RNN-Based Radio Resource Management on Multi-Core RISC-V Accelerator Architectures

Gianna Paulin, *Student Member, IEEE,* Renzo Andri, *Member, IEEE,* Francesco Conti, *Member, IEEE,* and Luca Benini, *Fellow, IEEE*

***Abstract*—Radio Resource Management (RRM) is critical in 5G mobile communications due to its ubiquity on every radio device and its low latency constraints. The rapidly evolving RRM algorithms with low latency requirements combined with the dense and massive 5G base station deployment ask for an on-the-edge RRM acceleration system with a trade-off between flexibility, efficiency, and cost making Application-Specific Instruction-Set Processors (ASIPs) an optimal choice. In this work, we start from a baseline, simple RISC-V core and introduce instruction extensions coupled with software optimizations for maximizing the throughput of a selected set of recently proposed RRM algorithms based on models using multi-layer perceptrons (MLP) and recurrent neural networks (RNNs). Furthermore, we scale from a single-ASIP to a multi-ASIP acceleration system to further improve RRM throughput. For the single-ASIP system we demonstrate an energy-efficiency of 218 GMAC/s/W, and a throughput of 566 MMAC/s corresponding to an improvement of 10× and 10.6×, respectively, over the single-core system with a baseline RV32IMC core. For the multi-ASIP system, we analyze the parallel speedup dependency on the input and output feature map size for Fully-connected and LSTM layers, achieving up to 10.2× speedup with 16 cores over a single extended RI5CY core for single LSTM layers and a speedup of 13.8× for single fully-connected layers. On the full RRM benchmark suite, we achieve an average overall speedup of 16.4×, 25.2×, 31.9× and 38.8× on two, four, eight, and 16 cores, respectively, compared to our single-core RV32IMC baseline implementation.** [1]

***Index Terms*—ASIP, RISC-V, Machine Learning, Neural Networks, RNN, LSTM, Radio Resource Management, RRM**

I. Introduction

PEOPLE'S demand for being mobile and continuously connected with reliable high-speed internet (e.g., for video streaming), and the increasing number of connected Internet-of-Things (IoT) devices make the new 5G radio communication standard a necessity for the advancement of the digital revolution. However, these advancements heavily tighten the already demanding requirements on the hardware (HW) and software (SW) layers used in radio communication, pushing industry and academia towards improving the efficiency of Radio Resource Management (RRM) [1].

G. Paulin, R. Andri and L. Benini are with the Integrated System Laboratory of ETH Zürich, Gloriastrasse 35, 8092 Zürich, Switzerland (e-mail: {pauling, benini}@iis.ee.ethz.ch).

R. Andri is with Huawei Technologies, Zurich Research Center, Thurgauerstrasse 40, 8050 Zürich, Switzerland (e-mail: renzo.andri@huawei.com).

F. Conti and L. Benini are with the Department of Electrical, Electronic and Information Engineering of University of Bologna, Viale del Risorgimento 2, 40136 Bologna, Italy (e-mail: f.conti@unibo.it, luca.benini@unibo.it).

This project is funded by Huawei Technologies Sweden AB.

Manuscript received February 24, 2021; revised May 15, 2021.

[1]Hardware, software and benchmarks have been open sourced on GitHub https://github.com/iis-eth-zurich/RNNASIP

RRM typically runs on a Radio Access Network (RAN) SoC on every base station. The RAN optimizes various tasks such as, e.g., limited radio-frequency communication spectrum utilization, transmission power control, error coding, beamforming within a very short time period. While doing so, various constraints need to be considered: The communication load needs to be appropriately balanced, every user device needs to be served fairly, overall throughput should be high, and ideally, everything should be performed with high energy efficiency. Figure 1 gives a high-level overview of some essential RRM tasks with the most common performance metrics [2]. Traditionally used optimization algorithms for RRM include exhaustive heuristic search methods, iterative algorithms [3], [4], non-linear non-convex optimization problems [3], game theory, or lagrangian relaxations [5].

Recently, however, algorithms based on Deep Learning (DL) have revolutionized many time-series analysis problems such as speech recognition [6], speech synthesis [7], automatic translation [8], biosignal analysis [9], and many more. Therefore, it is no surprise that research has started to tackle RRM using neural networks [10]–[20]. Compared to the aforementioned traditional iterative algorithms, DL-based models are capable of autonomously extracting high-level features and (non-linear) correlations with a much smaller computational cost, making their real-time implementation in the time range of milliseconds much easier [10].

Up to now, RRM tasks addressed by DL models are at the data-link layer of the OSI model [21]. Recurrent neural networks (RNNs), often the long short-term memory (LSTM) version, are successful at carrier sensing, collision detection, and have the capability to learn and compensate nonlinearities and imperfections that are invariant to the environment in RF components at runtime [17]. Dynamic resource scheduling of frequency bands, dynamic range control, and various other network optimizations are successfully addressed by convolutional neural networks (CNNs) and LSTM networks [11], [12]. Multi-layer perceptronss (MLPs) are used for optimal resource allocation [10], [13], dynamic transmit power control [14], dynamic multichannel access [15], beamformer design [10], dynamic spectrum allocation, transmission power control [16], and channel access optimizations [19].

RRM tasks are highly latency-critical and are ideally performed in-situ at the base stations. However, since RRM models and algorithms are typically rapidly evolving while the costs of the base stations should be amortized over a long time period, specialized hard-wired accelerators cannot provide enough flexibility. Using FPGA fabrics for RRM acceleration

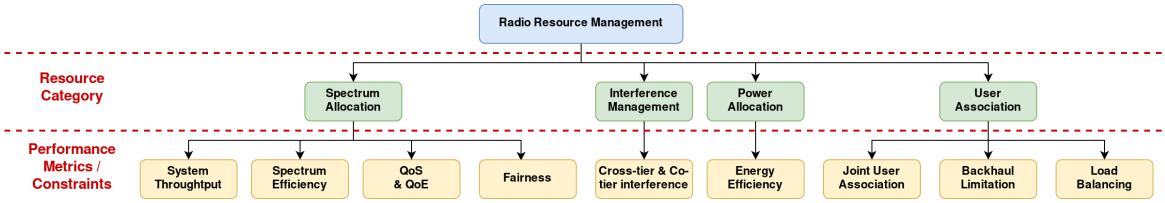


Fig. 1: Overview of various RRM tasks with their most common optimization constraints and metrics [2]. The resource categories are high-level and can include many fine-grained tasks, often also targeting metrics from other task categories.

	flexibility	cost-efficient	performance	integration
Hardwired accelerator	--	--	++	+
FPGA accelerator	-	-	+	--
Graphical processing unit (GPU)	+	+	-	-
Application-specific ISA processor	+	+	+/-	+
General-purpose ISA processor	++	+	--	+

Fig. 2: Overview of benefits and drawbacks of various HW platforms for RRM. '+' corresponds to 'high' and '-' to 'low'

would bring the needed flexibility, however, FPGAs are expensive when targeting massive and dense deployment as required in 5G networks and are hard to integrate into bigger Systems-on-Chips (SoCs). In 5G network deployments, an approach based on Application-Specific Instruction-Set Processors (ASIP) achieves an excellent trade-off between efficiency, costs, and flexibility. Furthermore, integrating an ASIP-based subsystem in complex 5G Systems-on-Chips (SoCs) is a very widely practiced approach. Additionally, the various high-quality open-source cores based on the open and royalty-free RISC-V ISA offer a widely supported standardized baseline to systematically explore the architectural needs of DL-based RRM applications. An overview of the benefits and drawbacks for various HW platforms is shown in Figure 2.

This work is an extension of Andri et al. [22]. We evaluate the architectural needs of DL-based RRM applications on an RV32IMC RI5CY open-source core [23], introducing further extensions and support for parallelization. We perform all our evaluations on open-source single-core and multi-core architectures based on the same core [24]. We make the following contributions:

1) We define a benchmark suite with multiple MLP- and RNN-based RRM applications running on a single-core cluster configuration with RI5CY and apply various software optimizations: `xPULP` extensions (4.0×), improved data reuse through output feature map (OFM) tiling (1.7×), and input feature map (IFM) tiling (3%).
2) We extend RI5CY with RRM-specific hardware instructions: custom activation (13% within LSTMs), a merged load and compute instruction (1.5×) with minimal area overhead (3.4%) and no increase of the critical path.
3) On a single-core configuration, our instruction extensions achieve an energy-efficiency of 218 GMAC/s/W, and a throughput of 566 MMAC/s in 22FDX technology at 0.8 V, corresponding to an improvement of 10× and 10.6×, respectively, over a baseline RISC-V IMC core. Furthermore, we define a multi-core RRM cluster configuration, showing that parallel speedups depend on the input and output feature maps (FM) size for fully-connected and LSTM layers. We achieve up to 10.2× speedup with 16 cores over a single extended RI5CY core for single LSTM layers, and up to 13.8× for single fully-connected layers.
4) We provide a detailed analysis of parallelizing the RRM benchmark suite on a range of cluster configurations taking into account speedups, parallelization overhead (average of only 2.55% on bigger RRM models), and memory transfers. We achieve a total overall speedup of 193.7×, 110.4×, and 132.0× on 16 cores; and 101.9×, 81.5×, and 93.4× on eight cores compared to the single-core RV32IMC baseline implementation when the complete models fit into 512 kB of L1 memory.

The paper is structured as follows. Section II describes the related work and the selected benchmark suite. Section III describes the system architecture and all microarchitectural and software optimizations. In Section IV we define an upper bound for the speedup on the multi-core system. Section V discusses experiments on the single and multi-core system.

II. Related Work

A. RNN for 5G

The expected benefits of 5G require a frequency band utilization in the range of 3 – 300 GHz, which goes beyond the previously utilized 300 MHz – 3 GHz spectrum. Using these higher frequency bands incurs higher path losses, reflections, and scattering, requiring an ultra-dense base station deployment. Additionally, 5G will be used in combination with other radio access technologies such as 2G, 3G, LTE-A resulting in higher interference. With the tightened device power consumption requirements and other equally essential requirements such as fairness or load balancing, Radio Resource Management for 5G becomes extremely complex [2].

Therefore, finding more efficient RRM algorithms is crucial for the successful deployment of 5G technology. With the recent Deep Learning revolution, it comes as no big surprise that industry and academia have started to tackle RRM with neural networks, which are now considered the SoA approach. The used DL models include deep MLPs [10], [13], CNNs, LSTM RNNs [11], [12]. While attention-based models like Transformer [25] have shown state-of-the-art results for all kind of time-series predictions, to the best of our knowledge, this new network type has not yet been applied to RRM tasks.

