Optimizing the Deployment of Tiny Transformers on Low-Power MCUs

Optimizing the Deployment of Tiny Transformers on Low-Power MCUs

Victor J.B. Jung, Alessio Burrello, Moritz Scherer, Francesco Conti, Luca Benini

Abstract—Transformer networks are rapidly becoming State of the Art (SotA) in many fields, such as Natural Language Processing (NLP) and Computer Vision (CV). Similarly to Convolutional Neural Networks (CNNs), there is a strong push for deploying Transformer models at the extreme edge, ultimately fitting the tiny power budget and memory footprint of Micro-Controller Units (MCUs). However, the early approaches in this direction are mostly ad-hoc, platform, and model-specific. This work aims to enable and optimize the flexible, multi-platform deployment of encoder Tiny Transformers on commercial MCUs. We propose a complete framework to perform end-to-end deployment of Transformer models onto single and multi-core MCUs. Our framework provides an optimized library of kernels to maximize data reuse and avoid unnecessary data marshaling operations into the crucial attention block. A novel Multi-Head Self-Attention (MHSA) inference schedule, named Fused-Weight Self-Attention (FWSA), is introduced, fusing the linear projection weights offline to further reduce the number of operations and parameters. Furthermore, to mitigate the memory peak reached by the computation of the attention map, we present a Depth-First Tiling (DFT) scheme for MHSA tailored for cache-less MCU devices that allows splitting the computation of the attention map into successive steps, never materializing the whole matrix in memory. We evaluate our framework on three different MCU classes exploiting ARM and RISC-V Instruction Set Architecture (ISA), namely the STM32H7 (ARM Cortex M7), the STM32L4 (ARM Cortex M4), and GAP9 (RV32IMC-XpulpV2). We reach an average of 4.79 × and 2.0 × lower latency compared to SotA libraries CMSIS-NN (ARM) and PULP-NN (RISC-V), respectively. Moreover, we show that our MHSA depth-first tiling scheme reduces the memory peak by up to 6.19 ×, while the fused-weight attention can reduce the runtime by 1.53 ×, and number of parameters by 25 %. Leveraging the optimizations proposed in this work, we run end-to-end inference of three SotA Tiny Transformers for three applications characterized by different input dimensions and network hyperparameters. We report significant improvements across the networks: for instance, when executing a transformer block for the task of radar-based hand-gesture recognition on GAP9, we achieve a latency of 0.14 ms and energy consumption of 4.92 µJ, 2.32 × lower than the SotA PULP-NN library on the same platform.

Index Terms—Deep Neural Networks, Transformers, Micro-Controller Units, Edge Computing, DNN Acceleration

1 Introduction

In the last few years, there has been a significant trend towards moving computing workload from centralized facilities towards the extreme edge of the Internet of Things (IoT), improving privacy, efficiency, and ensuring a predictable and constant latency, independent from the network coverage and congestion [1]. As a consequence, modern IoT endpoints have been changing from simple Micro-Controller Units (MCUs) equipped with the sensor infrastructure for data collection and transmission towards more complex heterogeneous Systems-on-Chip (SoCs) characterized by dedicated accelerators and enhanced Instruction Set Architectures (ISAs) to optimize efficient on-board computations – while still meeting to tight power, performance, and cost constraints. For this reason, extensive research has been conducted on how to squeeze complex Machine Learning (ML) models into such constrained devices with a strong emphasis on Deep Neural Networks (DNNs) due to their success in solving many challenging tasks. Standard hardware-agnostic optimization techniques include topological changes [2], data quantization [3], [4], and pruning [5]. These efforts are often collectively labeled as *TinyML* and are applied to State of the Art (SotA) Convolutional Neural Networks (CNNs), such as ResNets [6] or MobileNets [7], that dominates most of the tasks of the IoT domain, including image recognition, object detection, or time-series analysis.

At the same time, increasing efforts have been dedicated to finding a successor architecture to CNNs for IoT processing to increase tasks' accuracy further. A strong candidate is the Transformer architecture [8], which first emerged as SotA model in Natural Language Processing (NLP) and is now also SotA in other domains such as Computer Vision (CV) and Audio processing. Transformer models have been initially developed targeting "at scale" deployment in the cloud. Industry and academia are now increasing efforts toward Transformer models tuned for edge deployment. Recent advancements have focused on the minimization of Transformer model size [9] while still maintaining leading-edge accuracy when compared with CNN models of similar size, thus introducing a novel class of Tiny Transformers. The success of these early reduced-scale Transformer models confirms that Tiny Transformers are relevant and applicable in Edge computing scenarios.

However, the Multi-Head Self-Attention (MHSA) operation comes with many challenges, such as the high memory footprint of intermediate results and frequent data marshaling. Despite our early work demonstrating encoder-only Tiny Transformer deployment [9], the approach was highly model and platform-specific and not easily generalizable to multiple models and platforms. Moreover, early work did not tackle the critical problem of optimized data tiling and computation scheduling, which is essential for most practical Tiny Transformers.

To facilitate the deployment and the optimization of a wide range of encoder-only Transformer models for TinyML platforms on multiple MCU platforms, we significantly extend an existing deployment framework (DORY [10])

- *Victor J.B. Jung, Moritz Scherer, L. Benini are with the Integrated Systems Laboratory (IIS) of ETH Zürich, ETZ, Gloriastrasse 35, 8092 Zürich, Switzerland (e-mail: name.surname@iis.ee.ethz.ch).*
- *A. Burrello, F. Conti, L. Benini are with the Department of Electrical, Electronic and Information Engineering, University of Bologna, 40136 Bologna, Italy. E-mail: firstname.firstsurname@unibo.it*
- *A. Burrello is also with the Interuniversity Department of Regional and Urban Studies and Planning, Politecnico di Torino, 10129, Turin, Italy. E-mail: name.surname@polito.it*

- *This work has received funding from the Chips Joint Undertaking (Chips-JU) TRISTAN project under grant agreement No 101095947 and from the Convolve project under grant agreement No 101070374. The JU and CONVOLVE receive support from the European Union's Horizon Europe research and innovation program. Pre-print manuscript submitted for review to the IEEE Transactions on Computers. We thank Marco Fariselli from GreenWaves Technologies for the insightful discussions, particularly on the FWSA strategy.*

and quantization library (Quantlib). Thus, we present a comprehensive end-to-end and open-source[1] encoder-only Transformer deployment flow to SotA MCUs. Further, we explore optimizations at various levels of the stack, from pure *algorithmic techniques*, to *low-level code optimization*, *novel scheduling* and *tiling schemes*. Together, these optimizations significantly enhance the performance of Transformers deployment on low-power MCUs. In detail, we present three main contributions:

1) We designed a library of highly efficient kernels for both MHSA and Fused-Weight Self-Attention (FWSA). The library is tailored to minimize the overhead associated with data marshaling operations and maximize data reuse through carefully optimized data layout and loop reordering for each Attention layer. By reducing unnecessary data movements and optimizing memory access patterns, the proposed library significantly boosts the overall performance of Attention in an ISA-agnostic fashion, with comparable improvements for RISC-V and ARM with respect to SotA DNN libraries PULP-NN and CMSIS-NN.
2) We introduce two novel optimizations: a fusion-based tiling scheme for MHSA operations and an offline weight fusion schedule for the MHSA operation. By combining fusion-based tiling and pipelining-related computations, the memory footprint and frequency of memory transfers during MHSA computation are substantially diminished, reducing the memory peak to 6.19 ×. Furthermore, the weight fusion schedule reduces the latency by a factor of 1.53× while reducing the number of parameters by 25 %.
3) We extensively benchmark our library and optimizations using published transformers targeting three tasks, i.e., seizure detection from EEG signals, arrhythmia classification from ECG signals, and hand-gesture recognition from high-frequency short-range pulsed RADAR.

Experimental evaluations on the above-mentioned MCUs demonstrate an energy consumption reduction compared to SoA libraries of up to 5.1 × on GAP9 and 2.9 × on ARM-based platforms when executing the attention-building block of the three transformers. When considering end-to-end execution of the networks on GAP9, we obtain the best latency of 9.42 ms, 2.85 ms, and 5.49 ms for the three different tasks at the maximum frequency of 370 MHz. In the most energy-efficient configuration, we achieved 310 µJ, 90 µJ, 207 µJ energy consumption, respectively, still respecting the real-time constraint of the applications.

The rest of the paper is structured as follows. Sec. 2 provides the necessary background while Sec. 3 summarizes related works on MHSA optimization and TinyML platforms. In Sec. 4, we provide implementation details of the library, tiling scheme, and Weight-Fusion schedule. Finally, Sec. 5 presents experimental results, and Sec. 6 concludes the paper.

2 Background

2.1 Transformers and Multi-Head Self-Attention

Transformers are built around repeating an identical Transformer block. As shown in Figure 1, the encoder block comprises two stages, the MHSA and the Fully-Connected stage. To deploy transformers on MCUs, the MHSA stage is the most challenging step, given that the Fully-Connected stage is common in CNNs and its optimization has already been tacked in previous work.

1. https://github.com/pulp-platform/pulp-transformer

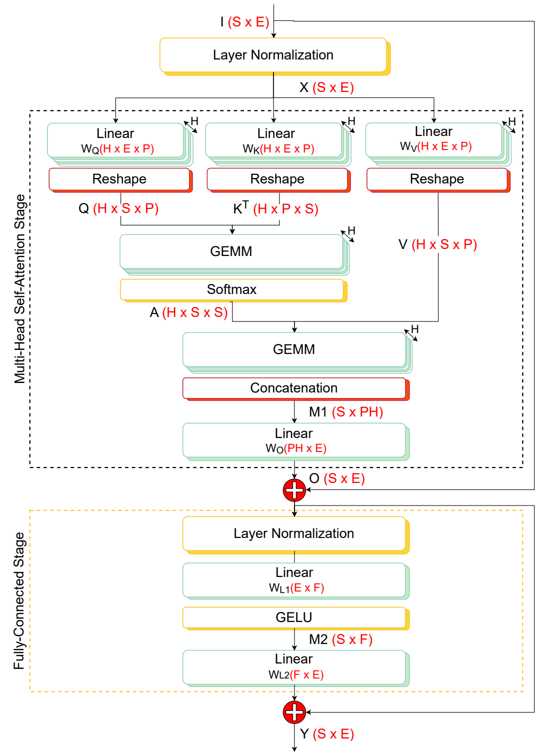


Fig. 1: Topology of the Transformer block, composed of a MHSA stage and a Fully-Connected stage. The dimensions of the tensors are indicated in red.

The dimensions of each operation in the MHSA are determined by four parameters: the *sequence length* S, the *embedding dimension* E, the *projection dimension* P and the *head dimension* H. The first step in the MHSA stage is the projection of the input sequence $X \in \mathbb{R}^{S\times E}$ into the query, key, and values, $Q, K, V \in \mathbb{R}^{S\times P}$, by means of three weights matrices, $W_{\text{query}}, W_{\text{key}}, W_{\text{value}} \in \mathbb{R}^{H\times E\times P}$.

