A Microcontroller is All You Need: Enabling Transformer Execution on Low-Power IoT Endnodes

A Microcontroller is All You Need: Enabling Transformer Execution on Low-Power IoT Endnodes

Alessio Burrello[1], Moritz Scherer[2], Marcello Zanghieri[1], Francesco Conti[1], Luca Benini[1,2]

***Abstract*—Transformer networks have become state-of-the-art for many tasks such as NLP and are closing the gap on other tasks like image recognition. Similarly, Transformers and Attention methods are starting to attract attention on smaller-scale tasks, which fit the typical memory envelope of MCUs. In this work, we propose a new set of execution kernels tuned for efficient execution on MCU-class RISC-V and ARM Cortex-M cores. We focus on minimizing memory movements while maximizing data reuse in the Attention layers. With our library, we obtain 3.4×, 1.8×, and 2.1× lower latency and energy on 8-bit Attention layers, compared to previous state-of-the-art (SoA) linear and matrix multiplication kernels in the CMSIS-NN and PULP-NN libraries on the STM32H7 (Cortex M7), STM32L4 (Cortex M4), and GAP8 (RISC-V IMC-Xpulp) platforms, respectively. As a use case for our TinyTransformer library, we also demonstrate that we can fit a 263 kB Transformer on the GAP8 platform, outperforming the previous SoA convolutional architecture on the TinyRadarNN dataset, with a latency of 9.24 ms and 0.47 mJ energy consumption and an accuracy improvement of 3.5%.**

***Keywords*—TinyML, Transformers, Deep Learning, Internet of Things**

I. Introduction

In the last few years, there has been a consistent trend towards moving computing workload from centralized facilities towards the extreme edge of the Internet of Things (IoT), improving privacy, decreasing latency, and decongesting communication networks [1]. IoT endpoints have been changing from simple Microcontroller Units (MCUs) dedicated to sensor data collection and transmission towards more advanced devices capable of performing near-sensor data analytics with traditional [2] and emerging Deep Learning algorithms [3] – while still being subject to the same strict power, performance, and cost constraints of much simpler MCUs.

For this reason, an extensive amount of research has been conducted on how to squeeze complex analytics algorithms into such constrained devices, with particular emphasis on Deep Neural Networks (DNNs) (by topological changes [4], data quantization [5], and pruning [6]). This set of efforts is often collectively labelled as *TinyML*. Even in TinyML, Convolutional Neural Networks (CNNs), a particular class of DNNs, are dominating for most tasks, from computer vision to temporal series analysis. On the other hand, Transformer [7], another class of Deep Learning algorithms, has emerged in the last few years competing and, in some cases starting to outmatch SoA CNNs. In 2017, Vaswani et al. [7] published the first results on the Transformer architecture, which outperformed SoA algorithms in different machine translation benchmarks. Transformers are composed mainly of stacks of Attention layers, a computational pattern that relates input elements together, regardless of their distance within the input sequence. Following their success in Natural Language Processing (NLP), Transformers have been successfully applied to tasks previously dominated by DNNs such as image classification [8]. However, Transformer models have so far been considered too large and too complex (requiring tens to hundreds of millions of parameters) to be executed on constrained platforms, such as MCUs, and have been applied only to large-scale problems unsuited to execution at the extreme edge of the IoT.

With this work, we aim to pave the way to the execution of Transformers on MCU-class devices – while simultaneously showing that *Tiny Transformers* can achieve high performance for realistically sized extreme-edge tasks. To this end, we present what, to the best of our knowledge, is the first effort to bring Transformer models on three commercial low-power MCUs – the single-core Cortex-M-based STM32H7[1] (M7) and STM32L4[2] (M4), and the 9-core (RISC-V IMC-Xpulp) GreenWaves Technologies GAP8[3]. More in detail, we propose the following novel contributions:

- We introduce a novel library of Attention kernels[4], which minimizes expensive data marshalling operations, such as reshaping and permutations, by using data layout and loop reordering tailored to each Attention sub-node.
- We compare our novel Attention kernels library for inference operations with implementations based on SoA public libraries, PULP-NN [9], and CMSIS-NN [10], on the STM32H7, the STM32L4, and GAP8. We obtain a speed-up of 3.4×, 1.8×, and 2.1×, respectively, reaching 0.61, 0.18, and 11.3 MAC/cycle.
- We show a pipeline to fuse CNNs with Transformer quantization and map full precision (`fp32`) architectures to ultra-low-power embedded devices. We reduce both memory occupation and complexity using `int8` quantization. Specifically, we integrate I-BERT [11] quantization strategies with the NEMO pipeline [12].

