ARCHIVIO ISTITUZIONALE
DELLA RICERCA

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

Deeploy: Enabling Energy-Efficient Deployment of Small Language Models on Heterogeneous Microcontrollers

This is the final peer-reviewed author’s accepted manuscript (postprint) of the following publication:

Published Version:

Scherer, M., Macan, L., Jung, V.J.B., Wiese, P., Bompani, L., Burrello, A., et al. (2024). Deeploy: Enabling Energy-Efficient Deployment of Small Language Models on Heterogeneous Microcontrollers. IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, 43(11), 4009-4020 [10.1109/tcad.2024.3443718].

Availability:

This version is available at: https://hdl.handle.net/11585/1000951 since: 2025-01-10

Published:

DOI: http://doi.org/10.1109/tcad.2024.3443718

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/).
When citing, please refer to the published version.

(Article begins on next page)

10 July 2025

arXiv:2408.04413v1 [cs.LG] 8 Aug 2024

Deeploy: Enabling Energy-Efficient Deployment of Small Language Models On Heterogeneous Microcontrollers

Moritz Scherer, *Graduate Student Member, IEEE*, Luka Macan, *Graduate Student Member, IEEE*, Victor J.B. Jung, *Graduate Student Member, IEEE*, Philip Wiese, *Graduate Student Member, IEEE*, Luca Bompani, *Graduate Student Member, IEEE*, Alessio Burrello, *Member, IEEE*, Francesco Conti, *Member, IEEE*, Luca Benini, *Fellow, IEEE*

***Abstract*—With the rise of Embodied Foundation Models (EFMs), most notably Small Language Models (SLMs), adapting Transformers for edge applications has become a very active field of research. However, achieving end-to-end deployment of SLMs on microcontroller (MCU)-class chips without high-bandwidth off-chip main memory access is still an open challenge. In this paper, we demonstrate high-efficiency end-to-end SLM deployment on a multicore RISC-V (RV32) MCU augmented with ML instruction extensions and a hardware neural processing unit (NPU). To automate the exploration of the constrained, multi-dimensional memory vs. computation tradeoffs involved in aggressive SLM deployment on heterogeneous (multicore+NPU) resources, we introduce Deeploy, a novel Deep Neural Network (DNN) compiler, which generates highly-optimized C code requiring minimal runtime support. We demonstrate that Deeploy generates end-to-end code for executing SLMs, fully exploiting the RV32 cores' instruction extensions and the NPU: We achieve leading-edge energy and throughput of** 490 µJ/Token, **at** 340 Token/s **for an SLM trained on the TinyStories dataset, running for the first time on an MCU-class device without external memory.**

***Index Terms*—Neural Networks, TinyML, Embodied AI, Foundation Models, Accelerators, Compilers**

I. INTRODUCTION

The latest evolutions in mainstream Artificial Intelligence (AI) have been driven by Transformers, which have taken over from Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs) as the leading edge models for language processing and multi-modal applications [1], [2]. The success of Transformers can be primarily attributed to the emergence of the Foundation Model (FM) paradigm: large Transformer models extensively pre-trained on datasets spanning trillions of tokens and then fine-tuned with a much lower volume of labeled data to solve domain-specific problems. Following the success of FMs in Natural Language Processing (NLP) [1], [3], an increasing number of fields are starting to formulate and adapt FMs for high dimensional sensor data that has traditionally been challenging to process, like decoding neural data [4], [5], or training embodied AI agents [6], [7], which may incorporate multi-modal sensor inputs.

Operating directly on sensory data and in a cyber-physical loop may lead to solving many outstanding challenges in fields such as brain-machine interfaces [5] and miniaturized robotics [7]. However, to materialize this promise, models of this class need to be *embodied* in physical devices as Embodied Foundation Models (EFMs), and they must cope with the strict constraints in terms of compute throughput, power consumption, and footprint typical of edge devices. Unlike datacenter-scale systems, which collect and aggregate sensor data over sharded resources for high-throughput processing, embodied AI systems must process sensor data with extremely low latency and memory capacity under tight power constraints. This is particularly challenging for the smallest class of AI-oriented computers: so-called "*tinyML*" devices operating at the extreme edge, based on microcontroller-class devices without complex operating systems or Memory-Management Units (MMUs), relying on user-level software to implement low-level hardware management functionalities. Despite many recent successes with previous-generation Deep Neural Networks (DNNs), the emergence of the tinyML paradigm for EFMs faces the dual challenge of reducing FMs to a manageable size and enabling their deployment on tiny devices.

A first concrete step in this direction is the recent introduction of Small Language Models (SLMs): FMs with tens to a few hundred million, rather than several billion parameters [8], [9]. While most currently available FMs are focused on processing natural language at a proof-of-concept scale, the effort towards embedded multi-modal sensor inputs with small-scale, application-specific FMs offers a highly promising path for the development of this novel class of models. Much like what happened with the initial emergence of Deep Learning [10], the evolution of advanced tinyML applications based on EFMs is currently prevented by the lack of suitable targets for deployment of these models and, even more, of deployment frameworks that enable utilizing existing specialized hardware to its full capabilities.

Deploying tiny EFMs requires overcoming several challenges specific to the tinyML domain. Large-scale AI inference systems typically employ heterogeneous computer architectures composed by a conventional host (e.g., an x86 processor) and a very large throughput-oriented accelerator (e.g., H100 [11], TPU [12]), which is fully exploited only at large batch sizes. Conversely, tinyML is used for latency-sensitive applications focusing on real-time inference without batching. As a consequence, tinyML AI inference typically employs much more specialized accelerator architectures [13],

---

M. Scherer, V. JB. Jung, P. Wiese, are with the Integrated Systems Laboratory, ETH Zürich, 8092 Zürich, Switzerland; e-mail{scheremo,jungvi,wiesep}@iis.ee.ethz.ch.

L. Macan, L. Bompani, and F. Conti are with the Department of Electrical, Electronic, and Information Engineering (DEI), University of Bologna, 40126 Bologna, Italy; e-mail{luka.macan, luca.bompani5, f.conti}@unibo.it

L. Benini is with the Integrated Systems Laboratory, ETH Zürich, 8092 Zürich, Switzerland and the Department of Electrical, Electronic, and Information Engineering (DEI), University of Bologna, 40126 Bologna, Italy; e-mail lbenini@iis.ee.ethz.ch

A. Burrello is with the Department of Control and Computer Engineering, Politecnico di Torino, 10129, Turin, Italy; e-mail alessio.burrello@unibo.it

This work was funded in part by the Spoke 1 on Future HPC of the Italian Research Center on High-Performance Computing, Big Data and Quantum Computing (ICSC) funded by MUR Mission 4 - Next Generation EU, in part by the Chips Joint Undertaking (Chips-JU) TRISTAN project under grant agreement No 101095947, the CONVOLVE project under grant agreement No 101070374, and the EU Horizon Europe project NeuroSoC under Grant 101070634. Chips-JU, CONVOLVE, and NeuroSoC receive support from the European Union's Horizon Europe research and innovation program.

[14], leading to more complex mapping and optimization challenges for DNN deployment. Furthermore, tinyML's strict constraints on energy efficiency and microcontroller-class computer architecture typically require platform-specific optimization, including memory-aware tiling, static memory allocation, and latency-hiding Direct Memory Access (DMA) scheduling, which require advanced compiler support to scale to complex DNNs like FMs. While several compilers have limited support for user-defined kernels [15], [16], configuring and extending them requires expert knowledge, and their top-down compilation approach often clashes with loosely coupled accelerators. Moreover, mainstream compilers do not address the strict memory constraints in extreme-edge devices.

In this paper, we aim to remove the first barrier towards developing EFM suited for deployment on tinyML platforms: the lack of deployment frameworks that enable their efficient execution. We demonstrate, to the best of our knowledge, the first end-to-end tool flow to deploy EFMs on heterogeneous microcontroller-class systems. Specifically, we demonstrate the end-to-end deployment of a TinyStories-class [8] network on *Siracusa*, an advanced microcontroller in TSMC 16 nm technology featuring embedded non-volatile memory (MRAM) and two heterogeneous compute engines, namely, an octa-core RV32 compute cluster with instruction extensions for ML and a multi-mode CNN Neural Processing Unit (NPU), *N-Eureka* [13]. We present the tooling and algorithms integrated within our deployment framework, Deeploy.

