + rs3$
rs3	10	rs2	rs1	rm*	rd	1000111	FMSUB.H	$(rs1 * rs2) - rs3$
rs3	10	rs2	rs1	rm*	rd	1001011	FNMSUB.H	$-(rs1 * rs2) + rs3$
rs3	10	rs2	rs1	rm*	rd	1001111	FNMADD.H	$-(rs1 * rs2) - rs3$
0000010		rs2	rs1	rm*	rd	1010011	FADD.H	$rs1 + rs2$
0000110		rs2	rs1	rm*	rd	1010011	FSUB.H	$rs1 - rs2$
0001010		rs2	rs1	rm*	rd	1010011	FMUL.H	$rs1 * rs2$
0001110		rs2	rs1	rm*	rd	1010011	FDIV.H	$rs1/rs2$
0101110		00000	rs1	rm*	rd	1010011	FSQRT.H	$\sqrt{rs1}$
0010010		rs2	rs1	000	rd	1010011	FSGNJ.H	rs1, sign of rs2
0010010		rs2	rs1	001	rd	1010011	FSGNJN.H	rs1, inv. sign of rs2
0010010		rs2	rs1	010	rd	1010011	FSGNJX.H	rs1, sign rs1 $\oplus$ sign rs2
0010110		rs2	rs1	000	rd	1010011	FMIN.H	min
0010110		rs2	rs1	001	rd	1010011	FMAX.H	max
1010010		rs2	rs1	010	rd	1010011	FEQ.H	equal
1010010		rs2	rs1	001	rd	1010011	FLT.H	less than
1010010		rs2	rs1	000	rd	1010011	FLE.H	less than or equal
1100010		00000	rs1	rm*	rd	1010011	FCVT.W.H	to sgn. word (32bit)
1100010		00001	rs1	rm*	rd	1010011	FCVT.WU.H	to usgn. word (32bit)
1101010		00000	rs1	rm*	rd	1010011	FCVT.H.W	from sgn. word (32bit)
1101010		00001	rs1	rm*	rd	1010011	FCVT.H.WU	from usgn. word (32bit)
1110010		00000	rs1	001	rd	1010011	FCLASS.H	classify

Conversions with F Standard Extension

0100000	00010	rs1	000	rd	1010011	FCVT.S.H	binary16 $\rightarrow$ binary32
0100010	00000	rs1	rm*	rd	1010011	FCVT.H.S	binary32 $\rightarrow$ binary16

RV32Xf8 Quarter-Precision Floating-Point Extension, bit[26,25]=11 (binary8)

imm[11:0]			rs1	000	rd	0000111	FLB	load
imm[11:5]		rs2	rs1	000	imm[4:0]	0100111	FSB	store
rs3	11	rs2	rs1	rm*	rd	1000011	FMADD.B	$(rs1 * rs2) + rs3$
rs3	11	rs2	rs1	rm*	rd	1000111	FMSUB.B	$(rs1 * rs2) - rs3$
rs3	11	rs2	rs1	rm*	rd	1001011	FNMSUB.B	$-(rs1 * rs2) + rs3$
rs3	11	rs2	rs1	rm*	rd	1001111	FNMADD.B	$-(rs1 * rs2) - rs3$
0000011		rs2	rs1	rm*	rd	1010011	FADD.B	$rs1 + rs2$
0000111		rs2	rs1	rm*	rd	1010011	FSUB.B	$rs1 - rs2$
0001011		rs2	rs1	rm*	rd	1010011	FMUL.B	$rs1 * rs2$
0001111		rs2	rs1	rm*	rd	1010011	FDIV.B	$rs1/rs2$
0101111		00000	rs1	rm*	rd	1010011	FSQRT.B	$\sqrt{rs1}$
0010011		rs2	rs1	000	rd	1010011	FSGNJ.B	rs1, sign of rs2
0010011		rs2	rs1	001	rd	1010011	FSGNJN.B	rs1, inv. sign of rs2
0010011		rs2	rs1	010	rd	1010011	FSGNJX.B	rs1, sign rs1 $\oplus$ sign rs2
0010111		rs2	rs1	000	rd	1010011	FMIN.B	min
0010111		rs2	rs1	001	rd	1010011	FMAX.B	max
1010011		rs2	rs1	010	rd	1010011	FEQ.B	equal
1010011		rs2	rs1	001	rd	1010011	FLT.B	less than
1010011		rs2	rs1	000	rd	1010011	FLE.B	less than or equal
1100011		00000	rs1	rm*	rd	1010011	FCVT.W.B	to sgn. word (32bit)
1100011		00001	rs1	rm*	rd	1010011	FCVT.WU.B	to usgn. word (32bit)
1101011		00000	rs1	rm*	rd	1010011	FCVT.B.W	from sgn. word (32bit)
1101011		00001	rs1	rm*	rd	1010011	FCVT.B.WU	from usgn. word (32bit)
1110011		00000	rs1	001	rd	1010011	FCLASS.B	classify