TABLE I: Benchmark suite of typical models used for RRM: long short-term memory (LSTM) RNNs, and multi-layer perceptrons (MLPs) which are built from multiple Fully-Connected Layers (FCLs).

	#Params	# Input Features N_I	Layer 1		Layer 2		Layer 3		Layer 4		Layer 5		Layer 6	
			Type	N_O	Type	N_O	Type	N_O	Type	N_O	Type	N_O	Type	N_O
Model A [11]	30k	10	LSTM	70	FCL	70	LSTM	4	-	-	-	-	-	-
Model B [12]	1k	8	LSTM	8	FCL	8	-	-	-	-	-	-	-	-
Model C [16]	160k	6	FCL	500	FCL	250	FCL	120	FCL	6	-	-	-	-
Model D [20]	150k	512	FCL	200	FCL	200	FCL	16	FCL	180	-	-	-	-
Model E [10]	85k	16	FCL	200	FCL	200	FCL	200	FCL	4	-	-	-	-
Model F [14]	37k	57	FCL	200	FCL	100	FCL	40	FCL	10	-	-	-	-
Model G [19]	24k	100	FCL	64	FCL	64	FCL	64	FCL	64	FCL	64	FCL	2
Model H [13]	1k	4	FCL	32	FCL	16	FCL	4	-	-	-	-	-	-
Model I [15]	9k	10×10×8	3x3-Conv	800	FCL	10	-	-	-	-	-	-	-	-

In this work, we have selected a comprehensive set of DL-based RRM models as a benchmark suite, see Table I. All models are used in a deep reinforcement learning (DL-RL) setup: We have an agent which interacts with an environment. At every timestep, the agent decides based on an observed state of the environment according to a policy what action to take, thereby putting the environment in a new state. Based on the new state, the agent receives a feedback. The policy is typically implemented as a function approximation, and in our case is implemented with MLP and/or LSTM models. Note that, in general, RL allows to perform policy updates online based on the received feedback. However, none of the selected benchmarks make use of this feature. Instead, the models are trained offline and are deployed on a base station (typically on a RAN SoC). Online training support increases the computational complexity from $\mathcal{O}(n_{layers} \cdot n^3)$ for the inference of an MLP with $N_I = N_O = n$ to $\mathcal{O}(n_{gradient\,iterations} \cdot n_{layers} \cdot n^3)$. While this can get problematic for the highly latency-critical RRM applications, offline training still allows updating the models as needed periodically (e.g., once per week) [12].

Furthermore, DNN based policies are trained with algorithms such as gradient-descent and backpropagation, which in contrast to the inference part of DNNs, typically still require high precision floating-point arithmetic. In contrast, inference works well with low-precision fixed-point arithmetic, such as 16-bit [26], 8-bit, or even fewer bits while keeping accuracy high [27]. As RRM mainly uses MLP and LSTM layers [28], the latter being well known for a quite high sensitivity to numerical precision, we use for our implementation the rather conservative 16-bit integer format, which requires only simple or even no quantization-aware training methods.

Models A and B both combine LSTM and MLPs. Model A maximizes throughput by adapting dynamic channel selection, carrier aggregation, and spectrum access under fairness constraints [11], Model B focuses more on the dynamic spectrum access for network utility maximization [12]. Model C is also a MLP and minimizes interference under latency constraints via channel selections and power control [16]. Model D targets a multichannel access problem for throughput maximization [20]. Model E, and F tackle the problem of interference channel power control and throughput maximization [10], [14]. Model G optimizes the sum throughput and α-fairness in time slot sharing among heterogeneous network nodes with different applied MAC protocols [19]. Model H optimizes throughput with a more general optimal resource allocation problem formulation. Finally, model I is the only CNN based model. It aims at maximizing throughput by power control [15]. For more detailed information about the individual benchmarks we refer the interested reader to the references of the corresponding models.

Currently, these RRM tasks are mainly executed on general-purpose processors on the 5G base stations [29]. The application-specific customization of these processors can cover the various needs of the RNN-based applications while still being flexible enough for adapting to rapidly evolving algorithms. In this work, we present the first, to the best of our knowledge, dedicated ASIP-based subsystem for running RNN-based RRM efficiently.

Up to now, all Radio Access Network (RAN) SoCs are heavily proprietary and closed-source. These designs tend to be offered by a very limited number of vendors and typically require a complete replacement when upgrading to newer protocols and standards as their HW and SW designs are heavily coupled. The virtualization coming with OpenRAN offers operators to run software-based network functions on standard (COTS) servers, allowing them to upgrade software code and hardware components more gradually [30]. General-purpose hardware running software defined stacks, however, is often suboptimal from a power viewpoint. Recently, a startup called EdgeQ [31] has announced a developer-accessible RISC-V based SoC with custom hardware instructions to accelerate algorithms used for 4G and 5G communication and signal processing. Our own approach is similar, but takes a further step towards openness, relying on (i) open ISA, (ii) open-source cores and compilers, (iii) open-source architecture [32].

B. ISA Extensions for Domain Specialization

The idea of extending general-purpose cores with application-specific instructions is not new. ARM and Intel both offer various matrix computation and vector processing extensions in high-performance oriented general-purpose processors. E.g., ARM has introduced the `AARCH64` Neon extensions with the ARMv8-A processor series, including SIMD instructions for sum-dot-products (e.g., `BFDOT`) and 2×2 matrix-matrix multiplications (e.g., `BFMMLA`) with 2-way SIMD in brain floating-point format `bfloat16`. Note that this processor series comes in various microarchitectures. So has, e.g. the CORTEX-A55 (ARMv8.2-A) a in-order superscalar microarchitecture, while, e.g., CORTEX-A75 (ARMv8.2-A)

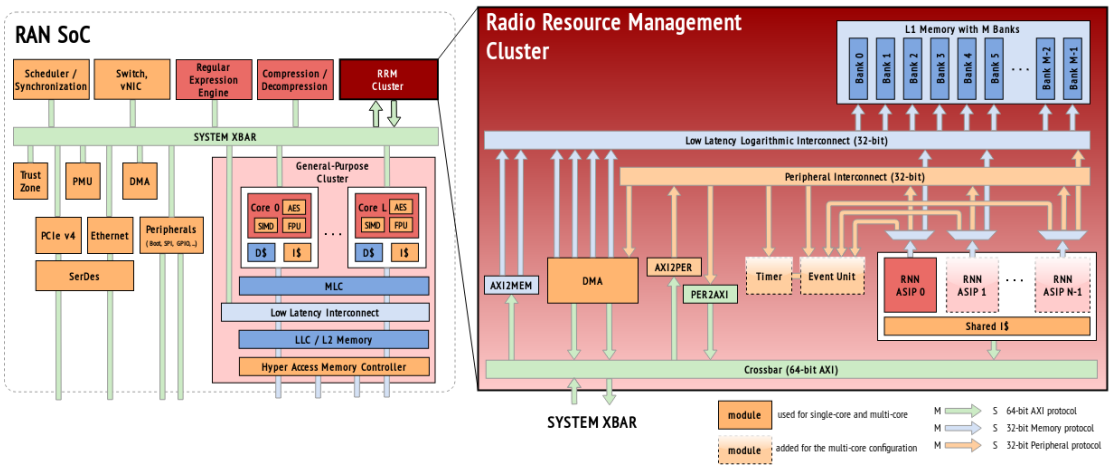


Fig. 3: ASIP RNN subsystem overview including its integration in a typical RAN SoC. The arrows in the RRM cluster show in master-to-slave direction. The modules with the dashed line as a border are only used for the multi-core cluster configuration, while the modules with the solid lines are used for the single-core and the multi-core cluster configuration.

incorporates an out-of-order superscalar pipeline. Intel's out-of-order SkyLake SP processors extend the x86 ISA with 512-bit wide vector units called `AVX512` enabling 16×32-bit SIMD vector operation for multiplication in single-precision float (FP16) and accumulations in double-precision float (FP32). In 2019 Intel announced its new x86 microarchitecture called Cascade Lake, which introduces the AVX512 Vector Neural Network Instructions (VNNI). `AVX512 VNNI` implements an 8-bit and 16-bit fixed-point vector product with 32-bit internal accumulation [33]. While Intel focuses mostly on the high-performance, high-cost processor market, ARM also offers microcontrollers in the low-cost and low-power range with the Cortex-M family. Recently, ARM introduced the Cortex-M55, an ultra-low-power in-order microprocessor with the Vector Extensions MVE (Helium). The Helium instructions support various single instruction–multiple data (SIMD) instructions (INT8/16/32, FP16/32), hardware loops, and interleaved post-increment load/stores [34]. The Helium extension shows that introducing custom ISA extensions is not only beneficial for high-performance general-purpose out-of-order cores and superscalar in-order cores, but also for small, energy-efficient, in-order cores with an IPC close to one.

Several academic proposals also introduce specialized instructions: [35] introduces specialized SIMD instructions optimized for biological sequence alignment algorithms. [36] proposes a custom fast fourier transformation (FFT) instruction for increased throughput on orthogonal frequency division multiplexing (OFDM)-based communication standards. Opportunities for ASIPs in 5G networks are surveyed in [37]. The authors also mention ASIPs for ML acceleration as a future direction in 5G applications. Our work takes a major step in this direction. We focus on RISC-V ISA extensions for two key reasons. First, RISC-V is designed for extensibility with significant parts of the opcode space reserved for custom extensions. Second, with the growing community around the open and royalty-free RISC-V ISA, the number of high-quality RISC-V based open-source cores and microcontroller systems has grown rapidly. Various open-source cores already support custom instructions, e.g., the RI5CY core from the PULP (Parallel Ultra Low Power) project supports custom `xPULP` instructions such as SIMD, HW loops and post-increment loads [23]. In this work, we use the open RISC-V ISA and start as a baseline with a multi-core cluster system based on the mentioned open-source RI5CY core.

C. Software Optimizations

The rise of previously described ISA extensions has led industry and academia to develop highly optimized SW kernels such as matrix-vector and matrix-matrix multiplications which make the best use of these extensions. The used techniques mainly include the utilization of parallel SIMD computations and the data reuse within the local register file with appropriate tiling for reduced memory data loads. The latter has been commonly used for tiling the output feature maps, where loaded input feature maps can be reused to compute multiple outputs in parallel [38], [39]. CNN can exploit the im2col concept of replicating and rearranging feature maps for being formulated as a matrix-matrix multiplication problem [38], [39]. This well researched reformulation enables the tiling of both the input and output FM spatially in $m \times n$-sized tiles, thereby enabling the reuse of both weights and input FM pixels which ultimately reduces the number of memory loads from $\mathcal{O}(mn)$ to $\mathcal{O}(m+n)$. Since both the MLP and (non-convolutional) LSTM layers are based on matrix-vector multiplications, this 2D tiling cannot be reused.