To demonstrate our Tiny Transformer flow on an end-to-end use case, we propose a novel Transformer-based architecture

[1]A. Burrello, M. Zanghieri, F. Conti, and L. Benini are with the Department of Electrical, Electronic and Information Engineering, University of Bologna, 40136 Bologna, Italy. name.surname@unibo.it

[2]M. Scherer and L. Benini are with the Department of Information Technology and Electrical Engineering at ETH Zurich, 8092 Zurich, Switzerland. lbenini@iis.ee.ethz.ch

978-1-6654-3156-9/21/$31.00 ©2021 IEEE

[1]www.st.com/en/microcontrollers-microprocessors/stm32h7-series.html

[2]www.st.com/en/microcontrollers-microprocessors/stm32l4-series.html

[3]www.greenwaves-technologies.com/gap8_smart_sensors/

[4]The library is released open source at https://github.com/pulp-platform/pulp-transformer

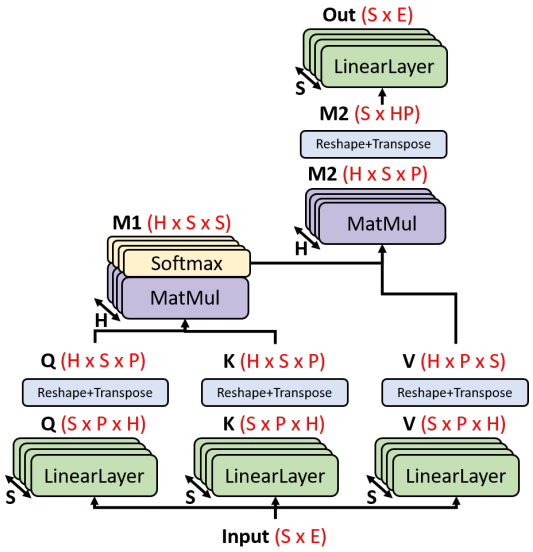

Fig. 1. Multi-Head Attention module.

for the TinyRadarNN dataset [13], which outperforms the baseline temporal DNN model presented in the same work by 3.5% in detection accuracy. When deployed onto GAP8 using our library, it consumes $0.47\,\mathrm{mJ}$ with a latency of $9.24\,\mathrm{ms}$, $9.6\times$ lower than the baseline with $6.3\times$ lower latency [13] and per-frame detection accuracy of 77.15%, an increase of 3.5% over the baseline.

II. Background & Related Work

A. Attention & Transformers

We call *attention*, in general, any layer used in machine learning models to emphasize "important" parts of data and depress "unimportant" ones, imitating the cognitive mechanism with the same name. Here we focus on *Multi-Head Self-Attention (MHSA)*, introduced by [7]. Sequence-to-sequence Transformers [14], [15] are mostly composed of stacks of MHSAs, and other kinds of Transformers also follow this line [8]. MHSA takes as input a tensor $\mathbf{X}$ of sequential data, with *sequence length* S and organized in E channels usually called *embeddings*; it produces an output of the same shape $S \times E$.

Internally, MHSA uses a set of H mutually independent parallel *heads*, all of which perform an identical set of operations divided into three steps. In the first step, each element of the sequence $\mathbf{X}$ is projected from the space of embeddings of size E to three separate *projection spaces* each of size P, known as *queries* $\mathbf{Q}$, *keys* $\mathbf{K}$ and *values* $\mathbf{V}$ – using three trainable linear transforms:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_{\text{query}} \qquad \mathbf{K} = \mathbf{X}\mathbf{W}_{\text{key}} \qquad \mathbf{V} = \mathbf{X}\mathbf{W}_{\text{value}} \tag{1}$$

where $\mathbf{W}_{\text{query}}$, $\mathbf{W}_{\text{key}}$ and $\mathbf{W}_{\text{value}}$ are all matrices of size $E \times P$. In the second step, $\mathbf{Q}$, $\mathbf{K}$ and $\mathbf{V}$ are used as inputs to the core attention mechanism, *scaled dot-product attention*, which is defined as

$$\text{Attention}(\mathbf{Q},\mathbf{K},\mathbf{V}) \doteq \mathbf{A}\mathbf{V} \doteq \underset{\text{over keys}}{\text{SoftMax}}\left(\frac{\mathbf{Q}\mathbf{K}^{\text{T}}}{\sqrt{P}}\right)\mathbf{V} \tag{2}$$

where $\mathbf{A}$ is an $S \times S$ matrix called the *attention matrix*. The intuition behind this is that MHSA can learn to understand which elements in the sequence are relevant with respect to one another (through $\mathbf{A}$), and can use this information to gate $\mathbf{V}$. Finally, in the third step, the output of scaled dot-product attention from all heads is projected back to the original embedding space with another linear layer, resulting in an output of size $S \times E$.

