NextRAN-AI

M1.1 AI-aided Physical Layer Processing

Report by:
Yichao Zhang, Ph.D. Student
Marco Bertuletti, Ph.D. Student
yiczhang, mbertuletti @iis.ee.ethz.ch

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

Integrated Systems Laboratory,
ETH Zürich,
Zürich, Switzerland

September 20, 2025

Contents

1 Introduction

The evolution of 5th-Generation (5G) and 6G Radio Access Networks (RAN) is rapidly advancing the network Quality of Service (QoS) in complex deployment scenarios with high user densities [1], enabling many new edge applications: *immersive communication* for immersive Extended Reality (XR) (remote multisensor telepresence and holographic communications), *massive communication* (connecting massive number of devices for smart cities, transportation, logistics, environment monitoring), *Artificial Intelligence and Communication* (assisted automated driving, autonomous collaboration between devices), and *Joint Communication and Sensing (JCAS)* (outdoor&indoor navigation, movement tracking).

Processing the uplink Physical Layer (PHY)-layer at the RAN edge is crucial for improving latency, performance, and system flexibility, but it stands out as one of the most computational and memory demanding RAN functions [2]. Recent research [3] has focused on Artificial Intelligence (AI)-for-RAN to enhance PHY-layer performance. Neural Network (NN)-based Orthogonal Frequency Division Multiplexing (OFDM) receivers have demonstrated improved Bit Error Rate (BER) performance compared to conventional Minimum Mean Squared Error (MMSE)-receivers [4, 5, 6, 7].

As a consequence of growing computational requirements for mixed AI & wireless workloads, base-station edge-processors are evolving from RAN-specialized Application Specific Integrated Circuits (ASICs) to high-performance many-core programmable processors [8, 9, 10, 11, 12]. In this milestone (M1.1) we review state-of-the-art (SoA) AI models for PHY processing and we extract the required Key Performance Indicators (KPIs) of a multi-cluster many-core architecture for edge mixed AI & wireless processing. More in detail:

1. In section 2 we describe our baseline TeraPool architecture, that was successfully used in a previous project to run the Physical Uplink Shared Channel (PUSCH) of 6G. The TeraPool is a building block for the computing architecture developed in NextRAN-AI.

2. In section 3 we review the SoA and we identify the KPI of AI-models for telecommunications to define the performance required to the NextRAN-AI computing platform and the improvements necessary on TeraPool.

3. In section 4 we describe the model proposed as a benchmark for the NextRAN-AI project and we further derive required features for our architecture.

4. In section 5 we describe the metodology adopted to extract system-level and cluster-level parameters of the NextRAN-AI compute platform and we obtain a dimensioning of it. We also estimate the final Power, Performance and Area (PPA). We then outline the next steps of the project.

M1.1 covers the SoA analysis, requirement analysis, and architectural specification parts of WP1.Y1, the exploration and selection tasks of WP2.Y1. We annotate each paragraph of the deliverable with the corresponding addressed work-packages.

2 WP1.Y1 TeraPool

NextRAN-AI project builds on the previous TeraPool project. TeraPool is a flexible and parametric many-core architecture for Software Defined Radio (SDR). It consists of a cluster of individually programmable cores, that share a low-latency L1 Scratchpad Memory (SPM) through a hierarchical, physical-design-aware, low-latency interconnect. This section details TeraPool's architecture bottom-up, starting with the Processing Element (PE).

2.1 TeraPool: Architecture

TeraPool's PEs are single-stage, 32-bit RISC-V *Snitch* cores [13]. The core Instruction Set Architecture (ISA) is the RV32IMA basic ISA, plus baseband specific extensions: XpulpIMG, Zfinx, Zhinx, complex floating point dot-product and Xsmallfloat with inputs and output operands in the integer register file.

Snitch contains a scoreboard that keeps track of outstanding memory accesses and allows to proceed with instructions as long as there are no read-after-write (RAW) hazards. To tolerate multi-cycle L1 access latency, the Load Store Unit (LSU) supports multiple outstanding memory requests, allowing Snitch to issue a series of loads and stores without waiting for the response. Given TeraPool's Non-Uniform Memory Access (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.

The *Tile* in Figure 1 is the *1st* implementation hierarchy. It contains 8 core complexes, 2 shared division and square-root floating-point units, 4 KiB of shared I$, and 32 KiB of scratchpad memory, divided into 32 banks, namely the Tightly Coupled Data Memory (TCDM). The cores' LSUs access the banks in 1-cycle through a local crossbar built with a logarithmic number of stages. A reservation table per bank handles atomic load&stores. Cores can hide TeraPool's NUMA interconnect latency by issuing up to 8 outstanding transactions. Issued load&stores are tracked by keeping their metadata in a transaction table. Unless a RAW dependency occurs, load&stores are non-blocking for the next instruction execution.

Figure 2 represents the hierarchical connections of Tiles that we adopted for software-defined lower-PHY applications. The *2nd* implementation hierarchy is a *SubGroup*, with 8 Tiles. All cores in a Tile have a shared request-response port to an 8×8 TCDM crossbar, addressing the memory of other Tiles in the SubGroup. The *3rd* implementation hierarchy is a *Group*, with 4 SubGroups. In a Group, the Tiles of a SubGroup access the TCDM of Tiles in other SubGroups via a shared request-response port, connected to one 8×8 TCDM crossbar per SubGroup. The *4th* level of the hierarchy is a *Cluster*, with 4 Groups. The 32 Tiles in a Group access the TCDM of Tiles in other Groups via 32×32 TCDM crossbars, and one shared request-response port per Group in each Tile.

As described in [14], an AXI interconnect is also instantiated in each hierarchical level. The AXI interconnect has three main functions. First, it connects the cores to the L2 memory,

the cluster peripherals, and a cluster-shared Direct Memory Access (DMA) engine. Second, it allows I$ refill. The cores and the I$ share the same AXI port at the Tile level. Third, it carries streamlined DMA transfers between the L2 and the shared-L1 scratchpad banks. A DMA frontend is individually programmed by each core through reads and writes on the AXI interconnect. Data transfers are initiated by the frontend. They run over the AXI interconnect and they are redistributed to banks by a DMA backend in each SubGroup. The DMA backend and the Tiles' AXI requests and responses share the same 512 b wide port at the SubGroup level. Therefore, the whole cluster has a 1024 B/cycle AXI link to L2 memory, containing both instructions and data.

2.2 TeraPool: programming model & inter-cluster parallelization

The big workloads of 5G lower PHY are parallelized over the many cores of the cluster, using a fork-join programming model. By default, all the cores execute the program in parallel. By runtime calls, each core reads its private ID from a status-register. The programmer can use this ID to index conditional branches and loops, distributing the workload between cores. At the end of the assigned parallel task, cores synchronize and enter a new parallel task. Adopting the same single-program multiple-data paradigm used for small TCDM clusters of 4 to 16 cores has the advantage of simplifying the programming model, but it could incur synchronization overheads. The cluster's C-runtime includes two main synchronization primitives [15]. In the *linear barrier*, cores atomically write a shared synchronization variable and go in a wait-for-interrupt (WFI) state; the last core fetching from the shared memory location resets the variable and wakes up all the others. The *logarithmic barrier* implements a synchronization tree: first, the cores synchronize in groups by atomic writes to multiple locations, then the last core in each group continues to the next level of the tree, where the same pattern repeats. The core passing through all the synchronization levels wakes up the cluster. The wake-up is centrally handled, via a shared wake-up register and hardwired wake-up trigger connections to each core. The wake-up triggers to each core can be asserted with a Tile granularity, implementing *partial synchronization* between subsets of cores in the cluster [15].

2.3 TeraPool: Runtime

TeraPool provides a bare-metal C runtime where cores are programmed with the same code, and the programmer can create branches for each core or let them work on specific data using their unique ID [14]. Control is handed over to the programmer once the stack is allocated in the sequential region and the runtime is initialized. The programmer can then use various runtime functions, such as allocators and barriers, and has complete control over every core to implement highly optimized applications.