The contributions of this paper are as follows:

- We describe *Deeploy*, a customizable, domain-specific compiler designed for generating bare metal code fitting the memory constraints of extreme edge devices. Deeploy supports all the key computational primitives needed for the execution of Transformer-based EFMs on heterogeneous extreme edge System-on-chips (SoCs) through its bottom-up compilation approach, which allows applying advanced code optimization on expert-optimized kernel templates. We further introduce a novel algorithm for solving the tiling and static memory allocation problems for multi-level software-managed caches and its integration into Deeploy.[1].
- We benchmark common Transformer configurations, demonstrating that code generated by Deeploy maximizes engine utilization in heterogeneous, multi-accelerator SoCs. We achieve data marshaling overheads of just 9 % for large workloads with high arithmetic intensity executing on the cluster cores and NPU collaboratively thanks to efficient data movement acceleration and low-overhead offloading mechanisms.
- As a concrete large-scale end-to-end use-case of Deeploy and its adaptability to heterogeneous hardware platforms, we demonstrate for the first time the deployment of a TinyStories-class SLM on Siracusa, a state-of-the-art heterogeneous Microcontroller (MCU). While using on-chip memory only, we achieve a throughput of 340 Token/s at an energy cost of 490 µJ for autoregressive inference. We show that using the flexible deployment flow enabled by Deeploy for the same SLM allows us to implement multi-layer KV caching using on-chip memory only, improving token throughput by 26 × compared to inference without caches.

The rest of this paper is organized as follows: in Section II, previous work in quantized neural networks, small language models, and neural network deployment for extreme edge devices is introduced and discussed. Section III introduces Deeploy and discusses its deployment flow for Transformers. Section IV discusses the SLM architecture used in this work and the approach to mapping it on Siracusa. In Section V, we present the Siracusa MCU platform. Section VI presents and discusses the end-to-end deployment results, comparing them to the state-of-the-art. Finally, Section VII concludes this paper, summarizing the results and contributions.

II. RELATED WORK

This Section gives an overview of the state-of-the-art on EFMs, focusing on developments towards improvements in energy efficiency and model size and tools to deploy DNNs on extreme edge devices.

A. *Small Foundation Models*

Recently, the development of decoder-only Large Language Models (LLMs) such as Llama [1], and Mixtral [2], and their associated Machine Learning (ML) pipelines led to a new model type: the Foundation Model (FM).

FMs are pre-trained LLMs, which can be fine-tuned for downstream tasks at a fraction of the cost of pre-training, making them particularly relevant for domain specialization. However, LLMs often contain several billion parameters, requiring GiB of storage space, making them incompatible with extreme edge inference.

Addressing this gap, the emerging field of SLMs has gained significant traction in the last year. The aim of SLMs is to compact LLMs down to tens to hundreds of MiB [8], [9], mirroring the evolution of compression of CNNs [17] over the past decade.

This paradigm shift towards compact FMs is particularly interesting for tinyML applications. Incorporating smaller FMs, like SLMs, into embedded devices may enable a new wave of intelligent, responsive, and autonomous devices built on EFMs. Such systems could bridge the gap between human-understandable inputs such as text and performing high-level planning and low-level control tasks [18] and make such advanced capabilities available at the edge, embodied in robots, appliances, and wearable devices.

In this work, we contribute to the growing field of SLM and EFM research and aim to lay the foundation for truly embedded SLMs by providing a foundational deployment flow that supports a wide range of FMs, from autoregressive decoder-only ones to encoder-only ones.

B. *Quantized Transformer Models*

Neural network quantization has been an active field of research for the past decade, as the promises of reduced parameter storage and higher compute efficiency on reduced-precision operands drive the development of increasingly aggressive quantization methods [17], [19].

Improvements in energy efficiency are significant when switching from floating-point point computation to integer arithmetic [20], [21] due to the reduced hardware complexity required to implement the fundamental operations using integer arithmetic. One commonly used approach to quantize DNNs is Quantization-Aware Training (QAT), where the model is trained to overcome quantization effects that occur when using lower-precision values for weights and activations [22]. However, QAT often requires computationally expensive retraining of the model and access to representative datasets, which are not readily available. Post-Training Quantization (PTQ) methods can be applied to quantize models without retraining while conserving full-precision accuracy [23]. Especially in the domain of FMs, PTQ has been successfully applied [24] to reduce the computational cost of quantization. In this work, we apply state-of-the-art PTQ on a publicly available pretrained SLM to achieve quantized inference without loss of accuracy, a prerequisite for energy-efficient inference on extreme edge devices.

---

[1] We will open-source all code required to reproduce our experiments under https://github.com/pulp-platform/deeploy

C. Neural Network Deployment for Extreme Edge Devices

Building on the trends of model quantization and compression, as well as research into more computationally efficient DNNs [25], DNN inference on mobile and embedded devices has become a flourishing field of research [13], [14], [26]. While model deployment on mobile devices like smartphones follows similar approaches to server-scale deployment, relying on the ample compute- and memory resources, hardware-managed caches, and operating systems to carry out task scheduling available to this class of devices, deeply embedded devices face much more severe constraints in deployment. This is especially true for the new generation of MCU-class devices focusing on AI applications. In contrast to their predecessors, these MCUs feature multi-core compute clusters, DNN accelerators, and on-chip memory of up to 10 MiB, split into multiple software-managed memory hierarchy levels [13], [14], [27].

To optimally leverage the compute capabilities of such complex systems, network deployment must simultaneously optimize the execution schedule and tiling of operators and orchestrate overlapping memory transfers using DMAs to achieve low data marshaling overheads and high compute utilization. While modern top-down compilers like MLIR and TVM [15], [16] allow integration of most common Instruction Set Architectures (ISAs) and accelerator APIs, their focus is not on meeting the stringent memory constraints of this class of tinyML devices. Prior work like Dory [28], CoSa [29], and others have addressed these challenges for CNNs by focusing on operator tiling to fit the target's memory constraints. However, these approaches assume a single-cluster memory hierarchy, with undivided memory at each level, and a simple lifetime model for network tensors, which are fundamentally stateless across inference rounds. These simplifying assumptions do not hold for complex heterogeneous multi-accelerator hardware and advanced SLM networks [30], [31].

Moving beyond these prior works, we propose a novel constraint programming algorithm that enables co-optimizing tiling and memory allocation, which overcomes the limitations of previous approaches by supporting data flows with complex lifetimes (e.g. KV caching) as required by EFMs.

III. DEEPLOY

In this section, we provide an overview of the Deeploy compilation flow. In contrast to most state-of-the-art compilers for DNNs, which lower DNN representations top-down into predefined primitives that need to be implemented by each backend [15], [16], [32], Deeploy employs a bottom-up compilation approach, where the compiler implements networks by composing user-provided C kernels, extending them with code generation passes to implement tiling and memory allocation. This bottom-up approach to compilation provides three key advantages: first, it supports reusing hand-optimized kernel libraries commonly available for most ISAs and accelerators. Second, it can be easily extended to support highly customized non-standard compute platforms, including heterogeneous SoCs featuring multiple accelerators for which a low-level compiler backend may not exist. Third, it allows easy integration of novel operators found in emerging Transformer architectures without invasive modifications to the deployment flow.

Deeploy is organized in three building blocks; the *Frontend* validates and transforms the graph representation into a representation that suits the platform and assigns kernel templates to each operator. The *Midend* performs all tiling and static memory allocation computations, guaranteeing that the computed program schedule may execute without unscheduled runtime memory spills. Finally, the *Backend* uses the optimized graph representation generated in the *Frontend*, and the generated tiling schedule and memory allocation map generated in the *Midend* to create executable code through a series of code generation passes. All deployment targets share the same execution flow, and Deeploy uses a configurable platform abstraction, the *Deployment Platform*, which allows it to steer operators' mapping, optimization, and lowering according to the platform's configuration. An overview of the Deeploy execution flow is shown in Figure 1.