MHSAs can be expressed with three input Linear layers, the scaled dot-product attention, and one output Linear layer by properly reorganizing the dimensions at each step. Fig. 1 shows a MHSA module expressed in this way, indicating the shape of all tensors (including the H dimension) and the required reshape/transpose operations.

B. Related Works

1) Vision Transformers: After the breakthroughs in NLP-related tasks [14], [15], Transformers have been applied in other fields, most remarkably in computer vision. For instance, [8] divides images into patches, compresses the intra-patch 2D space with a single convolutional layer, and uses a Transformer architecture to classify the image as a sequence of patches. [16] uses pointwise convolutions to compare pixel images with keys intended as prototypes. In addition, [17] explores ways to add more convolutions while preserving the Transformer's efficiency. In this work, we propose a Tiny Transformer module for radar-based gesture recognition. To the best of our knowledge, we are the first to apply such a Transformer-based architecture to a small-scale edge-related task, namely gesture recognition.

2) Quantized Transformers: I-BERT[5] [11] was the first integer-only quantization scheme for Transformers, but it does not push an actual device-level implementation. It quantizes both weights and activations to $8\,\mathrm{bit}$; its quantization is uniform (i.e., the relationship between real and quantized values is linear), and symmetric around 0. I-BERT Softmax is the only kernel computed in $32\,\mathrm{bit}$, though always integer (`int32`). Previously, only *fake-quantization* (also called *simulated quantization*) schemes were proposed, which are not fully integerized since they still keep `float32` computations for some layers or stages (e.g., the Softmax). Q-BERT [18] executes the whole inference in floating-point, quantizing only the outputs of Attention modules. Also, Q8-BERT [19] applies weights quantization to $8\,\mathrm{bit}$, but keeps `float32` operations. LQ-NET [18] uses non-uniform quantization, which can better approximate the uneven distributions of weights and activations compared to uniform approaches. For a more detailed overview of these quantization schemes for Transformers, we refer to [11] and [20].

III. Methods

A. Self-Attention Kernels

As shown in Fig. 1, the Multi-Head Self-Attention operator comprises four Linear and two Matmul layers, each followed by memory marshalling operations such as transposition, reshape, and concatenation, which can cause performance overheads up to 20% [21], [22]. Even when these kernels are

[5]https://github.com/kssteven418/i-bert

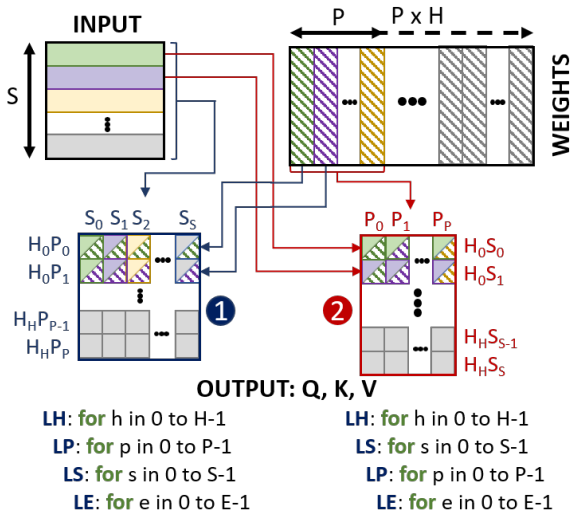

Fig. 2. Linear layer data flow for HPS and HSP out data layouts. Output matrices are filled from left to right, top to bottom.

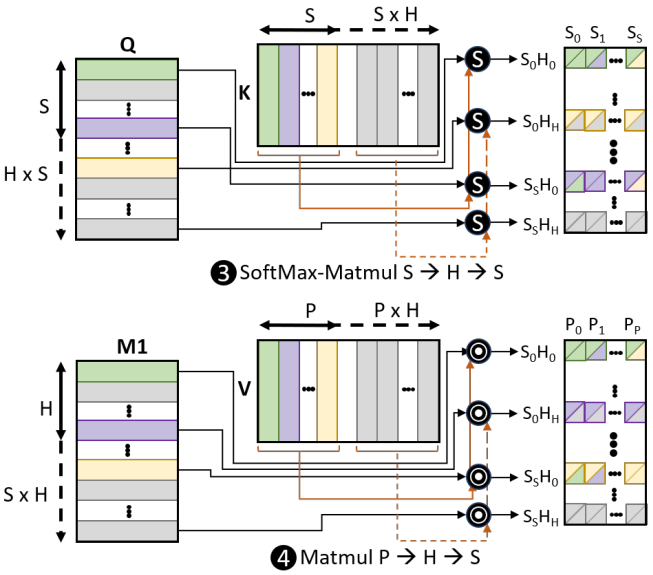

Fig. 3. Matrix multiplication designed to work with multiple heads with different data layouts.

individually optimized using state-of-the-art libraries [9], [10], they result in low data reuse and non-optimal parallelization scaling. Further, parallelizing a kernel on a dimension smaller than the number of cores leads to suboptimal speed-up. We present a new set of tailored kernels for each of the internal operations to address these issues. All kernels work on 8 bit input and output tensors, following I-BERT [11].