The OpenMP runtime for TeraPool is implemented in C, providing a shared-memory fork-join programming model [14]. The program is primarily executed by a single master core, which can then fork off to employ the cores waiting for work through wake-up triggers and

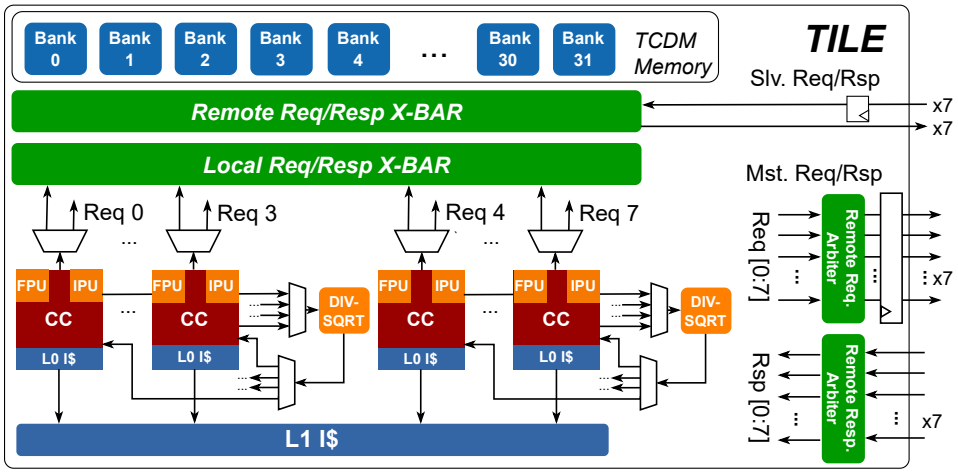


Figure 1: The TeraPool Tile architecture overview.

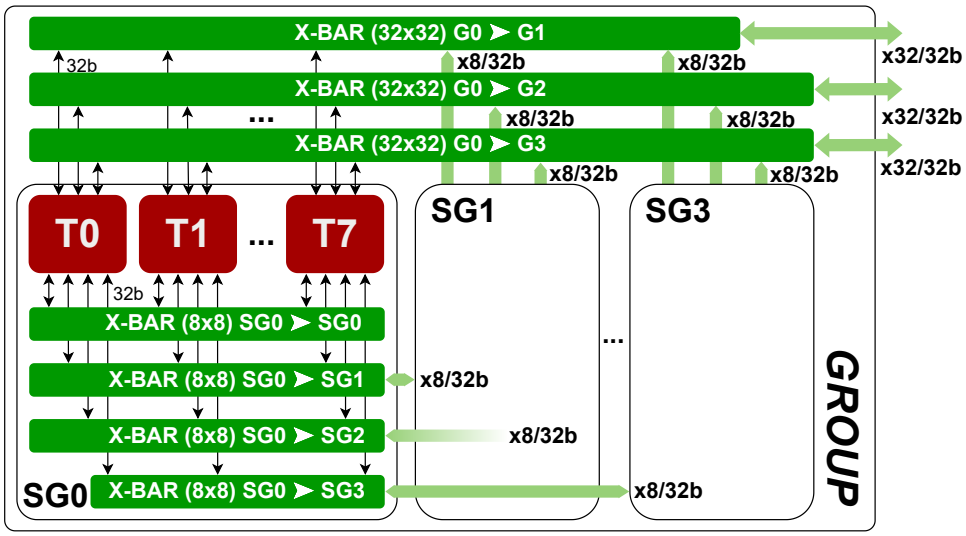


Figure 2: The TeraPool Interconnect architecture overview.

synchronization with RISC-V atomics. TeraPool supports static and dynamic loop scheduling; parallel sections; synchronization directives such as master, critical, atomic, and barrier; and reductions. OpenMP simplifies parallelizing a workload at the cost of runtime overhead.

3 WP2.Y1 AI-Models for the Physical Layer

This section describes the main strategies adopted by SoA NN models to improve QoS in the PHY. First, Convolutional Neural Network (CNN) based models of the full PHY processing are described. Second, the focus is on attention-based models and their successful application to channel estimation tasks. The basic-operators in the models dataflow are identified and the KPIs of the models are evaluated in terms of computational complexity, memory footprint and performance.

3.1 CNN-based Models

Among the CNN-based models of the PHY, [5] first introduced DeepRx, a deep learning-based receiver architecture for Multiple-Input, Multiple-Output (MIMO) systems. The architecture comprises a ResNet-based convolutional neural network combined with a transformation layer, which can be either a maximal ratio combining transformation or a fully learned multiplicative transformation. The latter leverages learned multiplicative layers to enhance performance. The study demonstrates that both transformation layers outperform conventional receivers, particularly in scenarios with sparse pilot configurations.

[7] builds upon [5]. It extends the approach from single to multi-user MIMO scenarios, and introduces a neural receiver that processes the entire 5G slot, jointly performing channel estimation, equalization, and demapping. The message-passing block, described in fig. 3 models inter-user interference and relationships, making the system well-suited to scenarios with dynamic user and layer configurations. The architecture is designed to generalize to arbitrary numbers of subcarriers, MIMO layers, and users without the need for retraining. Empirical results on the 3GPP 5G PUSCH show significant improvements in bit-error rate (BER) over conventional receivers.

Despite improving on [5] generalization capabilities, [7] significantly increases in computational complexity, as described in table 1. [6] mitigates the problem for runtime deployment of the model, by controlling the number of iterations of the message-passing layer. It further introduces retraining to improve the model performance, depending on the use-case.

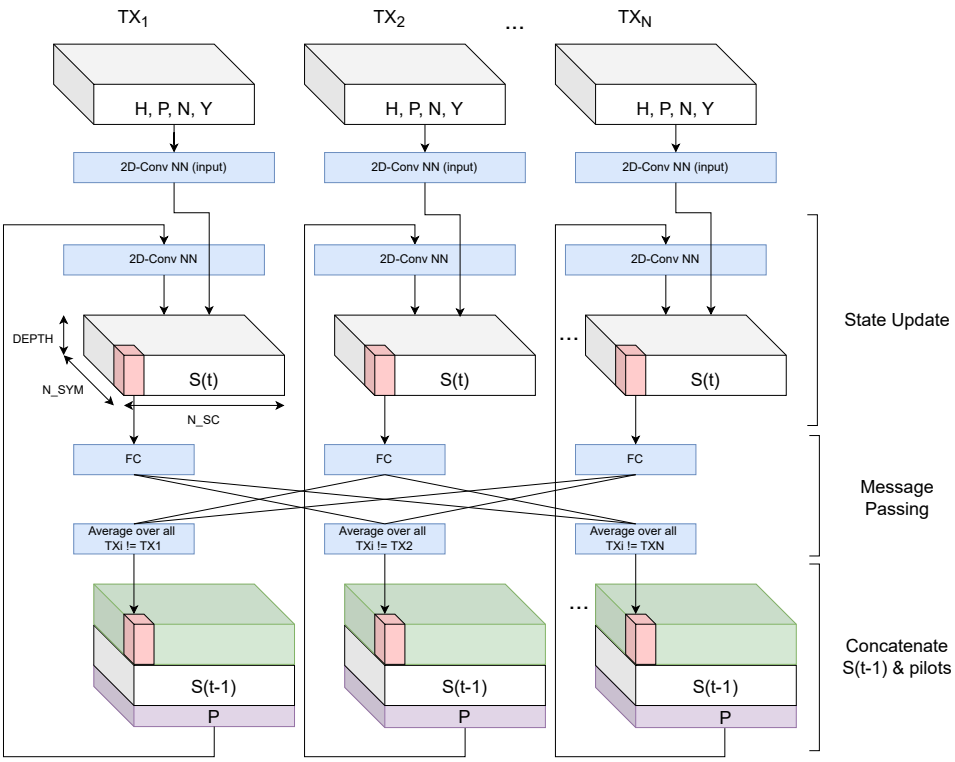


Figure 3: The Neural-RX model.

3.2 Attention-based Models for Channel Estimation

Alongside CNN-based models, attention-based models gained popularity for channel estimation tasks, given their ability to capture long-range dependencies between data.

In this context, [16], represented in fig. 4 (a), introduces a novel hybrid neural architecture, referred to as HA02, which combines a multi-head self-attention encoder with a residual convolutional decoder to improve channel estimation in OFDM-based systems. The attention mechanism allows the model to focus on the most informative Least squares pilot estimates, while the decoder refines these features to reconstruct the complete channel state. Experimental results demonstrate that HA02 significantly outperforms existing neural and traditional baselines, achieving up to 70% reduction in mean squared error compared to a ResNet-based model.