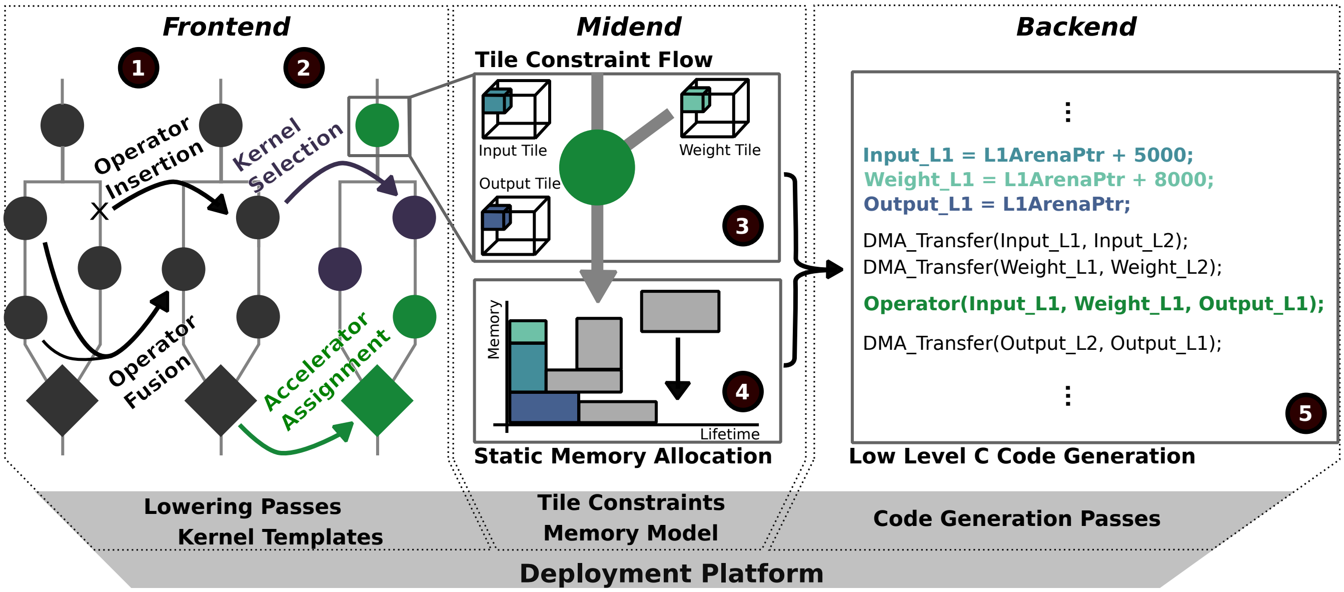
A. Data Structures

Deeploy distinguishes between three types of buffers: *Variable Buffer*s, *Transient Buffer*s, and *Constant Buffer*s. *Variable Buffer*s represent tensors that contain data that is not constant at compile-time, i.e., network inputs, outputs, and intermediate activations. *Constant Buffer*s represent compile-time constant data used in inference, i.e., network weights and other network parameters. Lastly, *Transient Buffer*s represent scratchpad memory locations for kernel execution, e.g., *im2col* buffers for convolution kernels [33], [34], or reorder buffers for efficient transposition kernels. Typically, the amount of space used in *Transient Buffer*s depends on the operator's parametrization, distinguishing them from *Variable Buffer*s. In contrast to simpler DNN topologies, EFMs employ data structures that require advanced allocation strategies, such as the KV caches of autoregressive SLMs, as they have more complex buffer lifetime requirements than intermediate tensors found in CNNs. Addressing these constraints requires a more sophisticated management of the buffers' lifetime and memory allocation than in other deployment tools targeting extreme edge devices [28], [29].

The distinction between global and local section buffers is relevant for code generation; global objects are allocated as global C variables, while local objects are only accessible in the inference code. As such, global variables are alive throughout an inference execution, while local variables are allocated and deallocated as the network's execution schedule requires.

B. Frontend

Deeploy's *Frontend* is designed around ingesting quantized Open Neural Network Exchange (ONNX) graphs produced by DNN and Transformer quantization tools like Quantlib [35]. Deeploy implements a configurable lowering pass system based on pattern matching of ONNX graphs to enable efficient and customizable graph-lowering strategies. Each lowering pass consists of a user-defined



```
⋮
Input_L1 = L1ArenaPtr + 5000;
Weight_L1 = L1ArenaPtr + 8000;
Output_L1 = L1ArenaPtr;
DMA_Transfer(Input_L1, Input_L2);
DMA_Transfer(Weight_L1, Weight_L2);
Operator(Input_L1, Weight_L1, Output_L1);
DMA_Transfer(Output_L2, Output_L1);
⋮
```

Fig. 1. Overview of the Deeploy Execution Flow. Steps ① and steps ② are part of the *Frontend*. In the first step, the graph is modified by fusing and inserting platform-specific operators, for example, transposition operators, to match data layout requirements. In the second step, datatypes for every tensor are inferred, the accelerator target is chosen, and kernel templates are selected. The first step in the *Midend*, step ③, is the Tile Constraint Flow, which computes geometrical constraints for the tile sizes of each tensor, adding them to a CP. The resulting tensor size variables are translated into a 2D bin packing problem in step ④. The solution of the co-constrained tiling and static memory allocation problem is computed by the ORTools CP-SAT solver and finally processed in step ⑤ in the *Backend*. Step ⑤ generates platform-specific C Code exploiting DMA transfers. Each step of the execution flow is highly configurable through the *Deployment Platform* object.

replacement function and a *source pattern*, which describes the subgraph that should be replaced. Using the replacement function, each lowering pass uses the matched sub-graph to generate a *target pattern*, which replaces the *source pattern*. Using this system, the first processing step in the *Frontend* is transforming the input graph into a custom, platform-specific ONNX dialect using lowering passes provided by the *Deployment Platform*. The user further defines operator mappings between custom operators and the engines available in the target platform to control the code generation on the level of individual operators. Common tinyML kernel libraries like CMSIS-NN and PULP-NN [33], [34] offer kernels for fused linear operators and activations, which can be lowered into by matching pairs of linear operators and quantization operators. Besides operator fusion optimization passes, Deeploy also supports the minimization and insertion of data marshaling operators like transpositions to match the data layout requirements of kernel libraries. An example of such an operator insertion pass is adding transpositions operators to optimize the data layout of the B matrix for General Matrix Multiplication (GEMM) kernels of type $Y = \alpha AB + \beta C$ for better data access locality.

The second step after transforming the input graph into the platform-specific dialect in the *Frontend* is parsing, during which every operator in the network is analyzed to construct an initial context of buffers used in the network's execution, and *Type Inference & Kernel Selection* where every buffer in the context is assigned a type. The types used in Deeploy correspond to standard C types (e.g., *int8_t*, *float32*) or custom data types, depending on the kernels used by the *Deployment Platform*. To guarantee a valid type assignment, Deeploy propagates type information top-to-bottom. The user must only provide the input types for every graph's input tensor to achieve this; then, using this information, Deeploy matches the input types of each operator with one of the kernel signatures provided by the *Deployment Platform*.

The final result of the *Frontend* is an assignment of low-level kernel templates to every operator in the lowered platform-specific ONNX, which satisfies the type constraints imposed by the network's operators.

C. *Midend*

The second stage of Deeploy's execution flow, the *Midend*, receives the platform-specific ONNX graph and the kernel assignment for each operator from the *Frontend*. The *Midend*'s purpose is to perform all optimization operations required to generate low-level optimized C code for the target platform in the *Backend*. The *Midend* is divided into two optimization steps: *Memory Level Annotation* and *Tiling & Memory Scheduling*. To model the CP used to compute the tiling and static memory allocation solution, Deeploy uses Google's ORTools.

1) Memory Level Annotation: The memory level annotation step annotates every buffer in the compilation context with a memory hierarchy level. The motivation for defining the storage location of every tensor is to model code generation constraints closely to the hardware; most embedded systems designed for tinyML applications use multiple memory or cache levels [13], [14] to optimize the trade-off between storage density and memory access latency. While Deeploy supports the tiling of buffers, directly assigning buffers' memory levels to lower cache levels can lead to performance improvements. When targeting accelerators that would otherwise be limited by the available bandwidth towards higher-level caches, controlling memory allocation has a significant performance impact [13].