1) Linear Layers: Fig. 2 depicts two distinct implementations for the three input Linear layers, i.e., layers that can be reduced to a matrix-matrix multiplication. Implementation ❶, which we call *Weight-Reuse Linear* (WRL), is used to project the $\mathbf{V}$ tensor from $\mathbf{X}$, while ❷, called *Input-Reuse Linear* (IRL), is used for $\mathbf{Q}$ and $\mathbf{K}$. These two kernels differ in i) loop ordering and ii) data layouts. We force the WRL kernel to produce output data in the HPS layout to remove the data reshaping operator (see Fig. 1). Therefore, we order the loop as $H \rightarrow P \rightarrow S \rightarrow E$, from outermost to innermost. When targeting multi-core platforms such as GAP8, we parallelize our kernel on the H dimension, thus splitting the outermost loop across cores.

On the other hand, the IRL's required layout is HSP, and thus we force the loop nest to $H \rightarrow S \rightarrow P \rightarrow E$, from outermost to innermost. In this kernel, at every iteration of the S loop, a single input time sample ($1 \times E$) is multiplied by a weight-head ($E \times P$). The parallelization is identical to the WRL case.

The last Linear layer, which projects the output tensor of the matrix multiplication to the final output, uses the $S \rightarrow E \rightarrow H \times P$ loop order, parallelizing on E.

2) Matrix Multiplications: To optimize matrix multiplications, we optimally order the loop executions and parallelize over the outermost dimensions to improve performance. Fig. 3 reports two different implementations for the two matrix multiplications in the Self-Attention kernel. The Softmax-Matmul ❸ merges the matrix multiplication with the Softmax operator; it uses the $S \rightarrow H \rightarrow S$ loop order; P is the dimension over which the reduction is performed. After completing each iteration of the H loop, the Softmax is applied to the data produced (e.g., the first one, S_0H_0). We store the output data in the SHS data layout to allow the second Matmul ❹ to ingest data sequentially. This implies reading input data for the Softmax-Matmul layer in non-sequential order, but still with a regular stride, thus not impairing the performance. For instance, a set of weights, H_0 (dimensions, $P \times S$), is multiplied with the corresponding activation buffer, H_0S_0 (dimensions, $1 \times P$). Then, the set of weights H_1 is multiplied with the input H_1S_0, which is stored S input buffers after the first one (with a stride of $S \times P$). After all head positions relative to sequence S_0 are multiplied with the K matrix, the cycle is repeated for each S_i of Q input.

Matmul ❹ produces the final tensor, which is then fed to the output projection Linear layer. Given the design of the previous layers ❶ and ❸, its implementation is straightforward. The loop execution order is $S \rightarrow H \rightarrow P$, with P as innermost dimension. The reduction dimension is the innermost S of the $\mathbf{M1}$ matrix.

3) Kernel execution loops: We follow three primary guidelines for kernel optimizations: i) keep the parallelization (when available on the target platform) as much as possible on the H dimension; ii) exploit output stationarity; and iii) produce the output tensors sequentially (i.e., element $i + 1$ in the innermost dimension is always generated immediately after the i-th element).

The first guideline is particularly convenient when a low number of cores is available, given the typically low number of heads present in networks (e.g., eight heads in original transformer [7]). Furthermore, the heads dimension is present in all the kernels[6]. The second guideline allows saving memory by avoiding the storage of many intermediate `int32` accumulators for the partial outputs, as described in [23]. Besides the memory saving, output stationarity enables optimal ex-

[6]The output Linear layer represents an exception, as we parallelize on E to maintain output stationarity.

```
Inputs: I, W; Outputs: Q, K, V
C = H / CORES
H_start = min(C * CORE_ID, H); H_end = min(H_start + C, H)
  LH: for (h = H_start; h < H_end; h++)
    LP: for (p = 0; p < P/2; p++)
      LS: for (s  = 0; s < S/4; s++)
        S0...7 = {0};
        LE: for (e = 0; e < E/4; e++)
          A1 = I(4s); A2 = I(4s + E);
          A3 = I(4s + 2E); A4 = I(4s + 3E);
          B1 = W(hpE + 2pE); B2 = I(hpE + (2p+1)E);
          S0 += sdot4(A1,B1); S1 += sdot4(A1,B2);
          S2 += sdot4(A2,B1); S3 += sdot4(A2,B2);
          S4 += sdot4(A3,B1); S5 += sdot4(A3,B2);
          S6 += sdot4(A4,B1); S7 += sdot4(A4,B2);
        O(h,2p,4s) = quant(S0);
        O(h,2p,4s+1) = quant(S1);
        O(h,2p,4s+2) = quant(S2);
        O(h,2p,4s+3) = quant(S3);
        O(h,2p+1,4s) = quant(S4);
        O(h,2p+1,4s+1) = quant(S5);
        O(h,2p+1,4s+2) = quant(S6);
        O(h,2p+1,4s+3) = quant(S7);
```