To address the challenges of channel estimation in extremely large-scale MIMO systems, [17], represented in fig. 4 (b), proposes *MAT-CENet*, a novel Transformer-based neural architecture. The model integrates a mixed attention mechanism comprising feature-map attention and spatial attention, enabling it to capture both high-level feature relevance and spatial dependencies within large-dimensional channel matrices. MAT-CENet demonstrates superior estimation accuracy compared least-squares estimation, across near-field, far-field, and hybrid-field scenarios.

Finally, [18], represented in fig. 4 (c), investigates a multi-layer data transmission scheme employing superimposed pilots to improve the throughput of MIMO-OFDM systems. A key challenge in such settings arises from signal coupling between antennas and data layers, which introduces severe interference and complicates receiver design. To address this, the authors propose a deep learning-based receiver architecture called *SANet*, which utilizes cascaded Muti-Head Self-Attention (MHSA) layers for parallel feature extraction across signal layers. Each attention head captures localized features, enabling effective separation of multi-layer bitstreams. Numerical results demonstrate that SANet achieves throughput gains of 7.01% with orthogonal pilots and 37.15% with superimposed pilots compared to conventional methods.

The required computational complexity, memory footprint and performance of the described attention-based models for channel estimation are reported in table 1. These models tend to bess expensive that CNN-based models in terms of computational complexity. In general, while CNN have beed used to model the entire PHY processing, smaller attention-models can be used to estimate the transmission channel with better performance than least-squares for only some users in the Transition Time Interval (TTI), when the QoS is excessively degraded.

3.3 Evaluation of Computational-Complexity and Memory-Footprint

The evaluation of the peak performance and memory footprint of the SoA models in the survey is required to guide the architecture definition phase. In particular, three main steps must be completed:

1. The NN-model must be divided into layers. The layers can be further split into **basic-operators**, implemented as C-microkernels on TeraPool. To further clarify this distinction

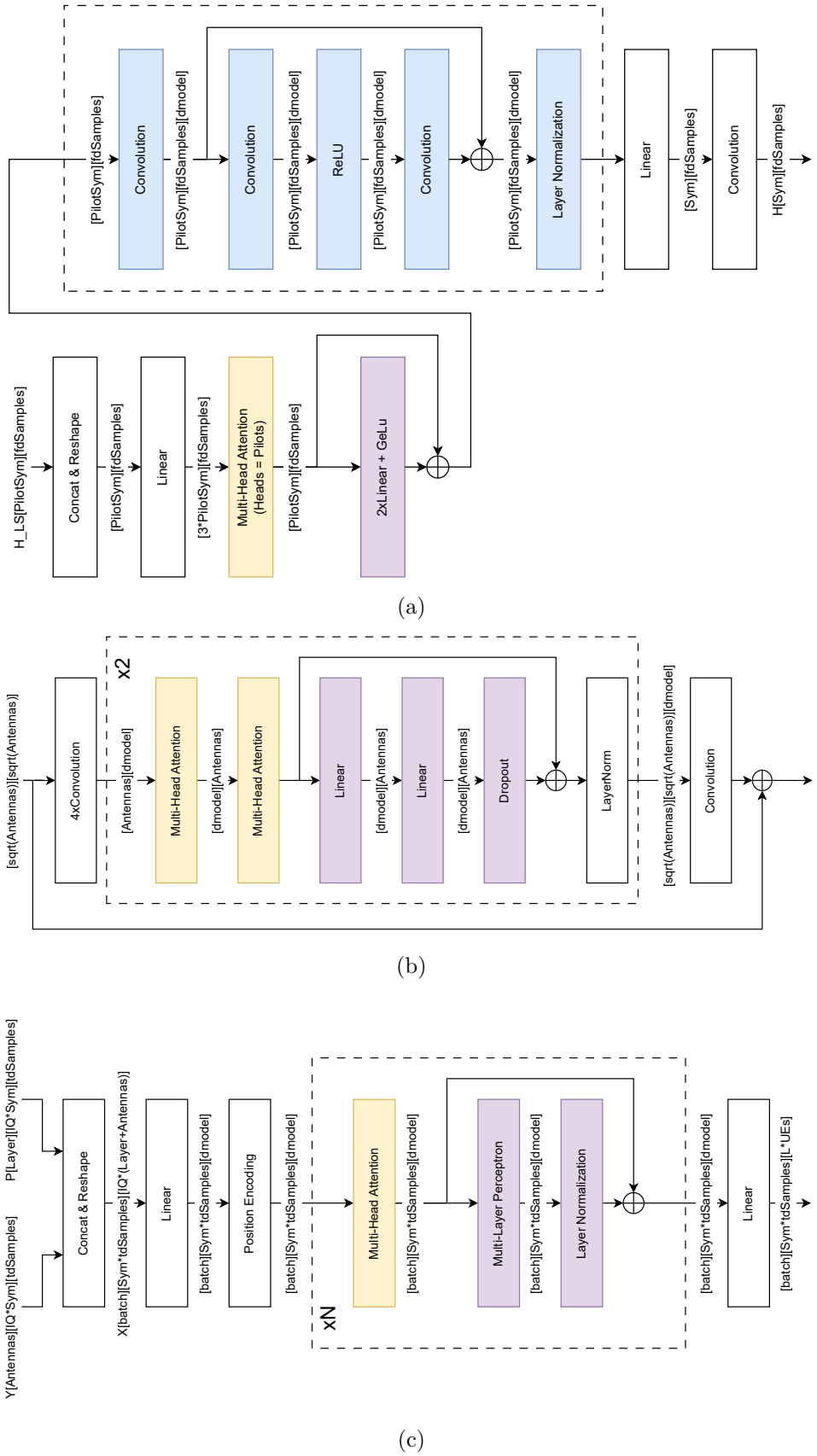


Figure 4: Transformer-based NNs for channel estimation.

an activation *layer* of a common feed-forward NN is generally composed in our nomenclature of two basic-perators: a matrix-vector multiplication and the output activation (e.g. softmax, ReLU, GeLU, ...). A neural network block diagram, representing the connections between basic-operators can be also created.

2. The **computational-complexity** of the basic-operators is computed in terms of FLOP. The peak-performance requirements is obtained dividing the computational complexity by the application runtime (0.5ms).

3. The **memory-footprint** for basic-operators includes the parameters, the inputs and the outputs of each operator. The memory footprint for the entire network must include the weights of all the network layers and the maximum of the sum between input and outputs for all the network layers. Assuming a sequential execution of the network on the available memory resources (the output of the previous layer is used as an input for the next layer), the so computed memory footprint represents the size of L1 required to execute the model without tiling of basic-operators. Tiling the basic-operators consists of splitting their inputs, outputs and weights across the L1 of separate compute units. This reduces the performance of reduction operations, and forces more data movement and duplication. Therefore we aim for large L1s and for compute units that can execute big operators or many independent operators in sequence without tiling.

From point 1 and point 2 we derive which basic-operators collect the largest number of FLOP and we identify possible acceration options. From point 3 we obtain the minimum required memory footprint to execute the model in L1 without tiling the basic-operators. This figure is enough to dimension the system's L1 footprint with data-reuse maximization as a target.

Evaluating other possible execution scenarios (sequential execution of operators on a single many-core cluster, execution of independent basic-operators across parallel clusters and execution of a single tiled basic-operator across parallel clusters), based on the workload distribution across clusters do not change the worst case requirements on L1 footprint to maximize data-reuse.

The computational-complexity, compute-performance and memory-footprint of the models discussed in section 3.1 and section 3.2 is in table 1.

Model	Complexity [FLOP]	Mem.-Footprint	Performance [FLOP/s]	SNR @BER/MSE=10^{-2}
Korpi [5]	10.2G	13.4MB	20.4T	BER: 7.5dB vs 8.5dB (LS + LMMSE)
Cammerer [7]	38.8G	437.4KB	77.6T	BER: 11dB vs 16db (LS + LMMSE)
Wiesmayr [6]	12.7G	142.9KB	25.4T	BER: 14dB vs 16dB (LS + LMMSE)
Luan [16]	365.5K	147.5KB	731M	MSE: 10dB
Shuangshuang [17]	165M	98.3KB	330M	MSE: 11dB vs 25dB (LS)
Zou [18]	5.5G	1.9MB	11T	BER: 0.0dB vs 1.5dB (LS)

Table 1: Computational-complexity, compute-performance and memory-footprint of the models described in section 3.1 and section 3.2

4 WP2.Y1 Analysis of the NextRAN-AI model

The NextRAN-AI model is the architecture proposed by the industry counterpart of the NextRAN-AI project as a benchmark. The model is inspired by [16, 17, 18] and is used for channel estimation in users with impared least-squares only performance. In the next section, we describe the model and discuss the similarities with SoA examples, to justify its adoption as a reference. We further compute its KPIs and evaluate processing bottlencks, to consider AI-oriented acceleration options.