2) Tiling: The second processing step in the *Midend* is *Tiling & Memory Scheduling*. For every kernel template chosen in the *Frontend*, the target platform must specify a Tile Constraint (TC). The TC models the geometric and platform-specific constraints for

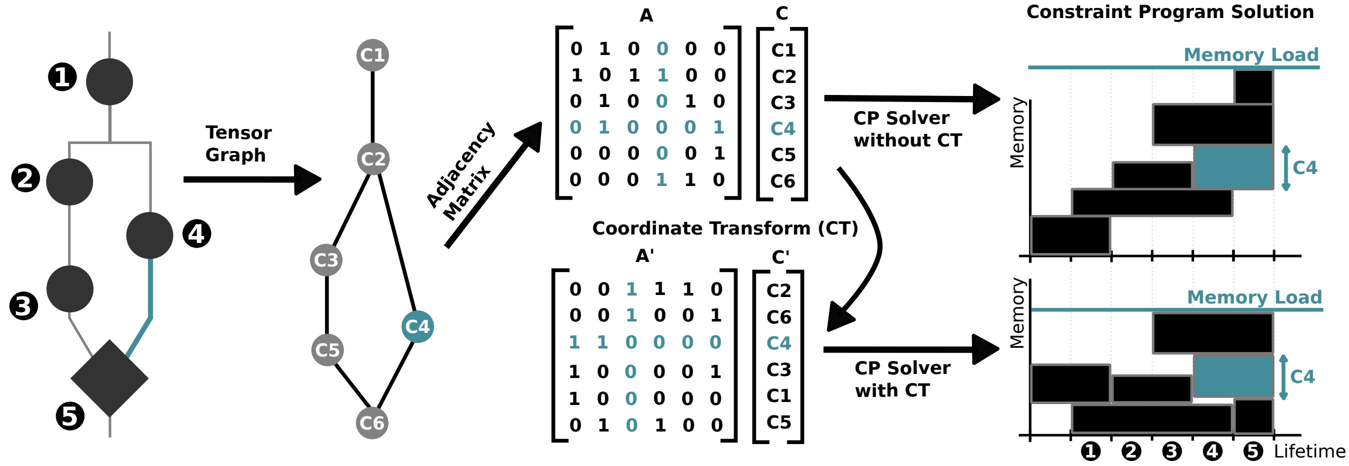


Fig. 2. Example of the co-optimization of tiling and static memory allocation algorithm for one memory level in Deeploy. First, the lifetime of each tensor in the graph is calculated under the execution schedule shown on the left. Next, the memory scheduler constructs an adjacency matrix of the tensor graph and extracts the cost vector from the tile constraint flow shown in the middle. Finally, Deeploy applies a coordinate transform within the CP. On the right-hand side, the 2D bin packing solution is presented with the naive solution on top, and the solution found by Deeploy is shown below.

tiling an operator. For a tiling solution to be correct, all geometric constraints must hold. For example, the spatial dimensions of a softmax activation's output tile must be the same as its input tile's dimensions. As such, geometric constraints do not depend on the implementation of an operator. While it is possible to tile large tensor operators down to single instructions when targeting processor cores, the same does not hold for accelerators. Specifying TCs and platform-specific constraints on a per-kernel basis is especially important for handling the tiling problem for loosely-coupled accelerators since they typically only support specific dimensions to be tiled, owing to their specialized datapaths [13], [14].

Similarly to the *Type Inference & Kernel Selection* flow, the Tile Constraint Flow (TCF) is applied top-to-bottom through the execution schedule of the network, adding the geometric and platform-specific tile constraints of every operator to the CP. Furthermore, the TCF adds one symbolic variable per dimension per tensor in the network to the CP and a symbolic variable for every tensor, representing its size as the product of all dimension variables. Using this formulation, the solution of the CP represents the size of the largest tile.

3) Memory Scheduling: After the geometrical constraints of every mapped kernel template in the network are collected and added to the CP, Deeploy's memory scheduler calculates the lifetime of every tensor in the network over the user-provided execution schedule of the ONNX graph as shown in Figure 2. As previously mentioned, this is an essential step for autoregressive Transformers that must accommodate short-lived tensors (e.g., intermediate activations, residuals) and long-lived buffers (such as KV caches).

Deeploy's memory scheduler computes a tiling path using the *Deployment Platform*'s memory hierarchy model to assign a sequence of memory transfers through the different memory levels. Using the calculated lifetimes and the tensor's size variable computed before, the memory scheduler models the problem of computing a static memory allocation schedule as a 2D bin packing problem [30], [36], where the horizontal axis represents lifetime, and the vertical axis represents memory address space.

Similar to other state-of-the-art algorithms [30], Deeploy's scheduling CP works with Tetris scheduling introduced in TetriSched [37], where memory buffers are scheduled one after another, adding to the maximum load of each of their lifetime's bins. To solve the tiling and allocation problem in a single shot, the memory allocation of each buffer is coupled to the tiling solution, which requires expressing the order in which they are scheduled within the CP as well.

The first step to modeling the memory allocation problem is to pick a random schedule of memory buffers and compute the adjacency matrix A of the tensor graph. We collect the memory size of each buffer, represented as an integer variable of the CP, in a cost vector C. For any permutation matrix P, $A' = P \times A \times P^T$ is a valid adjacency matrix with associated cost vector $C' = P \times C$. A valid $N \times N$ permutation matrix can be expressed as:

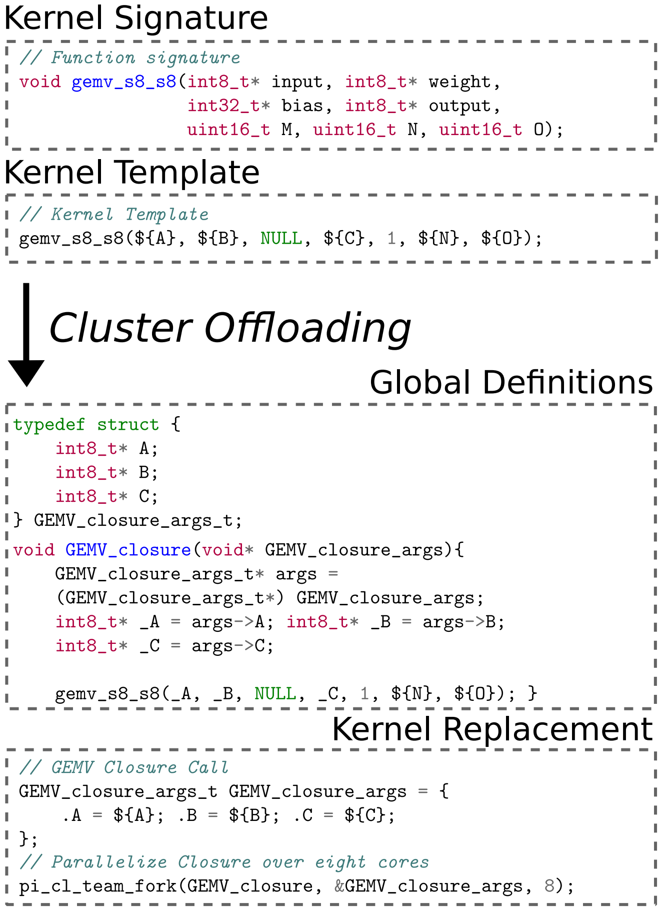
$$
\begin{aligned}
p_{i,j} &\in [0,1] && \forall i,j \in [0,N-1] \\
\sum_{i=0}^{N-1} p_{i,j} &= 1 && \forall j \in [0,N-1] \\
\sum_{i=0}^{N-1} p_{j,i} &= 1 && \forall j \in [0,N-1]
\end{aligned}
$$

Next, the total memory load is computed iteratively using $A' \& C'$: since we use Tetris scheduling, we add each buffer's memory size to the size of the last scheduled buffer whose lifetime overlaps. We use a vector of intermediate variables containing one entry for each buffer, H, representing the memory load in the lifetime region of each buffer. The vector H is computed as follows:

$$
\begin{aligned}
&H_0 = 0 \\
&H_j = \max_{i=0\ldots j-1}(A'[j,i] \cdot H_i) + C'_j
\end{aligned}
$$

The total worst-case memory load for all execution steps is then computed as memory load $= \max_{i=0\ldots N}(H_i)$.

In contrast to other static memory schedule algorithms, which focus on calculating an optimal solution for memory blocks of fixed size, our algorithm combines the constraints on tile sizes and memory layout calculation into a single CP; this allows Deeploy to simultaneously optimize static memory allocation as well as



```
// Function signature
void gemv_s8_s8(int8_t* input, int8_t* weight,
                int32_t* bias, int8_t* output,
                uint16_t M, uint16_t N, uint16_t O);
```

```
// Kernel Template
gemv_s8_s8(${A}, ${B}, NULL, ${C}, 1, ${N}, ${O});
```

```
typedef struct {
    int8_t* A;
    int8_t* B;
    int8_t* C;
} GEMV_closure_args_t;
void GEMV_closure(void* GEMV_closure_args){
    GEMV_closure_args_t* args =
    (GEMV_closure_args_t*) GEMV_closure_args;
    int8_t* _A = args->A; int8_t* _B = args->B;
    int8_t* _C = args->C;

    gemv_s8_s8(_A, _B, NULL, _C, 1, ${N}, ${O}); }
```

```
// GEMV Closure Call
GEMV_closure_args_t GEMV_closure_args = {
    .A = ${A}; .B = ${B}; .C = ${C};
};
// Parallelize Closure over eight cores
pi_cl_team_fork(GEMV_closure, &GEMV_closure_args, 8);
```

Fig. 3. Bottom-up offloading closure generation for a GEMV kernel. All arguments that refer to non-global *Variable Buffers* or *Constant Buffers* are captured and used to generate a closure struct typedef and a closure function that unpacks the argument struct and calls the original kernel. Finally, the kernel template is replaced with a function *pi_cl_team_fork*, which takes the newly generated closure as an argument and offloads its execution to all eight cluster cores.

tile sizing to control memory use during the entire inference process, which is critical to matching the memory constraints of extreme-edge SoCs with the complex buffer lifetime requirements of Transformers. An overview of the co-constrained tiling and static memory allocation algorithm is shown in Figure 2.