Listing 1: Example of kernel pseudocode of sub-layer ❶.

ploitation of the dot-product Single Instruction Multiple Data (SIMD) instructions, as demonstrated in [9]. Finally, producing output tensors sequentially prevents undesired additional operations in the innermost loop to compute storage locations.

Listing 1 reports an example of the pseudocode of layer ❶ with the `RV32IMCXpulpV2` ISA and GAP8 target. In the innermost loop, we exploit the `sdot4` operator to perform 4 Multiply-and-Accumulate (MAC) operations in a single instruction. Further, inspired by [9], we perform 8 `sdot4` operations in the same loop iteration, thus eliminating Read-After-Write (RAW) hazards and performing just 6 load operations (4 activations buffers and 2 weights buffers), better exploiting the data reuse in the register file. Incrementing the number of produced output values in a single iteration, e.g., to 16, would cause an increase in the number of utilized registers to 24 (16 for outputs, 4 for inputs, 4 for weights), causing additional load and store operations to spill variables from the register file to the stack to make room for operands. Conversely, reducing the number of registers employed causes a reduction in the MAC/load ratio and impairs the performance.

B. Quantization

Since commercial off-the-shelf MCUs feature memory in the order of hundreds of kilobytes up to one megabyte, quantization of both weights and activations is typically used to reduce the memory footprint of trained networks [6], [24]. Besides saving precious memory space on these tightly constrained devices, quantization to 8 bit allows specialized microcontrollers to leverage SIMD instructions, which lead to significant speed-up with respect to floating-point computations. To quantize the Self-Attention layers and allow for deployment onto commercial MCUs using our kernels as well as the baseline kernels, we use the NEMO toolchain [12]. The NEMO toolchain is used to perform post-training quantization on the floating-point model to an 8 bit integer model after training. We add dedicated quantized operators (e.g., Softmax) from I-BERT [11] to quantize the Self-Attention layers fully.

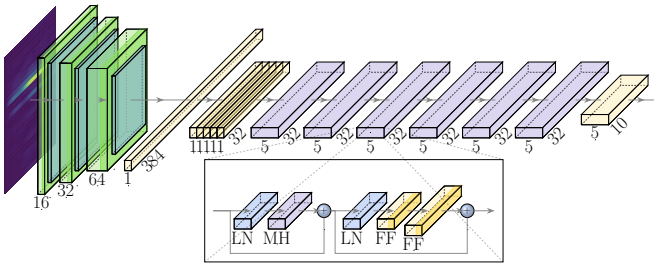

Fig. 4. Overall architecture of the proposed network. The front end is composed of 3 blocks of pointwise convolution, depthwise convolution and pooling, followed by a Linear layer. Each of the six encoder blocks (in purple) consists of layer norm layers (LN), a Multi-Head Attention layer (MH) and Linear layers (FF).

C. TinyRadar Transformer

We use the TinyRadar network and its corresponding dataset [13] as a use case to demonstrate both the feasibility and the advantages of porting Transformers on edge using the proposed kernels. TinyRadarNN consists of a CNN stage, a Temporal Convolutional Neural Network (TCN) stage and a dense layer stage [13]. The TinyRadar dataset consists of a total of over 10000 recordings of 11 hand gestures by 26 people made with a short-range radar. In the original paper, the data is arranged in spatio-temporal windows that contain samples of the radar reflection amplitude sampled after different amounts of time-of-flight, called range points, on the columns, with consecutive distance samples concatenated along the rows. The working principle of the network architecture is based on the splitting of spatial and temporal modelling of the gesture recognition problem.

Here, we preserve this structure but replace the TCN stage, which models the long-term temporal dependencies, with a more sophisticated 6-layer transformer encoder architecture. Special care was taken to fit the tight memory constraints of the GAP8 MCU, which features 512 kB of L2 RAM. The proposed architecture is shown in Figure 4. We use a CNN stage with 3 blocks of convolutions, each followed by a batchnorm, ReLU6 activation layer and pooling. At each block, the input resolution is reduced while the number of output channels scales from 16 to 64. A final Linear layer projects the input sample to the embedding dimension $E = 32$. In the second and third stage, we replace the convolutional layers of TinyRadarNN with separable convolutions made up of a depthwise layer followed by pointwise convolution, like in MobileNetV1 [4]. This allows to significantly reduce the size and the number of operations of the convolution frontend. Using $S = 5$ processed input time samples as a sequence, a 5×32 input is fed to the Transformer backend. The 6 layers which constitute the backend are identical to the ones of [8], with $S = 5$, $E = 32$, $P = 32$, and $H = 8$. Finally, the output of the encoder is fed to a dense layer which is used as a classifier, returning a prediction for each time step.

