Efficient Parallelization of 5G-PUSCH on a Scalable RISC-V Many-Core Processor

Marco Bertuletti
ETH Zürich
Zürich, Switzerland
mbertuletti@iis.ee.ethz.ch

Yichao Zhang
ETH Zürich
Zürich, Switzerland
yiczhang@iis.ee.ethz.ch

Alessandro Vanelli-Coralli
ETH Zürich
Zürich, Switzerland
Università di Bologna
Bologna, Italy
avanelli@iis.ee.ethz.ch

Luca Benini
ETH Zürich
Zürich, Switzerland
Università di Bologna
Bologna, Italy
lbenini@iis.ee.ethz.ch

Abstract—5G Radio access network disaggregation and softwarization pose challenges in terms of computational performance to the processing units. At the physical layer level, the baseband processing computational effort is typically offloaded to specialized hardware accelerators. However, the trend toward software-defined radio-access networks demands flexible, programmable architectures. In this paper, we explore the software design, parallelization and optimization of the key kernels of the lower physical layer (PHY) for *physical uplink shared channel* (PUSCH) reception on MemPool and TeraPool, two manycore systems having respectively 256 and 1024 small and efficient RISC-V cores with a large shared L1 data memory. PUSCH processing is demanding and strictly time-constrained, it represents a challenge for the baseband processors, and it is also common to most of the uplink channels. Our analysis thus generalizes to the entire lower PHY of the uplink receiver at gNodeB (gNB). Based on the evaluation of the computational effort (in multiply-accumulate operations) required by the PUSCH algorithmic stages, we focus on the parallel implementation of the dominant kernels, namely fast Fourier transform, matrix-matrix multiplication, and matrix decomposition kernels for the solution of linear systems. Our optimized parallel kernels achieve respectively on MemPool and TeraPool speedups of 211, 225, 158, and 762, 880, 722, at high utilization (0.81, 0.89, 0.71, and 0.74, 0.88, 0.71), comparable a single-core serial execution, moving a step closer toward a full-software PUSCH implementation.

Index Terms—Many-core, RISC-V, 5G, OFDM, MIMO

I. Introduction

To provide increased flexibility, performance, and efficiency, the 5G standard foresees the introduction of novel features in its air-interface, known as new radio (NR), such as larger bandwidths, higher spectrum frequencies, increased massive multi-user multiple-input multiple-output (MIMO), beamforming, etc. [1]. These enhancements require the processing of high-dimensional signals in a fraction of milliseconds. Over the last few years, a wide range of baseband processing application-specific integrated circuits (ASICs) [2]–[4] have been proposed. Industry stakeholders are, however, moving towards more flexible solutions based on radio access network (RAN) disaggregation and softwarization [5] to improve the time-to-market in diverse deployment scenarios.

A key direction in RAN softwarization and disaggregation is to exploit open software and hardware platforms, to ensure long-term scalability, to speed up the adoption of innovative community-developed solutions, and to reduce vendor captiv-ity issues. The RISC-V instruction set architecture (ISA) plays a strategic role in this context by enabling open software and hardware architectures and designs, without the constraints imposed by proprietary instruction sets. In this paper, we focus on the PUSCH lower PHY of the uplink receiver at the gNB by exploring the feasibility of implementing it on MemPool [6] and its scaled-up version TeraPool, two clusters of respectively 256 and 1024 fully programmable RISC-V cores with a shared low latency access L1 memory. The PUSCH lower PHY is indeed one of the most challenging processing parts of the entire receiving chain. The main contributions of this paper are:

- The identification of the most computationally complex kernels of PUSCH lower PHY;
- A local memory access parallel implementation of these key kernels, reducing the memory-related stalls to less than 10% of the execution time, in MemPool and TeraPool;
- A flexible scheduling policy that enables executing kernels on subsets of the cluster's cores, supported by the implementation of barriers for partial group synchronization;
- The evaluation of the speedup of our parallel software-defined PUSCH chain, compared to a single core serial execution, and of the achievable efficiency in terms of processor utilization and stall reduction.

The implemented parallel kernels achieve respectively on MemPool and TeraPool speedups of 211, 225, 158, and 762, 880, 722, at utilizations 0.81, 0.89, 0.71, and 0.74, 0.88, 0.71. The speedup obtained on the whole processing chain is 871. The execution time, constrained to a realistic clock frequency of 1GHz is 0.785ms, which is close to the 0.5ms per transmission specified by the 5G PUSH standard. Our analysis thus shows that a RISC-V-based "pool of processors" architecture, whose implementation feasibility was demonstrated in [6], is a promising candidate for a parallel software implementation of PUSCH on programmable cores.