Conversions with F Standard Extension

0100000	00011	rs1	000	rd	1010011	FCVT.S.B	binary8 $\rightarrow$ binary32
0100011	00000	rs1	rm*	rd	1010011	FCVT.B.S	binary32 $\rightarrow$ binary8

Conversions with Xf16 Extension

0100010	00011	rs1	000	rd	1010011	FCVT.H.B	binary8 $\rightarrow$ binary16
0100011	00010	rs1	rm*	rd	1010011	FCVT.B.H	binary16 $\rightarrow$ binary8

Xfvec Vectorial Floating-Point Ext. with Xf16, FLEN$\geq$32, vfmt=10 (binary16)

10	00001	rs2	rs1	0	10	rd	0110011	VFADD.H	$rs1 + rs2$
10	00001	rs2	rs1	1	10	rd	0110011	VFADD.R.H	$rs1 + rs2$, R
10	00010	rs2	rs1	0	10	rd	0110011	VFSUB.H	$rs1 - rs2$
10	00010	rs2	rs1	1	10	rd	0110011	VFSUB.R.H	$rs1 - rs2$, R
10	00011	rs2	rs1	0	10	rd	0110011	VFMUL.H	$rs1 * rs2$
10	00011	rs2	rs1	1	10	rd	0110011	VFMUL.R.H	$rs1 * rs2$, R
10	00100	rs2	rs1	0	10	rd	0110011	VFDIV.H	$rs1/rs2$
10	00100	rs2	rs1	1	10	rd	0110011	VFDIV.R.H	$rs1/rs2$, R
10	00101	rs2	rs1	0	10	rd	0110011	VFMIN.H	min
10	00101	rs2	rs1	1	10	rd	0110011	VFMIN.R.H	min, R
10	00110	rs2	rs1	0	10	rd	0110011	VFMAX.H	max
10	00110	rs2	rs1	1	10	rd	0110011	VFMAX.R.H	max, R
10	00111	00000	rs1	0	10	rd	0110011	VFSQRT.H	$\sqrt{rs1}$
10	01000	rs2	rs1	0	10	rd	0110011	VFMAC.H	$(rs1 * rs2) + rd$
10	01000	rs2	rs1	1	10	rd	0110011	VFMAC.R.H	$(rs1 * rs2) + rd$, R
10	01001	rs2	rs1	0	10	rd	0110011	VFMRE.H	$(rs1 * rs2) - rd$
10	01001	rs2	rs1	1	10	rd	0110011	VFMRE.R.H	$(rs1 * rs2) - rd$, R
10	01100	00001	rs1	0	10	rd	0110011	VFCLASS.H	classify
10	01101	rs2	rs1	0	10	rd	0110011	VFSGNJ.H	rs1, sign of rs2
10	01101	rs2	rs1	1	10	rd	0110011	VFSGNJ.R.H	rs1, sign of rs2, R
10	01110	rs2	rs1	0	10	rd	0110011	VFSGNJN.H	rs1, inv. sign of rs2
10	01110	rs2	rs1	1	10	rd	0110011	VFSGNJN.R.H	rs1, inv. sign of rs2, R
10	01111	rs2	rs1	0	10	rd	0110011	VFSGNJX.H	rs1, sign rs1 $\oplus$ sign rs2
10	01111	rs2	rs1	1	10	rd	0110011	VFSGNJX.R.H	rs1, sign rs1 $\oplus$ sign rs2, R
10	10000	rs2	rs1	0	10	rd	0110011	VFEQ.H	equal
10	10000	rs2	rs1	1	10	rd	0110011	VFEQ.R.H	equal, R
10	10001	rs2	rs1	0	10	rd	0110011	VFNE.H	not equal
10	10001	rs2	rs1	1	10	rd	0110011	VFNE.R.H	not equal, R