4.1 Model architecture and basic-operators

The NextRAN-AI model is represented in fig. 5. It integrates the most salient features attention-based models for channel estimation described in section 3.2. The design leverages a hybrid attention mechanism, combining MHSA (in yellow) for feature extraction across the feature-map (embedded) and temporal dimensions, as in the MAT-CENet framework [17]. Similarly to what was observed in [16], a residual convolutional decoder (in light-blue) refines the characteristics extracted by the two MHSA blocks. Furthermore, as in SANet [18], twenty MHSA blocks are cascaded.

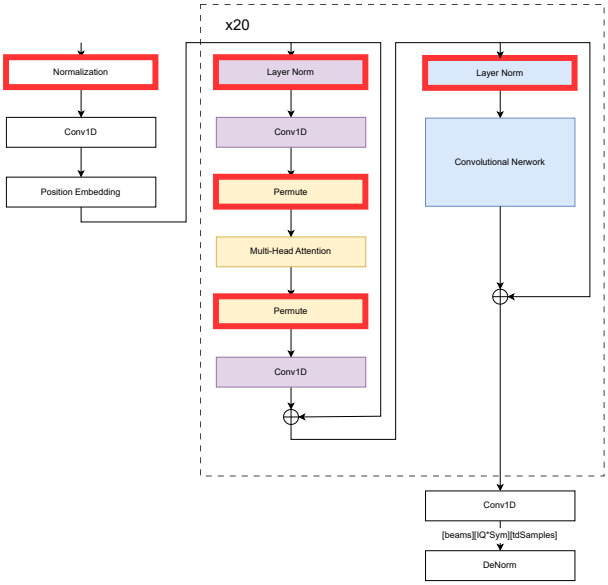


Figure 5: The block diagram of NextRAN-AI model.

4.2 Model computational-complexity and memory-footprint

Besides combining different multiple SoA solutions, the NextRAN-AI benchmark has comparamble computational-complexity and memory footprint to the most demanding models in our survey. As such it configures as a challenging and representative benchmark that can be used as a reference to tune the TeraPool architecture for mixed AI and wireless workloads.

Model	Complexity [FLOP]	Mem.-Footprint [MiB]	Performance [FLOP/s]
NextRAN-AI	4.2G	3.2MB	8.4T

Table 2: Computational-complexity, compute-performance and memory-footprint of the NextRAN-AI model.

4.3 Model processing bottlenecks

96% of the total FLOP required by the model is invested in the execution of 1D convolution, and matrix multiplication operations in the attention heads, including the computation of the attention matrix and the product between the values matrix and the attention matrix. The remaining 4% of the operation is used for layer normalization and activation operations. These findings suggest that TeraPool's compute architecture can better address the workload with specialized matrix-multiplication datapaths, which will increase the peak-performance over area and power efficiency of the computation. Yet programmable cores should be maintained to ensure that the wireless functions can still be executed and the activation and normalization function in the model can be pipelined with the matrix-multiplication computation, while the data is kept in the shared L1 memory of the cluster.

4.4 Parallelization of the model

In fig. 5 the operators that enforce data dependencies between all the elements of the computation are circled in red. The marked operators typically involve reductions or data shuffling. The input tensors computation must be concluded over all the dimensions before the processing of these operators can start. Therefore, the operators circled in red are hard synchronization standpoints. Forseeing the execution of the model on a processing platform that includes multiple TeraPool-like clusters, the workload in between these operators will be potentially parallelized over multiple clusters and across the cores of the single cluster. Figure fig. 6 explains this concept for the embedded attention block in the model (other parts of the model can be similarly parallelized). In the embedded attention block the Conv1D and Attention operators do not have data dependencies over the Beams and Embed dimension respectively. Therefore, they can be parallelized over multiple clusters. Within the single cluster the $Embed \times tdSamples$ Conv1Ds and $Beams \times tdSamples$ Attention problems can be parallelized across cores, as detailed in section 2.2. After parallel execution on multiple clusters, the clusters can synchronize. For synchronization, a memory-mapped registers synchronization method similar to the one described within the single cluster can be forseen.

5 WP1.Y1 Multi-TensorPool Architecture Proposal

Given the high computational complexity and peak performance required by running multiple instances of the NextRAN-AI model (each taking care of the channel estimation for a different user), we propose a multicluster architecture. Independent operator sequences in

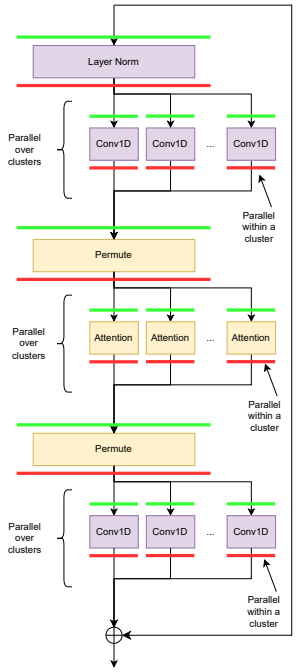


Figure 6: Parallelization of the Embedded attention block. Green and red lines represent load and stores from the L2 memory or from the L1 memory of other clusters via DMA transfers.

different model instances can be executed in parallel on different clusters. The data is moved across clusters and to the L2 memory through a NoC. A high level diagram of the system is represented in fig. 7

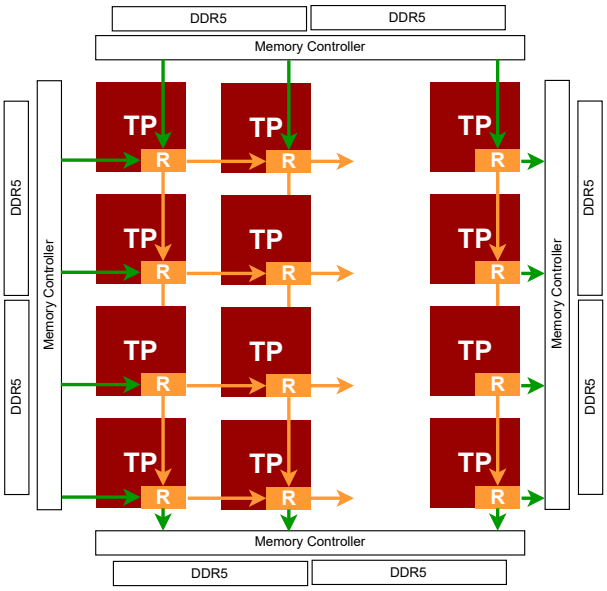


Figure 7: Block diagram of the Multi-Cluster TeraPool.

To achieve a high peak performance within the single cluster, we propose integrating RedMulE [19] into the TeraPool architecture. RedMulE is a tensor accelerator. Having a fixed function, RedMulE reduces the programmability of the TeraPool design, but offers more energy

efficiency at the same area cost. To compromise peak performance with flexibility, in our final design some of the Snitch Tiles in TeraPool are replaced by RedMulE Tiles. We refer to this modified TeraPool as **TensorPool**. The RedMulE Tile is represented in fig. 8. A Snitch core is kept to program the accelerator, RedMulE memory ports are connected to the local interconnect and to the Tile remote request interface, so as to access also the remote portions of L1 in the cluster. Snitch and RedMulE Tiles share the same L1 memory, thanks to TeraPool's low-latency interconnect.