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_{\text{query}}, \qquad \mathbf{K} = \mathbf{X}\mathbf{W}_{\text{key}}, \qquad \mathbf{V} = \mathbf{X}\mathbf{W}_{\text{value}}. \quad (1)$$

In the second step, Q, K, and V are combined in the *Attention* step, which is the core of the Transformer block. It is defined as: $\text{Attention}(\mathbf{Q},\mathbf{K},\mathbf{V}) := \text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^{\intercal}}{\sqrt{d}}\right)\mathbf{V}$, with d being the dimensionality of $\mathbf{K}$ used as a scaling factor and where the softmax function is applied to each row. The softmax of the i-th element of a row of size n is defined as follows:

$$\text{Softmax}(\boldsymbol{x})_i = \frac{e^{x_i - \max(\boldsymbol{x})}}{\sum_{j=1}^{n} e^{x_j - \max(\boldsymbol{x})}} \quad (2)$$

The output of this second step is the matrix $M1$. These first two steps are repeated for every head of the MHSA layer, leading to H parallel executions of the three Linear projection layers and the two GEneral Matrix Multiplications (GEMMs). The final step of the *MHSA* stage involves a Linear layer to project the matrix $M1$ back into the input embedding space E, fusing the computation from the different heads.

The second stage of the Transformer block, shown in Figure 1, is the *Fully-Connected* stage. It is an entirely sequential stage comprising a row-wise layer normalization, an element-wise Gaussian Error Linear Unit (GELU) [11], and two Linear layers.

2.2 Hardware Platforms: RISC-V and ARM MCUs

We consider platforms relevant to the TinyML context, typically featuring $\sim$1 MB of on-chip memory [1] and based on the two leading 32 bit ISAs for MCUs, namely the ARMv7 and RISC-V 32 bit (RV32). We open-source our framework and target commercially available MCUs to facilitate benchmarking and extensions.

2.2.1 ARM Cortex platforms

STM32L4 - Cortex-M4 [2]: The STM32L4 has a Cortex-M4 core coupled with 640 kB of Random-Access Memory (RAM) and 2 MB of Flash. Among the platforms chosen, the M4 shows the lowest average power consumption of 13.63 mW at a clock frequency of 80 MHz. Additionally, the core supports 16 bit Single Instruction Multiple Data (SIMD) operations. In detail, the board used is the STM32L476 [3].

STM32H7 - Cortex-M7 [4]: This architecture is representative of the high-performance end of the spectrum for ARM MCUs. At a frequency of 480 MHz, it consumes 234 mW on average. Among the additional features, we highlight a more complex memory hierarchy involving both data and instruction caches as well as a Direct Memory Access (DMA) for software-managed memory movements between the main memory (1 MB of RAM) and a smaller but faster data memory of 64 kB. Specifically, in this work, we use the NUCLEO-H743ZI2 board [5].

We do not target hardware platforms using the ARM Helium vector extension [6], such as the Cortex-M55 [7] and Cortex-M85 [8], given that they are not yet widely available on the market.

2.2.2 RISC-V platform: GAP9

GAP9 [9] is a low-power RISC-V multi-core processor commercialized by GreenWaves Technologies targeting DNN workloads for hearable and smart sensors. This platform features a RISC-V control core for I/O management and a compute cluster. The cluster comprises 9 RISC-V cores to parallelize computational intensive workload The RISC-V cores have a 4-stages in-order single-issue pipeline based on the RV32 XpulpV2 ISA extension [12], tailored for fast and efficient signal processing. These extensions reduce execution time by more than $5\times$ for different workloads with respect to RV32 baseline [12].

Every core in the cluster is tightly connected to an L1 memory of 128 kB via a single cycle logarithmic interconnect. The on-chip L2 memory communicates with the L1 through the Advanced eXtensible Interface (AXI) bus, and its size is 1.5 MB. To overlap computation with data transfers, GAP9 relies on two DMA units to explicitly move data between L1 and L2 or between L2 and external memories. The DMA core responsible for L1-L2 transfers reaches a peak bandwidth of 2 GB/s.

We do not include single-core RISC-V MCUs to the benchmark, such as the ESP32-C3, as they use the vanilla RV32 ISA and are dominated by the RV32-XpulpV2 cores in terms of performance and energy efficiency.

2. https://developer.arm.com/documentation/dui0553
3. https://www.st.com/resource/en/datasheet/stm32l476je.pdf
4. https://developer.arm.com/documentation/ddi0489
5. https://www.st.com/en/evaluation-tools/nucleo-h743zi.html
6. https://developer.arm.com/documentation/102102
7. https://developer.arm.com/documentation/101051
8. https://developer.arm.com/documentation/101924
9. https://greenwaves-technologies.com/gap9_processor/

3 Related Work

3.1 Attention Mechanism Optimizations

Since the release of the seminal attention paper in 2017 [8], the research community put considerable effort into optimizing the basic MHSA, the major building block in transformer-based architectures introduced in the previous section. These optimizations can be classified into topology optimizations, software optimizations, and hardware acceleration.

3.1.1 Topology optimizations

The first class of optimizations aims to modify the attention mechanism's topology to reduce deployment costs regarding memory footprint or number of operations.

These approaches, including linearized attention, usually seek to eliminate the quadratic scaling of memory and computational requirements relative to the sequence length. To linearize models, researchers use either the kernel trick [13], [14], [15], [16] or low-rank approximation [17]. However, linear transformers suffer from performance degradation on various tasks compared to traditional attention [18]. Hence, linear attention methods are currently not widely adopted by the research community [18]. Consequently, this work focuses on speeding up the traditional attention mechanisms.

Another popular topological change to the Attention mechanism is the Multi-Query Attention (MQA) [19]. It reformulates the MHSA to use only one K and V head instead of H heads. This modification drastically reduces the required memory bandwidth and increases data reuse. However, the impact on the output quality is non-negligible [20]. Grouped-Query Attention (GQA) is a variant of MQA proposed to reduce this accuracy drop. Instead of reducing the number of K and V heads to one, it groups a certain number of KV heads, allowing the programmer to mitigate the accuracy drop. GQA has been successful used in popular Large Language Models (LLMs) such as LLama-2 [21]. However, to the best of the authors' knowledge, these methods have yet to be successfully applied in the TinyML domain. Furthermore, the implementation of GQA kernels can be derived from the basic ones and does not require extra optimization steps.

3.1.2 Software optimizations

The second optimizations category encompasses every software optimization of the traditional MHSA operator. One can notice that the vast majority of these works focus on large-scale inference using server-class hardware [22], [23]. Due to the extreme memory constraints and the small number of cores present in TinyML platforms, many fundamental differences arise in the way one would optimize the MHSA operator. However, despite not being directly applicable to edge devices, most of these techniques can be used as starting points.

We excluded batch size-based optimizations, given that, due to the memory constraints of MCUs and the context of online inference, we always consider batch sizes of 1 in this work. Further, we do not consider transformer model ”partitioning” [24]: While this is crucial for large transformer models that are too large for a single compute unit (*e.g.* a Graphics Processing Unit (GPU)), tinyML systems are overwhelmingly based on a single MCU.

10. https://github.com/google/XNNPACK
11. https://github.com/pytorch/QNNPACK

Library	Supported Platform	Precision	Multi-Core Support	SIMD Support	Loop Unrolling	Fused Reshape
XNNPACK	ARM/x86 RISC-V	fp16/fp32	✗	✓	✓	✗
QNNPACK	ARM/x86	fp16/fp32	✗	✓	✓	✗
CMSIS-NN	ARM	int8/int16	✗	✓	✓	✗
PULP-NN	RISC-V	int8	✓	✓	✓	✗
TinyFormer (Ours)	RISC-V ARM	int8	✓	✓	✓	✓

TABLE 1: Comparison of the SotA kernel libraries for TinyML platforms.

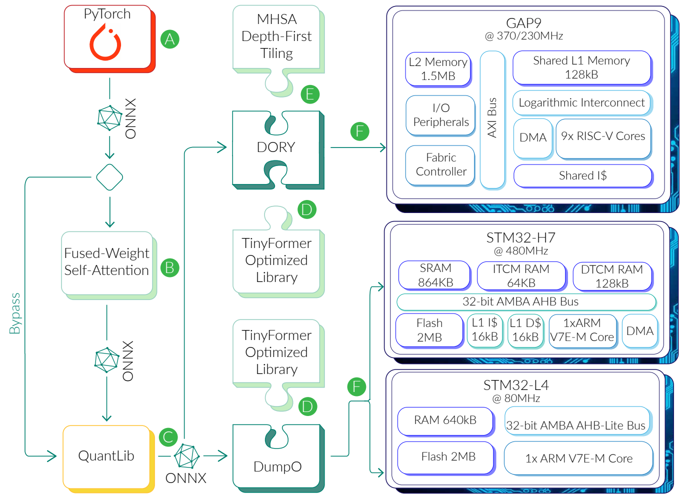


Fig. 2: Overview of our Tiny Transformers deployment flow. The floating point Pytorch model in A can be transformed by the FWSA node in B and is then fed to QuantLib C. Afterward, the quantized Open Neural Network Exchange (ONNX) graph is ingested by the deployment frameworks enhanced with our library D and Depth-First Tiling (DFT) optimization E. Finally, in F, the generated C code is deployed on the desired platforms.

Some works present optimizations partially applicable in the context of TinyML platforms [19], [20], [22]. FlashAttention and FlashAttention-2 use online softmax and normalization [25] to break the row dependencies, which results in better tiling. Together with carefully tuned kernels, FlashAttention significantly improves the performances of transformer inference of large models on GPUs and is now fully integrated into the PyTorch library. However, some of the optimizations FlashAttention uses are not beneficial for extreme edge platforms, given the smaller memory and fewer computational resources. For instance, the block tiling and partial softmax from [22] are not suited for MCUs as it induces non-negligible overhead to store partial sums and renormalize the softmax at each block. Additionally, many transformer models exhibit a high amount of sparsity. Therefore, various approaches exploiting this sparsity have been studied to accelerate the inference [26]. However, sparsity exploitation generally requires dedicated hardware support to be beneficial, which is not available in the current generation of MCUs.

3.1.3 Hardware accelerators

Hardware accelerators optimize efficiency by tailoring architecture and circuits to specific computational patterns at the price of flexibility. Many have been proposed to accelerate MHSA in all kinds of contexts, from server-class to the edge [27]. However, hardware accelerators usually employ a fixed dataflow and attention flavor, reducing the adaptability to different attention mechanisms (*i.e.* such as GQA or MQA) and network topologies. Their extra cost and limited flexibility limit their adoption in the current generation of MCUs. Hence, we focus on the most general and accessible way to port transformers to the edge: computing MHSA on the ARM or RISC-V cores.

3.2 DNN Kernel Libraries for TinyML Platforms

Optimized kernel libraries are at the core of every DNN deployment stack. These libraries contain manually optimized kernels to efficiently run operators commonly found in DNNs such as convolutions or matrix multiplications. Efficient kernel design relies heavily on hardware-specific expert knowledge and is very time-consuming. Despite some attempts of automatizing their generation [28], the vast majority of kernels in well-known libraries such as CUDA [10] and BLAS [29] are still handwritten by experts. However, to the best of our knowledge, there are no optimized libraries specifically tailored for transformer execution on edge devices.