In contrast to CNNs, RNNs require the use of non-linear activation functions such as hyperbolic tangent and sigmoid. As their transcendental computation is computationally complex various acceleration approaches have been proposed: (i) piece-wise linear approximation (PLA) [38], (ii) low-order Taylor series expansion (e.g., 2nd order [40]), (iii) look-up-table (LUT) with adaptive value granularity [41], and (iv) small neural network [42]. For our extension we apply the PLA approach and exploit unlike other works the symmetry property of `tanh` and `sig`. Additionally, we go a step further than, e.g., ARM's CMSIS-NN library and evaluate the error introduced

by different numbers of interpolation intervals and take the applied fixed-point quantization into account [38].

III. ASIP-based RNN Acceleration

A. Baseline RISC-V Architecture

In this work, we propose an ASIP-based RNN RRM acceleration system. We start with designing a single-core ASIP, which we then integrate into a cluster with up to $N = 16$ ASIPs. As a starting point for the ASIP, we use PULP's RI5CY core , a 4-stage pipeline processor supporting the RISC-V standard extended with custom instructions (i.e., RV32IMFCXpulp ISA) [23]. We further extend the RI5CY core with a specialized dot product, hyperbolic tangent, and sigmoid instructions for efficient MLP and LSTM execution [22]. We evaluate these custom instructions for throughput and energy-efficiency on the selected RRM benchmark suite in an acceleration sub-system of up to 16 RNN-enhanced RI5CY cores. An overview of the proposed architecture and its integration into a state-of-the-art Radio Access Network (RAN) System-on-Chip (SoC) is shown in Figure 3. In the right half, we show the proposed RRM acceleration cluster. We perform the evaluations on a cluster configuration with a single RNN ASIP, in which case the modules with a dashed border in Figure 3 would be removed, and in a multi-core configuration where all shown modules are used. For the integration of the proposed ASIP-based acceleration sub-system into a large 5G RAN SoCs, we propose to connect one of the proposed systems via a crossbar, e.g., by connecting it to the system crossbar in Marvell's Octeon TX2 CN98xx architecture [29] as shown in the left side of Figure 3.

The proposed RRM cluster is designed around a configurable number (up to 16) of enhanced-RI5CY cores. The cluster has no data cache-hierarchy in the traditional sense. Instead, all ASIP cores share a single-cycle accessible L1 tightly-coupled data memory (TCDM) with a banking factor of two, composed of word-interleaved single-port SRAM macros as memory banks. The programmer is responsible to ensure that the correct data is loaded via DMA from the L2 memory before the ASIP cores try to access them. The required synchronization schemes are implemented by an event unit. We use three different cluster configurations for our evaluations, in which we have 1, 8, or 16 ASIPs and a 1MB, 512KB, or 64KB L1 memory. The L2 memory is mapped into the last level cache (LLC) memory of the RAN SoC.

A DMA engine is connected to the L1 memory via the logarithmic interconnect, to manage data transfers between L2 and the small L1 embedded within the RRM cluster. The DMA can be controlled by a single core from the cluster side and supports blocking and non-blocking 64-bit/cycle data transactions in both direction concurrently. The ASIPs fetch their instructions from a shared instruction cache. Multi-port memory banks are used for the shared tag and instruction memory. The I-Cache has access to the 64-bit AXI cluster bus to fetch off-cluster data from the L2 in the case of a cache-miss. The 64-bit AXI cluster bus also serves DMA data transfers from L2 to L1 TCDM. Over a so-called peripheral interconnect, the ASIPs can control and program the DMA engine, event unit, or further peripherals like, e.g. timers.

B. Neural RRM Models

The selected set of benchmarks are based on two DL models: multi-layer perceptrons (MLP) and long short-term memory (LSTM) recurrent neural networks (RNNs). *MLPs* include at least three layers: one input, at least one hidden, and one output layer. These individual layers, also called *Fully-Connected Layers (FCL)*, are biased matrix-vector multiplications transforming N_X input features $\mathbf{x}_t$ at time t into N_O output activations $\mathbf{y}_t$ with the weight $\mathbf{W}_x \in \mathbb{R}^{N_O \times N_X}$ and bias $\mathbf{b}_y \in \mathbb{R}^{N_O}$:

$$\mathbf{y}_t = \mathbf{W}_x \mathbf{x}_t + \mathbf{b}_y \tag{1}$$

RNNs [43] are a linear superposition of multiple FCLs activated by a non-linear activation function *act* (typically hyperbolic tangent or sigmoid function) with a feedback over the *hidden state* $\mathbf{h}_t = (h_1, h_2, ..., h_{N_H})$ with N_H elements:

$$\mathbf{h}_t = \text{act}(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h) \tag{2}$$

RNN networks can contain multiple RNN layers by feeding the *hidden* state of one RNN layer as *input state*) to the next RNN layer. The final output of the network is computed from the *hidden* state of the last RNN layer:

$$\mathbf{y}_t = \text{act}(\mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y) \tag{3}$$

LSTM neural networks [44] are a subclass of RNNs specialized in learning both short-term and long-term time-series dependencies. Besides the hidden state $\mathbf{h}_t$ LSTMs include an internal *cell* state $\mathbf{c} = (c_1, c_2, ..., c_{N_H})$. Their computation include matrix-vector multiplication, point-wise vector-vector additions/multiplications and point-wise applied sigmoid and hyperbolic tangent activation functions:

$$\mathbf{i}_t = \sigma(\mathbf{W}_{xi}\mathbf{x}_t + \mathbf{W}_{hi}\mathbf{h}_{t-1} + \mathbf{b}_i), \tag{4}$$

$$\mathbf{f}_t = \sigma(\mathbf{W}_{xf}\mathbf{x}_t + \mathbf{W}_{hf}\mathbf{h}_{t-1} + \mathbf{b}_f), \tag{5}$$

$$\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_{xc}\mathbf{x}_t + \mathbf{W}_{hc}\mathbf{h}_{t-1} + \mathbf{b}_c), \tag{6}$$

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t, \tag{7}$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_{xo}\mathbf{x}_t + \mathbf{W}_{ho}\mathbf{h}_{t-1} + \mathbf{b}_o), \tag{8}$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t), \tag{9}$$

including the *input gate* $\mathbf{i}$, *forget gate* $\mathbf{f}$, *output gate* $\mathbf{o}$, the *cell* state $\mathbf{c}$.

C. Enhanced RISC-V ISA

In this section, we summarize our custom ISA extensions used in our ASIP cluster. For more details, we refer the interested reader to [22].

1) xPULP Extensions: The baseline RI5CY core already supports various specialized HW instructions such as SIMD, HW loops, and post-increment loads under the name `xPULP`, which we leverage in the first optimization step.

2) Tanh and Sigmoid Extension (HW): The two non-linear activation functions used for neural networks such as LSTM networks are sigmoid `sig` and hyperbolic tangent `tanh`. Their execution is often emulated in software with the help of a linear approximation technique requiring multiple iterations until the required precision is reached. This emulation can quickly become a major contributor to the overall execution

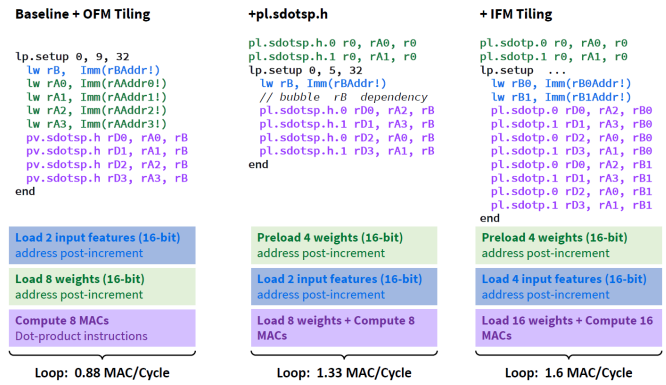


Fig. 4: Assembly Code comparison baseline+output FM tiling, +`pl.sdotsp.h` instruction, +input FM tiling.

time of LSTM networks as, for example, [11] and [12] show that the calculation of `tanh` and `sig` require together 10.3% and 33.6% of the overall computation cycles.

Therefore, we introduce two new single-cycle HW instructions `pl.tanh rD, rA` and `pl.sig rD, rA`. They are implemented as linear function approximation within certain intervals. The pre-computed approximation parameters m, q are loaded from look-up tables (LUTs) for a certain interval. After the linear approximation, the result is mirrored if needed (symmetry property). For more details on the implementation and an error evaluation, we refer the interested reader to [22].

3) Load and Compute VLIW Instruction (HW): An evaluation of instruction counts in the benchmarks (see Table IIIc) shows that the two `xPULP` instructions `lw!` and `pl.sdotsp.h` are by far the two most executed instructions within our benchmark suite. Thus, we introduce a new instruction which combines the two instructions within a single `pl.sdotsp.h` instruction which is capable of loading data and of calculating the 16-bit packed SIMD sum-dot-product:

```
rD[31:0] += rA[31:16]*rB[31:16]
          + rA[15:0]*rB[15:0]
```

`rA` contains the memory address, loaded from memory by the load/store unit (LSU) and is incremented for the next data access. The instruction makes use of two special-purpose registers SPR which are written and read in an alternating way (using `pl.sdotsp.h.0` and `pl.sdotsp.h.1` instructions), which helps to avoid a 2-cycle latency and thus unnecessary stalling. As the presented customized instructions were implemented for the RI5CY core, Figure 6 shows the extended RI5CY datapath where the datapath for the extended `pl.sdotsp.h` instruction is highlighted in blue.

A comparison between the assembly code with (middle) and without (left), including the output feature map tiling of size $d = 4$ is shown in Figure 4. In the middle, the first two `pl.sdotsp.h` instructions before the HW loop pre-load the two SPR with the first two weights. The corresponding input feature map is loaded in line 4, followed by a bubble caused by the latency of the load word instruction and the following instructions' data-dependency.