D. Backend

Every kernel template picked in the *Frontend* is assigned a list of code generation passes by the *Deployment Platform*. Each code generation pass operates on a code segment, starting from the original kernel template, and may add to or modify its code segment. Besides enabling integration of custom passes, Deeploy offers standard code generation passes required for generating correct code, e.g., memory allocation and deallocation generation, which inserts calls to heap-based allocators or sets pointers to predefined memory locations calculated during *Tiling & Memory Scheduling*.

An essential set of code generation passes is centered around generating closures for code segments. In the context of Deeploy, closure generation consists of three parts: the closure function itself, which encapsulates a code segment; the closure environment, which contains every free variable used within the code segment and must be passed to the closure function; and the closure invocation, which is either an offloading function or a call to the closure function.

Deeploy implements closures as standard C functions by generating a function call around the target code segment and passing the closure environment as a struct pointer. Deeploy captures the relevant free variable expressions by analyzing the Abstract Syntax Tree (AST) of the underlying code segment using the Mako templating library [38]; since the function signature of the kernel template is known to Deeploy, it can extract arguments used in the kernel template that refer to local buffers, and pass them to the closure using an argument struct. During code generation, the closure generation pass hoists the closure function definition into the global context, inserts code for constructing the argument struct and returns the function call to the hoisted closure as the new code segment for subsequent code generation passes.

An important application for Deeploy's closures is to facilitate operator offloading, which is required for programming processor-based accelerators like compute clusters or loosely-coupled, memory-mapped accelerators like NPUs. An example of closure generation for operator offloading to the octa-core cluster is shown in Figure 3.

Tiling code generation is implemented as a pass as well. Deeploy supports DMA engines and uses them in tiling code generation to move tiles between different memory hierarchy levels according to the tiling solution computed in *Tiling & Memory Scheduling*. To hide the latency of DMA transfers, Deeploy can configure tiling for operators to use double-buffering, which constrains the tiling solution to reserve twice the required space for every input- and output tile. During code generation, Deeploy schedules data fetching and writeback to occur in parallel with kernel execution to minimize latency.

IV. TINYSTORIES LLAMA MODEL

As a concrete example of our deployment flow for next-generation EFMs, we quantize and deploy an SLM on a heterogeneous MCU, Siracusa, introduced in Section V. We chose a Llama2 model pre-trained on the tinyStories dataset [8] from HuggingFace[2], with a hidden size $d_m = 64$, $h = 16$ parallel attention heads, $N = 8$ layers and an intermediate size $d_{ff} = 256$ for the feed-forward layer. The model architecture is shown in Figure 4. Note that, however, any SLM fitting the memory constraints of the target platform can be deployed with the same flow.

Like all other decoder-based language models, the Llama model we use in this work has two fundamental inference modes, which we refer to as autoregressive inference mode and parallel inference mode, and generates its response in two distinct phases, the *prompting phase* and *generation phase*; the prompting phase ingests the initial sequence of user input tokens, whereas the generation phase generates the model's output tokens autoregressively.

A. Prompting Phase

Inferences follow a two-pass regime: First, the text input is translated into a sequence of tokens, typically referred to as the prompt. The prompt can have an arbitrary sequence length S_p, up to the size of the context window of the model.

In the first pass of the model, the prompt is processed to produce the first output token. Since all tokens of the prompt are available

---

[2] https://huggingface.co/Maykeye/TinyLLama-v0

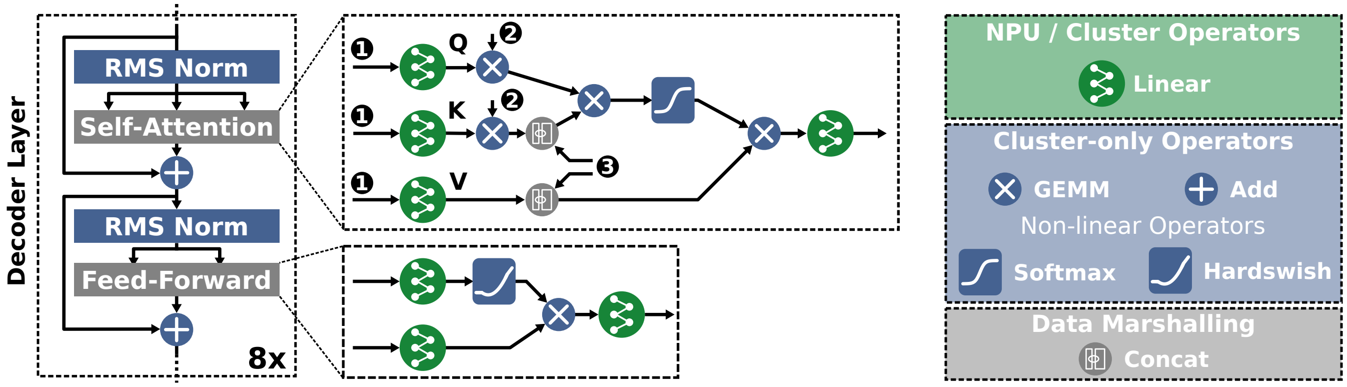


Fig. 4. Overview of the Llama model deployed in this work. The eight decoder layers of the model are shown on the left and consist of an *RMSNorm - Self-Attention - RMSNorm - Feed-Forward* layer stack. Input ① in the self-attention inset corresponds to the token input. Input ② corresponds to the rotational embedding used in Llama models. Input ③ are the KV cache inputs used during autoregressive inference. Notably, during autoregressive inference, the new row of the K and V matrices computed on the input token are appended to the KV cache.

ab initio, the decoder can process them in a parallel single-shot fashion by applying causal-masking of the attention matrix [39]. This first pass generates the first token output and the K and V matrices, which may be reused in the subsequent *generation phase*. This process parallels the function of encoder layers used in the first Transformer models [39].

B. Generation Phase

In the generation phase of the inference process, output tokens are generated one at a time using the previous token outputs as the model's input. While every step of the generation phase may use the same parallel inference mode described in the previous Section, doing so would require recomputing all previous tokens' K and V submatrices. Therefore, the K and V matrices of previous inference steps are typically cached in memory to avoid the quadratic cost of recomputing them [39].

As the parallel inference mode and autoregressive inference mode require different trade-offs in memory allocation for KV caching and storage of intermediate results we deploy them using separate ONNX models which reflect these trade-offs: For the parallel inference mode we export an ONNX model with a single input and output for the token sequence and outputs for the computed KV submatrices which are stored for the next generation phase. For the autoregressive inference mode, we use an ONNX model that additionally requires cached KV submatrices. While computing outputs using KV caches is significantly more efficient regarding the absolute number of operations, loading and storing the KV caches induces significant data movement, and the smaller operator dimensions make the generation phase much more challenging to accelerate.

C. Quantization Setup

To quantize the SLM for deployment on extreme edge devices with integer-focused Single Instruction Multiple Data (SIMD) processors and DNN accelerators, we used QuantLib [35] with the Trained Quantization Thresholds (TQT) algorithm for PTQ [22]. QuantLib inserts requantization layers after operators which results in higher bitwidth outputs. Furthermore, it harmonizes scaling factors for operators like addition and concatenation and replaces various operators with their quantization-aware equivalents. Following this, we use a single token to execute PTQ over three inference epochs. Initially, we collect statistics to initialize the clipping bound for all activations and weights. At the end of the second epoch, we quantize all linear operations, and in the final epoch, we quantize non-linear operations, including Softmax and RMSNorm. Subsequently, the model is projected to the integer domain and exported as an ONNX graph. To leverage the advanced hardware support for SIMD operations in the PULP Cluster and Siracusa's NPU, *N-Eureka*, we chose to quantize all activations and weights used in matrix multiplication to 8 bit integer precision. We use I-BERT's approximation for Softmax [21] and the Hardswish approximation for Swish activations [25]. Moreover, we perform all divisions in RMSNorm [40] layers with 32 bit numerators and denominators to preserve accuracy.