On the other hand, given that Attention layers mainly exploit GEMMs and Linear layers, SotA DNN libraries can be reused and extended to execute transformers. Table 1 provides a qualitative description of the existing DNN libraries commonly used for TinyML platforms. XNNPACK is a library commonly used to accelerate high-level machine learning frameworks such as TensorFlow or PyTorch. As indicated in Table 1, it supports a wide range of platforms but does not support multi-core or efficient data reshaping. It is also used in a TinyML context when using the TensorFlow Lite [30] deployment framework to execute code on Raspberry Pi [11]. XNNPACK has been expanded to optimize efficiency further by targeting 8 bit integer quantized neural networks, which are now the SotA networks in terms of efficiency.

CMSIS-NN [31] is the first kernel library for the deployment of DNNs that explicitly targets MCU class devices, namely platforms relying on ARM Cortex-M cores. CMSIS-NN is tailored for single-core platforms and cannot handle parallelization over any input/output dimensions. It exploits SIMD operations on 16 bits when supported, and its main focus is on optimizing the data reuse in the register file to minimize the additional load/store latency, which was observed to be one of the sources of significant overhead. PULP-NN [12] is another kernel library aiming to perform inference of DNNs on RISC-V MCUs efficiently. It targets explicitly Parallel Ultra Low Power (PULP) SoCs such as GAP9 based on the RV32 XpulpV2 ISA. Similarly to CMSIS-NN, it supports SIMD and Loop Unrolling, boosting the computational intensity for quantized workloads and minimizing memory access stalls. However, both libraries lack support for fused data marshaling that has been demonstrated to be worth 20 % of the latency of transformer networks execution [32]. Hence, they struggle to efficiently execute operators involving data-layout reconfiguration, such as Transposition.

The four libraries described above have a few limitations when boosting the performance of transformers' execution. For instance, they do not consider the specific attention layer topology, they do not provide flexible parallelization dimension, and they do not fuse data marshaling and transpose operators, thus missing crucial optimization opportunities. Finally, they do not provide efficient softmax implementation, a key operator in MHSA. To the best of the authors'

10. https://docs.nvidia.com/cuda/

11. https://www.raspberrypi.org/

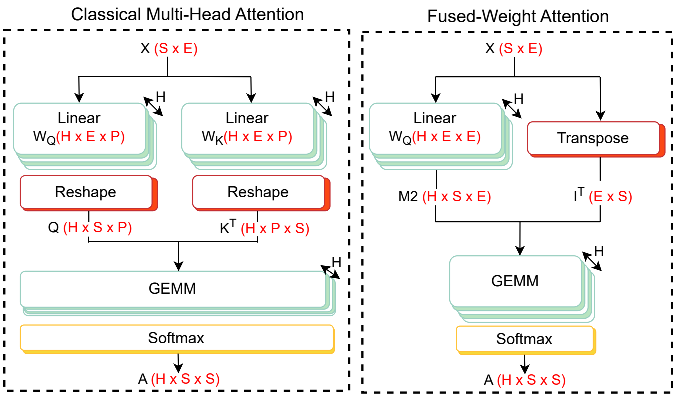


Fig. 3: Diagram of the Classical Attention used in the MHSA and the proposed Fused-Weight Attention. The dimension of each tensor is specified in red.

knowledge, we introduce the first attention-tailored library for MCUs. Our library addresses the limitations of the SotA DNN libraries by providing an optimized parallelization scheme and loop ordering, alleviating the need for expensive data marshaling operations.

4 METHODS

This section describes our end-to-end deployment toolchain depicted in Figure 2; this framework unlocks the execution of Transformer networks on several MCUs. We first give an overview of every tool we use; then, we detail our contributions at each step of the toolchain. Finally, we introduce the three Transformers that we deploy to benchmark our method.

4.1 Deployment Toolchain Overview

The input of our toolchain is a PyTorch Transformer floating point model A represented as an ONNX graph. Then, this graph goes through the optional FWSA transformation B before being processed by QuantLib [12] C, our open-source library for model quantization and graph topology transformation. QuantLib's output is a quantized ONNX graph compatible with the downstream deployment tool. The deployment frameworks are enhanced by our kernel library D and a DFT scheme for MHSA E; their output is code that can be executed on the target platforms F.

We exploit two different deployment tools depending on the target platform. For commercial ARM-based MCUs such as STM32, we utilize DumpO, an internally developed automated code generation tool leveraging CMSIS-NN or our optimized transformer library as backend kernels. GAP9 and other PULP platform SoCs require additional deployment steps, such as tensor tiling, as they employ an explicitly managed memory hierarchy instead of conventional caches. For this reason, in this work, we rely on and enanched DORY [10], a SotA open-source tool used to deploy DNNs on MCUs with software-managed caches such as GAP9.

4.2 Fused-Weight Self-Attention B

In small-scale Transformers, the embedding size E is often tiny, unlike in larger transformers. We introduce an isomorphic transformation of the MHSA, fusing the weights for the

12. https://github.com/pulp-platform/quantlib

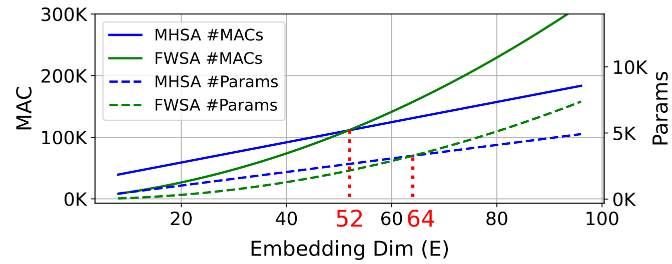


Fig. 4: Number of parameters and Multiply-Accumulate (MAC) as a function of the embedding dimension E and for $S = 32$, $P = 32$ and $H = 8$. The intersection points happen at $E = 52$ and $E = 64$ for the number of MAC and the number of parameters, respectively.

Q and K Linear projection offline to exploit this fact. We call this transformation *Fused-Weight Self-Attention*. Note that this transformation only reorders the matrix multiplications used in the computation of the attention matrix A. Additionally, it is mathematically equivalent to the MHSA. Hence, it does not affect the accuracy of the transformers. This optimization aims to speed up the inference by performing one of the matrix multiplications offline, hence reducing the number of operations to execute at inference-time on the MCU and the number of parameters to store on-chip. In the equations below, we decompose the matrix product between Q and K^T and demonstrate the equivalence between the MHSA and the FWSA. A denotes the attention matrix, X the input, ϕ the softmax, and W_{QK} the weights of the Q and K linear projection. We define the fused weight W^* such as: $W^* = W_Q \cdot W_K^T$ Additionally, $X \in \mathbb{R}^{(S \times E)}$, $W^* \in \mathbb{R}^{(H \times E \times E)}$, $W_{QK} \in \mathbb{R}^{(H \times E \times P)}$, $Q \in \mathbb{R}^{(H \times S \times P)}$, $K^T \in \mathbb{R}^{(H \times P \times S)}$, and $A \in \mathbb{R}^{(H \times S \times S)}$.

$$A = \phi(Q \cdot K^T) \tag{3}$$

$$A = \phi(X \cdot W_Q \cdot (X \cdot W_K)^T) \tag{4}$$

$$A = \phi(X \cdot W_Q \cdot W_K^T \cdot X^T) \tag{5}$$

$$A = \phi(X \cdot W^* \cdot X^T) \tag{6}$$

In the equations above, we reorder the operations using the associativity of the matrix product and the transposition's reversal order of the product. The fused weight matrix W^* is computed offline and considered as fixed parameters during the inference. In detail, we reduce the number of matrix multiplications to perform online from three to two. Figure 3 shows the computational graph of the classical Multi-Head Attention (left) and the FWSA (right), we indicate the dimension of the tensors in red.

To evaluate when this transformation is beneficial, we compare the number of operations O and parameters θ between the FWSA and the MHSA in Eq. 7&8 below:

$$\begin{cases} O_{MHSA} = 2HSPE + HS^2P \\ O_{FWSA} = HSE^2 + HS^2E \end{cases} \tag{7}$$

$$\begin{cases} \theta_{MHSA} = 2HPE \\ \theta_{FWSA} = HE^2 \end{cases} \tag{8}$$

Looking at Eq. 7&8, we notice that using the FWSA is indeed beneficial for some conditions of S, P, and E values. More specifically, in Eq. 9, we provide the inequalities that need to be satisfied to benefit from the FWSA in terms of number of operations (left) and parameters (right).

$$E < P - \frac{S}{2} + \frac{\sqrt{4P^2 + S^2}}{2} \qquad E < 2P \tag{9}$$

In Figure 4, we show the relationship between E and the number of MAC/Parameters for the MHSA and FWSA. For $E < 52$, FWSA always shows a lower number of MAC and parameters. When considering the intermediate condition of $52 \leq E < 64$, the FWSA reduces the number of parameters but increases the number of MACs. Finally, in the third case ($E \geq 64$), the MHSA is more advantageous than the FWSA.

4.3 Transformers Quantization with QuantLib C

To quantize Transformers, we employ and extend the open-source quantization library QuantLib. QuantLib transforms the ONNX graph representing the floating-point model of a DNN to an integerized ONNX graph. This transformation from a floating-point to an integerized graph is done in two steps.

All operation nodes in the graph, such as matrix multiplication, convolution, or softmax, are replaced by their integer equivalent. As the softmax, GELU, and layer normalization present in Transformers do not have an integer equivalent, we extend QuantLib to support the integer version of these operands from I-BERT.

Then, we add a re-quantization step after operations whose output is not represented in 8 bit. For instance, the output type of an integer GEMM is *int32*; thus, to convert it to 8 bit, we have to re-quantize it. We use the uniform symmetric quantizer from the SotA Trained Quantization Threshold (TQT) method [3]. Below, we describe the functional behavior of the TQT quantizer denoted ψ:

$$\psi(x) = \max(2^{b-1}, \min((x \cdot \epsilon_{mul}) >> \epsilon_{div}, 2^{b-1} - 1)) \quad (10)$$

The number of bits of the quantized values is b, while ϵ_{mul} and ϵ_{div} are the re-quantization parameters. This quantizer constrains the scale factor to a powers-of-2 (i.e., $2^{\epsilon_{div}}$) and uses per-tensor scaling of activations and weights, making it hardware-friendly as we can use the bit-shift operation to apply ϵ_{div}. Additionally, we use Quantization Aware Training (QAT) to adjust weights to reduce the accuracy loss induced by quantization. Note that we could use Post-Training Quantization (PTQ) instead of QAT, we choose to use QAT because it mitigates even further the quantization accuracy loss, compared to PTQ.

After applying the necessary transformations and obtaining an integerized ONNX graph, we integrate into QuantLib a new export module to format the graph such that it can be ingested by the deployment framework (DORY or DumpO).

4.4 Deployment Framework Integration

For ARM platforms, we use DumpO to compile the integerized ONNX graph into C code. DumpO first transforms the graph so that each node can be executed by one kernel from the available libraries. Then, it parses the graph to match every node with the appropriate kernel and generates the code for the target platform. We modify the kernels to use the appropriate intrinsics for vector packing and SIMD calls to leverage our optimized library in ARM platforms.