Algorithm 1 Fully-Connected Layer with Output FM Tiling

Require: All weights w_{mn}, biases b_n and input activations x_m for all input channels $m \in N_I$ and output channels $n \in N_O$ in memory

1: **for all** d in $\{8, 4, 2, 1\}$ **do**
2: **for all** remaining d-sized output channel tiles $\tilde{y}_k = \{y_{k \cdot d}, \ldots, y_{(k+1) \cdot d}\}$ **do**
3: **for all** output channels y_s in d-sized tile $\tilde{y}_k$ **do**
4: temp_out[s] = Mem(b_s)
5: **end for**
6: **for all** input channels x_r in x **do**
7: temp_in=Mem(x_r)
8: #unroll following loop
9: **for all** output channels y_s in d-sized tile $\tilde{y}_k$ **do**
10: w=Mem($w_{s,r}$)
11: temp_out[s] += temp_in * w
12: **end for**
13: **for all** output channels y_s in d-sized tile $\tilde{y}_k$ **do**
14: temp_out[s] = temp_out[s] >> 12 // quantize
15: Mem(y_s) = temp_out[s]
16: **end for**
17: **end for**
18: **end for**
19: **end for**

D. Loop Tiling and Double Buffering

1) Baseline: As a baseline, we have developed a straightforward implementation for the FCL and LSTM kernels. E.g., the matrix-vector multiplication makes use of a double nested loop over all inputs and output. All weights and activations are encoded into the 16-bit $Q_{3.12}$ fixed-point format, and all computations were validated against a 32-bit floating-point implementation. The 16-bit quantization can be applied on the benchmark suite without fixed-point aware retraining, and therefore offers a good compromise between accuracy/robustness and energy-efficiency/throughput.

2) Output Feature Map Tiling (SW): A single MAC operation requires two inputs, which need two memory loads: one for the input feature and one for the weight. While the weights differ for each input feature, the input features can be reused for several outputs. The next improvement step exploits this fact by reorganizing the output features in tiles of d output features over which a loaded input feature is reused. The partial sums of the d output features are kept in d processor registers. They are written back to the memory once all input features have contributed their part to the corresponding results. Algorithm 1 gives a high-level overview of the aforementioned output feature map tiling.

The loaded input feature (line 7) can be used by d `pl.sdotsp` instructions (line 11) each performing two `mac` operations on 16-bit data. In total, this results in $\mathcal{O}(1 + 1/N)$ loads needed per `pl.sdotsp` instruction. The feature map tile size d is optimally smaller or equal to the number of available processor registers, as in these cases, the partial results can be kept locally. Going beyond this limitation would decrease efficiency because the intermediate results have to be pushed back into the memory before being reused.

Additionally, the compiler can rearrange the instructions and hide the load latency. We empirically determined an optimal tile size of $d_{optimal} = 8$ for our implementation, which means that each core works on multiple rounds of tiles of size $d \in \{8, 4, 2, 1\}$ until no remainder is left. The input features of

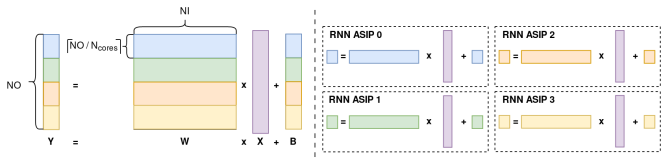


Fig. 5: Example of the tiling for a linear layer which is distributed on a 4-core ASIP cluster.

TABLE II: TCDM contention in % with all single core optimizations enabled with a weight offset of 6 against no weight offset on a FCL layer with $N_I = N_O = 64$ evaluated on a 16-core cluster configuration with 32× L1 memory banks.

Weight Offset	1 Core	2 Cores	4 Cores	8 Cores	16 Cores
0	0.0%	27.5%	10.9%	14.8%	14.3%
6	0.0%	0.1%	1.9%	2.7%	14.5%

convolutional layers can be rearranged and replicated (i.e., im2col) in such a way that its computation is mapped on a matrix-matrix multiplication (as shown by [38], [39]) enabling the same tiling optimization.

3) Input Feature Map Tiling (SW): To get rid of the bubble as shown in Figure 4, another optimization was implemented where the loop loads two input data words, which corresponds to four 16-bit input features. This doubles the number of `pl.sdotsp.h` instructions in the innermost loop as shown on the right side in Figure 4.

4) Multi-Core Tiling: For the evaluation of the cluster-based configuration on up to 16× enhanced RI5CY cores, the kernels and computations are tiled and distributed along the output dimension, which results in every core working on $\tilde{N}_O$ output features in parallel, as shown in Equation (10).

$$\tilde{N}_O = \left\lceil \frac{N_O}{N_{cores}} \right\rceil \tag{10}$$

In contrast to tiling along the input dimension, this method does not require any additional overhead for combining partial results across multiple cores. Figure 5 shows conceptually how the computation on the output feature map tiles of a linear layer are distributed on an exemplary cluster with four RNN ASIPs. Our tiling approach is commonly used [38], however, we adapt the tiling to overlap as many DMA transfers with computation phases for optimal performance. As a first step, every core has to compute its assigned tile size, the start address, and their tile size of the assigned weights and features, which cost us up to 14 instructions.

5) Storing Weights with Address Offset: When multiple cores access the cluster memory, various cores may want to access the same memory bank at the same time to load the weights or features of the model. When this happens, only one of the cores gets its request granted and receives the requested data, while the other cores' memory load requests get stalled and have to wait. These so-called banking conflicts can result in many wasted cycles and should be prevented. As a counter-measure, we implemented an offset in weight storage in memory, allowing every core to start at a different address. Table II shows the cycles lost due to TCDM banking conflicts with a weight offset of six against no weight offset.

6) Double Buffering: The double-buffering concept uses two buffers to store input data. While the data in the first buffer is being processed (e.g., weights of current network layer), new data (e.g., weights of next network layer) is loaded into the second buffer. This alternating buffer usage allows for overlapping the computation and DMA transaction phases and reduces overall run time. However, entirely hiding the DMA transaction is only possible if the computation time of a network tile is longer than the DMA data transfer time for loading the next tile parameters.

7) Batching: If the DMA transaction outlasts the computation time, the cores need to wait and are stalled, thereby reducing the speedup. We increase the computation time by batching the input activations, which means that the parameters of a layer or model are loaded once and are consumed by computing not only on one feature but also on one or multiple following input features. However, introducing this form of batching increases the latency, which consequently might violate the tight timing constraints for RRM decisions. Hence batching can be applied to a limited extent.

IV. Multi-Core Speedup Model

To verify the optimality of the parallelized implementation, we introduce an upper bound model for the speedup based on Amdahl's Law and the innermost loop behavior. Amdahl's Law, shown in Equation (11), describes the theoretical speedup SU, which can be achieved by running a model on multiple cores in parallel. P is the fraction of the code that can be parallelized with a speedup S.

$$SU = \frac{1}{(1-P) + \frac{P}{S}} \tag{11}$$

Out of these parameters, P is static and can be measured, whereas S depends on the computational load. For example, if the load is perfectly dividable by the number of available cores, the speedup achieves its maximum $S_{ideal} = N_{cores}$. However, any load imbalance can cause a non-ideal speedup $S_{estimate}$ which we estimate with our model. The following evaluation targets the FCL kernel. For simplicity reasons, we assume that N_I and N_O are even numbers.

As the multi-core implementation follows the second level of Output FM tiling, each core works on $\tilde{N}_O$ output features, as described in Equation (10). Following the first level of Output FM tiling performed on the $\tilde{N}_O$ of a single core, each core works on a round of tiles of size $d \in \{8, 4, 2, 1\}$ until no remainder is left. Equation (12) defines I_d, the number of instructions in the parallelizable part P of the kernel for a single tile of size d. The total amount of instructions can then be computed by following Equations (12) to (15).

$$I_d = \left(\underbrace{2 \cdot d}_{\substack{\text{preload bias} \\ \text{\& get address}}} + \underbrace{\frac{N_I}{4}}_{\substack{2\times \text{ data} \\ \text{in form of v2s}}} \cdot \Big(2 \cdot d + 2\Big) + \underbrace{2 \cdot d}_{\substack{\text{shift} \\ \text{\& store}}} \right) \tag{12}$$

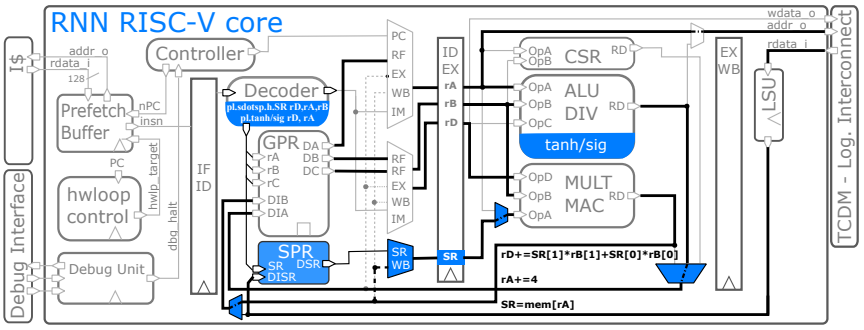


Fig. 6: RNN RISC-V Core with extensions to RI5CY core [23] in blue and datapath for `pl.sdotpsp` instruction marked in bold lines.

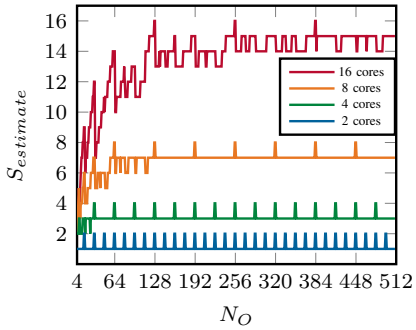


Fig. 7: $S_{estimate}$ for various N_O and for various number of active cores.

TABLE III: Cycle and instruction count for the entire RRM benchmark suite (**RISCY in bold**, **new extensions in blue**)

a) w/o opt (RV32IMC)			**b) +SIMD/HWL (Xpulp)**			**c) +Out-FM Tile./tanh/sig**			**d) +pl.sdotsp instruction**			**e) +Input FM Tiling**		
Instr.	*k*cycles	*k*instrs	Instr.	*k*cyc.	*k*instrs	Instr.	*k*cyc.	*k*instrs	Instr.	*k*cyc.	*k*instrs	Instr.	*k*cyc.	*k*instrs
addi	3'269	3'269	**lw!**	2'432	1'621	**lw!**	894	893	**pl.sdot**	811	811	**pl.sdot**	817	817
bltu	3'248	1'627	**pv.sdot**	811	811	**pv.sdot**	811	811	**lw!**	166	83	**lw!**	83	83
lh	3'248	3'248	addi	22	22	lw	9	9	lw	9	9	lw	39	35
sw	1'627	1'627	jal	10	5	sw	8	8	sw	8	8	sw	16	16
lw	1'627	1'627	sh	10	10	add	7	6	add	7	6	d.srai	8	8
mac	1'621	1'621	srai	10	10	**tanh,sig**	0.4	0.4	**tanh,sig**	0.4	0.4	**tanh,sig**	0.4	0.4
oth.	43	32	oth.	28	27	oth.	26	26	oth.	30	29	oth.	17	10
Σ	14'683	13'051	Σ	3'323	2'506	Σ	1'756	1'753	Σ	1'028	943	Σ	980	969
Impr.	Baseline (1×)		Impr.	4.4×		Impr.	8.4× (1.9×)		Impr.	14.3× (1.7×)		Impr.	15.0× (1.05×)	