II. 5G PUSCH Kernels Complexity

This section reviews the key kernels in PUSCH processing. Fig. 1 represents the reference PUSCH lower PHY receiving chain. PUSCH transmission is based on orthogonal frequency

division multiple access (OFDMA) [7]. User equipments (UEs) are multiplexed on a time and frequency grid (Fig. 2). Each orthogonal frequency division multiplexing (OFDM) symbol consists of N_{SC} orthogonal sub-carriers. N_{symb} are sent during one slot transmission. PUSCH may be interleaved in time and frequency with other channels, however, in the worst case for PUSCH computational complexity the whole spectrum is allocated to this channel. OFDM symbols are received by a set of N_R antennas.
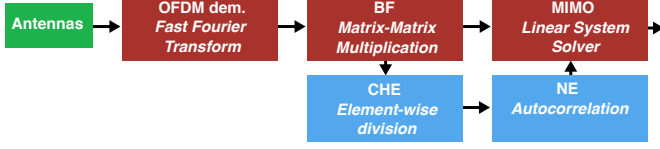

Fig. 1. PUSCH processing chain steps: OFDM demodulation, beamforming (BF), MIMO, channel estimation (CHE) and noise estimation (NE). The steps involving pilot symbols are reported in blue.
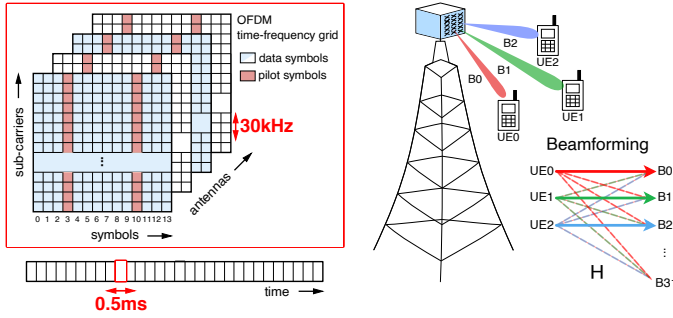

Fig. 2. Time-frequency grid of an OFDM system and Beamforming.

At the beginning of the baseband digital signal processing (DSP) chain, the signal received by each antenna is translated to the frequency domain via a Fast Fourier Transform (FFT). The complexity of this stage can be estimated as $N_{SC} \times log(N_{SC})$ complex multiply and accumulate operations (MACs), and the kernel is run for each antenna and each OFDM symbol. As shown in Fig. 2, beamforming linearly combines the signal received by different antennas and creates N_B receiving beams. This results in a matrix-matrix multiplication (MMM) with known coefficients, that requires $N_R \times N_B \times N_{SC}$ complex MACs for each OFDM symbol. After beamforming, a $\mathbf{y} \in \mathbb{C}^{N_B}$ signal is obtained for each sub-carrier. The relation between this signal and the $\mathbf{x} \in \mathbb{C}^{N_L}$ signal transmitted by N_L UEs can be modeled as:

$$\mathbf{y} = \mathbf{H}\mathbf{x} + \mathbf{n} \qquad (1)$$

where $\mathbf{H} \in \mathbb{C}^{N_B \times N_L}$ is the channel matrix and $\mathbf{n} \in \mathbb{C}^{N_B}$ is additive white gaussian noise. In the MIMO stage, the transmitted signal is extracted from the received signal through least minimum mean squared error estimation. Before this step, the channel matrix and the noise variance are estimated. Introducing the variance of the Gaussian noise σ^2, the identity matrix $\mathbf{I}$, the estimated channel matrix $\hat{\mathbf{H}}$, its hermitian $\hat{\mathbf{H}}^H$,

and the Gramian matrix $\mathbf{G}$, the MIMO stage of PUSCH consists of the following:

$$\mathbf{x} = \left(\hat{\mathbf{H}}^H\hat{\mathbf{H}} + \sigma^2\mathbf{I}\right)^{-1}\hat{\mathbf{H}}^H\mathbf{y} = \mathbf{G}^{-1}\hat{\mathbf{H}}^H\mathbf{y} \qquad (2)$$

As suggested in [8], the computationally intensive matrix inversion required by MIMO can be avoided by resorting to a Cholesky decomposition of matrix $\mathbf{G}$, followed by the solution of two triangular systems. The complexity of these steps is respectively $N_L^3/3$ and $2N_L^2$, for each sub-carrier and each data OFDM symbol. The channel matrix and the variance of noise used in (2) are estimates based on pilot symbols known both at the base-station and at the UE side.
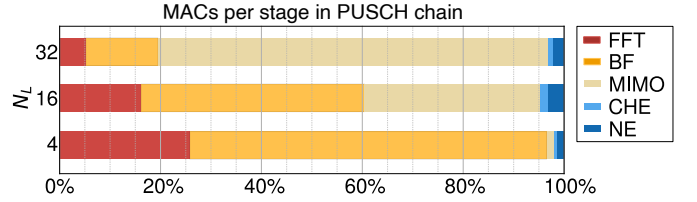

Fig. 3. Complex OFDMs allocated to each PUSCH stage for different number of UEs transmitting at the same frequency.