This simple code generation strategy does not work for the GAP9 target due to the nature of the hardware architecture. As mentioned in Sec. 4.1, unlike ARM processors, the family of GAP processors does not have hardware-managed data caches, meaning that software must handle data movement and tiling. While increasing the complexity of the deployment framework, this choice unlocks several optimizations, such as tiling and double-buffering. An example of implementing these optimizations is provided in Listing 1, comparing the generated code between the ARM/DumpO and GAP/DORY cases.

DumpO-generated pseudocode for ARM MCUs

```
Inputs: X; Output: Y
kernel_params = {param1, param2, etc...};
Kernel(X, Y, kernel_params);
```

DORY-generated pseudocode for GAP9

```
Inputs: X; Output: Y
kernel_params = {param1, param2, etc...};
for (i = 0; i < N_tile; i++)
    DMA_wait(L1_X_next); swap(L1_X_next, L1_X_current);
    DMA_transfer_async(L1_X_next <- L2_X[i]);
    Kernel(L1_X_current, L1_Y_current, kernel_params);
    DMA_wait(L1_Y_previous); swap(L1_X_current, L1_X_previous);
    DMA_transfer_async(L2_Y[i] <- L1_Y_previous);
```

Listing 1: Example of code generated by DORY and DumpO for executing a DNN layer. The input and output tensors are noted as X and Y, respectively. The code generated by DORY features tiling with double-buffering, it uses the DMA core to fetch input tiles (noted $L1_X$) and send back output tiles (noted $L1_Y$) asynchronously to the execution of the kernel.

For this reason, we employ and extend a different tool, DORY, when targeting GAP9. Similarly to DumpO, DORY transforms the graph to fuse operands and features a parser to match layers to kernels. Furthermore, it supports multi-level layer-wise tiling with double-buffered DMA transfers to overlap data movement behind computation [10]. The tile dimensions are chosen using constrained optimization [10] and relies on handcrafted heuristics such as maximizing the L1 memory utilization. DORY also integrates the PULP-NN library [12] in its backend natively.

The baseline version of DORY focuses exclusively on convolutional DNNs and cannot handle Transformers. Furthermore, the baseline DORY targets feed-forward Convolutional DNNs, allowing only one skip connection branch. For example, it can support a network such as MobileNet V2 [7], but not SSD [33]. We extend DORY by adding support for Transformer-related operators such as MHSA GELU, layer normalization, and softmax, adding our specialized library of efficient MHSA and FWSA kernels D as a new backend for DORY. Additionally, we integrated support for a new tiling scheme E and added support for multiple skip connection branches. In the following subsections, we describe in detail this tiling scheme tailored for MHSA for MCUs without hardware-managed caches and our optimized kernel library.

4.5 MHSA Depth-First Tiling E

Tiling is usually done layer per layer; the intermediate activation tensors are entirely moved from one level to the other after each DNN layer. This approach is referred to as *Layer-Wise Tiling (LWT)* and is by far the most common one for deployment at the edge [10], [34].

On the other hand, recently, *Depth-First Tiling (DFT)* has been gaining traction. The idea behind this is to tile several layers together, effectively using Layer-Wise Tiling but on a group of layers [35], [36].

We propose a DFT scheme specifically tailored for MHSA on MCUs and inspired by the flash-attention approach [22]. This scheme aims to reduce the memory peak encountered when materializing the attention matrix A. Since the attention's matrix dimensions are $(H \times S \times S)$, its size grows quadratically with respect to the sequence length S. Hence, the materialization of A can bottleneck systems where on-chip memory is very limited, such as MCUs. To cope with this problem, we propose an output stationary dataflow for our DFT scheme, meaning that we

```
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(4sE+4e); A2 = I((4s+1)E+4e);
         A3 = I((4s+2)E+4e); A4 = I((4s+3)E+4e);
         B1 = W(hPE+4e); B2 = W(hPE+2pE+4e);
         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) = requantization(S0);
       O(h,2p,4s+1) = requantization(S1);
       O(h,2p,4s+2) = requantization(S2);
       O(h,2p,4s+3) = requantization(S3);
       O(h,2p+1,4s) = requantization(S4);
       O(h,2p+1,4s+1) = requantization(S5);
       O(h,2p+1,4s+2) = requantization(S6);
       O(h,2p+1,4s+3) = requantization(S7);
```

Listing 2: Example of kernel pseudocode ❶. *requantization* denotes the function ψ from Equation 10 and *sdot4* represents the 4-way dot-product SIMD instruction.

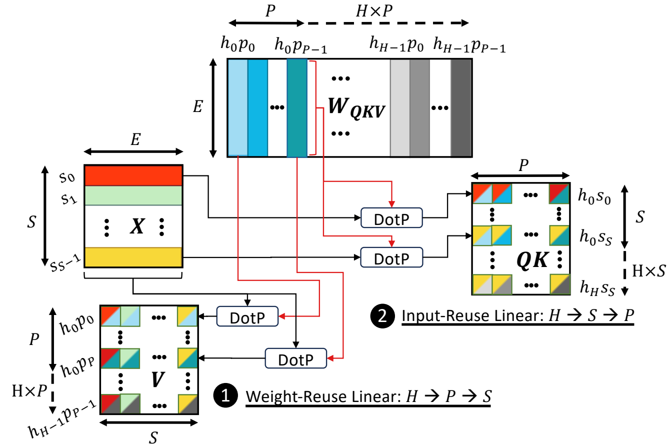


Fig. 5: Linear layer dataflow for generating Q, K, and V. The output data layout is HPS and HSP. Matrices are filled from top left to bottom right.

do not store partial results to accumulate them later. Instead, we directly compute the final output, avoiding overhead of storing partial outputs. The proposed DFT scheme tiles the two GEMM layers (kernels ❸ and ❹) of the MHSA stage depicted in Figure 1, as well as the softmax operator together. Our DFT scheme takes x rows of the Q tensor as input. Note that the choice of the number of rows to process in one tile depends on the memory hierarchy of the platform and the specific dimensions of the transformer. Additionally, one entire head of K and V is necessary to apply our scheme. The rows of Q are multiplied with one head of K to create x rows of the attention matrix A; then we consume these rows of A by multiplying them with one head of V to generate x rows of $M1$. Hence, to use our DFT, the L1 memory needs to hold at least one complete head of the K and V tensors as well as one row of Q, A, and $M2$. Thus, the L1 memory size required to apply this tiling scheme and produce x rows of $M2$, noted Mem_{DFT}, can be expressed as follows:

$$Mem_{\mathrm{DFT}}(x) = (2P + S)x + 2PS \tag{11}$$

where Px bytes are used to store a tile of Q, Sx to store the intermediate tile of A, and Px bytes for the output tile of $M2$. $2PS$ bytes are used to store a head of K and V. The DFT scheme cannot be used if $Mem_{\mathrm{DFT}}(1) > \mathrm{L1}_{\mathrm{size}}$, in this situation we use the LWT instead. When we cannot fit L1 memory given the smallest possible x, LWT is used as a fallback solution. The result section compares the LWT and DFT approaches in all evaluation scenarios.

4.6 Multi-Head Self-Attention Kernel Library Ⓓ

Every layer of the MHSA, described in Figure 1, is followed by transpositions, reshape, or concatenation operations. These memory marshaling operations are hard to execute efficiently on multi-core processors as they only comprise data movement. Hence, hiding the data transfer latency behind computation is impossible. Therefore, their impact on performances is non-negligible, causing overheads up to 20 % in some network execution [32]. Furthermore, SotA DNNs libraries [12], [31] do not tune data layout to avoid data marshaling operations and do not fine-tune their parallelization scheme for MHSA. Hence, they perform poorly for executing MHSA, exhibiting low data reuse and sub-linear scaling on multi-core processors. We propose our optimized library of tailored kernels for each layer in the MHSA to address these issues. Each kernel of the library embeds the re-quantization step and takes 8 bit input and output tensors. The scaling factors and the bias of the linear projection are stored on 16 bit.

4.6.1 Kernel Execution Loop

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).

When the memory constraints of the platform allow it, we parallelize our kernels on the H dimension. Spatially unrolling the heads has several advantages. First, this dimension is present in every kernel of the MHSA. Second, it allows for minimizing synchronization among cores since the heads perform independent computation, avoiding race conditions. Note that this parallelization strategy, both for the MHSA and for the FWSA, would not be efficient on high-performance devices such as GPUs. These high-performance platforms have thousands of cores, which is considerably more than the typical number of heads. For instance, Llama3-405B[13] one of the largest transformer model available have 128 heads.

The second guideline saves memory by avoiding the storage of many intermediate `int32` accumulators for the partial outputs, as described in [10]. Besides the memory saving, output stationarity maximizes the exploitation of the dot-product SIMD instructions, as demonstrated in [12].

Finally, our motivation to enforce the last guideline is to prevent potential computational overhead induced by the additional operations in the innermost loop to compute storage locations.

4.6.2 Linear Layers

Figure 5 depicts two distinct implementations for the Linear layers, used to project the input to the Q, K, and V matrices and compute the output of the MHSA. In each tensor from Figure 5, the rows and columns are indicated by the lowercase letter of the dimension and a subscript that denotes the index of the row/column. For instance, the fifth row of

13. https://huggingface.co/meta-llama/Meta-Llama-3.1-405B

$I \in \mathbb{R}^{S \times E}$ is noted as s_4. For concatenated dimensions, the innermost element is specified by the rightmost letter, *i.e.* in $W_Q \in \mathbb{R}^{E \times HP}$, h_2p_3 is associated with the $2 * P + 3$-ith column. Implementation ❶, is called Weight-Reuse Linear (WRL) and is used to project the $\mathbf{V}$ tensor from $\mathbf{X}$, while we use the implementation ❷, named Input-Reuse Linear (IRL), for $\mathbf{Q}$ and $\mathbf{K}$. These two implementations have a different loop ordering and data layout.

From a functional point of view, for a single head, one can write the functions of the WRL and IRL kernels as follows:

$$\begin{cases} \varphi_{IRL}(X, W^T) = O \\ \varphi_{WRL}(X, W^T) = O^T \end{cases} \tag{12}$$

Where φ is the function of the kernel, X, W, and O are the Input, Weight, and Output tensors, respectively.

WRL kernel produces output data in the HPS order, allowing the subsequential matrix multiplication to ingest data sequentially without stridden access. In this way, we remove the data-reshaping operator, reducing the overall number of memory accesses. At every iteration of the P loop, the whole input ($S \times E$) matrix is multiplied by a single weight sample ($1 \times E$). From outermost to innermost, we order the loop as $H \rightarrow P \rightarrow S \rightarrow E$. On the other hand, the IRL requires the output layout to be HSP to allow the following matrix multiplications to read data sequentially. Therefore, the loop order in this case is $H \rightarrow S \rightarrow P \rightarrow E$, from outermost to innermost. Contrary to WRL, at every iteration of IRL, the S loop, a single input sample ($1 \times E$), is multiplied by a weight-head ($E \times P$).

The last Linear layer, which projects the output tensor of the matrix multiplication, noted M, to the final output, uses the $S \rightarrow E \rightarrow H \times P$ loop order. Since the H dimension is a part of the reduction dimension, we parallelize this kernel over S, the outermost loop.