Besides the changes in architecture, we down-sample the inputs by employing 1 sensor and 246 range points instead of 2 sensors and 492 range points to reduce the total number of operations without impairing accuracy.

IV. Experimental Results

For the sake of space, here we report the results of our kernels on a single Multi-Head Self-Attention layer with $H = 16$, $P = 64$, $E = 64$, and $S = 32$. Note that we benchmark our library in a wide range of (H, P, E, S) tuples, obtaining similar results. As benchmarking platforms, we selected three state-of-the-art single and multi-core MCUs; i) the STM32H7 and the STM32L4, comprising one core ARM M7 and one core ARM M4, respectively. ii) GAP8, a commercial PULP SoC including a RISC-V processor (*fabric controller*) and a *cluster* of 8 additional RISC-V cores, using the `RV32IMCXpulpV2` Instruction Set Architecture (ISA); We set the operating frequencies of the GAP8, STM32H7, and STM32L4 at 100 MHz, 480 MHz, and 80 MHz, with a corresponding average power consumption of 51 mW, 234 mW, and 10 mW, respectively. We choose these three operating points as they are the most energy-efficient ones according to [23].

A. Kernel performance

Column 'Our Work' of Table I details the performance of our library on top of the three MCUs. The layer analyzed with $H = 16$, $P = 64$, $E = 64$, and $S = 32$ has 10.48 MMAC and 262 k parameters and thus fits the on-chip memory of all three platforms. Overall, our library allows Attention layers to achieve performance comparable to the best performing convolutional layers with both ISAs, achieving 11.29 MAC/cycle and 0.61 MAC/cycle on GAP8 and STM32H7, respectively, compared to 12.86 MAC/cycle and 0.71 MAC/cycle for convolutions [23]. The benchmark layer runs on the three platforms in 929 kcycles, 59.4 Mcycles, 17.2 Mcycles, respectively, with a latency of 9.29 ms (GAP8), 35.77 ms (STM32H7), and 741.93 ms (STM32L4). This performance gap is caused mainly by two factors: first, GAP8 has 8 cores vs the single-core in STM32L4/STM32H7, with a maximum speedup of 8×. Second, we fully exploit `RV32IMCXpulpV2` ISA 8 bit SIMD operations to speed up our code. At the same time, with ARM ISA, we are limited to 16 bits SIMD, which results in a reduction of peak single-core performance of 2× and also requires additional data casting[7]. The remaining speedup of GAP8 (18.5× vs STM32H7, and 63.9× vs STM32L4 in MACs/cycle) is attributable to `RV32IMCXpulpV2` micro-architectural features such as hardware loops.

1) Comparison with the state-of-the-art: We compare our transformer deployment with state-of-the-art CNN libraries, CMSIS-NN [10] and PULP-NN [9]. In particular, we exploit the optimized Linear layer kernels of these libraries as a base function for the matrix multiplications and projection layers, adding extra external loops, the SoftMax operator and the memory marshalling operators. Fig. 5 depicts a detailed study on performance improvement of every single part of the Attention layer. On the STM32L4, GAP8, and STM32H7, we obtain a speedup of 1.8×, 2.1×, and 3.3×, respectively. However, the different sublayers demonstrate different speedsup on the three platforms. Starting from GAP8, the three different components, Linear layers, matrix multiplications,

[7]The new generation ARM M55 and the new ARM ISA will also support 8 bits SIMD instructions.

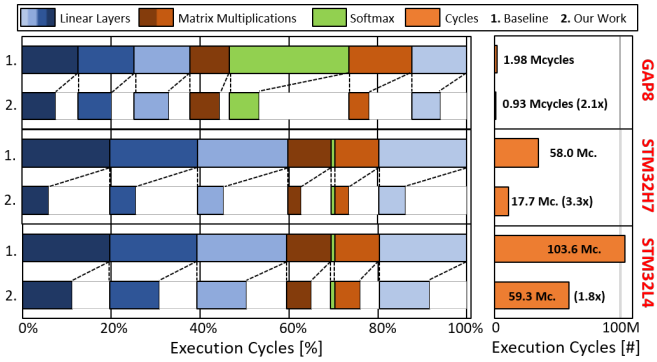

Fig. 5. Performance description of baselines and our kernels on the three different platforms. The x scales are different for each platform given the extremely different upper limits.