V. DEPLOYMENT PLATFORM

This Section introduces the hardware platform used in this work as a deployment target to deploy the SLM introduced in Section IV and goes over the NPU-specific *Deployment Platform* implementation in Deeploy.

A. Siracusa

Siracusa [13], is a low-power, heterogeneous RISC-V MCU implemented in TSMC 16 nm technology, which is the multi-accelerator SoC targetted in this work. Siracusa is designed for efficient AI inference, which can leverage its dedicated NPU, *N-Eureka*, and generalistic Digital Signal Processing (DSP) tasks, which can exploit both dedicated XpulpNN ISA extensions [33] enabling SIMD processing of low-precision integers, as well as an accelerator cluster of eight RISC-V cores which enable Single Program Multiple Data (SPMD) processing.

To enable single-latency access from cluster cores to the L1 Tightly-coupled Data Memory (TCDM), all cores and the 16 L1 memory banks are connected through a TCDM interconnect using one 32-bit port each, granting a total memory bandwidth of 256 bit/cycle to the compute cluster. The cluster's TCDM memory banks are also accessible from the *N-Eureka* accelerator using 9-bank wide, 288 bit accesses. To manage contention on accesses to the single-ported memory banks, Siracusa integrates a lightweight, programmable access arbiter, which allows the set the maximum number of stall cycles for the accelerator; if accesses from the core-side interconnect cause accelerator access to stall for the programmed number of cycles, the arbiter will stall core accesses and grant it to *N-Eureka*.

The *N-Eureka* accelerator uses a mixed-weight-precision bit-serial datapath, which is optimized for executing dense 3×3, depthwise 3×3, and dense 1×1 convolution operations with 8 bit activations and 2 bit to 8 bit convolution weights [13]. To support the bit-serial nature of the datapath, *N-Eureka* requires its weights to be stored in a non-standard bit-interleaved data format, which requires offline transposition, padding, and bit shuffling of CNN weight tensors. *N-Eureka* is designed as an output-stationary accelerator, opting to cache small input tiles and streaming weights. To execute operations larger than its internal buffers, it integrates a hardware tiler with a programmable number of tiles and strides between dimensions and fixed tile sizes that match the buffer sizes. To increase the available memory bandwidth for *N-Eureka*'s weights and minimize off-chip access to fetch weights, the cluster integrates a Neural Memory Subsystem (NMS), which contains two dedicated 4 MiB memory subsystems, implemented in Static Random Access Memory (SRAM) and Magnetoresistive Random Access Memory (MRAM) technology respectively, which are designed to hold weights for the *N-Eureka* accelerator and are attached through a dedicated 256 bit/cycle weight data port.

The compute cluster and *N-Eureka* are located in a shared clock domain, the heterogeneous cluster, which communicates with the rest of the SoC, mainly consisting of a controller core, 2 MiB L2 memory, and peripherals, through a 64 bit wide Advanced eXtensible Interface Bus (AXI) bus, which can be used by a DMA integrated within the cluster, to transfer data between the L1 and L2 memories autonomously.

While Siracusa is equipped with significant computing capabilities through two dedicated accelerators and sizeable on-chip memory, deploying an advanced neural network on this device is a challenging problem. While weight storage for layers that can be executed on *N-Eureka* is plentiful, all other layers' activation, weight, and output tensors must be tiled to fit within 256 KiB of L1 memory. Furthermore, memory transfers between L2 and L1 should be orchestrated using the DMA to minimize stalling.

B. Deeploy Integration

We address the deployment challenges posed by Siracusa's heterogeneity through an augmented *Deployment Platform* model. This subsection gives an overview of the additions implemented to use Deeploy for deploying SLMs on Siracusa and, more generally, of the modifications needed to support a generic new platform in our deployment tool.

As Deeploy's core primitives are optimized kernels, we chose the PULP-NN [33] kernel library, which integrates parallel kernels as well as single-core implementations, as our target for utilizing the octa-core cluster. The PULP-NN kernels focus on efficient implementations of fused linear and quantization layers. We support fused layers through lowering passes that match the supported operator combinations and merge them in the *Frontend* of Deeploy. We further added fused linear operator TCs, which add kernel-specific constraints besides providing general geometric constraints.

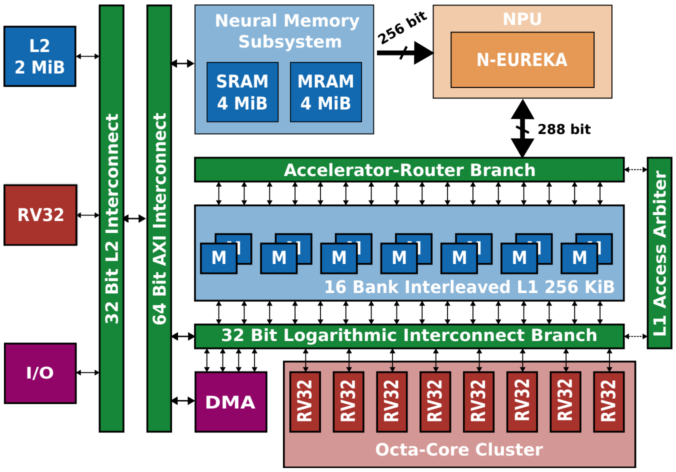


Fig. 5. Overview of the Siracusa SoC featuring its DSP-enhanced octa-core RISC-V cluster and host controller (red), NPU (orange), complex memory hierarchy with two levels of scratchpad memory and a Neural Memory Subsystem (blue), two arbitrated interconnects towards the L1 memory and an AXI interconnect (green), and peripherals such as the cluster DMA and chip-level I/O (purple).

We implement function offloading to both the NPU and the octa-core compute cluster in Siracusa using the closure system, as detailed in Section III-D.

The *N-Eureka* accelerator provides greater compute capabilities than the octa-core cluster for CNN operators, achieving a peak throughput in the range of hundreds of GOp/s for pointwise and 3×3 convolutions. Even though SLMs do not employ these types of operations, we add a custom *linear layer to pointwise convolution* lowering pass that converts GEMM operators with compile-time constant weight matrices into pointwise convolutions. This method allows us to deploy all linear layers in Transformer models, as shown in Figure 4, on the NPU.

For this lowering pass, we consider GEMM operation of type $Y = \alpha AB + \beta C$, where Q are appropriately integer-quantized numbers:

$$A \in Q^{M\times N} \qquad B \in Q^{N\times O} \qquad C \in Q^{M\times O} \qquad Y \in Q^{M\times O}$$

Similarly, we define the pointwise convolution operator as $Y = A \otimes B + C$, with the same dimension definition used in PyTorch:

$$\begin{aligned} &A \in Q^{H\times W\times C_{in}} && B \in Q^{C_{out}\times 1\times 1\times C_{in}} \\ &C \in Q^{H\times W\times C_{out}} && Y \in Q^{H\times W\times C_{out}} \end{aligned}$$

We map the dimensions of the pointwise convolution to those of a GEMM operation by setting $H := 1$, $W := M$, $C_{in} := N$, and $C_{out} := O$. For this mapping to succeed, the C operand in the GEMM operation must be reducible to a dimension of $[1\times O]$, i.e., all rows in the matrix are identical.

Lastly, we annotate all pointwise convolution weights previously transformed from the GEMM operators to be allocated in the NMS, allowing the accelerator to leverage its significantly larger bandwidth.

C. Deployment Setup

As explained in Section IV, the dual inference modes of decoder-only models require different deployment strategies, as

TABLE I

COMPILER PERFORMANCE METRICS FOR THE 128TH AUTOREGRESSIVE INFERENCE STEP USING THE *NPU with NMS Deployment* SCENARIO WITH VARYING NUMBERS OF DECODER LAYERS

Number of decoder layers	1	2	3	4	5	6	7	8
Number of operators	32	64	96	128	160	192	224	256
Deeploy compilation time [s]	1	2.65	5.17	7.92	11.72	16.9	22.25	28
Text section size [kB]	42.4	61.8	84.4	105.7	128.1	150.1	171.5	194.8
Data section size [kB] (input and output buffers)	10.4	18.6	26.8	35.0	43.2	51.4	59.5	67.7
Data section size [kB] (requantization parameters)	7.0	15.3	23.6	31.9	40.2	48.5	57.8	66.1
Weight memory section size [kB] (pointwise convolution parameters)	64	128	192	256	320	384	448	512

the autoregressive inference mode requires significant memory for KV caching. We deploy two model prototypes to accommodate this difference, one for autoregressive inference mode and one for parallel inference mode.