Listing 2 reports an example of the pseudocode of layer ❶ with the `RV32IMCXpulpV2` ISA and GAP9 target. In the innermost loop, we exploit the 4-way dot-product SIMD instruction (noted `sdot4`) to perform 4 MAC operations in a single instruction. Once a final output is generated, we apply the re-quantization function ψ from Equation 10. Additionally, we perform loop unrolling in the same fashion as [12]. We select the window size of the loop unrolling to be 4×2, meaning that we perform 8 dot-products on 8 register-allocated accumulators at each loop iteration. This optimization allows us to avoid Read-After-Write (RAW) hazards and increase the register files data reuse. Similarly to PULP-NN [12], incrementing the number of produced output values in a single iteration, e.g., to 16, would cause an increase in the number of required registers to 24 (16 for outputs, 4 for inputs, 4 for weights); we observe that the compiler[14] can not generate valid code, in this case, keeping all accumulators allocated to registers: some of them are spilled to the stack to make room for operands, causing extra load/store operations and reducing the overall performance. Conversely, reducing the number of registers employed causes a reduction in the MAC/load ratio and impairs the performance.

4.6.3 Matrix Multiplications

To optimize matrix multiplications, we optimally order the loop executions and parallelize over the outermost dimensions to improve performance. Figure 6 reports two different

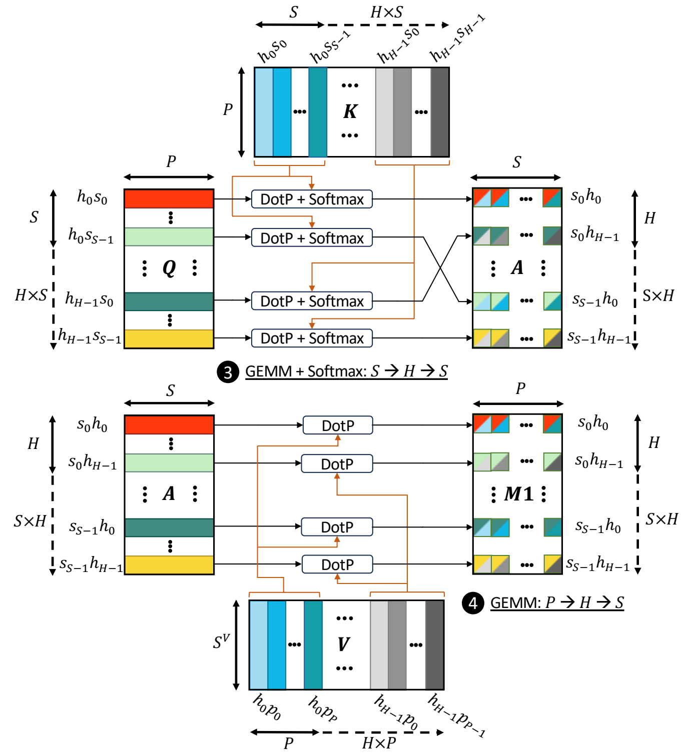


Fig. 6: Dataflow of the matrix multiplication kernels designed to work with multiple heads with different data layouts.

implementations for the two GEMMs in the MHSA kernel. The notation used to denote rows and columns of the tensors is the same as Figure 5 and a detailed description is provided in section 4.6.2.

The Matmul-Softmax ❸ fuses the matrix multiplication with the integer softmax; it uses the $S \rightarrow H \rightarrow S$ loop order; P is the dimension over which the reduction is performed. Each iteration of the H loop produces a new row of A. We apply the softmax to the produced row (e.g., the first one, S_0H_0). The final Linear layer of the MHSA (see Figure 1) performs the reduction over the concatenated dimension of H and P. Hence, to avoid this concatenation operator, we choose the output data layout to be SHS, interleaving the heads with the sequence length. For instance, the first row of Q ($S \times 1$) is multiplied with the first head of K ($P \times S$), then the softmax is applied, resulting in the first row of A ($1 \times S$). Next, the process is repeated with the second head of K, resulting in the second row of A. Once the first row of Q has been matched with the whole K matrix, we reiterate using the next row of Q.

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

4.6.4 Fused-Weight Self-Attention Kernel

Figure 3 shows the computational graph of the FWSA, and Sec. 4.2 describes its equation. Like the other kernels of our library, the FWSA kernel uses SIMD and loop unrolling to maximize hardware utilization. The FWSA kernel is broken down into two parts, one responsible for the computation of $M2$ and the other for A.

The kernel to compute $M2$ is derived from the Linear IRL kernel ❶ used for the Linear projection of Q and K

14. A precompiled RISC-V toolchain for GAP IoT Application Processor compiled with gcc-9.4.0 available at https://github.com/GreenWaves-Technologies/gap-riscv-gnu-toolchain

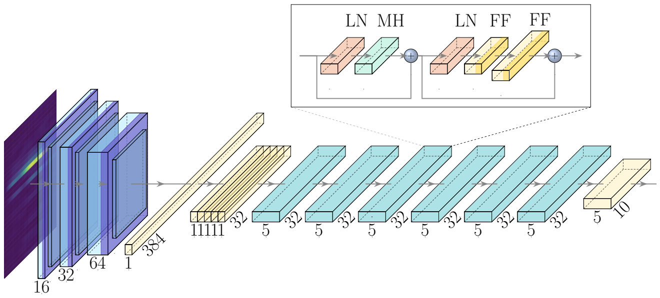


Fig. 7: Overall architecture of TR-Former. The front end comprises 3 Tokenizer blocks, each containing Pointwise Convolution, Depthwise Convolution, and Pooling, followed by a Linear layer. Each of the six encoder blocks consists of layer normalization layers (LN), a MHSA layer (MH), and Linear layers (FF).

and described in detail in Sec. 4.6.2. Its loop ordering is $H \rightarrow S \rightarrow E$ to create output with the HSE layout. The second part of the kernel perform the Matmul-Softmax on $I^T \in \mathbb{R}^{E\times S}$ and $M2 \in \mathbb{R}^{H\times S\times E}$; its loop ordering is identical to ❸ ($S \rightarrow H \rightarrow S$). However, inputs of ❸ are $Q \in \mathbb{R}^{H\times S\times P}$ and $K^T \in \mathbb{R}^{H\times P\times S}$, but the input tensors of this kernel are $M2 \in \mathbb{R}^{H\times S\times E}$ and $I^T \in \mathbb{R}^{E\times S}$, hence the innermost loop performing the dot-product is over E instead of P in ❸.

4.7 End-to-end Tiny Transformers Applications

To demonstrate the performances of our deployment toolchain, we decided to use three real-world Tiny Transformer networks targeting very different applications, from biosensing wearables use cases like arrhythmia classification or seizure detection to more traditional hand gesture recognition.

Two of the three transformers we are benchmarking in this paper have been introduced in [37], demonstrating better performance than the CNN counterpart. The other one, TR-Former, has been first introduced in [9]. In the following subsections, we summarize the rationale and describe the architecture of TR-Former, EEGFormer, and ECGFormer.

Recent research has empirically shown that Transformers can tolerate a quantization down to 4 bit [38]. However, quantizing to 4 bit would negatively impact execution efficiency for all the targeted MCUs due to the lack of hardware support for 4 bit SIMD. The only benefit would be to divide by two the memory footprint compared to 8 bit quantization. Because of the performance impact and potential accuracy degradation, we use a 8 bit quantization scheme. It's worthy to note that we deploy already-quantized transformer models and do not explore other quantization schemes further.

4.7.1 Hand Gestures Recognition: TR-Former

The first application is hand-gesture recognition based on short-range radar. We use the Tiny Transformer proposed in [9] and trained on the TinyRadar dataset. It demonstrates the feasibility and the advantages of Transformers on TinyML platforms. The dataset consists of over 10K recordings of 11 hand gestures by 26 people made with a short-range radar. The architecture is shown in Figure 7. Using $S = 5$ processed input time samples as a sequence, a 5×32 input is fed to the Transformer backend. The 6 layers constituting the backend are identical to the ones of [39], 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.

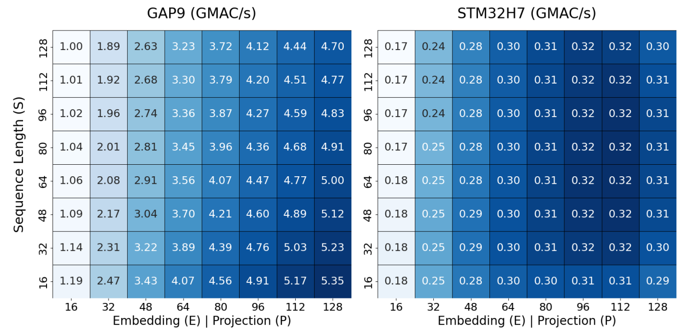


Fig. 8: Heatmap representing the throughput of our MHSA kernel on GAP9 and STM32H7 for various input dimensions of $(S \times E)$. The projection dimension P is equal to E and the number of heads (H) is set to 8.

4.7.2 Seizure Detection: EEGFormer

The second Tiny Transformer targets a non-obtrusive and non-stigmatizing Electroencephalogram (EEG) acquisition setup to detect seizures. The encoder contains a MHSA layer of the following dimensions: $S = 81$, $E = 32$, $P = 32$, and $H = 8$. A detailed description of the architecture can be found in [37].

4.7.3 Arrhythmia Classification: ECGFormer

The last task we target is arrhythmia classification; the dataset used is MIT-BIH. ECGFormer's [40] architecture is also based on BioFormer and targets ultra-low power applications. ECGFormer features a single transformer encoder block with MHSA dimensions of $S = 66$, $E = 16$, $P = 2$, and $H = 8$. More details on the architecture can be found in [40].

5 Results And Discussion

In this section, we detail our evaluation setup, and we benchmark our library against the SotA libraries, CMSIS-NN for ARM platforms and PULP-NN [12], [31] for RISC-V ones, to showcase the advantage of our method for various input dimensions, number of cores, and hardware platforms. Afterward, we demonstrate the usage of the library with the deployment tools to deploy three end-to-end SotA transformers for different edge applications. Different input dimensions and architectural hyperparameters characterize each application. To conclude, in Sec. 5.4, we provide an ablation study of the impact of the FWSA and the DFT on latency and memory footprint on these applications.

5.1 Evaluation Setup

We benchmarked and compared our library with SotA kernels on the three platforms introduced in Sec. 2.2. On GAP9, we use internal performance counters to measure the cycles of both single attention blocks and the entire Transformers execution. Furthermore, to simulate larger dimensions of the GAP9 internal memories, we rely on the event-driven simulator GVSoC from GreenWaves Technologies, which enables cycle-accurate simulations. To measure the power consumption, we use Nordic Semiconductor's Power Profiler Kit 2 (PPK2) [15] with a sampling frequency of 100 kHz. We measure the power of the cluster and fabric controller and the off-chip RAM and Flash. We consider a hot start, meaning that we neglect the movements of weights from L3 to L2, executed a single time for consecutive inferences.

15. https://www.nordicsemi.com/Products/Development-hardware/Power-Profiler-Kit-2

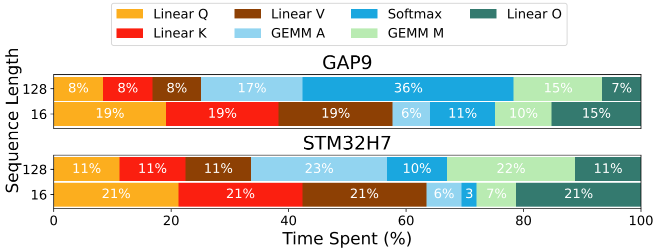


Fig. 9: Breakdown of the execution time of each MHSA layer on GAP9 and STM32H7 for sequence lengths of 16 and 128. The other dimensions are fixed to $E = 64$, $P = 64$, and $H = 8$.