10	10010	rs2	rs1	0	10	rd	0110011	VFLT.H	less than
10	10010	rs2	rs1	1	10	rd	0110011	VFLT.R.H	less than, R
10	10011	rs2	rs1	0	10	rd	0110011	VFGE.H	greater than or equal
10	10011	rs2	rs1	1	10	rd	0110011	VFGE.R.H	greater than or equal, R
10	10100	rs2	rs1	0	10	rd	0110011	VFLE.H	less than or equal
10	10100	rs2	rs1	1	10	rd	0110011	VFLE.R.H	less than or equal, R
10	10101	rs2	rs1	0	10	rd	0110011	VFGT.H	greater than
10	10101	rs2	rs1	1	10	rd	0110011	VFGT.R.H	greater than, R
10	11000	rs2	rs1	0	10	rd	0110011	VFCPKA.H.S	2xbinary32 → binary16 *op0,1*

Xfvec Vectorial Floating-Point Ext. with Xf8, FLEN≥16, vfmt=11 (binary8)

10	00001	rs2	rs1	0	11	rd	0110011	VFADD.B	$rs1 + rs2$
10	00001	rs2	rs1	1	11	rd	0110011	VFADD.R.B	$rs1 + rs2$, R
10	00010	rs2	rs1	0	11	rd	0110011	VFSUB.B	$rs1 - rs2$
10	00010	rs2	rs1	1	11	rd	0110011	VFSUB.R.B	$rs1 - rs2$, R
10	00011	rs2	rs1	0	11	rd	0110011	VFMUL.B	$rs1 * rs2$
10	00011	rs2	rs1	1	11	rd	0110011	VFMUL.R.B	$rs1 * rs2$, R
10	00100	rs2	rs1	0	11	rd	0110011	VFDIV.B	$rs1/rs2$
10	00100	rs2	rs1	1	11	rd	0110011	VFDIV.R.B	$rs1/rs2$, R
10	00101	rs2	rs1	0	11	rd	0110011	VFMIN.B	min
10	00101	rs2	rs1	1	11	rd	0110011	VFMIN.R.B	min, R
10	00110	rs2	rs1	0	11	rd	0110011	VFMAX.B	max
10	00110	rs2	rs1	1	11	rd	0110011	VFMAX.R.B	max, R
10	10111	00000	rs1	0	11	rd	0110011	VFSQRT.B	$\sqrt{rs1}$
10	01000	rs2	rs1	0	11	rd	0110011	VFMAC.B	$(rs1 * rs2) + rd$
10	01000	rs2	rs1	1	11	rd	0110011	VFMAC.R.B	$(rs1 * rs2) + rd$, R
10	01001	rs2	rs1	0	11	rd	0110011	VFMRE.B	$(rs1 * rs2) - rd$
10	01001	rs2	rs1	1	11	rd	0110011	VFMRE.R.B	$(rs1 * rs2) - rd$, R
10	01101	rs2	rs1	0	11	rd	0110011	VFSGNJ.B	rs1, sign of rs2
10	01101	rs2	rs1	1	11	rd	0110011	VFSGNJ.R.B	rs1, sign of rs2, R
10	01110	rs2	rs1	0	11	rd	0110011	VFSGNJN.B	rs1, inv. sign of rs2
10	01110	rs2	rs1	1	11	rd	0110011	VFSGNJN.R.B	rs1, inv. sign of rs2, R
10	01111	rs2	rs1	1	11	rd	0110011	VFSGNJX.B	rs1, sign rs1 ⊕ sign rs2
10	01111	rs2	rs1	0	11	rd	0110011	VFSGNJX.R.B	rs1, sign rs1 ⊕ sign rs2, R
10	10000	rs2	rs1	0	11	rd	0110011	VFEQ.B	equal
10	10000	rs2	rs1	1	11	rd	0110011	VFEQ.R.B	equal, R
10	10001	rs2	rs1	0	11	rd	0110011	VFNE.B	not equal
10	10001	rs2	rs1	1	11	rd	0110011	VFNE.R.B	not equal, R
10	10010	rs2	rs1	0	11	rd	0110011	VFLT.B	less than
10	10010	rs2	rs1	1	11	rd	0110011	VFLT.R.B	less than, R
10	10011	rs2	rs1	0	11	rd	0110011	VFGE.B	greater than or equal
10	10011	rs2	rs1	1	11	rd	0110011	VFGE.R.B	greater than or equal, R
10	10100	rs2	rs1	0	11	rd	0110011	VFLE.B	less than or equal
10	10100	rs2	rs1	1	11	rd	0110011	VFLE.R.B	less than or equal, R
10	10101	rs2	rs1	0	11	rd	0110011	VFGT.B	greater than
10	10101	rs2	rs1	1	11	rd	0110011	VFGT.R.B	greater than, R