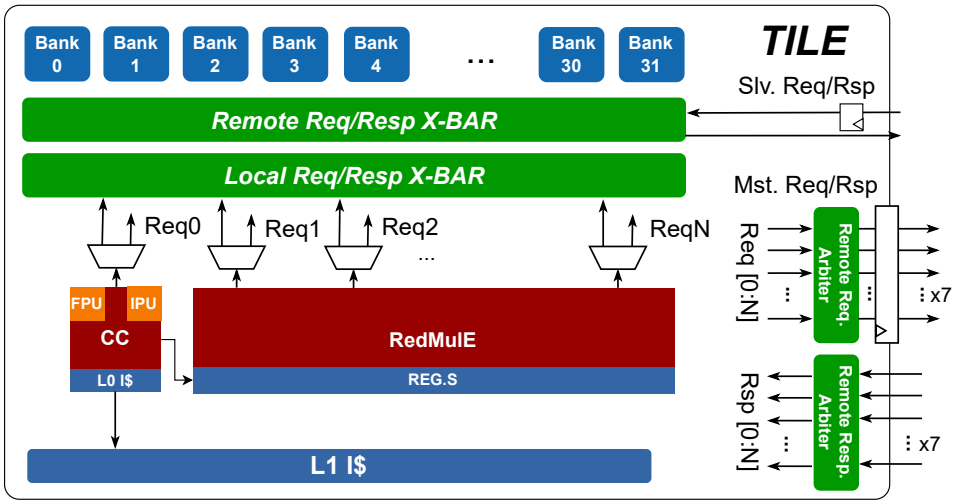


Figure 8: Block diagram of the RedMulE Tile.

Given these premises, we need to determine system-level requirements (peak compute per cluster, peak cluster-NoC bandwidth, peak cluster L1 size). Given the system level requirements we can design the cluster interconnect.

5.1 System-level requirements

At the system level we need to evaluate the peak-performance required to run the NextRAN-AI model in a TTI, and determine the configuration that allows to achieve the required peak-performance per cluster. Once the peak performance is determined, we need to further determine the L1 memory size per cluster and the cluster-NoC bandwidth, to ensure that model execution is not memory bound. Assuming double buffering we can choose the L2 memory type (e.g. SRAM, HBM, ...) and size based on the compute latency.

5.1.1 Peak-performance

Each model requires 4.2 billion operations. The model runs in 0.5ms. It is clear that the peak performance cannot be reached by a single cluster. The current configuration of TeraPool assumes a single cluster of 1024 programmable processors, which can execute transformer workloads in parallel with an average utilization of 70%, as demonstrated by [20]. In the current implementation in GF12 the maximum operating frequency of the cluster can be

established equal to 920MHz. In such framework the runtime of a model can be estimated as $4.2G/(1024 \times 0.7 \times 920MHz) = 6.36ms$.

Therefore, we assume multicluster configurations and we evaluate how many clusters would be necessary to achieve the required runtime performance. We assume full FP16 utilization, zero transfer overhead: the architecture delivers its peak performance. We assume operation at 1GHz in a 7nm process (920MHz in 12nm). Given the total number of FLOPS=FLOP/s required by a model and the $\text{FLOPS}_{\text{cluster}} = FLOP_{\text{cluster}}/s$ offered by the compute fabric:

$$N_{\text{clusters}} = \frac{\text{FLOPS}}{\text{FLOPS}_{\text{cluster}}} \tag{1}$$

Peak-Performance [FLOP/cycle/cluster]	1024	2048	4096	8192
NumClusters	115	58	29	14

Table 3: Sweep of the number of clusters as a function of the peak-performance achieved within a cluster.

From the previous analysis, we obtain the number of clusters in the NoC as a function of the peak performance per cluster in table 3. On the one hand, clusters with low peak performance would result in a large total number of clusters in the NoC, which increases data splitting and the traffic in the network. Very large mesh topologies tipically incur in this case in intense data traffic in the central network switches. On the other hand, clusters with very large compute units, able to provide up to 8192 FLOPs/cycle can incur under-utilization. We set us in the sweet spot of 4096 FLOPs/cycle and we round up the total required clusters to 32.

A more accurate estimate for the cluster peak performance takes into account the amount of computation allocated to cores and to RedMulEs, respectively α_{core} and α_{RM}. We can use the numbers from section 4.3. In this case we know that most of the 4096 FLOPs/cycle required to each cluster should be provided by matrix-multiplication specific datapaths, to optimize PPA.

$$N_{\text{clusters}} = \max\left(\frac{\alpha_{RM}\text{FLOPS}}{\text{FLOPS}_{RM}}, \frac{\alpha_{cores}\text{FLOPS}}{\text{FLOPS}_{cores}}\right) \tag{2}$$

It is also possible to include into the discussion the amount of compute required by the wireless signal processing itself: OFDM demodulation i.e. Fast Fourier Transform (FFT), beamforming, least-squares channel estimation and MIMO MMSE detection. We choose a 100MHz, $N_{TX} = 256$ transmitters configuration and a TTI with 14 data symbols and 2 Demodulation & Reference Symbols (DMRSs). We assume that beaforming will be executed on the RedMulE tensor cores, while the rest of the workload can be handled by Snitch cores. Evaluating the computational complexity of the processing similarly to what was done for the AI-models we obtain 1.8 GFLOP per TTI. With such a large number of antennas and beams the computational complexity is mostly allocated to beamforming tasks (89.8%). The processing requirements can be adjusted as in table 4:

It should be noted that the NextRAN-AI processing increases of 33× the computational complexity of the classical wireless processing chain. Although not in scope for this project,

	Complexity/TTI [FLOP]	Fraction MatMul	Fraction Others
NextRAN-AI	N×(4.2G)	96.8%	3.2%
Wireless Processing	1.8G	89.8%	10.2%
TOT	60.5G	96.6%	4.4%

Table 4: Computational-complexity, compute-performance and memory-footprint of the NextRAN-AI model.

a reduction of the model complexity should be traded off with the BER vs signal to noise ratio performance. Nevertheless this result confirms that the number of clusters and the peak performance per cluster required to run jointly the wireless processing and the AI processing set respectively at 32 and 4096 FLOPs/cycle.

5.1.2 L1-size

To determine the L1 memory size, a worst-case analysis is performed assuming single-cluster execution of the NextRAN-AI model. The memory usage of each layer includes its input, weights and output, with the output buffer reused as input for the next layer. Skip connections require retaining intermediate outputs, which contributes to peak memory usage. Thus, the L1 size is determined by evaluating the input, weight, output, and skip connection sizes, for the longest skip connection:

$$M = \sum_{l=0}^{L} W_l + \max_{l \in [0,L]} (I_l + O_l) + S$$

W_l denotes the number of weights in layer l. The summation $\sum_{l=0}^{L} W_l$ represents the total memory required to store the model parameters. The term $\max_{l \in [0,L]}(I_l + O_l)$ estimates the peak memory usage due to input and output activations, assuming layer-wise sequential processing, S is the memory required to store the skip connection.

As can be clearly seen in the formula, the L1 memory size depends only on the model characteristics. In particular, we obtain a requirement of approximately 3.2MB, which we round up to 4MiB.

5.1.3 Cluster-NoC bandwidth

To evaluate the Cluster-NoC bandwidth we enforce the memory balance between the cluster running at peak performance and the NoC. Matrix multiplication is the most typical operator in our workload, as previously observed. The analysis assumes execution of a $C = A \times B$ squared matrix multiplication: $A \in \mathbb{R}^{M \times N}, B \in \mathbb{R}^{N \times P}$, $C \in \mathbb{R}^{M \times P}$, and $M = N = P$. We rely on double-buffering to overlap transfers and compute. The problem size of the matrix multiplication in bytes is $2 \times 2 \times (MN + NP + MP)$. The first factor 2 accounts for double-buffering and the second factor 2 accounts for 16b precision (each data item is 2 Bytes). We choose the problem size to completely fill the cluster L1 memory:

$$
\begin{aligned}
&ProblemSize = 2 \times 2 \times (MN + NP + MP) = 12 \times N^2 \\
&ProblemSize = Mem \implies N = \frac{\sqrt{Mem}}{2\sqrt{3}}
\end{aligned}
\tag{3}
$$