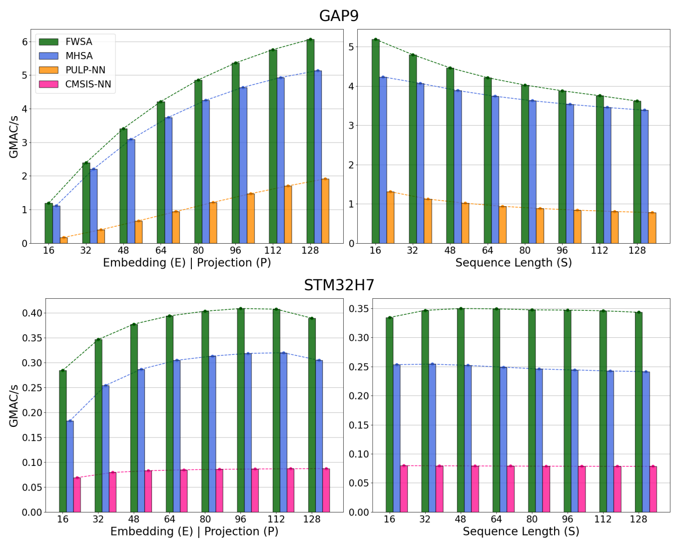


Fig. 10: Thoughtput comparison of attention block on GAP9 and STM32H7 for various embedding sizes and sequence lengths. We use our MHSA and FWSA kernels and SotA kernel libraries PULP-NN or CMSIS, depending on the platform.

For STM32L4 and STM32H7, we use internal hardware counters to measure the number of cycles, and we consider constant power consumption measured under an identical workload to estimate the energy. For STM32H7, we activate the data and instruction caches and store the weights and activation tensors in the Static Random-Access Memory (SRAM) to maximize the performance. For the STM32L4 platform, as we do not have caches in this processor, we store every constant tensor (input and weights) in the read-only data section of the Flash.

5.2 Micro-benchmarking: MHSA and FWSA

5.2.1 Input Size Scaling

First, we show the performance of our library for various dimensions of the input tensor X. To benchmark kernel optimizations on GAP9 without considering data movements, we increase the L1 memory size using GVSoC and directly store both weights and activations at this level. We report only STM32H7 performance for the ARM-based platforms, given that the same conclusions can be drawn for the STM32L4, which shares the same ISA and architecture.

Figure 8 shows the performance in terms of GMAC/s when executing a MHSA with different input sizes. On the x-axis, we increase $E|P$ dimensions, while on the y-axis, we increase the S dimension. The number of heads has been kept constant at 8 to maximize parallelization on GAP9 and at 1 for the STM32H7. In the figure, we observe two significant trends for GAP9: firstly, a notable decrease in efficiency occurs with an increase in sequence length; secondly, an improvement in efficiency when E and P increase.

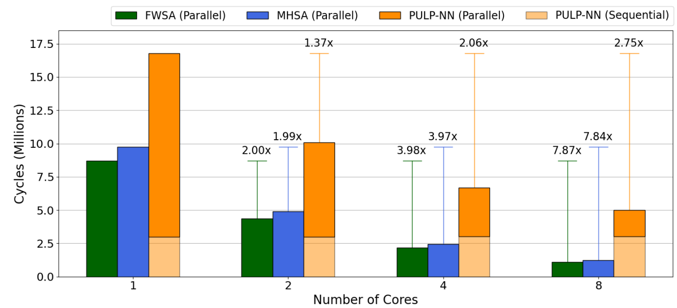


Fig. 11: Parallelization of the MHSA on GAP9 for the three SotA libraries. For PULP-NN, we indicate the breakdown of sequential and parallel execution time. Fused-Weight and Vanilla are completely parallelized. The dimensions of the MHSA are ($S = 64$, $E = 64$, $P = 64$, and $H = 8$).

TABLE 2: Performances of end-to-end applications on GAP9 at maximum frequency and at the most energy-efficient configuration.

			370MHz/50mW		230MHz/20mW	
Task	MACs	Cycles	Lat. (ms)	E. (µJ)	Lat. (ms)	E. (µJ)
EEG	7.35M	3.33M	9.42	460	13.76	310
ECG	0.97M	1.15M	2.85	120	4.28	90
TR	6.00M	1.92M	5.49	315	8.52	207

The first effect is due to the latency of the softmax nonlinearity. We measure the complexity of the MHSA with the number of MACs of the Linear and GEMM layers. Thus, the softmax counts as zero MACs but strongly impacts the overall latency. The latency of softmax rises proportionally with the dimension of the vector on which it is computed, S. The effect of the softmax on performance for large sequences is quantified in Figure 9. For both platforms, the proportion of time spent on the softmax drastically increases by a factor of $3.3\times$ when going from a sequence length of 16 to 128.

The efficiency growth with E and P is well visible in Figure 8: up to $4.7\times$ better between $E|P = 16$ and $E|P = 128$ with $S = 128$. This is because P and E are the dimensions over which the reduction is made in five out of the six matrix multiplications of the MHSA. Since we produce the output tensors sequentially, as explained in subsection 4.6.1, increasing the reduction dimension leads to better utilization of the SIMD units and less overhead due to loop indexes and pointer computations. For the STM32H7 platform, when we increase $E|P$, we observe the same behavior as for GAP9: the throughput is $1.72\times$ higher for $E|P = 128$ than for $E|P = 16$ with $S = 64$. We notice that an increase of S leads to a reduction in performance only for low $E|P$ values. When E is higher, the softmax operation's latency on STM32H7 is negligible compared to the other blocks, and, therefore, it does not negatively impact the throughput.

Figure 10 reports $E|P$ scaling with constant S, and S scaling with constant $E|P$ on GAP9 and STM32H7 for our MHSA and FWSA kernels, comparing them with the two SotA kernel baselines, i.e., PULP-NN and CMSIS-NN. It can be noticed that our kernels (MHSA and FWSA) consistently show higher throughput than PULP-NN and CMSIS-NN on both platforms. On GAP9, for the various values of S and $E|P$, the average throughput of the MHSA and FWSA kernels is $4.04\times$ and $4.53\times$ higher than the PULP-NN implementation. In STM32H7, compared to CMSIS-NN, the throughput is $3.28\times$ and $4.45\times$ higher on average for the MHSA and FWSA, respectively.

TABLE 3: Ablation study of the effect of Fused-Weight attention and Depth-First Tiling on the runtime and L2 memory peak for the Attention Stage of the three transformer studied.

	Memory Peak			Cycles	
	MHSA		FWSA	MHSA	FWSA
Tasks	LWT	DFT	LWT		
EEG	129.3 KB	**97.1** KB	121.2 KB	1.07M	**1.01M**
ECG	39.0 KB	**6.3** KB	38.5 KB	**553K**	569K
TR	34.2 KB	34.2 KB	**24.9** KB	52K	**34K**

5.2.2 *Parallelization Scaling on GAP9*

Figure 11 details the performance of an attention block using the MHSA, the FWSA, and the baseline PULP-NN kernels on GAP9 with 1, 2, 4, and 8 cores. As can be noticed, the speed-up of the PULP-NN baseline from 1 to 8 cores is only $2.75\times$ while our kernels reach $7.87\times$ and $7.84\times$ speed-up for FWSA and MHSA, respectively. We identify three main reasons for the better parallelization scaling: (i) unlike PULP-NN, in both our kernels, we parallelize over the outermost loop, requiring fewer synchronization steps. (ii) Moreover, since the GEMM in the PULP-NN implementation exploits individual Linear kernels, the cores split the computation on the same dimension on which the softmax has to be executed. In this way, all the cores must be synchronized before the softmax, executed from CORE 0, strongly impacting the parallelization. (iii) Similarly, our approach eliminates the data marshaling operations executed sequentially from CORE 0 inside PULP-NN. These last two problems can be observed in Figure 11, which shows the sequential part of the kernel in light colors: while it is constant for the PULP-NN baseline, we eliminate it in our kernels.

5.3 End-to-end Transformers performance

Table 2 describes the latency and energy of the end-to-end execution of the three Tiny Transformers introduced in Sec. 4.7. We here report the end-to-end performance on GAP9 only, given that we can exploit all the optimizations and features of our kernels, i.e., the influence of parallelization, FWSA, and DFT. A more detailed analysis of the performance of the attention blocks of these three networks for all hardware platforms is reported in Sec. 5.5. Here, we report only the results obtained with the best combination of optimizations to minimize latency. In detail, we use the FWSA on EEGFormer and TR-Former to reduce the latency, while we use MHSA for ECGFormer. Additionally, we do not use the DFT as it only reduces the L2 memory consumption, and GAP9 features a large enough L2 memory when executing a single transformer. Note that this is given by the specific shapes of the networks of our use cases, where the memory transfers are entirely hidden by computation with double buffering. Therefore, reducing the memory-transfer time does not improve the overall latency of the network. Further, we show two different hardware configurations: the first one minimizes the latency and runs at 370 MHz with a power consumption of 50 mW. The second configuration runs at 230 MHz, consuming 20 mW and is the most energy-efficient point. For the three networks (EEGFormer, ECGFormer, and TR-Former), we obtain a best latency of 9.42 ms, 2.85 ms, and 5.49 ms. Additionally, all three networks respect the real-time constraints imposed by their respective task. The most efficient configuration leads to an energy consumption of 310 µJ, 90 µJ, and 207 µJ for the three networks.

5.4 Ablation Study: Optimizations Impact

In this subsection, we analyze the individual impact of the proposed optimizations for the GAP9 platform. Figure 12 provides a breakdown of the impact of each optimization performed on the runtime for the three transformers studied. Compared to the PULP-NN baseline, on GAP9, our optimizations provide a total speedup of $5.79\times$, $5.8\times$, and $2.79\times$ on EEGFormer, ECGFormer, and TR-Former, respectively. It is crucial to notice that the impact of the optimizations considerably varies depending on the parameters and the structure of the network. For instance, the parallelization of the softmax is the major contributor of speedup for the networks whose sequence length S is larger than the projection dimension P. On one hand, ECGFormer, with $S = 66$ and $P = 2$ is an extreme example of this effect, where only parallelizing the softmax brings a $4.2\times$ speedup. On the other hand, on TR-Former ($S = 5$ and $P = 32$), the parallelization of the softmax only speeds up the execution by 20 %. Concerning EEGFormer, after the parallelization of the softmax, the improved parallelization of the kernels is the greatest contributor to the speedup. The overhead of data marshalling operators is negligible compared to the PULP-NN baseline when the runtime is dominated by sequential softmax and suboptimal parallelization scheme. However, when compared to the runtime post optimization, the overhead of data marshalling operators is non-negligible and represents 9.6 %, 0.9 %, and 17 % of the runtime on EEGFormer, ECGFormer, and TR-Former, respectively.