TABLE I
PUSCH KERNELS AND COMPUTATIONAL COMPLEXITY

PUSCH stage	Key kernel	Complex MACs
OFDM dem.	Fast Fourier transform	$N_{symb} \times N_R \times N_{SC} \times log(N_{SC})$
BF	Matrix-matrix multiplication	$N_{symb} \times N_{SC} \times N_R \times N_B$
MIMO	Cholesky decomposition	$N_{data-symb} \times N_{SC} \times N_L^3/3 \times 2N_L^2$
CHE	Element-wise division	$N_{pilot-symb} \times N_{SC} \times N_B \times N_L$
NE	Autocorrelation	$N_{pilot-symb} \times N_{SC} \times 2N_B \times N_L$

In this paper, the block-type arrangement described in [9], is assumed and pilots are allocated to a whole OFDM symbol, as shown in Fig. 2. The channel estimation block is based on least squares estimation and it consists of an element-wise matrix division. The computational cost of this kernel is $N_B \times N_L$ MACs for each sub-carrier and for each OFDM symbol. The noise variance is estimated by computing the autocorrelation of the difference between the received signal and the expected transmission output, obtained from the estimated channel and the pilots. The complexity of this kernel is $2N_B \times N_L$ complex MACs for each sub-carrier and pilot symbol. Tab. I reports the kernels of the PUSCH, and the number of complex MACs required for each one of them.

Let us consider a typical NR use-case. According to the 3GPP NR numerology, we consider a bandwidth of 100MHz, with sub-carrier spacing 30KHz, corresponding to 3276 sub-carriers. We assume 14 symbols per transmission and 2 pilot symbols, 64 receiving antennas, and 32 beams. Fig. 3 represents the complexity allocated to each kernel of the PUSCH processing chain as a percentage fraction of the total. Most of the effort is in OFDM demodulation and beamforming stages, the impact of MIMO stage depends on the number of UEs involved. According to Amdahl's law, this analysis shows that
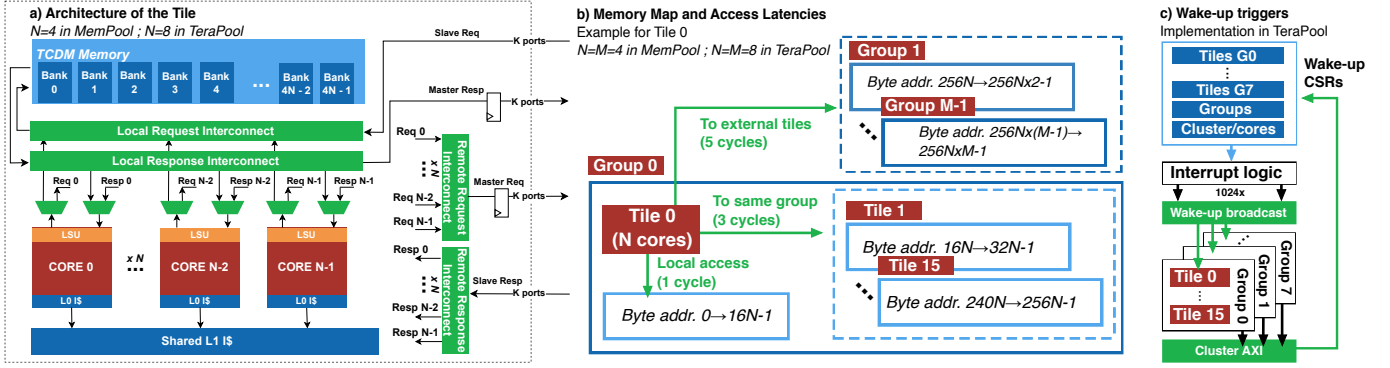
Fig. 4. (a) Architecture of tiles. (b) Access latency to each section of the cluster memory for the first tile. (c) Wake-up triggers in TeraPool cluster.

the throughput of the chain would greatly benefit from the speedup of FFT, MMM, and Cholesky decomposition.