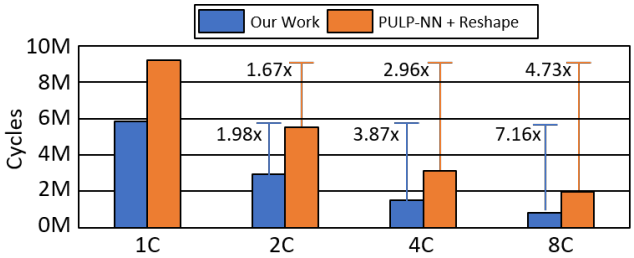

Fig. 6. Parallelization of the attention layer with 1, 2, 4, and 8 cores on GAP8.

and Softmax, demonstrate a speedup of 1.7×, 2.0×, and 4.0×, respectively. These speedups are due to better data reuse and the removal of reshaping layers. Furthermore, with the parallelization scheme on heads, with each core taking care of a portion of the heads of the Multi-Head Attention, we can speed up the SoftMax execution by up to 8×. On the other hand, parallelizing on S requires synchronization of all the cores after the computation of a single parallelized sequence S of data[8]. Fig. 6 details further the speedups on GAP8 of our library compared to the PULP-NN baseline. While the baseline only achieves 4.73× speedup with 8 cores compared to one, we can reach 7.16×, with an improvement of 1.51×.

On the STM32 platforms, the speedup is solely concentrated in Linear and matrix multiplication layers. Remarkably, despite using the same ISA, we observe a dramatic higher speedup of 3.3× compared to 1.8× between the STM32H7 to the STM32L4. This difference is mostly given by exploiting the dual-issue pipeline and the cache refill on the H7. Our kernels significantly boost data locality and reuse, allowing for better cache utilization than the CMSIS-NN baseline. To confirm this fact, disabling the caches reduces the relative speedup of our kernels compared to the baseline from 3.3× to 2.4×.

B. TinyRadar Transformer performance

We compare the proposed TinyRadar Transformer architecture both with the original TinyRadarNN and with a modified TCN architecture where we apply all changes explained in Section III-C, but we keep the TCN encoder, resulting in a

[8]We observe a speedup of only 4× in this case since we execute four tiles with 4 heads on GAP8 due to the memory constraint of using L1 [23].

TABLE I
COMPARISON OF OUR KERNEL LIBRARY WITH PULP-NN AND CMSIS-NN ONTO THREE COMMERCIAL MCUS.

MCU	1×RISC-V + 8×RISC-V		1×CortexM4		1×CortexH7	
Power [mW]/ Freq. [MHz]	51 mW / 100 MHz		10 mW / 80 MHz		234 mW / 480 MHz	
Kernels	Our Work	PULP-NN+Reshape	Our Work	CMSIS-NN+Reshape	Our Work	CMSIS-NN+Reshape
Layer: Attention - Heads: 16, Projection: 64, Sequence: 32, Embedding: 64, Operations: 8.4M						
Cycles	929k	1.95M	59.4M	103.59M	17.17M	57.0M
Time/Inference [ms]	9.29	19.52	741.93	1294.92	35.77	118.74
Energy[mJ]	0.47	1.00	7.42	12.95	8.37	27.79
MACs/cycle	11.29	5.37	0.18	0.10	0.61	0.18
Throughput [GMAC/s]	1.13	0.54	0.014	0.008	0.29	0.09
En.Efficiency [GMACs/s/W]	22.13	10.53	1.41	0.81	1.25	0.38

TABLE II
PERFORMANCE OF OUR PROPOSED ARCHITECTURE AT FR = 100MHZ ON GAP8 PLATFORM. ABBREVIATIONS: AT.: ATTENTION. FF: LINEAR LAYERS IN TRANSFORMER BACKEND.

	PULP-NN + Reshape			Our Work		
	CNN	Att.	FF	CNN	Att.	FF
MACS [#]	1.87M	1.06M	185k	1.87M	1.06M	185k
Cycles	717.4k	261.6k	94.5k	717.4k	112.5k	94.5k
Lat. [ms]	7.17	2.62	0.95	7.17	1.12	0.95
E. [mJ]	0.37	0.13	0.05	0.37	0.06	0.05

very small network ($\sim$30.7 kparam). We report all accuracy values post-quantization at `int8` precision. Our transformer-based network architecture achieves an accuracy of 77.15% on the TinyRadar dataset, outperforming the original architecture by 3.5% and the modified TCN architecture by $\sim$5%.