Unless RV32D Supported

10	01100	00001	rs1	0	11	rd	0110011	VFCLASS.B	classify
10	01100	00010	rs1	0	11	rd	0110011	VFCVT.X.B	to vector of sgn. bytes
10	01100	00010	rs1	1	11	rd	0110011	VFCVT.XU.B	to vector of usgn. bytes
10	01100	00011	rs1	0	11	rd	0110011	VFCVT.B.X	from vector of sgn. bytes
10	01100	00011	rs1	1	11	rd	0110011	VFCVT.B.XU	from vector of usgn. bytes

Conversions when F Standard Extension Supported

10	11000	rs2	rs1	0	11	rd	0110011	VFCPKA.B.S	2xbinary32 → binary8 *op0,1*
10	11000	rs2	rs1	1	11	rd	0110011	VFCPKB.B.S	2xbinary32 → binary8 *op2,3*

Conversions when Xf16 Extension Supported

10	01100	00111	rs1	0	10	rd	0110011	VFCVT.H.B	*op0,1* binary8 → binary16
10	01100	00111	rs1	1	10	rd	0110011	VFCVTU.H.B	*op2,3* binary8 → binary16
10	01100	00110	rs1	0	11	rd	0110011	VFCVT.B.H	binary16 → binary8 *op0,1*
10	01100	00110	rs1	1	11	rd	0110011	VFCVTU.B.H	binary16 → binary8 *op2,3*

When Xfvec Extension Supported, FLEN≥32, vfmt=10 (binary16)

10	01011	rs2	rs1	0	10	rd	0110011	VFDOTPEX.S.H	fp32(dotp(rs1,rs2))
10	01011	rs2	rs1	1	10	rd	0110011	VFDOTPEX.S.R.H	fp32(dotp(rs1,rs2)), R
10	01011	rs2	rs1	0	10	rd	0110011	VFNDOTPEX.S.H	fp32(ndotp(rs1,rs2))
10	01011	rs2	rs1	1	10	rd	0110011	VFNDOTPEX.S.R.H	fp32(ndotp(rs1,rs2)), R
10	0011110110		rs1	0	00	rd	0110011	VFSUMEX.S.H	fp32(sum(rs1,rd))
10	1011110110		rs1	0	00	rd	0110011	VFNSUMEX.S.H	fp32(sum(rs1,rd))

When Xfvec Extension Supported, FLEN≥32, vfmt=10 (binary16)

10	01011	rs2	rs1	0	00	rd	1110111	FCDOTPEX.S.H	cdotp(rs1,rs2)
10	01011	rs2	rs1	1	00	rd	1110111	FCNDOTPEX.S.R.H	cndotp(rs1,rs2), R
10	11101	rs2	rs1	0	00	rd	1110111	FCCDOTPEX.S.H	cdotp(rs1,rs2*)
10	11101	rs2	rs1	1	00	rd	1110111	FCCNDOTPEX.S.R.H	cndotp(rs1,rs2*), R