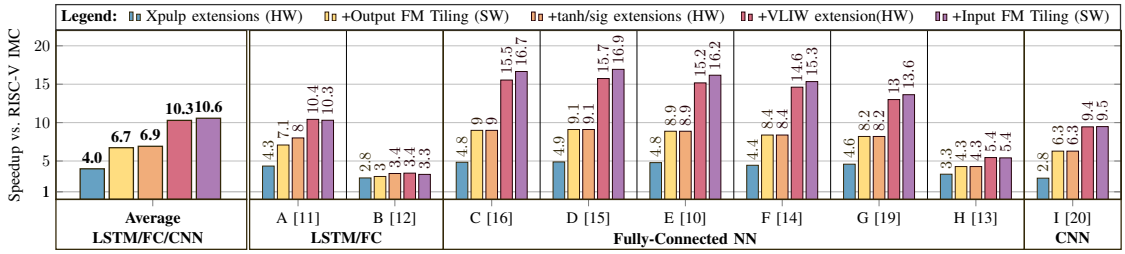


Fig. 8: Speedup with respect to the RISC-V IMC baseline implementation for a typical Neural Networks workload in RRM.

$$I_{tot,N_{cores}} = \sum_{d\in\{8,4,2,1\}} N_{tiles,d} \cdot I_d \tag{13}$$

$$N_{tiles,d} = \left\lfloor \frac{N_{O,d}}{d} \right\rfloor \tag{14}$$

$$N_{O,d} = \begin{cases} \tilde{N}_O, & \text{if } d = d_{max} = 8 \\ (\tilde{N}_O \% (2 \cdot d)), & \text{if } d \in \{4,2,1\} \end{cases} \tag{15}$$

With the help of these equations the number of instructions of the parallelizable part P can be estimated for the number of used cores $N_{cores} \in \{1,2,4,8,16\}$, which allows to estimate a more accurate achievable speedup $S_{estimate,N_{cores}}$ by following Equation (16).

$$S_{estimate,N_{cores}} = \frac{I_{tot,1}}{I_{tot,N_{cores}}} \tag{16}$$

We analyzed $S_{estimate}$ for two configurations:

- $N_I \in \{4,5,\ldots,512\}$ and $N_O = 128$
- $N_O \in \{4,5,\ldots,512\}$ and $N_I = 128$

As the speedup is independent from the number of input features N_I we only show the dependency of $S_{estimate}$ on the number of output features N_O of the FCL kernel in Figure 7. We define the following properties:

1) $\boldsymbol{N_I}$**-independency:** $S_{estimate}$ is independent from N_I.
2) $\boldsymbol{N_O}$**-dependency:** $S_{estimate}$ depends on N_O.
   a) **Saturation:** A minimum amount of output features N_O is needed to fill up the various cores.
   b) **Optimality Condition:** Once a core is saturated, optimal $S_{estimate} = S_{ideal} = N_{cores}$ can be achieved when $\frac{N_O}{N_{cores}} \% d_{optimal} = \tilde{N}_O \% d_{optimal} \overset{!}{=} 0$ is fulfilled where $d_{optimal} = 8$.

Following these properties, for a single-core, a multiple of 8 output features are optimal. Other configurations behave similarly, 2 cores with a multiple of 16, 4 with 32, 8 with 64, and 16 cores with 128.

V. Evaluations and Results

First, we discuss the results for the single-core acceleration system. We introduce three different cluster configurations for the evaluation of the multi-core acceleration system. Finally, we present the HW impacts of the ISA extensions. All performance

numbers were obtained from an event-based C++ virtual platform implementation called *GVSoC* whose cycle-accuracy was calibrated with RTL simulation measurements [45].

A. Single-Core Evaluation

The single-core configuration has an L1 memory size of 1MB and only one core, meaning that the modules with a dashed border in the RRM cluster on the right side of Figure 3 are non-existent ($N = 1$ in Figure 3). The performance results for this cluster configuration are presented in the same order in which we applied them. After every step, we focus on the largest cycle and instruction count of the optimized implementation to optimize the highest contributors in the next optimization step (see Table III). The straight-forward C-implementation is compiled with standard GCC 7.1.1 for RISC-V RV32IMFC ISA and run on the single-core cluster configuration with the unoptimized baseline RI5CY core and an L1 memory size of 1MB. With this rather large L1 memory size, all models can be stored locally, allowing an evaluation of a single ASIP independent of any model tiling effects. The instruction count for the entire benchmark suite is shown in Table IIIa and serves as a baseline for all HW and SW optimizations implemented on the single-core cluster configuration. Table IIIb shows that the three optimization techniques SIMD, HWL, and post-increment load as described in Section III-C1 achieve a 4.0× reduction in the number of instructions w.r.t. to the unmodified RISC-V IMC baseline.

The optimal OFM Tiling into tiles of size $d_{optimal} = 8$ as described in Section III-D2 brings an additional improvement of 1.89× on the RRM benchmark suite, as shown in Table IIIc. A more detailed insight into the various benchmarks is given in Figure 8. While most networks improved between 1.79× [19] and 1.87× [15], those with smaller OFM sizes N_O suffer from the higher overhead and achieve a less speedup, e.g., 1.07× [13] and 1.30× [12]. The first HW extensions for `tanh` and `sig` as described in Section III-C2 are only used by the LSTM based benchmarks [11], [12]. They allow a further cycle count reduction from 51.2 to 44.5 *k*cycles which results in a 13.0% improvement. The Load and Compute HW instruction described in Section III-C3 can again be exploited by all benchmarks and reduces the overall cycle count again by 1.7×, as can be seen in Table IIId. The additional IFM Tiling, described in Section III-D3, gives and additional modest gain of 1.05× (or 4.9%), see Table IIIe, since loads and stores from the stack increase by 1.4× as more registers are needed.

In summary, the achieved overall speedup of 10.6× w.r.t. the RISC-V IMC baseline come from using SIMD and HWL from the `xPULP` extension (4.0×), the OFM tiling (1.7×), the activation function instruction (3.0%), the merged load and compute instruction (1.5×) and the IFM tiling (3.0%). In total we achieve an additional speedup of 3.4× compared to the XPulp implementation. The relative benefits of each optimization step for the full benchmark are shown in Figure 8. While the input FM tiling had a positive effect for most networks, benchmarks with smaller FM need slightly more cycles caused by the increased stack operations.

B. Multi-Core Evaluation

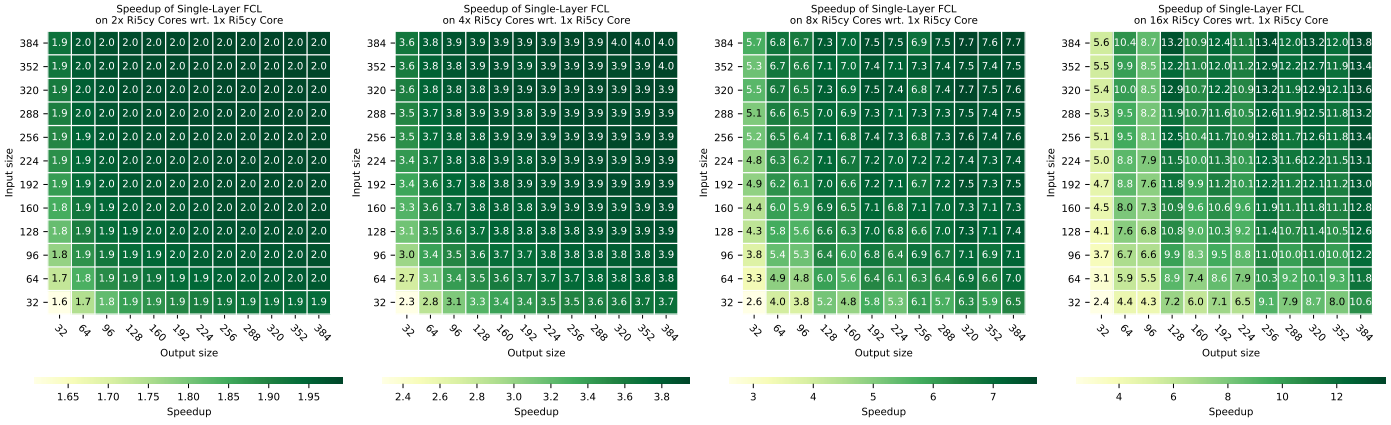
1) Multi-Core Cluster Configurations: We evaluate the individual kernels and the benchmark suite on three multi-core cluster configurations:

1) *Evaluation Cluster* - 16×ASIPs, 1MB L1, min. 1MB L2
2) *Large Cluster* - 16× ASIPs, 512KB L1, min. 1MB L2
3) *Small Cluster* - 8× ASIPs, 64KB L1, min. 512KB L2

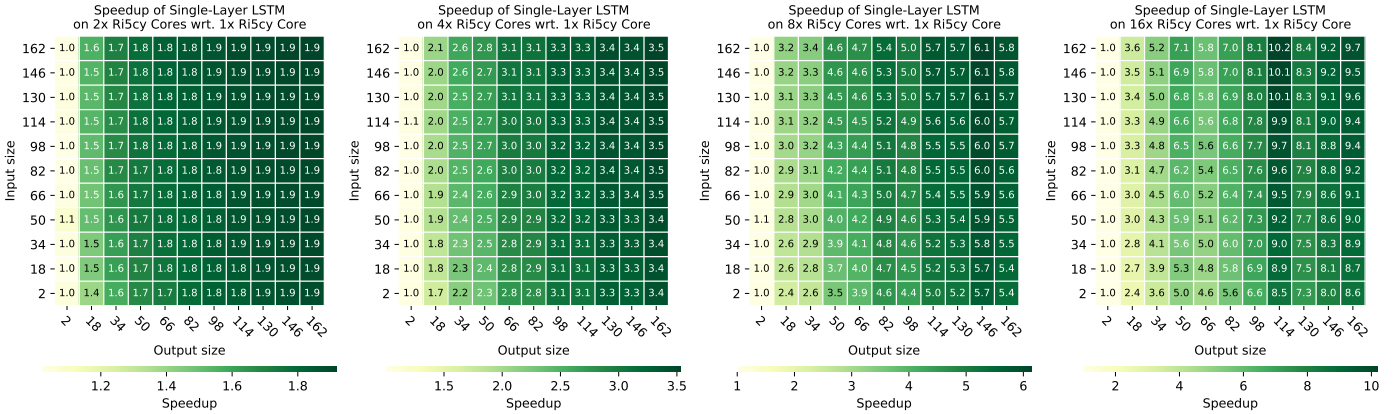
The first *evaluation cluster* configuration is only used to analyze the speedups for the LSTM and FCL kernels. Note that the large L1 memory, if brought on silicon, would impose a large cost in term of area in a 5G SoC. The second *large cluster* configuration allows to keep most of the benchmark suite locally in the L1 memory. For optimal allocation of the various resources for wireless communication, often multiple models are run on the same platform. Therefore, being capable of loading a complete model at a time comes in handy. Anyway, an evaluation on this configuration will give an upper limit for the achievable speedup for smaller cluster configurations where some form of tiling is either needed to adjust the load of imbalanced layer sizes, or simply because the complete model does not fit into L1. The memory size of the *small cluster* configuration is the most affordable in term of silicon real-estate in a 5G SoC, and furthermore has in similar form already been taped-out successfully [24], [46] making it a reasonable choice for a final evaluation of the RRM benchmark suite. As the parallelization of convolutional neural networks are already analyzed thoroughly in related work [39], we will ignore the CNN-based model I [15] for the following experimental results.