III. MEMPOOL-TERAPOOL ARCHITECTURE

In this section, we present the hierarchical architecture of TeraPool, a general-purpose compute cluster extended from the scalable many-core architecture of MemPool [6]. The clusters compute unit is *Snitch* [10], a single-stage 32-bit RISC-V core supporting the RV32IMAFD custom extensible ISA [1]. Instructions whose execution requires more than one cycle are offloaded to pipelined functional units through a dedicated port. The load store unit (LSU) handles memory transactions and issues up to 8 outstanding loads and stores, hiding the L1 interconnect latency. Fig. 4 (a) shows the architecture of a *tile* [6], which is the first building block allowing massive replication. In MemPool each tile contains 4 Snitch cores, sharing 2 KiB of L1 instruction cache and 16 banks, with 1 KiB each of local L1 data memory. Similarly, in TeraPool a tile has 8 Snitch cores with 4 KiB instruction cache and 32 banks, with 1 KiB each of local L1 data memory. Each core in a tile accesses the local memory in one cycle, through a fully connected interconnection.

The next hierarchy level is the *group* [6]. In both MemPool and TeraPool, each group has 16 tiles. The main bottleneck in a large shared memory many-core architecture is interconnection routing. A tile-to-tile crossbar though the whole cluster would not allow physical feasibility. A 16 x 16 fully connected crossbar is thus restricted to the group level. Each tile is connected with the K groups of other tiles in the cluster through master request and slave response ports, that are used for access to remote memory banks in the same local group within 3 cycles and to remote groups in 5 cycles, as Fig. 4 (b) shows. Overall, MemPool has 256 Snitch cores, 4 groups, and 1024 1 KiB banks of L1 memory, equal to 1 MiB of SRAM. Similarly, TeraPool has 1024 cores, 8 groups, and 4096 1 KiB banks of L1 memory, equal to 4 MiB of SRAM. We do not discuss here physical implementation strategies for Terapool and Mempool, as our focus is on software design and optimization. The interested reader is referred to [6].

[1] In this paper we do not discuss PUSH specific ISA extensions: this step is left for future work.

IV. PROGRAMMING MODEL AND SYNCHRONIZATION

In this section, we present the fork-join programming model adopted for the parallel execution of the PUSCH kernels. The sequential execution of the PUSCH kernels is split into portions without data dependencies, that are executed in parallel over multiple cores of the cluster. At the end of a parallel task, cores are synchronized, ensuring the consistent write-back of the results. To execute the kernels on a subset of cores, we implement partial synchronization barriers.

When a kernel runs on the whole cluster, the cores ending a parallel task atomically increment a barrier variable and enter a wait for interrupt (WFI) sleep state. The last core incrementing the barrier variable writes in a wake-up control status register (CSR) of the system and activates a broadcasted wake-up trigger, waking up all the cores, as shown in Fig. 4 (c). A core can also selectively wake up another one, writing its ID in the wake-up CSR. This allows to synchronize a subset of cores, but the last core completing the parallel task must individually wake up the processing elements involved in the computation. To simultaneously assert a subset of the wake-up triggers, we add one CSR to selectively wake up groups and one CSR per group to selectively wake up its tiles. Enabling the wake-up of a subset of cores allows to introduce fast partial synchronization barriers. When a kernel runs in parallel on a subset of cores, the cores terminating the execution of the parallel task increment a barrier variable in their local memory. The last core completing the task sends wake-up triggers with different granularity, depending on the total number of cores involved.

V. IMPLEMENTED KERNELS

According to Amdahl's law, the key kernels that must be efficiently parallelized to boost the throughput of the PUSCH processing chain are FFT, MMM, and matrix decomposition. These kernels are implemented assuming that the input and output data reside in L1 memory and their parallelization targets the multi-banked memory structure of MemPool and TeraPool clusters. In such a large interconnected memory, contentions may occur when two cores in the same tile access the same local bank, or the same remote group. This generates stalls of the LSU and increases the access latency. The problem

is addressed, and contentions are avoided by carefully placing the data structures in memory, emphasizing local loads and stores. When local data access is impossible, the access pattern of the cores can be rearranged to avoid simultaneous access to the same group from cores in the same tile. In the following subsections, the parallel implementation of the most computationally intensive PUSCH kernels is described.