When Xfvec Extension Supported, FLEN≥16, vfmt=11 (binary8)

10	01011	rs2	rs1	0	11	rd	0110011	VFDOTPEXA.S.B	fp32(dotp(rs1,rs2))
10	01011	rs2	rs1	1	11	rd	0110011	VFDOTPEXA.S.R.B	fp32(dotp(rs1,rs2)), R
10	01011	rs2	rs1	0	11	rd	0110011	VFDOTPEXB.S.B	fp32(dotp(rs1,rs2))
10	01011	rs2	rs1	1	11	rd	0110011	VFDOTPEXB.S.R.B	fp32(dotp(rs1,rs2)), R
10	0011110111		rs1	0	10	rd	0110011	VFSUMEX.H.B	fp16(sum(rs1,rd))
10	1011110111		rs1	0	10	rd	0110011	VFNSUMEX.H.B	fp16(sum(rs1,rd))

B IPU instructions

The *Xpulpimg* extensions are a subset of the *xpulpv2* extensions. The *xpulpv2* extensions are documented in `https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf`. The subset implemented in TeraPool's IPU is:

- **SCALAR arithmetic**:
  - P.ABS, P.SLET, P.SLETU,
  - P.MIN, P.MINU, P.MAX, P.MAXU,
  - P.EXTHS, P.EXTHZ, P.EXTBS, P.EXTBZ,
  - P.CLIP, P.CLIPU, P.CLIPR, P.CLIPUR,
  - P.MAC, P.MSU
- **SIMD2 arithmetic:** These instructions are available also in the .SC and .SCI format, where the second term of the computation is either the lower half of the second input register or an immediate.
  - PV.ADD.H, PV.SUB.H,
  - PV.AVG.H, PV.AVGU.H, PV.MIN.H, PV.MINU.H, PV.MAX.H, PV.MAXU.H,
  - PV.SRL.H, PV.SRA.H, PV.SLL.H,
  - PV.OR.H, PV.XOR.H, PV.AND.H, PV.ABS.H,
  - PV.DOTUP.H, PV.DOTUSP.H, PV.DOTSP.H, PV.SDOTUP.H, PV.SDOTUSP.H, PV.SDOTSP.H
- **SIMD4 arithmetic:** These instructions are available also in the .SC and .SCI format, where the second term of the computation is either the lower half of the second input register or an immediate.
  - PV.ADD.B, PV.SUB.B,
  - PV.AVG.B, PV.AVGU.B, PV.MIN.B, PV.MINU.B, PV.MAX.B, PV.MAXU.B,
  - PV.SRL.B, PV.SRA.B, PV.SLL.B,
  - PV.OR.B, PV.XOR.B, PV.AND.B, PV.ABS.B,
  - PV.DOTUP.B, PV.DOTUSP.B, PV.DOTSP.B, PV.SDOTUP.B, PV.SDOTUSP.B, PV.SDOTSP.B.
- **SIMD shuffle:**
  - PV.EXTRACT.H, PV.EXTRACT.B, PV.EXTRACTU.H, PV.EXTRACTU.B,
  - PV.INSERT.H, PV.INSERT.B, PV.SHUFFLE2.H, PV.SHUFFLE2.B,
  - PV.PACK, PV.PACK.H

Additionally, in the Snitch core, we also support the post-increment load and store instructions of the *xpulpv2* extension: P.LBU.IRPOST, P.LH.IRPOST, P.LHU.IRPOST, P.LW.IRPOST, P.LB.RRPOST, P.LBU.RRPOST, P.LH.RRPOST, P.LHU.RRPOST, P.LW.RRPOST, P.LB.RR, P.LBU.RR, P.LH.RR, P.LHU.RR, P.LW.RR, P.SB.IRPOST, P.SH.IRPOST, P.SW.IRPOST, P.SB.RRPOST, P.SH.RRPOST, P.SW.RRPOST, P.SB.RR, P.SH.RR, P.SW.RR.