2) Multi-Core - Standalone Kernel - Evaluation Cluster: The following results of our parallelized LSTM and FCL kernels are measured on the *evaluation cluster* configuration. The complete evaluation for the FCL kernel as shown in Figures 9a to 9d covers a sweep over the input and output FM sizes $N_I, N_O \in \{32, 64, \ldots, 384\}$. When only two (Figure 9a) or four cores (Figure 9b) in the cluster are enabled (while the others are shut off), the achieved speedup on the FCL kernel saturates for already rather small N_I and/or N_O towards its upper limit of $SU_{ideal} = N_{cores}$. In contrast, when eight or 16 cores are active, it needs a higher output FM size N_O than an input FM size N_I for the achieved speedup to incline towards the ideal speedup. These observations are aligned with the *Saturation* and the N_I*-independency* property of our simplified model in Section IV. The latter property shows itself in the smooth increase of $SU_{measured}$ in N_I-direction. The N_O*-dependency* of the stated *Saturation* gets visible for all plots, however, the *Optimality* property for the N_O*-dependency* can be observed better for eight or 16 active cores, see Figures 9c and 9d. Over the complete sweep of the FCL kernel, the highest speedups are 2×, 4×, 7.7×, and 13.8× for respectively two, four, eight and 16 cores.

The same properties can again be observed in Figures 9e to 9h for the sweep of the LSTM kernel over the IFM and OFM sizes $N_I, N_O \in \{2, 34, \ldots, 162\}$. The highest used FM size for the LSTM sweep are smaller than those used for the FCL sweep since a LSTM kernel is larger than a FCL kernel



(a) FCL - Speedup for 2 cores. (b) FCL - Speedup for 4 cores. (c) FCL - Speedup for 8 cores. (d) FCL - Speedup for 16 cores.



(e) LSTM - Speedup for 2 cores. (f) LSTM - Speedup for 4 cores. (g) LSTM - Speedup for 8 cores. (h) LSTM - Speedup for 16 cores.

Fig. 9: Speedup on the *evaluation cluster* of a single FCL or LSTM layer for various IFM and OFM sizes.

TABLE IV: Achieved Op/Cycle for the larger models on the *large cluster* configuration and the *small cluster* configuration with and without batching. The relative difference is compared to the achieved operations/cycle on the *large cluster* configuration.

	Large Cluster				**Small Cluster - w/o Batching**				**Small Cluster - with Batching**			
Operation/Cycle	*1 Core*	*2 Cores*	*4 Cores*	*8 Cores*	*1 Core*	*2 Cores*	*4 Cores*	*8 Cores*	*1 Core*	*2 Cores*	*4 Cores*	*8 Cores*
Model C [16]	3.22	6.05	10.94	19.52	2.84 / -11.8%	5.47 / -9.6%	6.94 / -36.6%	-	3.03 / -5.9%	6.03 / -0.3%	10.17 / -7.0%	-
Model D [20]	3.31	6.46	11.55	18.08	2.97 / -10.3%	4.85 / -24.9%	6.58 / -43.0%	-	3.17 / -4.2%	5.62 / -13.0%	8.36 / -27.6%	-
Model E [10]	3.14	6.01	10.60	15.59	2.98 / -5.1%	5.10 / -15.1%	6.40 / -39.6%	-	2.99 / -4.8%	5.44 / -9.5%	8.55 / -19.3%	-
Model F [14]	2.93	5.29	8.14	11.49	2.69 / -8.2%	4.29 / -18.9%	5.55 / -31.8%	-	2.80 / -4.4%	5.06 / -4.3%	7.73 / -5.0%	-
Average	*3.15*	*5.95*	*10.31*	*16.17*	*2.87 / -8.9%*	*4.93 / -17.2%*	*6.37 / -38.2%*	-	*3.00 / -4.8%*	*5.54 / -7.0%*	*7.95 / -15.6%*	-

and therefore saturate faster the L1 memory of the *evaluation cluster* configuration. Over the complete sweep of the LSTM kernel, the highest achieved speedups are 1.9×, 3.5×, 6.1×, and 10.2× for respectively two, four, eight, and 16 cores.

3) Multi-Core - Benchmark Suite - Large Cluster: When using the *large cluster* configuration instead of the single-core configuration, the implementation was extended with the necessary synchronization and tile offset computation (see Section III-D4) implementation. The already mentioned Table V compares the achieved MAC/cycle for the implementations when only one core in the *large cluster* configuration is active, including synchronization and tile offset computation, against the single-core configuration. On average, we lose 0.08 Operations/Cycle due to the parallelization overhead, which corresponds to approximately 2.6%. Taking this overhead into account, Figure 10 shows the achieved speedups in relation to the speedups achieved on the single-core implementation, including the synchronization and tile offset computation.

A first observation shows that model B [12] and model H [13] achieve no or even a worse speedup when using more cores. As listed in Table I all layers of these models have an output FM size of 32 or even mostly less, resulting in highly starved cores achieving a maximum speedup of 1.4× against the ”parallelized” single-core version. Again, this starvation is corresponds to the *Saturation* property as stated in Section IV.

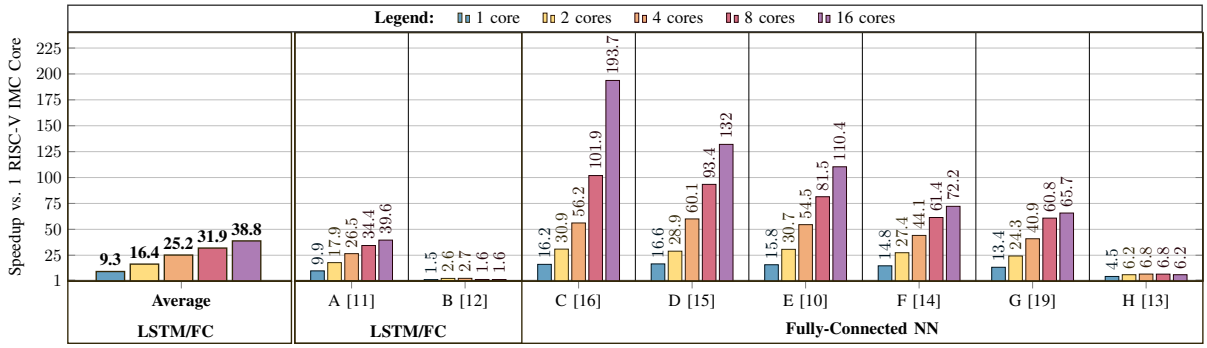
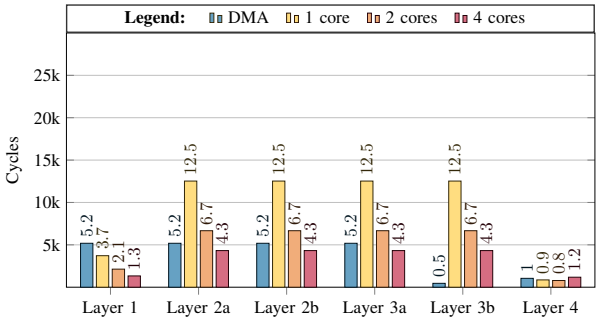
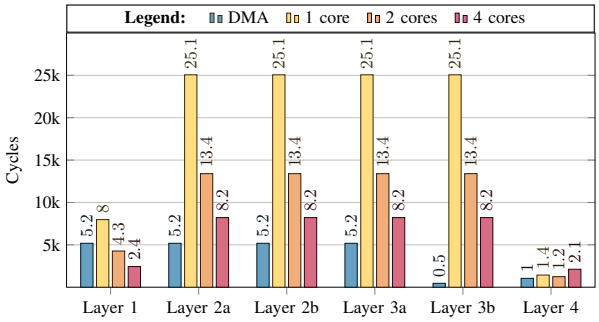


Fig. 10: Speedup when using multiple cores for typical Neural Networks workload in RRM when the whole model fits into L1. The single core speedup corresponds to the achieved speedup with all single-core implementations.



(a) Model E with tiling.



(b) Model E with tiling and batching.

Fig. 11: Layerwise breakdown of the computation time of each layer including the DMA transactions time for model E [10].

TABLE V: Measured operations/cycle on a optimized *single-core* implementation against a parallelizable *single-core* implementation, showing the parallelization overhead such as synchronization, tile determination etc.

	1 Core single-core [Op/Cycle]	**1 Core large cluster [Op/Cycle]**	**Parallelization Reduction [Op/Cycle]**
Model A [11]	1.92	1.85	0.07
Model C [16]	3.31	3.22	0.09
Model D [20]	3.37	3.31	0.06
Model E [10]	3.21	3.14	0.07
Model F [14]	3.04	2.93	0.11
Model G [19]	2.68	2.63	0.05
Average	*2.92*	*2.85*	*0.08 (2.6%)*

For these model sizes, only two cores in the *large cluster* configuration should be enabled as using more cores brings very limited performance gain while consuming more power.

The medium-sized models A [11], F [14], and G [19] with 24k–37k parameters achieve slightly higher speedups of 3.9×, 4.8×, and 4.7× when making use of all 16 available cores, respectively. The small speedup gain when using 16 PEs instead of 8 PEs clearly shows that using 16 PEs is again a waste of energy and area for these medium-sized models.

Models with bigger layers such as model C [16], D [20], and E [10] which each have multiple layers with $\geq$ 100 OFM sizes gain much more by using more cores and achieve speedups of 11.7×, 6.9× and 7.8× when using all 16 cores and speedups of 6.2×, 5.1× and 5.5× when using 8 cores against using the parallelized implementation on a single core.