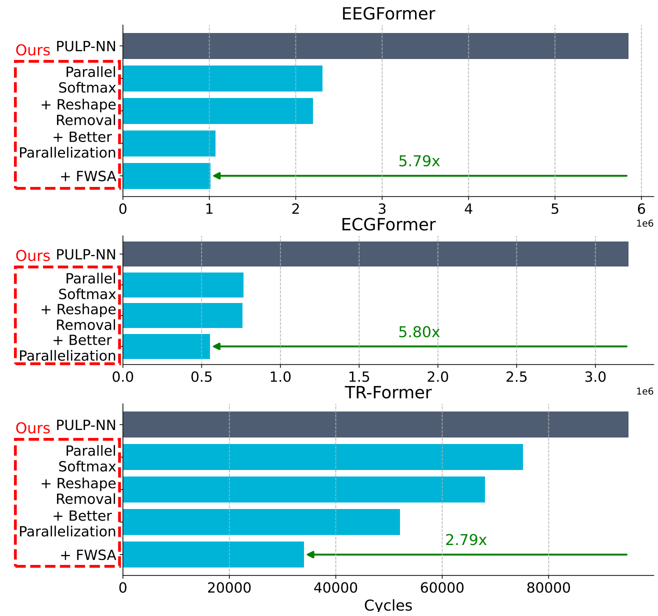


Fig. 12: Breakdown of the impact of each optimization of the proposed method onto the runtime of EEGFormer, ECGFormer, and TR-Former.

Table 3 provides an ablation study of the effect of the FWSA and the DFT on the three attention stages of the networks detailed in the previous section in terms of memory and latency saved. As said above, given that the DFT does not impact the latency, we only report memory saving for this optimization. However, note that in actual application scenarios, such as object detection, multiple networks often run on the same platform. Thus, reducing the memory footprint of each single network is crucial. For the FWSA, we measure both the impact on the memory peak and the number of cycles.

Concerning the FWSA, its memory peak and the number of operations can be computed offline with Eq. 7. Using this equation, we find that the number of operations of FWSA compared to the MHSA is reduced by 11 % and 23 % for EEGFormer and TR-Former, respectively, while it increases by 30 % for ECGFormer. We measure a reduction of the

TABLE 4: Comparison of our kernel library with PULP-NN and CMSIS-NN onto three commercial MCUs. For each application, we only report our fastest kernel.

Platform	GAP9		STM32L4		STM32H7	
Core(s)	9 RISC-V		1 Cortex-M4		1 Cortex-M7	
Power (mW) / Frequency (MHz)	50mW / 370MHz		10mW / 80MHz		234mW / 480MHz	
Kernels	Ours	PULP-NN [31]	Ours	CMSIS-NN [31]	Ours	CMSIS-NN [31]
EEGFormer MHSA (S=81 , E=32 , P=32, H=8)						
Cycles	1.01M	5.85M	39.71M	60.17M	13.6M	36.47M
Time/Inference (ms)	2.84	15.96	496.37	752.12	28.33	75.98
Energy (µJ)	136	683	4963.8	7521.3	662.98	1777.87
MACs/cycle	5.94	1.03	0.15	0.10	0.44	0.16
Throughput (GMAC/s)	2.20	0.38	0.012	0.0079	0.21	0.079
Energy Efficiency (GMAC/s/W)	46.09	8.88	1.21	0.79	9.07	3.38
ECGFormer MHSA (S=66 , E=16 , P=2, H=8)						
Cycles	553K	3.21M	3.28M	5.75M	1.77M	2.98M
Time/Inference (ms)	1.63	8.70	41	71.87	3.69	6.08
Energy (µJ)	62.1	316.0	410	718.75	86.28	142.34
MACs/cycle	0.37	0.06	0.05	0.03	0.10	0.06
Throughput (GMAC/s)	0.14	0.02	0.0043	0.0024	0.048	0.029
Energy Efficiency (GMAC/s/W)	3.63	0.66	0.43	0.24	2.04	1.24
TR-Former MHSA (S=5 , E=32 , P=32, H=8)						
Cycles	34K	95K	1.18M	1.71M	357K	1.04M
Time/Inference (ms)	0.14	0.26	14.75	21.37	0.74	2.16
Energy (µJ)	4.92	11.40	147.5	213.73	17.4	50.70
MACs/cycle	5.19	1.85	0.17	0.12	0.58	0.20
Throughput (GMAC/s)	1.92	0.68	0.014	0.010	0.28	0.095
Energy Efficiency (GMAC/s/W)	58.51	15.60	1.4	0.97	11.89	4.08

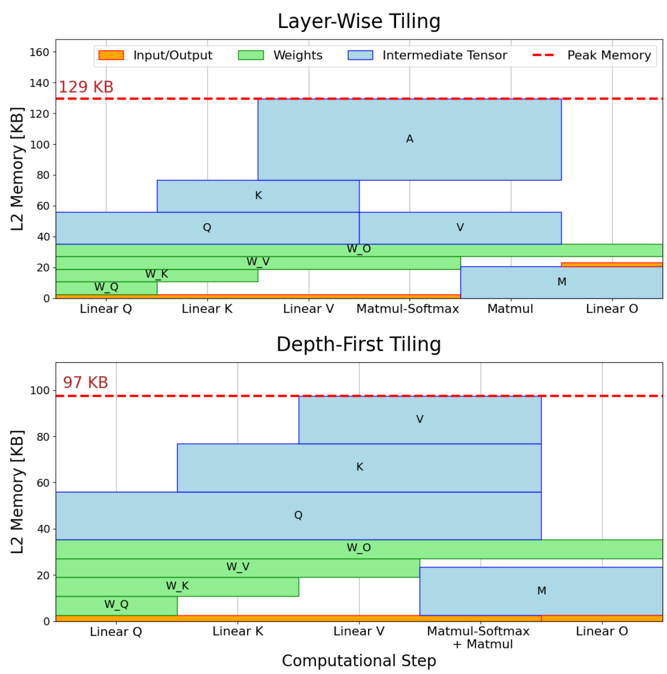


Fig. 13: L2 memory allocation breakdown for each layer of the MHSA for Classical and Depth-First tiling schemes. The dimensions of the MHSA are from EEGFormer [37] ($S = 81$, $E = 32$, $P = 32$, and $H = 8$). Peak L2 memory utilization is indicated in red.

number of cycles of 6 % for EEGFormer and 35 % for TR-Former. The difference is explained by the hyperparameters of the transformers that influence the efficiency of the single layers and by the softmax, whose operations are not included in the number of operations of Eq. 7. Reciprocally, for ECGFormer, the number of cycles increases less than expected, by only 3 %.

This effect is caused by the modification of the reduction dimension of the GEMM inside the ECGFormer attention block, from $P = 2$ for the MHSA to $E = 32$ for the FWSA, which strongly improves the usage of the SIMD and loop unrolling.

Concerning the L2 memory peak, Table 3 shows that the FWSA reduces the memory peak only for TR-Former. Compared to MHSA, the FWSA does not store the Q and K tensors to generate A, effectively skipping the *Linear* V computation step. Therefore, the FWSA reduces the overall memory peak of the network only when this step is the most memory-demanding.

We evaluate the impact of the DFT both at the scale of the Attention Stage and at the network scale. While the DFT reduces the memory peak of the Attention Stage, if the memory peak is present in another stage, such as the Tokenizer or the Fully-Connected (FC) Stage, it may not reduce the memory peak at the network scale. Thus, we first evaluate the memory peak locally in Table 2 and Figure 13, then we show the impact on the whole networks in Figure 14.

Figure 13 details the L2 memory allocation at each step of the sequential and depth-first tiling for the Attention Stage of EEGFormer. The x-axis shows the individual computational steps of the MHSA while the y-axis represents the L2 memory allocation. A computational step ends when the tiled output reconstructs the complete output tensor. As one can notice, DFT reduces the number of computational steps by one. As explained in Sec. 4.5, this is because we tile the two GEMMs of the MHSA together and do not materialize the attention matrix A. Consequently, the memory peak moves from 129 kB to 97 kB mainly due to skipping the storage of the matrix A. As shown in Table 2, at the scale of the Attention Stage, using DFT allows a substantial reduction of the memory peak of 24 % and 84 % for EEGFormer and ECGFormer's attention block, respectively. Indeed, our DFT scheme aims at avoiding storing the attention matrix A of dimension $(H \times S \times S)$. Hence, transformer networks with a large ratio between sequence length and projection dimension will benefit more from it. Concerning TR-Former, the memory peak in the Attention Stage happens in the *Linear* V computation step; therefore, the DFT does not reduce it.

Figure 14 shows the memory peak of the three stages of the transformer, (Tokenizer, Attention, and FC), for EEGFormer, ECGFormer, and TR-Former. The reduction of the Attention Stage memory peak is represented by the hashed bar. At the network scale, the DFT reduces the memory

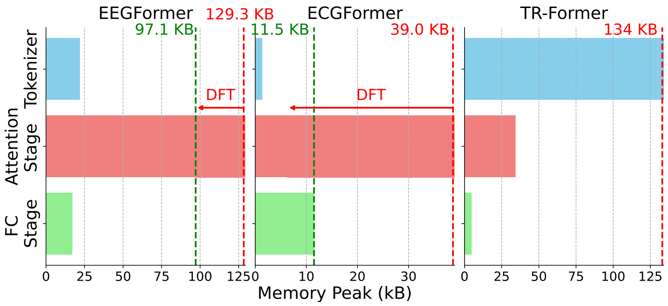


Fig. 14: Impact of the DFT on the memory peak at the network scale. The green line denotes the memory peak after applying DFT while the red one indicates the memory peak before using DFT.

peak by 24 % for EEGFormer and 71 % for ECGFormer. Interestingly, the effect of DFT at the network scale for each transformer is different. On EEGFormer, the DFT reduces the memory peak at the network scale, but the memory peak is still caused by the attention stage. In the case of ECGFormer, the network's memory peak moves from the attention stage to the FC stage when one apply the DFT. Finally, for TR-Former, the DFT does not reduce the memory peak of the attention stage and the tokenizer is completely dominating the memory peak.

5.5 Comparison with State-of-the-art

We demonstrated in Sec. 5.2.1 and Sec. 5.2.2 the higher efficiency and scaling capabilities of our kernels compared to the SotA. In this section, we compare them on the three real transformer networks, characterized by layers dimensions often unsuited to exploit kernel efficiency. For instance, the value of the projection dimension P of ECGFormer is 2. Hence, we cannot use SIMD for kernels where P is the most internal loop, such as the first GEMM operation (see Fig. 1).

Table 4 showcases latency, energy, and different efficiency metrics for the attention blocks of our three use cases onto three hardware targets. We reported the reference SotA kernel results for each target compared to our best kernel alternative, i.e., FWSA for EEGFormer and TR-Former and MHSA for ECGFormer. Specifically, to compare with SotA PULP-NN and CMSIS-NN kernels libraries, we leverage their optimized linear layers kernels, add additional loops, data marshaling operations, and I-BERT's integer softmax [41] to implement the attention layer. The average improvement in terms of latency of our best kernels on the three attention blocks is 4.80 ×, 1.57 ×, and 2.43 × for GAP9, STM32L4, and STM32H7, respectively. Interestingly, the significant improvement compared to the SotA for the three different hardware platforms is always associated with a different attention block.