A. Fast Fourier transform

We chose a radix-4 decimation in frequency Cooley-Turkey FFT approach. The radix is chosen to ease the memory accesses in local banks of MemPool and TeraPool, where each core has 4 local banks. In the kth stage of an N-points FFT, the radix-4 butterfly gets 4 inputs at a distance $N/(4 \times 4k)$. Each core computes 4 butterflies. For a 64-points FFT the accessed elements are reported in different colours in Fig. 5. Since the input vector unrolls over the whole memory, the access to 3 out of 4 elements will likely be external and generate conflicts. The input vector is thus folded in the local banks so that each set of the $N/(4 \times 4k)$ inputs is stored in a memory row. At the end of the computation, each core stores the results with the same folding scheme in the local banks of cores that are using them in the following FFT stage. The stage-by-stage division of the FFT in smaller FFTs computed over a sub-set of cores helps reduce the synchronization overhead because only the cores producing the inputs of the same FFT for the following stage need to be synchronized. Depending on the size of the input vector only $N/4$ cores are used. The rest of the cores in the cluster are allocated to the computation of other FFTs of the same size: MemPool fits $(256 \times 4)/N$ FFTs and TeraPool $(1024 \times 4)/N$ FFTs. Cores working on different FFTs are independently synchronized.
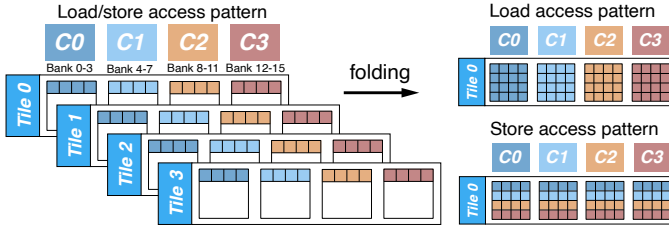

Fig. 5. FFT folded access pattern for a 64-points FFT.

B. Matrix-matrix multiplication

To efficiently handle the MMM on our architecture, 4x4 windows of the output matrix are computed at a time for two reasons. First, we achieve maximum utilization of the register files in Snitch, using all the 30 registers available for programming purposes in its ISA: 8 registers for inputs, 16 for the accumulation of temporary results, 3 for address increment, and 3 for loop control. Second, the large window size increase data reuse. Computing a 4x4 window requires 8 loads of 32-bit words per 16 MACs. This memory accesses vs. computing operations ratio is lower than respectively 12 or 16 memory loads per 16 MACs, required in a 4x2 or a 2x2 window kernel. The parallelization scheme of the implemented
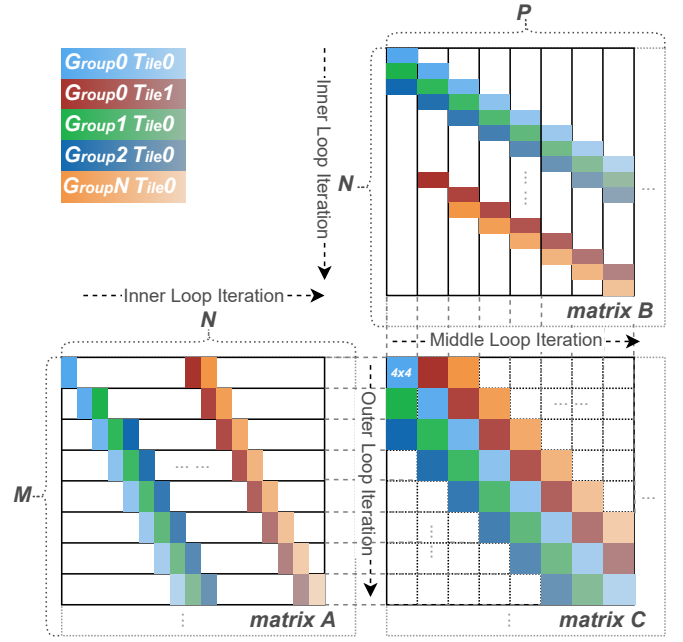

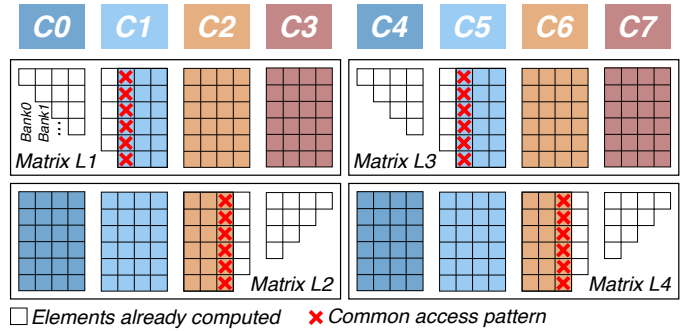Fig. 6. Kernel of MMM with 4x4 compute-window based optimization.