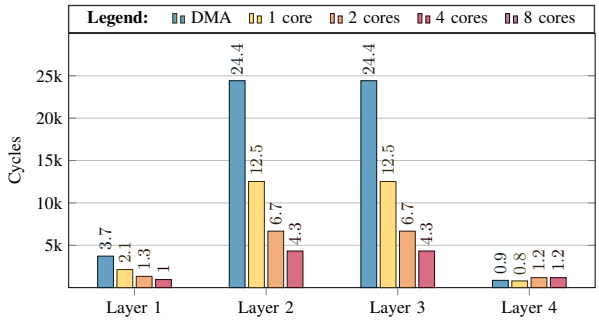


Fig. 12: Layerwise breakdown of the computation time of each layer including the DMA transcation time, which are overlapped with the computation for model E [10].

Directly comparing them against our baseline single RISC-V IMC core implementation, this corresponds to total speedups of 193.7×, 110.4×, and 132.0× on 16 cores and 101.9×, 81.5×, and 93.4× on eight cores.

4) Multi-Core - Benchmark Suite - Small Cluster: With the *smaller cluster* configuration including eight cores and only 64KB L1 memory, some form of model tiling is necessary. The first straight forward approach of tiling on the layer-level means that the model is loaded and computed in a layer-by-layer fashion with the double-buffering approach described in Section III-D6. However, model C [16], D [20], and E [10] have layers that do not entirely fit into the L1 memory and thus require further tiling. As most models are highly unbalanced,

any form of tiling is improving load balance. We focus on the OFM tiling because IFM tiling causes additional overhead for combining results across the cores.

In an exemplary case study, we show a detailed analysis of the applied tiling with its consequences on model E [10], covering both cases of layer imbalance and too big layer sizes. The first layer-wise computation cycle breakdown with staggered DMA transaction cycles (see Section III-D6) is shown in Figure 12. As the individual layers would be too big for the *small cluster* configuration, these specific measurements were taken from the *evaluation cluster* configuration. A first look on layer 1 shows clearly that the DMA transaction for the second layer (which is progressed in parallel to the layer 1 computation) is much higher than the computation time for the layer. As layers 2 and 3 are both too big to fit into L1, their necessary tiling coincidentally improves this load imbalance. Figure 11a shows that even with tiled layers, the aforementioned DMA transaction for the second layer cannot be hidden, and we are bandwidth-limited. When using more than two cores, the situation even gets worse, and we are bandwidth limited over almost all layers, making the usage of four or more cores questionable. We have applied similar tiling to all reasonable big models and have listed the achieved operations/cycles in Table IV. We count $1\times$ MAC as $2\times$ Operations. Overall we achieve up to 6.94 Op/cycle, 5.42 Op/cycle, and 2.98 Op/cycle for four, two, and one core, respectively. With the peak DMA bandwidth of 8 B/cycle, we lose on average 8.9%, 17.2%, 38.2% to the achieved optimal Op/Cycle on one, two, and four cores due to the heavy load-imbalance of the RRM benchmarks. We conclude that using four or more cores brings very limited gain unless the data bandwidth can be extended to $>$8 B/cycle.

5) Multi Core - Benchmark Suite Batching - Small Cluster: One possibility to improve the situation is batching. Meaning that we are computing on multiple IFM at the same time, allowing us to reuse the loaded parameters. As this doubles the computation time while keeping the DMA transaction time constant, we can push our example model E for most layers from bandwidth-limited into computation-limited operation. Figure 11b shows the tiled model E, where each layer is working simultaneously on two batched IFM. Batching solves the bandwidth limitations completely when using only a single core and almost entirely when using two or four cores. Additionally, the fully bandwidth-limited situation shifts from four cores to eight cores. Only for the single-core implementation, we reach the optimal case as in the *large cluster* configuration. However, it improves the situation significantly, as shown in Table IV. We achieve up to 10.17 op/cycle, 6.03 op/cycle, and 3.17 op/cycle for four, two, and one core correspondingly. The batching, therefore, enables an average loss of only 15.6% on the average optimal 10.3 op/cycle, making the usage of four cores in combination with a data bandwidth of 8 B/cycle reasonable. It should be noted, that this form of batching increases the latency of the model, which might violate the tight timing constraints for RRM decisions at the physical layer.

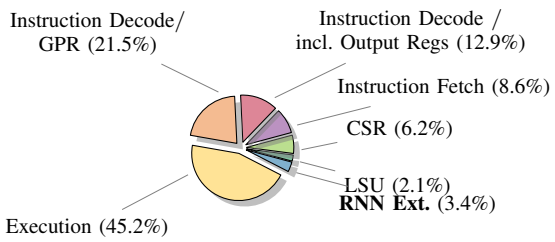


Fig. 13: Area Distribution of extended RI5CY core.

C. Hardware Implementation

To analyze the implementation of the extended RI5CY core, an 8-track low-threshold (LVT) standard cell library of the Globalfoundries 22 nm FDX technology was used. We used Synopsys Design Compiler 18.06 for synthesis and Cadence Innovus 18.11 for the back-end flow. The gate-level simulations with back-annotated delays for the power estimates were run with Modelsim Questa v2019.1 on the final layout.

The implemented extensions have no influence on the critical path which lays between the load-store unit and the memory in the write-back stage. The extended RI5CY achieves 380 MHz at 0.65 V at typical conditions at room temperature. However, the extensions introduce a small area overhead of 2.3 kGE which corresponds to 3.4% of the total core area. The area breakdown, shown in Figure 13 results from the final placed-and-routed layout. From the performance perspective, a single extended core performs the relevant benchmarks *on average* $10.6\times$ faster than the standard RISC-V core with RV32-IMC instructions and achieves 566 MMAC/s instead of 21 MMAC/s. When the core is using the extensions, the power consumption rises from 1.73 mW to 2.61 mW (51% total increase). While the decoder contributes little more power (approx. 5 µW), the higher power consumption is mainly due to the higher utilization of the compute units (ALU and MAC unit, i.e., 0.57 mW/33% of the total power), the increased GPR usage (0.16 mW/9%), and the higher use of the load-store unit (0.05 mW/3%). However, the overall energy efficiency at 218 GMAC/s/W shows a $10\times$ improvement. The power consumed by the memories L1, L2, an the rest of the system is not taken into account. Taking these energy costs into account would most likely result in energy-efficiency gains closer to the observed performance gains. The given $10\times$ energy-efficiency improvement can be taken as a safe lower bound on the overall energy-efficiency improvements for the RRM acceleration system.

D. Comparison with Related Work

Table VI compares performance and energy efficiency numbers of various related work: a hardwired accelerator [47], a GPU [48], a vector processor [49], our baseline RV32IMC core and our proposed RRM-ASIP. The GPU comes with the highest performance and slightly lower energy-efficiency than the hardwired accelerator. However, with our complete benchmark set of 3.2 MOp (an average of 320 kOp per model), the compute throughput of 11 TOp/s is disproportionately large. The vector processor supports floating-point operations, offering more numerical precision than needed by our applications,

resulting in an increased area cost and lower energy efficiency than our RRM ASIP. The hardwired accelerator provides the highest energy efficiency, however, hardwired accelerators are not flexible to adapt to the rapidly changing RRM field as new algorithms would require a costly HW redesign [37]. Even though our implementation focuses on 16 bits instead of 8 bits, on a single-ASIP system, we achieve >2× higher OP/cycle than a comparable commercially available dual-issue core (STM32H743). Adapting our HW and SW optimizations for 8-bit could be promising for less arithmetic sensitive applications. Overall, our customized ISA instructions are tailored to the needs of the targeted LSTM and MLP RRM models, while maintaining flexibility. Additionally, the number of compute units is adequately scaled to the average benchmark size.

TABLE VI: Comparison with related work.

	Tech. [nm^2]	Area [mm^2]	Arith.	Freq. [MHz]	Perf. [$\frac{OP}{cyc}$]	Perf. [$\frac{GOP}{s}$]	En.Eff. [$\frac{GOP/s}{W}$]
GPU [a][48]	12	<350	int8	1'377	-	11k	733.6
GPU [a][48]	12	<350	fp16	1'377	-	22k	366.6
Vector P. [49]	22	0.44	fp64	1'250	-	4.91	35.6
Vector P. [49]	22	2.16	fp64	1'040	-	32.4	40.8
ASIC [47]	65	14.4	int8	200	-	409.6	1'060
STM32 [50]	40	<49	int8	480	1.42	0.68	2.92
RV32IMC	22	0.91 / 0.0128 [c]	int16	380	0.11	0.042	24.3 [b]
RRM ASIP	22	0.92 / 0.0133 [c]	int16	380	2.98	1.13	433.7 [b]

[a] 8x64 tensor cores [b] power: core-only [c] complete system area / core-only area

VI. Conclusion

In this work we have identified a selected set of recently proposed real-world RRM-targeted benchmarks based on multi-layer perceptrons (MLP) and recurrent neural networks (RNNs). Starting from a baseline, simple RISC-V core, we first introduce instruction extensions coupled with software optimizations for the selected RRM benchmarks, and evaluate them on a single-core and multi-core cluster acceleration system. For the single-core acceleration system we demonstrate an energy-efficiency of 218 GMAC/s/W, and a throughput of 566 MMAC/s corresponding to an improvement of 10× and 10.6×, respectively, over the single-core system with a baseline RV32IMC core. For the multi-core acceleration system, we analyze the parallel speedup dependency on the input and output FM size for Fully-connected and LSTM layers, achieving up to 10.2× speedup with 16 cores over a single extended RI5CY core for single LSTM layers, and a speedup of 13.8× for single fully-connected layers. On the full RRM benchmark suite, we achieve an average overall speedup of 16.4×, 25.2×, 31.9× and 38.8× on two, four, eight, and 16 cores, respectively, compared to our single core RV32IMC baseline implementation.

Acknowledgment

We thank Matteo Spallanzani for the valuable discussions.

References

[1] N. D. Tripathi, J. H. Reed, and H. F. VanLandingham, *Radio resource management in cellular systems*. Springer Science & Business Media, 2006, vol. 618.

[2] S. Manap, K. Dimyati, M. N. Hindia, M. S. Abu Talip, and R. Tafazolli, "Survey of Radio Resource Management in 5G Heterogeneous Networks," *IEEE Access*, vol. 8, pp. 131 202–131 223, 2020.

[3] M. Naeem, K. Illanko, A. Karmokar, A. Anpalagan, and M. Jaseemuddin, "Optimal power allocation for green cognitive radio: fractional programming approach," *IET Communications*, vol. 7, no. 12, pp. 1279–1286, 2013.

[4] Q. Shi, M. Razaviyayn, Z.-Q. Luo, and C. He, "An iteratively weighted MMSE approach to distributed sum-utility maximization for a MIMO interfering broadcast channel," in *2011 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2011, pp. 3060–3063.

[5] K. I. Ahmed, H. Tabassum, and E. Hossain, "Deep Learning for Radio Resource Allocation in Multi-Cell Networks," *IEEE Network*, 2019.