To obtain the cluster to NoC bandwidth we make sure the transfer time is less than the compute time at peak performance and peak utilization. In practice, this ensures that in a double buffering regime the compute units will never wait for memory transfers to complete:

$$
\begin{aligned}
&T_{\text{transfer}} = \frac{ProblemSize}{BW} = \frac{12 \times N^2}{BW} \\
&T_{\text{compute}} = \frac{ProblemFLOPs}{\text{FLOPS}_{\text{RM/cores}}} = \frac{M \times N \times P}{\text{FLOPS}_{\text{RM/cores}}} = \frac{N^3}{\text{FLOPS}_{\text{RM/cores}}} \\
&T_{\text{transfer}} \leq T_{\text{compute}} \implies \frac{12 \times N^2}{BW} \leq \frac{N^3}{\text{FLOPS}_{\text{RM/cores}}} \\
&BW \geq \frac{12 \times \text{FLOPS}_{\text{RM/cores}}}{N} = \frac{24\sqrt{3} \times \text{FLOPS}_{\text{RM/cores}}}{\sqrt{Mem}}
\end{aligned}
\tag{4}
$$

Notice that the requirement on BW is more stringent by considering the peak performance provided by RedMulE, because it will be higher than the peak performance provided by the cores. Moreover, it does not make sense to sum the two. In fact, RedMulE and the cores generally do not work on the same matrix multiplication. It is way more common to have the cores working on another part of the processing, for example an activation. In this case, the memory balance should take into account the slowest between RedMulE and the cores in compute:

$$
BW \geq \frac{12 \times N^2}{\max\left(\frac{N^3}{\text{FLOPS}_{\text{RM}}}, \frac{N}{\text{FLOPS}_{\text{cores}}}\right)}
$$

However, given the assumptions on N, RedMulE is always the slowest compute element for any number of Snitch in the cluster. In this case, the total read&write bandwidth required from a cluster to the interconnect is ≥ 83 GBps $= 664$ Gbps.

5.1.4 Main Memory and NoC-Main-Memory Bandwidth

Our L2 memory is the system main memory because we assume that the large scale of TeraPool L1 memory allows to efficiently hide the latency of the main memory by double buffering. To verify this, related to our previous matrix multiplication assumption and referring to fig. 7, we need to demonstrate the chosen memory technology can deliver up to $\geq 8 \times 664$Gbps on the upper/bottom sides, and $\geq 4 \times 664$Gbps on the left/right sides of the die.

Most of the commercially available platforms for baseband processing are bound to DDR memory technology, as reported in table 5. Based on this technology limitation, while we keep the assumption on matrix multiplication to define the bandwidth between clusters, we need to reevaluate the assumption at the chip boundary. In this case it is enough to assume that the

main memory will be able to transfer in the system the data of a TTI in the required 0.5ms timeframe.

In our demanding use case, where we assume 256 antennas, 100MHz transmission bandwidth, 14 symbols, 2 DMRS, and 16b in-phase and quadrature data, the total input data for a TTI is 60 MB. We first notice that the full system with 32 clusters, given the specifications on L1 size in section 5.1.2, can accomodate up to 2 TTIs, which makes our first assumption valid. We then observe that to transfer a TTI in 0.5ms, the main-memory must have 1920 Gbps read&write bandwidth. Under the assumption that a DDR5 link can provide up to 563.2 Gbps at full bandwidth, we conclude that we need to plug at least 8 DDR5 memories in the NoC fabric, to keep a safe margin of 50% maximum utilization.

	Memory Class	Freq. [GHz]	channels	Data-Rate [Gbps]
PC802	LPDDR4	3.2	1	204.8
Qualcomm X100	LPDDR5	3.2	1	409.6
Marvell Octeon	DDR5	2.8	6	2150.4
Ours	DDR5	4.4	8	4505.6

Table 5: Main Memory technologies for commercial baseband platforms.

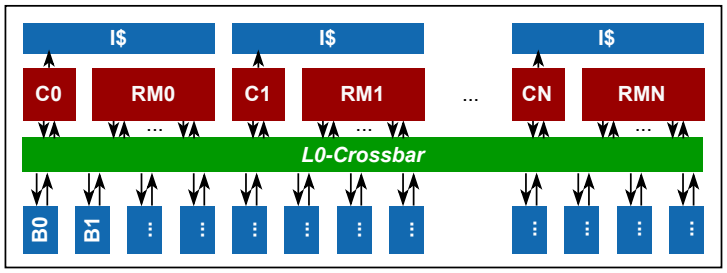
5.1.5 Optional on-chip L2 and potential for 3D integration

Since the main-memory bandwith was tuned according to the input and output data size for a TTI we must assume that the trasformer model weights, and other parameters needed by wireless processing, such as pilots for least-squares estimation and beamforming coefficients will be kept on chip, together with the intermediate outputs of the processing. In the proposed system, there are 32 clusters with 4MiB of memory each. The distributed L1 can be used as a buffer for the intermediate outputs of the computation and a processing pipeline can be created where clusters transfer data between each other without the need to resort to expensive main-memory transfers.

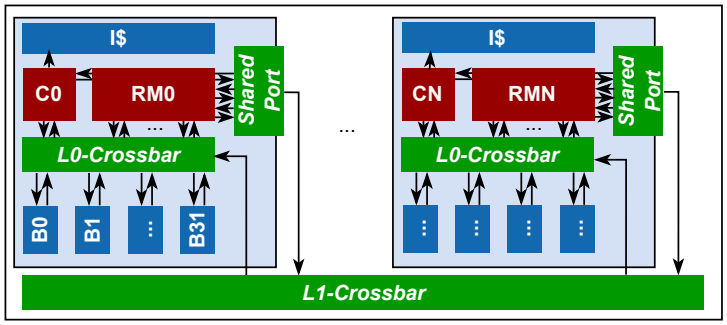
Ad intermediate memory hierarchy might be however necessary when switching the function of cluster at runtime, to repurpose them from pure physical layer computations to the processing of the transformer model. In this case the model weights residing in the cluster L1 should be loaded from main memory, which would conflict with the continuous stream of the TTI input-outputs. To cover this case the implementation of an intermediate L2 buffer could be considered. The size of this buffer should be enough to contain the model weights. Additionally we can assume the L2 to be connected to an NoC node, which would require the same 664Gbps bandwidth delivered between clusters. Similarly to how the OPs are estimated we computed the memory footprint of a transformer model equal to 180KiB. Providing each cluster with the possibility to temporarily store the model parameters, we obtain that 8MiB of L2 are required.

To maximize the area efficiency and bandwidth of the low-latency L2 buffer, we will orient the design of the main compute die to be 3D stacked with an L2 memory die. The methodology

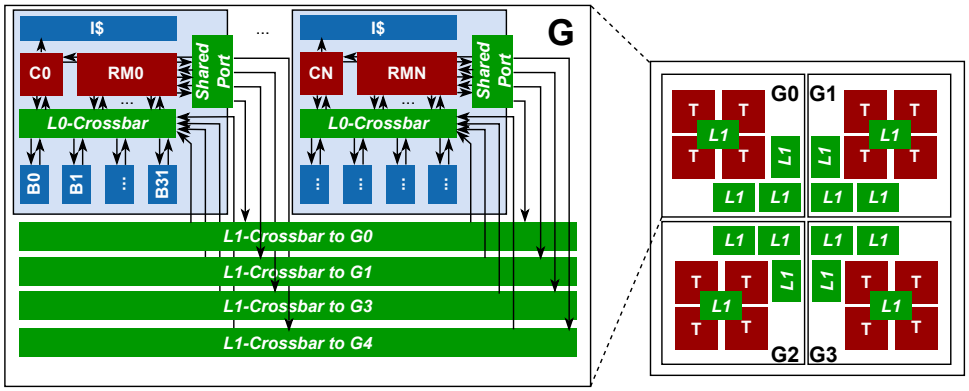
will follow the proposal described by [21]. We will implement the cluster's 2D floorplan and propose heuristics to optimally place the TSVs signal and power connections. Regarding timing: we will design the system to initiate all transactions to L2 on the bottom die to simplify timing analysis, and we will evaluate timing closure across dies by annotating the upper L2 die with the insertion delay measured on the bottom die. A similar approach, if successful, could also be surveyed for the L1 of the cluster.