Fig. 7. Cholesky decomposition parallelization scheme and replication over the cluster for a 16x16 matrix.

kernel is represented in Fig. 6. An *MxN* matrix **A** and an *NxP* matrix **B** are multiplied to obtain an *MxP* matrix **C**. The kernel consists of three loops, where cores span over the whole input matrices rows and columns to compute an entire output window without reductions. In the outer loop, each core is allocated 4 rows of matrix **A**.

To maximize the utilization of cores in the cluster, cores from different tiles can operate on the same group of four rows, to generate different windows in the output matrix. A conflict occurs when cores in the same tile access data in the same group. To avoid this and fully utilize the ports for external accesses, cores from the same tile are forced to work on rows, whose elements are located in different groups. In the middle loop, a core assigned to a row of matrix **A** spans over multiple groups of 4 columns of matrix **B**, to complete the computation of multiple output 4x4 windows. If cores of the same tile generate a bank conflict, the loop starting point for one of the cores shifts to the neighboring four columns,
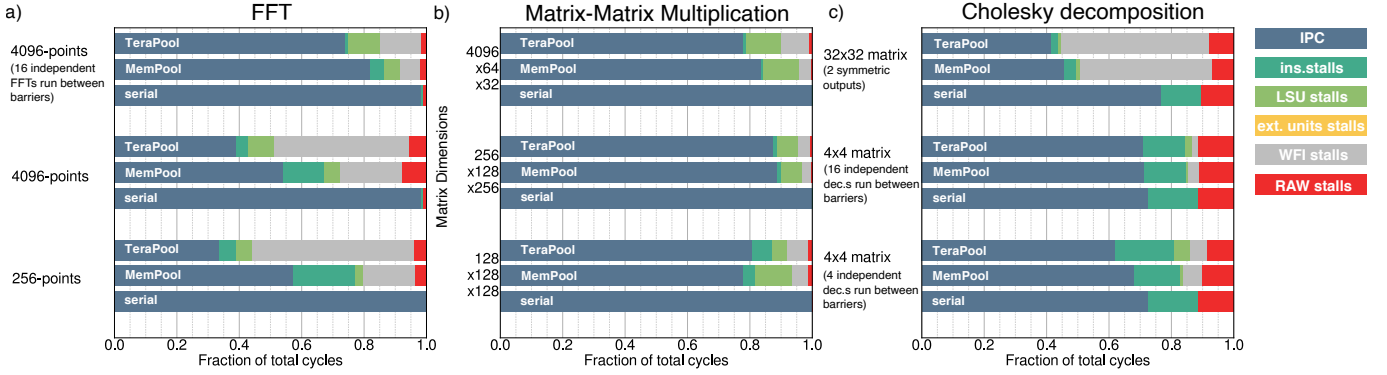
Fig. 8. Fraction of instructions and stalls over the total cycles for the PUSCH kernels.

and round-robins back to complete the loop. The inner loop specializes in the computation of the 4x4 output window. To avoid conflicts in accesses to the same locations of the output matrix, the cores working on the same output window shift their starting point on both rows and columns, then they round-robin back.

C. Cholesky decomposition

The Cholesky decomposition of matrix $\mathbf{G}$ in the lower and upper triangular matrices $\mathbf{L}$ and $\mathbf{L}^H$ follows Cholesky-Crout algorithm, which computes the output matrix $\mathbf{L}$ column by column. At each iteration, a new column of matrix $\mathbf{L}$ is generated, and all the already computed elements of a row must be accessed to produce the new row element on this column. In the parallel implementation, each core computes 4 rows in the output matrix. To avoid conflicts in the access to the elements of a row, the output matrix is folded in memory, rows are stored in the same bank. The staircase pattern of the kernel allocates more computations to the cores accessing the bottom rows of the output lower triangular matrix, unbalancing the workload and increasing the synchronization overhead. We thus replicate two instances of the kernel with different input matrices and mirrored outputs, as represented in Fig. 7. Depending on the input matrix size, the fine-grained parallelization uses a different number of cores. The remaining cores in the cluster can work on the decomposition of other matrices and be independently synchronized.