The autoregressive inference mode model uses additional network inputs corresponding to the previous sequences' KV caches. Other than that, the deployment setup between both models is equal. We allocate all graph inputs and outputs as global *Variable Buffer*s in Siracusa's L2 memory, and annotate all local *Variable Buffer*s modeling intermediate tensors in L2 as well. In deployment scenarios that use Siracusa's NMS, we allocate all linear layer weights in the NMS but use L2 for all activations.

Unless stated differently, all network operators are executed on the cluster and use Deeploy's TCF to generate tiled inference code, which orchestrates transfers of input, weight, and output tensors between the L2 memory and the L1 memory. For operators executed on the NPU, weights are stored in the NMS in their entirety and ingested by the accelerator without moving them into L1 first, leveraging the increased available bandwidth from the NMS.

VI. RESULTS

This section discusses the measurement results of deploying the TinyStories SLM on Siracusa and benchmarking results of general Transformer layers. First, we discuss the setup used to measure performance results on Siracusa. Finally, we present our benchmarking and end-to-end silicon measurements, as well as profiling experiments of our compiler.

A. Deployment Evaluation Setup

To evaluate the model's performance in autoregressive mode and for causally masked parallel inference, we measure each inference step individually with code generated by Deeploy. We start from empty KV caches for causally masked parallel inference and process N input tokens simultaneously. We start from the KV caches of the previous inference step for all experiments in autoregressive mode. To calculate the average throughput and energy per token, we take the average over all 256 inference steps.

We report all power numbers measured on a Siracusa prototype board using a Keysight N6715C DC, supplying all operating voltages and measuring current. We perform all experiments under nominal conditions, i.e., 0.8 V supply voltage and 360 MHz operating frequency of the cluster domain. We measure power consumption for every inference by averaging the power consumption of the model run in a continuous loop.

We measure four distinct deployment scenarios: In the first scenario, *Single Core Deployment*, we only generate code using a single RISC-V core. In the second scenario, *Octa-Core Deployment*, we generate code using all eight RISC-V cores of the cluster without using the NPU. In the third scenario, *NPU without NMS Deployment*, we generate code using all eight RISC-V cores and *N-Eureka* without offloading weights to the NMS. In the last scenario, *NPU with NMS Deployment*, we generate code using all eight RISC-V cores and *N-Eureka* with the NMS. We use the Siracusa *Deployment Platform* in Deeploy to generate code for all scenarios.

B. Microbenchmarking Results

To validate our approach of offloading GEMM operators on *N-Eureka*, we first measure the performance of *N-Eureka* and the RISC-V cluster on GEMM kernels. Specifically, we study the performance of the Q, K, and V projections in the attention layer and linear layer performance in feed-forward layers for different sequence lengths S in parallel inference mode. For the Llama model we study in this paper, these projections use dimensions $256 \rightarrow 64$ and $64 \rightarrow 256$. Our measurements are shown in Figure 6.

Transitioning from single-core to octa-core cluster execution, we measure a performance improvement of 6.2 ×, thanks to the low-overhead parallelization on the cluster cores. Transforming the linear layer operators into pointwise convolutions, as explained in Section V-B, enables execution on the NPU, which reduces latency by 25 × compared to the octa-core implementation due to the NPU's significant compute resources for convolution operations. Furthermore, we reduce data movement by allocating the convolution weights to the NPU's NMS, increasing the effective memory bandwidth available to *N-Eureka*. These optimizations improve performance, especially on memory-bound tasks, like linear layers in attention blocks with low sequence length, by 2.1 × compared to NPU execution without the NMS.

We further profile the execution performance of a representative encoder layer as commonly found in non-regressive Transformer models. For our benchmarking, we chose a configuration with hidden size $d_m = 64$ and $h = 16$ parallel attention heads and an intermediate size $d_{ff} = 256$, paralleling the decoder layer in Figure 4. We measure an increase in throughput of 17.8 × when leveraging the NPU to compute linear layers, improving end-to-end performance for encoder layers by 61 %. We further quantify the overheads due to tiling and data marshaling overheads, measuring an end-to-end overhead of only 9 %.

C. Compiler Evaluation

To study the scalability of our code generation approach, we break down the SLM network, measuring compiler metrics for varying network depth. For each configuration, we profile code size, constant data size, and input- and output buffer size for a single inference step in *NPU with NMS Deployment*, namely the 128th autoregressive

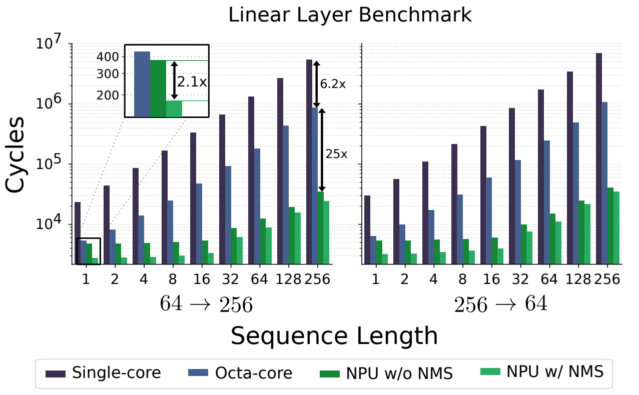


Fig. 6. Performance results for linear layer operators offloaded on *N-Eureka* using Deploy code generation. The highlighted inset shows that the NMS' added storage and bandwidth leads to performance gains of up to 2.1 × in memory-bound operator configurations. In large linear layer configurations, the speedup achieved by the NPU is 25 × compared to the octa-core implementation, and another 1.6 × when using the NMS for weights.

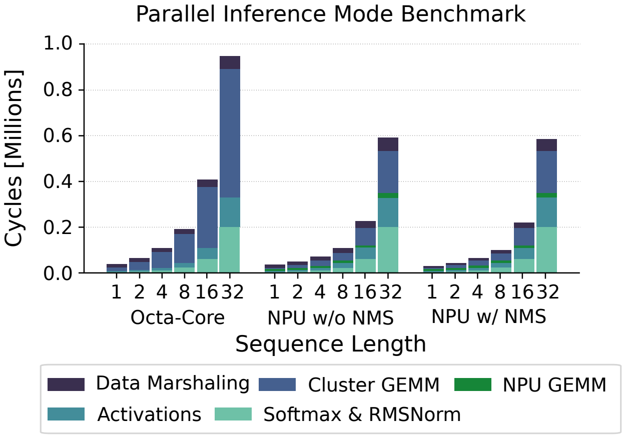


Fig. 7. Cycle breakdown of parallel inference in the studied SLM. Due to the larger contribution of operations from matrix multiplications using the NMS performance of offloaded GEMM operators increases by 17.8 ×, and end-end-performance improves by 61 % for sequence length 32 while maintaining low overheads of only 9 %, even when fully leveraging both the cluster and NPU.

TABLE II
CUMULATIVE LATENCY AND ENERGY FOR A 256-STEP INFERENCE OF THE SLM ON SIRACUSA USING THE NPU WITH NMS

	Parallel Inference	Autoregressive Inference	Speedup & Energy Reduction Ratio
Cumulative Latency [s]	17.6	0.75	23 × faster
Cumulative Energy [mJ]	3193	125	26 × more efficient

inference step. We generate all code with Deeploy and compile the resulting C Code using clang-15. Our results are shown in Table I.

We notice that while code size grows proportionally to the number of operators in the workload, the total size of the binary is dominated by weight storage in the NMS. We further see that while compilation time grows superlinearly with the number of operators in the network, the maximum compilation time of 28 s does not pose a bottleneck for practical purposes.

D. *End-to-end Deployment Results*

We thoroughly evaluate the SLM deployed on Siracusa by benchmarking the two operating phases required to execute SLM, namely the *prompting phase* and the *generation phase*.

Table II displays the cumulative runtime and energy for executing a 256-step inference in parallel mode and in autoregressive mode, where KV caching is used. The autoregressive mode outperforms the parallel mode, achieving a 23 × speedup and a 26 × improvement in energy efficiency. These improvements directly result from avoiding the costly recomputation of KV matrices. Averaging the autoregressive inference mode's cumulative latency and energy over 256 steps, we achieve an average throughput of 340 token/s at an average energy cost of 490 µJ/token.