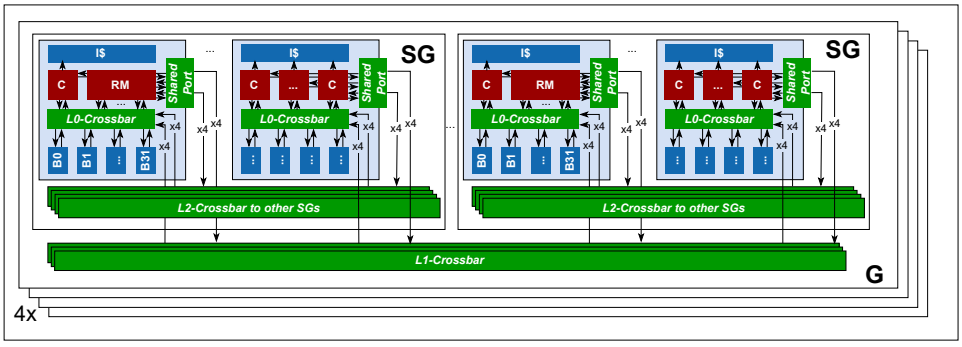
(a)



(b)



(c)



(d)

Figure 9: Interconnects of the TensorPool cluster.

5.2 Interconnect sizing

To build a shared-memory cluster capable of achieving 4k FLOP per cycle, we propose integrating multiple RedMulE tiles with Snitch tiles. The design seeks to balance performance

and programmability, raising the question of how much area and power can be allocated to flexibility (i.e., the Snitch tiles) without compromising the overall efficiency of the cluster. A key focus is on engineering the tile-to-memory interconnect to ensure sufficient bandwidth delivery to the RedMulE tiles. We raise the following design hypothesis:

1. We select a 16×16 RedMulE array as the most cost-effective configuration, achieving the highest ratio of FLOP per cycle (256 FLOP/cycle) per bandwidth (512b/cycle).

2. To maintain efficient memory access, we guarantee that RedMulE ports can access full bandwidth to a subset of memory banks, in order to reduce contention.

3. The complexity of the interconnect is limited to a maximum of 32×32 to ensure scalability and manageable implementation. The feasibility of this interconnect size was demonstrated in previous work [22] for GF12nm technology, with an achieved target frequency of 920MHz in tipycal corner. The evaluation in 7nm technology decided for this project will support the more ambitious target of 1GHz.

4. if hierarchical structures beyond individual tiles are introduced (e.g., at the Group or SubGroup level), we reproduce the hierarchy instance by multiples of 4, to preserve central symmetry in the floorplan, facilitating regular layout and balanced connectivity.

From these guidelines, all possible configurations are reported in table 6. Configuration 1 corresponds to the baseline TeraPool, the other TensorPool cluster configurations are reviewed in the following points:

- If we consider clusters with RedMulEs only the first option is a flat connection of RedMulEs to the banks, as in fig. 9 (a). Since at least 16 RedMulEs are necessary to achieve 4k FLOP per cycle, this connection is not possible, because it would result in a 512x512 interconnect, as in configuration 2. This proves that the Tile level is necessary.

- Since at least 512b/32b banks are necessary to connect the RedMulE ports to the banks in a Tile the only possible configuration of the RedMulE Tile has one RedMulE per Tile, as in fig. 9 (b), and configuration 4. With two RedMulEs the Tile interconnect becomes 64x64, which is unfeasible, as in configuration 3.

- Adding the Group hierarchy, as in fig. 9 (c) increases the bandwidth that RedMulE can inject in the shared interconnect, as in configuration 5: the connection between Tiles is replicated four times. It is worth noticing that the floorplan takes advantage of hypothesis 4, because the group is repeated four times. The conditions between groups are similar as in TeraPool.

- Adding the Snitch Tiles, the bandwidth of RedMule in the Group interconnect must be reduced, moreover, the number of instances to place and route in the Group floorplan grows and complicate the design, as in configuration 6. Hence, we resort to the SubGroup

hierarchy as in fig. 9 (d). The connection between SubGroups is similar to that in TeraPool. In this configuration the RedMules have more bandwidth to local SubGroups and shared bandwidth with Snitch Tiles to the remote SubGroups. This corresponds to configuration 7.

With understanding from the TeraPool project and based on the previous discussion we select the configurations 6 and 7 in table 6 as the most promising for the physical implementation.

	FLOP/ cycle	NumCores	NumRedMulEs	NumBanks Tile	NumTiles	NumGroups	NumSubGroups	NumRMPorts in G-Xbar	NumRMPorts in SG-Xbar	Level-0	Level-1	Level-2
1	1024	1024	0	32	128	4	4	0	0	8x32	32x32	8x8
2	4112	16	16	512	1	0	0	0	0	512x512	0	0
3	4112	16	16	32	8	0	0	0	0	64x64	0	0
4	4112	16	16	32	16	0	0	0	0	32x32	0	0
5	4112	16	16	32	16	4	0	8	0	32x32	32x32	0
6	4496	400	16	32	64	4	0	4	0	32x32	32x32	0
7	4496	400	16	32	64	4	16	4	8	32x32	32x32	12x12

Table 6: Configurations of the TensorPool interconnect with RedMulE Tiles and Snitch Tiles.

5.3 RedMulE latency estimate

An open point in the interconnect configuration is the latency that RedMulE memory requests must tolerate to ensure a continuous operation of the compute elements. To verify that the selected configurations allow to keep low-latency access to the shared-memory for RedMulE, we develop a semi-analytical model of the interconnect, representing each node of the hierarchy as $N \times K$ switches: each K output port can serve only one of the N initiators at a time, other concurrents wait in a list. In each cycle, the masters post a request on the crossbar to a random memory address with probability $req/cycle$. RedMulE masters will access the subsequent 512b to a randomly selected memory address. We can measure the rate of $resp/core/cycle$ and the average latency of a response.

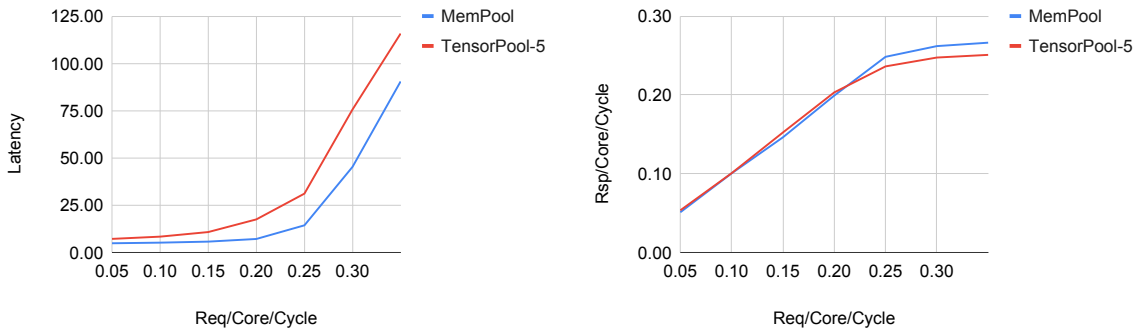


Figure 10: Latency and Throughput of the interconnect for the TensorPool-5 configuration.

Figure 10 represents a comparison between MemPool [14] and the the TensorPool-5 configuration. The two configurations have the same interconnect hierarchies (Tile, Group, Cluster). Initiators achieve similar performance, respectively 0.27 and 0.25 maximum throughput. A continuous operation of RedMulE requires fetching 512b at consecutive positions in memory

every 4 cycle. The experiment executed on the analytical interconnect model for this particular case yields 10 latency cycles on average for a response. In TensorPool the RedMulE memory port will therefore require modifications to support longer latency accesses.

5.4 Target architecture configuration & PPA estimation

The previous analysis delivers two possible cluster configurations, that are reported in table 7, alongside the baseline TeraPool configuration. We also report an estimate of the area of the cluster based on synthesis runs of the TeraPool Tile in 12nm and of RedMulE in 7nm, adapted to 7nm by means of technology scaling. We take into account the interconenct area based on the result for the place and route of TeraPool in 12nm. We finally report estimates from technology scaling of power simulations of a matrix multiplication on a RedMulE instance placed and routed in 22nm and on the 12nm placed and routed TeraPool cluster. The energy efficiency estimate is a projection that takes into account the relative number of FLOP for each compute unit (Snitch or RedMulEs) in the final architecture.

Finally, we report possible system configurations, including different number of TensorPool and TeraPool clusters. The configurations with only TeraPools is not feasible on a single chip. The configuration with TeraPools and TensorPools-5 requires at least 3 TeraPools to complete the parts of the workload that are not matrix multiplication dominated. As such it would require splitting of the workload across heterogeneous clusters. The TensorPools-7 configuration is therefore the most promising for the execution of the full workload on a single chip, with significant parts of the processing executed on a single cluster without workload splitting.