VI. RESULTS

Fig. 8 represents the instructions per cycle (IPC) of a *serial* implementation of the kernels run on a single TeraPool core and the average IPC for the parallel implementations running on MemPool and TeraPool. We also represent a breakdown of idle time due to synchronization (WFI stalls) or architectural stalls: instruction stalls, LSU stalls, stalls of the external pipelined units, and read after write (RAW) stalls. The latter originate when the register file of Snitch must wait for the output of the multiplication and division unit and the LSU. In Fig. 8 (a), the parallel implementations take into account the replication of independent FFTs, to employ all the cores of a cluster: MemPool fits 16 256-points FFTs and 1 4096-points

FFT, TeraPool fits 64 256-points FFTs and 4 4096-points FFTs. For larger input vectors the impact of synchronization overhead is reduced because there are fewer groups of cores simultaneously writing in the system CSRs to trigger an interrupt. Having a larger cluster, TeraPool has a larger fraction of WFI stalls with respect to MemPool. The same stage of different independent FFTs is run between the synchronization barriers to reduce the synchronization overhead. Running 16 independent 4096-points FFTs between the barriers we obtain 16 FFTs on MemPool and 64 FFTs on TeraPool. The IPC is increased to respectively 0.82 and 0.74.

The IPC of the single core and parallel implementations of MMM for different input dimensions are shown in Fig. 8 (b). As for the FFT, the kernel implemented on TeraPool exhibits more WFI stalls. Since the same kernel is parallelized over the entire cluster, cores in the larger TeraPool configuration get fewer instructions. This increases the fraction of instruction stalls. The few leftover LSU stalls are caused by conflicts from the cross-accesses in the two input matrices. The relative fraction of these stalls is smaller in TeraPool than in MemPool because the LSU stalls are hidden by the overlapping instruction stalls. For the 256x128x256 problem, MemPool achieves 0.89 IPC and TeraPool 0.88 IPC, which leads respectively to 145 and 558 MACs/cycle. The example use-case described in section II requires a 4096x64x32 MMM. In this case, the irregular matrix shape unbalances the workload assigned to the cores, nevertheless, MemPool achieves 0.84 IPC and 134 MACs/cycle TeraPool achieves 0.78 IPC and 487 MACs/cycle.

As shown in Fig. 8 (c), both the single-core and the parallel versions of the Cholesky decomposition kernel are influenced by its *staircase* structure. The inner loops of the algorithm count a different number of elements at each column iteration, making it difficult to hide the RAW stalls on data produced by the multiplication and division units. The cores working on the matrix central rows are assigned a smaller workload and conclude their task in advance, increasing the synchronization overhead. In the parallel implementations, independent decompositions are replicated to fit all the banks of a cluster. Respectively on MemPool and TeraPool we can fit 256 and 1024 single-core decompositions of 4x4 matrices, 32 and 128 fine-grained parallel decompositions on couples of

32x32 symmetric output matrices.

The same subset of cores can also generate multiple decompositions on independent inputs before the barrier to reduce the synchronization overhead. Using this strategy and running respectively 16x256 and 16x1024 single core Cholesky decompositions of 4x4 matrices on the whole MemPool and TeraPool clusters, we achieve an IPC of 0.71.
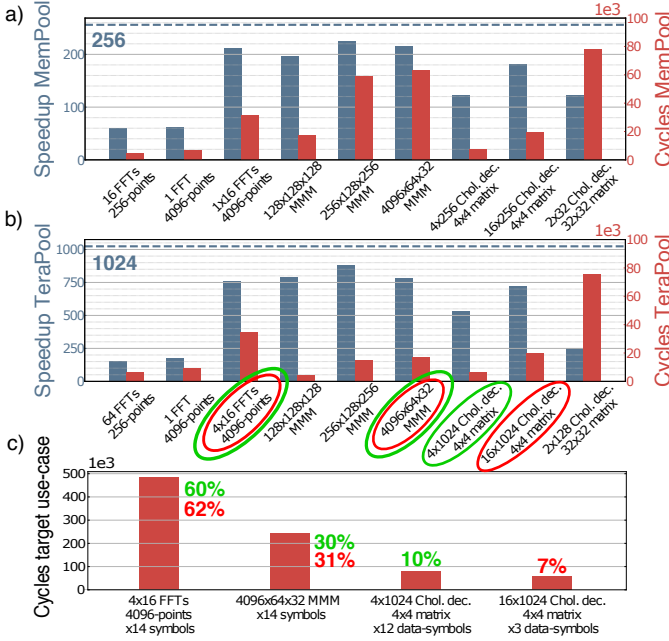

Fig. 9. (a-b) Speedup with respect to a serial single core execution, and total number of execution cycles. The use-case benchmarks are circled in green and red. (c) Total number of cycles per use-case kernel and percentage over the total execution time.