For GAP9, our kernels reach the top latency improvement of 5.80 × on EEGFormer where our kernels can maximally exploit the SIMD usage on the P dimension and exploit the parallelization also on the S dimension. As discussed in Sec. 5.2.2, the speed-up of our library over PULP-NN on GAP9 is primarily due to parallelizing the execution of the softmax and getting rid of sequential data marshaling operations such as transpositions. For the STM32 platforms, we obtain significant latency gains of 1.73 × and 2.91 × for STM32L4 and STM32H7 on the ECGFormer and TR-Former attention block, respectively. Noteworthy, these two networks further highlight the ability of our kernels also to manage *non-ideal* MHSA parameters, e.g., P=2 for ECGFormer or S=5 for TR-Former, which, on the other hand, strongly impair the performance of SotA kernels. Compared to CMSIS-NN, our kernels feature higher data reuse for these single-core platforms thanks to loop reordering and data marshaling operations fusion. Additionally, we measured the overhead of the re-quantization step performed after each layer and report an average overhead of 7.5% for three networks studied, this results confirm the benefits of executing the network completely in 8 bit integers.

6 Conclusion And Future Work

In this work, we proposed an end-to-end flow to enable efficient deployment of small Transformer models onto commercial MCUs. Our kernel library, tailored for MHSA, together with our optimized schedule and tiling strategy, allows us to speed up the execution of the attention block by a factor of 2.94× on average on RISC-V and ARM platforms. Furthermore, we demonstrate the efficiency of our flow by deploying three Tiny Transformers onto the GAP9 MCU, reaching an average energy consumption and latency of 202 µJ and 5.92 ms, respectively. Our work is open-source at https://github.com/pulp-platform/pulp-transformer. Future research direction would extend our framework to support emerging transformers such as encoder-decoder and decoder-only.

References

[1] C. R. Banbury, V. J. Reddi, M. Lam, W. Fu, A. Fazel, J. Holleman, X. Huang, R. Hurtado, D. Kanter, A. Lokhmotov *et al.*, "Benchmarking TinyML Systems: Challenges and Direction," *arXiv preprint arXiv:2003.04821*, 2020.

[2] J. Lin, W.-M. Chen, Y. Lin, C. Gan, S. Han *et al.*, "MCUNet: Tiny deep learning on IoT devices," *Advances in Neural Information Processing Systems*, vol. 33, pp. 11 711–11 722, 2020.

[3] S. Jain, A. Gural, M. Wu, and C. Dick, "Trained quantization thresholds for accurate and efficient fixed-point inference of deep neural networks," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 112–128, 2020.

[4] C. Gong, Y. Chen, Y. Lu, T. Li, C. Hao, and D. Chen, "VecQ: Minimal loss DNN model compression with vectorized weight quantization," *IEEE Transactions on Computers*, vol. 70, no. 5, pp. 696–710, 2021.

[5] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016*, 2016.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[7] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[8] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[9] A. Burrello, M. Scherer, M. Zanghieri, F. Conti, and L. Benini, "A microcontroller is all you need: Enabling transformer execution on low-power IoT endnodes," in *2021 IEEE International Conference on Omni-Layer Intelligent Systems (COINS)*, 2021, pp. 1–6.

[10] 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," *IEEE Transactions on Computers*, vol. 70, no. 8, pp. 1253–1268, 2021.

[11] D. Hendrycks and K. Gimpel, "Gaussian error linear units (GELUs)," *arXiv preprint arXiv:1606.08415*, 2016.

[12] 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.

[13] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret, "Transformers are RNNs: Fast autoregressive transformers with linear attention," in *International Conference on Machine Learning*. PMLR, 2020, pp. 5156–5165.

[14] K. M. Choromanski, V. Likhosherstov, D. Dohan, X. Song, A. Gane, T. Sarlós, P. Hawkins, J. Q. Davis, A. Mohiuddin, L. Kaiser, D. B. Belanger, L. J. Colwell, and A. Weller, "Rethinking attention with performers," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[15] D. Bolya, C.-Y. Fu, X. Dai, P. Zhang, and J. Hoffman, "Hydra attention: Efficient attention with many heads," in *European Conference on Computer Vision*. Springer, 2022, pp. 35–49.

[16] B. Peng, E. Alcaide, Q. Anthony, A. Albalak, S. Arcadinho *et al.*, "RWKV: Reinventing RNNs for the transformer era," in *Findings of the Association for Computational Linguistics: EMNLP 2023*, H. Bouamor, J. Pino, and K. Bali, Eds. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 14 048–14 077.

[17] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, "Linformer: Self-attention with linear complexity," *arXiv preprint arXiv:2006.04768*, 2020.

[18] Z. Qin, X. Han, W. Sun, D. Li, L. Kong, N. Barnes, and Y. Zhong, "The devil in linear transformer," in *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. Abu Dhabi, United Arab Emirates: Association for Computational Linguistics, 2022, pp. 7025–7041.

[19] N. Shazeer, "Fast transformer decoding: One write-head is all you need," *arXiv preprint arXiv:1911.02150*, 2019.

[20] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebron, and S. Sanghai, "GQA: Training generalized multi-query transformer models from multi-head checkpoints," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, H. Bouamor, J. Pino, and K. Bali, Eds. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 4895–4901.

[21] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[22] T. Dao, "Flashattention-2: Faster attention with better parallelism and work partitioning," in *The Twelfth International Conference on Learning Representations*, 2024.

[23] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with PagedAttention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 611–626.

[24] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently scaling transformer inference," *Proceedings of Machine Learning and Systems*, vol. 5, 2023.

[25] M. Milakov and N. Gimelshein, "Online normalizer calculation for softmax," *arXiv preprint arXiv:1805.02867*, 2018.

[26] L. Liu, Z. Qu, Z. Chen, F. Tu, Y. Ding, and Y. Xie, "Dynamic sparse attention for scalable transformer acceleration," *IEEE Transactions on Computers*, vol. 71, no. 12, pp. 3165–3178, 2022.

[27] T. J. Ham, Y. Lee, S. H. Seo, S. Kim, H. Choi, S. J. Jung, and J. W. Lee, "ELSA: Hardware-software co-design for efficient, lightweight self-attention mechanism in neural networks," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 692–705.

[28] S. G. Bhaskaracharya, J. Demouth, and V. Grover, "Automatic Kernel Generation for Volta Tensor Cores," *arXiv preprint arXiv:2006.12645*, 2020.

[29] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Trans. Math. Softw.*, vol. 5, no. 3, pp. 308–323, Sep. 1979.

[30] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: a system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. USA: USENIX Association, 2016, p. 265–283.

[31] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: Efficient neural network kernels for ARM cortex-M CPUs," *arXiv preprint arXiv:1801.06601*, 2018.

[32] A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, and T. Hoefler, "Data movement is all you need: A case study on optimizing transformers," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 711–732, 2021.

[33] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot MultiBox detector," in *Lecture Notes in Computer Science*. Springer International Publishing, 2016, pp. 21–37.

[34] L. Mei, P. Houshmand, V. Jain, S. Giraldo, and M. Verhelst, "ZigZag: Enlarging joint architecture-mapping design space exploration for DNN accelerators," *IEEE Transactions on Computers*, vol. 70, no. 8, pp. 1160–1174, 2021.

[35] L. Mei, K. Goetschalckx, A. Symons, and M. Verhelst, "DeFiNES: Enabling fast exploration of the depth-first scheduling space for dnn accelerators through analytical modeling," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 570–583.

[36] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[37] P. Busia, A. Cossettini, T. M. Ingolfsson, S. Benatti, A. Burrello, V. J. B. Jung, M. Scherer, M. A. Scrugli, A. Bernini, P. Ducouret, P. Ryvlin, P. Meloni, and L. Benini, "Reducing false alarms in wearable seizure detection with EEGformer: A compact transformer model for MCUs," *IEEE Transactions on Biomedical Circuits and Systems*, pp. 1–13, 2024.

[38] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "OPTQ: Accurate quantization for generative pre-trained transformers," in *The Eleventh International Conference on Learning Representations*, 2023.

[39] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[40] P. Busia, M. A. Scrugli, V. J.-B. Jung, L. Benini, and P. Meloni, "A noisy beat is worth 16 words: A tiny transformer for low-power arrhythmia classification on microcontrollers," *arXiv preprint arXiv:2402.10748*, 2024.

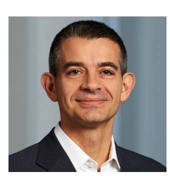
[41] S. Kim, A. Gholami, Z. Yao, M. W. Mahoney, and K. Keutzer, "I-BERT: Integer-only bert quantization," in *International Conference on Machine Learning*. PMLR, 2021, pp. 5506–5518.

Victor Jean-Baptiste Jung received his Bachelor's Degree in Computer Science and Engineering Physics from Juniata College, and the Master's Degree in Computer Science from the Institut Supérieur de l'Electronique et du Numérique of Lille (ISEN Lille) in 2022. After working as a research intern with KU Leuven's MICAS research group supervised by Prof. Marian Verhelst, he started his Ph.D. at ETH Zürich in the Integrated Systems Laboratory with Prof. Dr. Luca Benini. His current research interests include ML Compilers for TinyML platforms, Tiny Transformers, Scheduling and Quantization.

Alessio Burrello is currently a research assistant at Politecnico di Torino. He received his M.Sc. and Ph.D. degrees in Electronic Engineering at the Politecnico of Turin, Italy, and the University of Bologna, respectively, in 2018 and 2023. His research interests include parallel programming models for embedded systems, machine and deep learning, hardware-oriented deep learning, and code optimization for multi-core systems. He has published over 70 papers in peer-reviewed international journals and conferences. His work has been awarded different times, including best paper awards at IEEE ISVLSI 2023 and IEEE BioCAS 2018.

Moritz Scherer received the B.Sc. and M.Sc. degree in electrical engineering and information technology from ETH Zürich in 2018 and 2020, respectively, where he is currently pursuing a Ph.D. degree at the Integrated Systems Laboratory. His current research interests include the design of ultra-low power and energy-efficient circuits and accelerators as well as system-level and embedded design for machine learning and edge computing applications. Moritz Scherer received the ETH Medal for his Master's thesis in 2020.

Francesco Conti (Member, IEEE) received the Ph.D. degree in electronic engineering from the University of Bologna, Italy, in 2016. He is currently a Tenure-Track Assistant Professor with the DEI Department, University of Bologna. From 2016 to 2020, he held a research grant with the University of Bologna and a Post-Doctoral Researcher with ETH Zürich. His research is centered on hardware acceleration in ultra-low power and highly energy-efficient platforms, with a particular focus on System-on-Chips for Artificial Intelligence applications. His research work has resulted in more than 90 publications in international conferences and journals and was awarded several times, including the 2020 IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS I: REGULAR PAPERS Darlington Best Paper Award.



Luca Benini holds the chair of digital Circuits and systems at ETH Zürich and is Full Professor at the Università di Bologna. He received a PhD from Stanford University. His research interests are in energy-efficient parallel computing systems, smart sensing micro-systems and machine learning hardware. He is a Fellow of the ACM, a member of the Academia Europaea and of the Italian Academy of Engineering and Technology.