Cluster Configuration	TeraPool	TensorPool-5	TensorPool-7
FLOP/cycle/cluster	1024	4112	4496
NumCores	1024	16	400
NumRedMulETiles	0	16	16
NumSnitchTiles	128	0	48
NumTiles	128	16	64
NumGroups	4	4	4
NumSubGroups	16	0	16
Area-Cluster (TSMC7) [mm^2]	27.81	5.26	20.26
Energy-Efficiency (TSMC7) [$GFLOP/s/W$]	276.48	4743.00	1946.04
Area-Efficiency (TSMC7) [$GFLOP/s/mm^2$]	36.84	779.16	221.89
Energy/Area-Efficiency (TSMC7) [$GFLOP/s/W/mm^2$]	9.94	902.26	65.92
System Configuration	**TeraPools**	**TensorPools-5 + TeraPools**	**TensorPools-7**
NumTensorPoolClusters	-	29	32
NumTeraPoolClusters	60	3	-
Cluster L1-size	4MiB	4MiB	4MiB
Cluster-NoC BW	664 Gbps	664 Gbps	664 Gbps
Cluster-L2 BW	4505.6 Gbps	4505.6 Gbps	4505.6 Gbps
Area-TOT (TSMC7) [mm^2]	1779.71	235.87	648.40

Table 7: Configurations of TeraPool and TensorPool. Different configurations for a TeraPools/TensorPools multicluster system.

In table 8 we report the PPA comparison between RedMulE and SoA RISC-V tensor-core accelerators. We further report a comparison between our TensorPool-5 and TensorPool-7 architectures with leading edge A100 Graphics Processing Unit (GPU) [23].

	Tech.	Precision	Supply [V]	Freq. [MHz]	L1-TOT [MB]	Power [mW]	Area [mm^2]	GFLOPS/s	GFLOPS/s/W	GFLOPS/s/mm^2
[24]	16nm	INT8	-	200	-	40	0.62	204	4680	329
[25]	12nm	INT8	0.68	800	-	43.8	6.55	52400	15000	8000
RedMulE	7nm	FP16	0.75	1000	-	-	0.13	256	-	1914
[23]	7nm	FP16	-	1410	27	300×10^3	632	312000	1040	493.67
TensorPool-5	7nm	FP16	0.75	1000	128	27×10^3	235.87	131072	4743	555.68
TensorPool-7	7nm	FP16	0.75	1000	128	72×10^3	648.40	143872	1946	221.89

Table 8: Comparison of TensorPool with SoA architectures for NN-inference.

5.5 Next steps

In the next steps of the project:

- We will develop an RTL description of the most promising TensorPool configurations and we will target the implementation in the TSMC N7 node.
- We will analyze further the workload and provide C-microkernel implementations of the basic operators to be run on the TensorPool programmable accelerator.
- We will provide the PPA for the execution of this operators on a placed&routed TensorPool instance, by means of RTL-simulation and post-layout power simulation.
- We will evaluate the whole system performance relying on an higher level simulator (e.g. the GVSoC simulator), to speed-up verification and performance analysis at a system level, without the time-constaints of RTL-simulation. We will therefore be able to provide results on the end-end system performance.

Acknowledgments

Huawei Technologies Sweden AB supported this work.

References

[1] Walid Saad, Mehdi Bennis, and Mingzhe Chen. “A vision of 6G wireless systems: Applications, trends, technologies, and open research problems”. In: *IEEE network* 34.3 (2019), pp. 134–142.

[2] Emil Björnson, Jakob Hoydis, Luca Sanguinetti, et al. “Massive MIMO networks: Spectral, energy, and hardware efficiency”. In: *Foundations and Trends® in Signal Processing* 11.3-4 (2017), pp. 154–655.

[3] Chuan Zhang et al. "Artificial intelligence for 5G and beyond 5G: Implementations, algorithms, and optimizations". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 10.2 (2020), pp. 149–163.

[4] Mikko Honkala, Dani Korpi, and Janne MJ Huttunen. "DeepRx: Fully convolutional deep learning receiver". In: *IEEE Transactions on Wireless Communications* 20.6 (2021), pp. 3925–3940.

[5] Dani Korpi et al. "DeepRx MIMO: Convolutional MIMO detection with learned multiplicative transformations". In: *ICC 2021-IEEE International Conference on Communications*. IEEE. 2021, pp. 1–7.

[6] Reinhard Wiesmayr et al. "Design of a standard-compliant real-time neural receiver for 5G NR". In: *arXiv preprint arXiv:2409.02912* (2024).

[7] Sebastian Cammerer et al. "A neural receiver for 5G NR multi-user MIMO". In: *2023 IEEE Globecom Workshops (GC Wkshps)*. IEEE. 2023.

[8] *Data Processing Units, Empowering 5G carrier, enterprise and AI cloud data infrastructure*. URL: https://www.marvell.com/products/data-processing-units.html.

[9] 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: *GLSVLSI '24: Proceedings of the Great Lakes Symposium on VLSI 2024*. 2024. DOI: 10.1145/3649476.3658735.

[10] *How we Won the Acceleration Architecture Debate*. URL: https://www.qualcomm.com/news/onq/2023/03/how-we-won-the-acceleration-architecture-debate.

[11] NVIDIA Corporation. *NVIDIA H100 Tensor Core GPU Architecture*. White Paper. Accessed: 2024-05-29. NVIDIA Corporation.

[12] Nevine Nassif et al. "Sapphire rapids: The next-generation intel xeon scalable processor". In: *2022 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE. 2022.

[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 (2021), pp. 1845–1860. DOI: 10.1109/TC.2020.3027900.

[14] 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.

[15] Marco Bertuletti et al. "Fast Shared-Memory Barrier Synchronization 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*. 2023, pp. 241–254. DOI: {10.1007/978-3-031-46077-7_16}.

[16] Dianxin Luan and John Thompson. *Attention Based Neural Networks for Wireless Channel Estimation*. 2022. arXiv: 2204.13465 [eess.SP]. URL: https://arxiv.org/abs/2204.13465.

[17] Shuangshuang Li and Peihao Dong. "Mixed Attention Transformer Enhanced Channel Estimation for Extremely Large-Scale MIMO Systems". In: *2024 16th International Conference on Wireless Communications and Signal Processing (WCSP)*. 2024, pp. 394–399. DOI: 10.1109/WCSP62071.2024.10827075.

[18] Jian Zou et al. "Deep Receiver for Multi-Layer Data Transmission with Superimposed Pilots". In: *ICASSP 2025 - 2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2025, pp. 1–5. DOI: 10.1109/ICASSP49660.2025.10890516.

[19] Yvan Tortorella et al. *RedMule: A Mixed-Precision Matrix-Matrix Operation Engine for Flexible and Energy-Efficient On-Chip Linear Algebra and TinyML Training Acceleration*. 2023. arXiv: 2301.03904 [cs.AR]. URL: https://arxiv.org/abs/2301.03904.

[20] Bowen Wang et al. "A Dynamic Allocation Scheme for Adaptive Shared-Memory Mapping on Kilo-Core RV Clusters for Attention-Based Model Deployment". In: *2025 IEEE 36th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2025, pp. 9–16. DOI: 10.1109/ASAP65064.2025.00012.

[21] H. Ekin Sumbul et al. "A 3D Design Methodology for Integrated Wearable SoCs: Enabling Energy Efficiency and Enhanced Performance at Iso-Area Footprint". In: *2025 Design, Automation Test in Europe Conference (DATE)*. 2025, pp. 1–7. DOI: 10.23919/DATE64628.2025.10992815.

[22] 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.

[23] NVIDIA Corporation. *NVIDIA A100 Tensor Core GPU*. White Paper. Accessed: 2025-07-25. NVIDIA Corporation. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf.

[24] Xiaoling Yi et al. *OpenGeMM: A High-Utilization GeMM Accelerator Generator with Lightweight RISC-V Control and Tight Memory Coupling*. 2024. arXiv: 2411.09543 [cs.AR]. URL: https://arxiv.org/abs/2411.09543.

[25] "11.3 Metis AIPU: A 12nm 15TOPS/W 209.6TOPS SoC for Cost- and Energy-Efficient Inference at the Edge". In: *2024 IEEE International Solid-State Circuits Conference (ISSCC)*. Vol. 67. 2024, pp. 212–214. DOI: 10.1109/ISSCC49657.2024.10454395.