Fig. 9 (a) and (b) represent the speedup of the parallel implementations, over a serial single core execution, and the total execution time of the kernels. The theoretical limit, corresponding to the total number of cores used, is represented as a blue dotted line. Considering 4 UEs active on the same frequency, the use-case described in section II is addressed executing 64 4096-points FFTs and a 4096x64x32 MMM for each of the 14 OFDM symbols, 4096 Cholesky decompositions of 4x4 matrices for 12 data symbols. The overall speedup achieved on TeraPool, using the kernels circled in green in Fig. 9 (b) is 848, and the corresponding execution cycles and their percentage over the total is represented in Fig. 9 (c). If 4x4096 Cholesky decompositions of 4x4 matrices are scheduled every 4 data symbols, the IPC of the last stage increases. The overall speedup obtained using the kernels circled in red in Fig. 9 (b) is 871. The total execution cycles are shown in Fig. 9 (c) with a breakdown on the kernels. Executing the full PUSCH then requires 785 thousand cycles, corresponding to 0.785ms when the cluster runs ad 1GHz. An implementation analysis of the MemPool Architecture [6] demonstrates that this speed is achievable in FinFET technology (12nm and beyond), assuming a speedup of 30% with respect to the less advanced 22nm FDSOI technology

used in [6]. The 0.5ms timing constraint for one transmission in 5G PUSH can be met with customization of the RISC-V cores with domain-specific instructions (e.g. FFT butterfly), which will be explored in future work.

VII. CONCLUSIONS

In this paper, we proved the flexibility of MemPool and TeraPool many-core architectures, leveraging their shared memory structure in the parallelization of the key kernels of the lower PHY receiving chain of 5G PUSCH. The efficient parallelization of kernels with different memory access patterns on architectures with a large shared data memory was achieved. It was demonstrated that parallelizing the most computationally complex kernels in PUSCH processing, namely FFT, MMM, and Cholesky decomposition, TeraPool can provide speedup up to 871 with respect to the serial execution on a single RISCV core. The total execution time, at a realistic clock frequency of 1GHz, is 0.785ms, which can be further improved toward the 0.5ms target execution time by implementing domain-specific instruction extension in the RISC-V cores. Our work represents a first concrete step toward a 5G software-defined RAN over a fully open-source parallel RISC-V architecture.

REFERENCES

[1] X. Lin, J. Li, R. Baldemair, J.-F. T. Cheng, S. Parkvall, D. C. Larsson, H. Koorapaty, M. Frenne, S. Falahati, A. Grovlen, and K. Werner, "5G new radio: Unveiling the essentials of the next generation wireless access technology," *IEEE Communications Standards Magazine*, vol. 3, no. 3, pp. 30–37, 2019.

[2] L. Bertaccini, L. Benini, and F. Conti, "To buffer, or not to buffer? a case study on FFT accelerators for ultra-low-power multicore clusters," in *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2021, pp. 1–8.

[3] G. Peng, L. Liu, S. Zhou, S. Yin, and S. Wei, "A 2.92-Gb/s/W and 0.43-Gb/s/MG flexible and scalable CGRA-based baseband processor for massive MIMO detection," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 2, pp. 505–519, 2020.

[4] C. Jeon, O. Castañeda, and C. Studer, "A 354 Mb/s 0.37 mm^2 151 mW 32-user 256-QAM near-MAP soft-input soft-output massive MU-MIMO data detector in 28nm CMOS," in *ESSCIRC 2019 - IEEE 45th European Solid State Circuits Conference (ESSCIRC)*, 2019, pp. 127–130.

[5] E. Inc., "EdgeQ and Mavenir Partner on Software-defined 4G and 5G Mobile Networks," https://www.eetasia.com/edgeq-and-mavenir-partner-on-software-defined-4g-and-5g-mobile-networks/, Sep. 2022.

[6] M. Cavalcante, S. Riedel, A. Pullini, and L. Benini, "MemPool: A Shared-L1 Memory Many-Core Cluster with a Low-Latency Interconnect," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 701–706.

[7] 3GPP, "Physical Channels and Modulation," 3rd Generation Partnership Project (3GPP), Technical Specification (TS) 38.211, 2017, release 17.

[8] S. Shahabuddin, M. H. Islam, M. S. Shahabuddin, M. A. Albreem, and M. Juntti, "Matrix decomposition for massive MIMO detection," in *2020 IEEE Nordic Circuits and Systems Conference (NorCAS)*, 2020, pp. 1–6.

[9] Y. Liu, Z. Tan, H. Hu, L. J. Cimini, and G. Y. Li, "Channel estimation for OFDM," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 1891–1908, 2014.

[10] F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini, "Snitch: A tiny pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads," *IEEE Transactions on Computers*, vol. 70, no. 11, pp. 1845–1860, 2021.