This is achieved without sacrificing deployability: the TinyRadar Transformer contains a total of 263 k parameters, which fits the L2 on-chip memory of GAP8. Tab. II reports the network's performance running on GAP8 at 100 MHz. The network achieves as low as 9.24 ms latency and 0.47 mJ, 9.6× and 6.3× lower than the original TinyRadarNN. Specifically, using our new library, we improve the performance of the Attention part of 2.32×, with a 14% direct reduction of the overall cycles of the network. We remark that the indirect effect of the insertion of the Transformer back-end is much more dramatic: speed-oriented architectural changes such as replacing standard convolutions with separable ones would not be attractive due to the related accuracy loss – but the Transformer back-end more than compensates this effect.

V. CONCLUSIONS

In this work, we paved the way to the deployment of Transformer networks on MCUs, with a set of optimized yet general kernels, which exploit ARM and RISC-V ISA to improve attention layers' performance. We showed a speed-up from 1.8× to 3.3× on three commercial MCUs compared to State-of-the-Art (SoA) convolutional libraries augmented with data reshaping and operators needed by the attention algorithm. Furthermore, we demonstrate with a use case the application of transformers to the TinyML field, improving the SoA performance on the TinyRadar dataset by 3.5%. Deploying this Tiny Transformer on the GAP8 platform, we reach a latency of 9.24 ms, with an energy consumption of 0.47 mJ per classification, with an an improvement 9.6×, 6.3× over the performance and energy of the previous SoA network, which is fully dominated.

REFERENCES

[1] S. Madakam *et al.*, "Internet of Things (IoT): A Literature Review," *Journal of Computer and Communications*, vol. 3, no. 05, p. 164, 2015.

[2] A. Burrello *et al.*, "Embedded Streaming Principal Components Analysis for Network Load Reduction in Structural Health Monitoring," *IEEE Internet of Things Journal*, 2020.

[3] D. Rossi *et al.*, "4.4 A 1.3 TOPS/W@ 32GOPS Fully Integrated 10-Core SoC for IoT End-Nodes with 1.7 μW Cognitive Wake-Up From MRAM-Based State-Retentive Sleep Mode," in *2021 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2021.

[4] A. G. Howard *et al.*, "Mobilenets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv preprint arXiv:1704.04861*, 2017.

[5] J. Choi *et al.*, "PACT: Parameterized Clipping Activation for Quantized Neural Networks," 2018.

[6] S. Han *et al.*, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," in *Proc. International Conference on Learning Representations*, 2016.

[7] A. Vaswani *et al.*, "Attention is All You Need," *arXiv preprint arXiv:1706.03762*, 2017.

[8] A. Dosovitskiy *et al.*, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," *arXiv preprint arXiv:2010.11929*, 2020.

[9] A. Garofalo *et al.*, "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.

[10] L. Lai *et al.*, "CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs," *arXiv preprint arXiv:1801.06601*, 2018.

[11] S. Kim *et al.*, "I-BERT: Integer-only BERT Quantization," *arXiv preprint arXiv:2101.01321*, 2021.

[12] F. Conti, "Technical Report: NEMO DNN Quantization for Deployment Model," 2020.

[13] M. Scherer *et al.*, "TinyRadarNN: Combining Spatial and Temporal Convolutional Neural Networks for Embedded Gesture Recognition with Short Range Radars," *IEEE Internet of Things Journal*, pp. 1–1, 2021.

[14] J. Devlin *et al.*, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[15] Y. Liu *et al.*, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[16] B. Wu *et al.*, "Visual Transformers: Token-based Image Representation and Processing for Computer Vision," 2020.

[17] H. Wu *et al.*, "CvT: Introducing Convolutions to Vision Transformers," 2021.

[18] D. Zhang *et al.*, "LQ-Nets: Learned Quantization for Highly Accurate and Compact Deep Neural Networks," 2018.

[19] O. Zafrir *et al.*, "Q8bert: Quantized 8bit Bert," *arXiv preprint arXiv:1910.06188*, 2019.

[20] S. Singh *et al.*, "The NLP Cookbook: Modern Recipes for Transformer based Deep Learning Architectures," 2021.

[21] A. Ivanov *et al.*, "Data Movement is All You Need: A Case Study on Optimizing Transformers," *arXiv e-prints*, pp. arXiv–2007, 2020.

[22] H. Wang *et al.*, "SpAtten: Efficient Sparse Attention Architecture with Cascade Token and Head Pruning," *preprint arXiv:2012.09852*, 2020.

[23] A. Burrello *et al.*, "DORY: Automatic End-to-End Deployment of Real-World DNNs on Low-cost IoT MCUs," *IEEE TComp*, 2021.

[24] S. Han *et al.*, "Learning Both Weights and Connections for Efficient Neural Networks," in *Proc. NIPS*, 2015, p. 1135–1143.