[6] A. Y. Hannun and Others, "Deep Speech: Scaling up end-to-end speech recognition," *Computing Research Repository*, vol. abs/1412.5, 2014.

[7] H. Ze, A. Senior, and M. Schuster, "Statistical parametric speech synthesis using deep neural networks," in *IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, 2013, pp. 7962–7966.

[8] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Ł. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation," *arXiv*, pp. 1–23, 2016.

[9] M. Zanghieri, S. Benatti, A. Burrello, V. Kartsch, F. Conti, and L. Benini, "Robust real-time embedded emg recognition framework using temporal convolutional networks on a multicore iot processor," *IEEE transactions on biomedical circuits and systems*, vol. 14, no. 2, pp. 244–256, 2019.

[10] H. Sun, X. Chen, Q. Shi, M. Hong, X. Fu, and N. D. Sidiropoulos, "Learning to optimize: Training deep neural networks for wireless resource management," in *2017 IEEE 18th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*. IEEE, 2017, pp. 1–6.

[11] U. Challita, L. Dong, and W. Saad, "Proactive Resource Management for LTE in Unlicensed Spectrum: A Deep Learning Perspective," *arXiv*, 2017.

[12] O. Naparstek and K. Cohen, "Deep Multi-User Reinforcement Learning for Distributed Dynamic Spectrum Access," *IEEE Transactions on Wireless Communications*, vol. 18, no. 1, pp. 310–323, 2019.

[13] M. Eisen, C. Zhang, L. F. O. Chamon, D. D. Lee, and A. Ribeiro, "Learning Optimal Resource Allocations in Wireless Systems," *IEEE Transactions on Signal Processing*, vol. 67, no. 10, pp. 2775–2790, 2019.

[14] Y. S. Nasir and D. Guo, "Multi-Agent Deep Reinforcement Learning for Dynamic Power Allocation in Wireless Networks," *arXiv*, 2018.

[15] W. Lee, M. Kim, and D.-H. Cho, "Deep power control: Transmit power control scheme based on convolutional neural network," *IEEE Communications Letters*, vol. 22, no. 6, pp. 1276–1279, 2018.

[16] H. Ye and G. Y. Li, "Deep reinforcement learning for resource allocation in V2V communications," in *2018 IEEE International Conference on Communications (ICC)*. IEEE, 2018, pp. 1–6.

[17] M. Yao, M. Sohul, V. Marojevic, and J. H. Reed, "Artificial Intelligence Defined 5G Radio Access Networks," *IEEE Communications Magazine*, vol. 57, no. 3, pp. 14–20, 2019.

[18] E. Ghadimi, F. Davide Calabrese, G. Peters, and P. Soldati, "A reinforcement learning approach to power control and rate adaptation in cellular networks," in *IEEE International Conference on Communications*. Institute of Electrical and Electronics Engineers Inc., jul 2017.

[19] Y. Yu, T. Wang, and S. C. Liew, "Deep-reinforcement learning multiple access for heterogeneous wireless networks," *ieeexplore.ieee.org*, 2017.

[20] S. Wang, H. Liu, P. H. Gomes, and B. Krishnamachari, "Deep reinforcement learning for dynamic multichannel access in wireless networks," *IEEE Transactions on Cognitive Communications and Networking*, 2018.

[21] International Organization for Standardization/International Electrotechnical Commission and others, "Information Technology—Open Systems Interconnection—Basic Reference Model: The Basic Model," *ISO/IEC*, vol. 427, 1994.

[22] R. Andri, T. Henriksson, and L. Benini, "Extending the RISC-V ISA for Efficient RNN-based 5G RAdio Resource Management," in *Proceedings of the 57th Annual Design Automation Conference 2020*. ACM, 2020.

[23] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.

[24] A. Pullini, D. Rossi, I. Loi, G. Tagliavini, and L. Benini, "Mr.Wolf: An Energy-Precision Scalable Parallel Ultra Low Power SoC for IoT Edge Processing," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 7, pp. 1970–1981, jul 2019.

[25] I. Vaswani, Ashish and Shazeer, Noam and Parmar, Niki and Uszkoreit, Jakob and Jones, Llion and Gomez, Aidan N and Kaiser, Lukasz and Polosukhin, "Attention is all you need," *Advances in Neural Information Processing Systems*, vol. 30, pp. 5998–6008, 2017.

[26] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *International Conference on Machine Learning*, 2016, pp. 2849–2858.

[27] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2704–2713.

[28] G. L. Santos, P. T. Endo, D. Sadok, and J. Kelner, "When 5G meets deep learning: A systematic review," p. 208, sep 2020.

[29] Marvell, "Marvell OCTEON TX2 DPDK Overview," Tech. Rep., 2020.

[30] M. Yang, Y. Li, D. Jin, L. Su, S. Ma, and L. Zeng, "Openran: a software-defined ran architecture via virtualization," *ACM SIGCOMM computer communication review*, vol. 43, no. 4, pp. 549–550, 2013.

[31] "EdgeQ." [Online]. Available: edgeq.io

[32] "OpenHW." [Online]. Available: www.openhwgroup.org

[33] Intel Corp., "Intel®Architecture Instruction Set Extensions and Future Features Programming Reference," 2019. [Online]. Available: https://software.intel.com/en-us/download/intel-architecture-instruction-set-extensions-and-future-features-programming-reference

[34] J. Yiu, "Introduction to Armv8.1-M architecture," *ARM*, no. February, pp. 1–14, 2019.

[35] N. Neves, N. Sebastiao, D. Matos, P. Tomas, P. Flores, and N. Roma, "Multicore SIMD ASIP for Next-Generation Sequencing and Alignment Biochip Platforms," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 7, pp. 1287–1300, jul 2015.

[36] X. Guan, Y. Fei, and H. Lin, "Hierarchical design of an application-specific instruction set processor for high-throughput and scalable FFT processing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 3, pp. 551–563, mar 2012.

[37] O. S. Shahriar Shahabuddin, Aarne Mämmelä, Markku Juntti, "ASIP for 5G and Beyond: Opportunities and Vision," *IEEE Transactions on Circuits and Systems II: Express Briefs68*, vol. 68, no. 3, pp. 851 – 857, 2021.

[38] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs," *2018 International Conference on Hardware/Software Codesign and System Synthesis*, pp. 1–2, 2018.

[39] A. Garofalo, M. Rusci, F. Conti, D. Rossi, and L. Benini, "PULP-NN: accelerating quantized neural networks on parallel ultra-low-power RISC-V processors," *Philosophical Transactions of the Royal Society A*, vol. 378, no. 2164, p. 20190155, 2020.

[40] C.-W. Lin and J.-S. Wang, "A digital circuit design of hyperbolic tangent sigmoid function for neural networks," in *2008 IEEE International Symposium on Circuits and Systems*. IEEE, 2008, pp. 856–859.

[41] K. Leboeuf, A. H. Namin, R. Muscedere, H. Wu, and M. Ahmadi, "High speed VLSI implementation of the hyperbolic tangent sigmoid function," in *2008 Third International Conference on Convergence and Hybrid Information Technology*, vol. 1. IEEE, 2008, pp. 1070–1073.

[42] C.-H. Tsai, Y.-T. Chih, W. H. Wong, and C.-Y. Lee, "A hardware-efficient sigmoid function with adjustable precision for a neural network system," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 11, pp. 1073–1077, 2015.

[43] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[44] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, nov 1997.

[45] É. F. Zulian, G. Haugou, C. Weis, M. Jung, and N. Wehn, "System simulation with pulp virtual platform and systemc," in *Proceedings of the Conference on Rapid Simulation and Performance Evaluation: Methods and Tools*, 2020, pp. 1–7.

[46] VentureBeat.com, "GreenWaves Technologies unveils Gap8 processor for AI at the edge," 2018.

[47] S. Yin, P. Ouyang, S. Tang, F. Tu, X. Li, S. Zheng, T. Lu, J. Gu, L. Liu, and S. Wei, "A High Energy Efficient Reconfigurable Hybrid Neural Network Processor for Deep Learning Applications," *IEEE Journal of Solid-State Circuits*, vol. 53, no. 4, pp. 968–982, apr 2018.

[48] NVidia Corporation, "Jetson AGX Xavier Developer Kit," *Nvidia*, pp. 1–36, 2019. [Online]. Available: https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit

[49] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini, "Ara: A 1-ghz+ scalable and energy-efficient risc-v vector processor with multiprecision floating-point support in 22-nm fd-soi," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 530–543, 2019.

[50] A. Burrello, A. Garofalo, N. Bruschi, G. Tagliavini, D. Rossi, and F. Conti, "DORY: Automatic End-to-End Deployment of Real-World DNNs on Low-Cost IoT MCUs," aug 2020.

Gianna Paulin received her B.Sc. and M.Sc. degrees in Electrical Engineering and Information Technology from the Swiss Federal Institute of Technology Zurich (ETHZ), Switzerland, where she started as a Ph.D. student with the Integrated Systems Laboratory at the beginning of 2019. Her main interests lay in reduced precision deep learning from the algorithmic and hardware acceleration aspect with a focus on time series applications and low power embedded systems.

Renzo Andri received the B.Sc., M.Sc. and Ph.D. degree in Electrical Engineering and Information Technology at ETH Zurich in 2013, 2015, and 2020, respectively. His research focuses on energy-efficient machine learning acceleration from embeeded system design to full-custom IC design. He is currently working as senior researcher at the Computing Systems Laboratory, Zurich Research Center, Huawei Technologies Switzerland. In 2019, he has won the IEEE TCAD Donald O. Pederson Award.

Francesco Conti received the Ph.D. in electronic engineering from the University of Bologna, Italy, in 2016. He is an Assistant Professor in the DEI Department of the University of Bologna. From 2016 to 2020, he was a postdoctoral researcher at the Integrated Systems Laboratory of ETH Zürich in the Digital Systems group. He focuses on the development of deep learning based intelligence on top of ultra-low power, ultra-energy efficient programmable Systems-on-Chip – from both the hardware and software perspective. His work has resulted in 40+ publications in international conferences and journals and has been awarded several times, including the 2020 IEEE TCAS-I Darlington Best Paper Award.

Luca Benini is the Chair of Digital Circuits and Systems at ETH Zürich and a Full Professor at the University of Bologna. He has served as Chief Architect for the Platform2012 in STMicroelectronics, Grenoble. Dr. Benini's research interests are in energy-efficient system and multi-core SoC design. He is also active in the area of energy-efficient smart sensors and sensor networks. He has published more than 1'000 papers in peer-reviewed international journals and conferences, four books and several book chapters. He is a Fellow of the ACM and of the IEEE and a member of the Academia Europaea.