Since the autoregressive mode maximizes the data reuse across the whole inference process, this mode can be considered both during the *prompting* and *generation* phases detailed in Section IV. However, this strategy leads to sub-optimal results as running in parallel mode for the *prompting* phase enables better utilization of the NPU without excessive recomputation of KV matrices, as tokens are not fed back in this phase.

The parallel inference mode's performance for the SLMs studied in this work follows the trend of the benchmark shown in Figure 7. While we benchmark the end-to-end performance of decoder-only models in this work, the results in Figure 7 also apply to encoder-based transformer models, as the parallel inference mode is equivalent to encoder layer execution in such networks. In autoregressive mode the speedup achieved by employing the NPU is only 19 %, which can be attributed to the mode's smaller operator sizing, leading to stalling of the accelerator due to reconfiguration overheads. Additionally, the average proportion of time spent for data marshaling is 40 % for the autoregressive versus just 14 % for the parallel modes, underlining the memory access intensity inherent to KV caching, which drastically reduces the number of computations leading to reduced arithmetic intensity. A detailed analysis of runtime and breakdown of operator intensity for end-to-end autoregressive inference is shown in Figure 8 plots ① and ②.

E. *Deployment Overheads*

An important metric for the quality of generated code is the utilization of the system's compute engines. To profile the quality of our code, we measured the overheads incurred by Deeploy for each autoregressive inference step in *NPU without NMS Deployment* and *NPU with NMS Deployment*, shown in Figure 8, plot ③. The main difference between the two scenarios is whether Siracusa's NMS is used for compile-time constant GEMM weights. While the reduction in overheads decreases from 33 % to 7 % with increasing sequence lengths and arithmetic intensity, the weight memory drastically reduces the relative time spent on data movement in the first steps of inference. This reduction of overheads is a crucial advantage of the bottom-up compilation approach employed by Deeploy; while other compilers might not consider low-level architectural features like memory hierarchy or only expose a simplified model, Deeploy allows complete control over memory allocation and code generation to leverage knowledge of the target architecture fully.

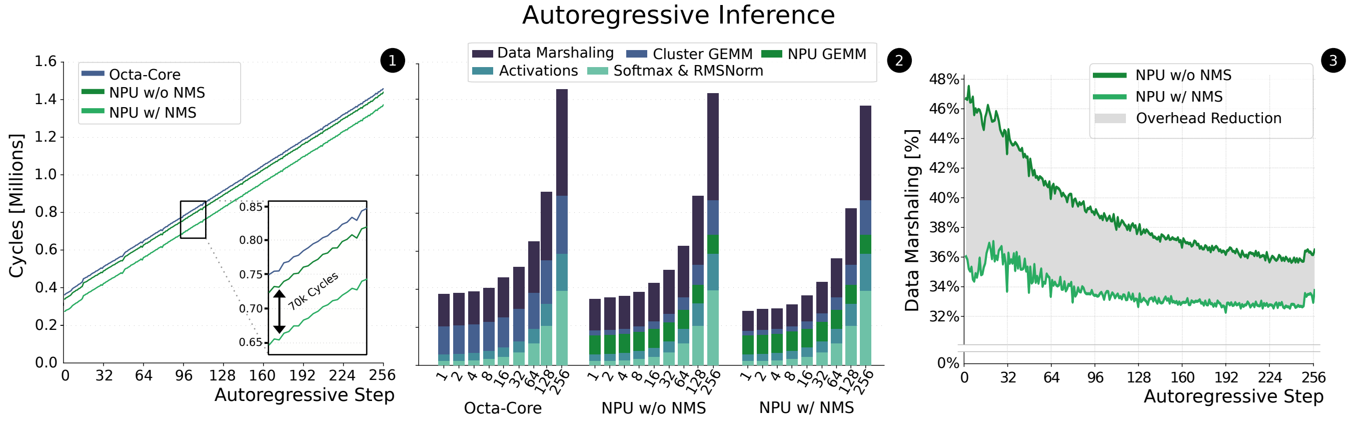


Fig. 8. Performance results of end-to-end autoregressive inference. Plot ① shows the runtime of each autoregressive inference step in three scenarios corresponding to *Octa-Core Deployment*, *NPU without NMS Deployment*, and *NPU with NMS Deployment*. The plot shows that autoregressive inference on Siracusa is highly memory-bound in all scenarios, which is due to the transfer of KV caches between L2 and L1; *NPU with NMS Deployment* reduces the runtime of every step by approximately 70 kcycles since weights are stored untiled in the NMS, reducing the required L2 - L1 data transfers. The second plot ② shows a breakdown of the runtime in the different operators of the network and data marshaling overheads. Evidently, the higher compute throughput of *N-Eureka* is unused due to the overall memory-boundedness. Finally, plot ③ shows that the data movement overhead reduction afforded by the NMS decreases with increasing sequence lengths as the overhead of transferring KV caches increases.

F. Comparison with tinyML Compilers

While we designed Deeploy to deploy state-of-the-art and emerging SLMs, we also report results on more classical CNN and Artifical Neural Network (ANN) workloads, as defined in the MLPerf tiny benchmark [41]. We compare Deeploy with the state-of-the-art open-source Dory tool [28] using the same open-source CNN kernels for PULP MCUs [33] we used in this work. To ensure a fair, compiler-focused comparison, we do not use the NMS or the NPUs of Siracusa. In this mode, both compilers only deploy cluster kernels with equivalent memory constraints. As a third data point, we add measurements of Deeploy-generated code on Siracusa when using the NMS and NPU. Our results are shown in Table III. We find that Deeploy generates code with an equivalent latency of Dory up to 1 % of variation, underlining that even though Deeploy chooses a more general compilation approach than Dory, it does not incur any performance penalties.

G. Comparison with the State-of-the-art

Currently, most efforts on EFM deployment target models with more than a billion parameters on high-end Microprocessors (MPUs) and embedded processors such as the I.MX95 or NVIDIA Orin or mobile phone chips, featuring multi-GiB external memories and multi-W power envelopes [42], [43]. Even though our performance and efficiency are extremely competitive, quantitative comparisons against these deployments would be unfair in our favor as we target much smaller SLMs.

Considering SLMs in the 100s million parameters range, we compare our implementation on Siracusa with another small-scale Llama model for edge devices, MobileLLM, by Liu et al. [44]. Liu et al. deploy a 125 MParameter SLM on an iPhone 13 featuring an A15 Bionic chip in 5 nm technology using the highly optimized Metal Performance Shaders (MPS) backend for Apple devices, achieving a throughput of 64 Token/s. While their paper does not profile the exact energy consumption of their models during inference, Liu et al. optimistically estimate the energy consumption of their setup with 12.5 mJ per token. Compared to this estimate on the iPhone 13's A15 processor, the implementation of our SLM on the Siracusa microcontroller uses 26 × less energy per token while achieving 5 × more throughput, for a total 130 × higher energy efficiency. When normalizing throughput with the number of operations per token of their network, we find that they achieve an equivalent of 4800 TinyStories Llama tokens per second. Under this estimate, our end-to-end energy efficiency on Siracusa implemented in an older 16 nm TSMC technology node is 1.7× higher.

TABLE III
LATENCY RESULTS OF DORY AND DEEPLOY ON THE MLPERF TINY BENCHMARK, RUNNING ON SIRACUSA AT A CLOCK FREQUENCY OF 360 MHz.

Benchmark	Siracusa w/o NPU Dory	Siracusa w/o NPU Deeploy	Siracusa w/ NPU Deeploy
DS-CNN	1.4 ms	1.4 ms	0.39 ms
MobileNetv1	5.6 ms	5.6 ms	0.69 ms
ResNet	3.7 ms	3.7 ms	0.60 ms
ToyAdmos	0.24 ms	0.24 ms	0.11 ms

A comparison with a similar-scale (10s million parameters) model as ours is possible against the *llama2.c* [45] implementation of the TinyStories-15M model on a Samsung Galaxy Watch 4, demonstrated to achieve 22.1 Token/s [46] using an Exynos W920 dual-core ARM Cortex-A55 processor [47]. Neglecting the power consumption of Dynamic Random Access Memory (DRAM) accesses, only considering a power consumption of 300 mW per core in Samsung 5 nm technology [48], we estimate the power consumption during inference as 600 mW. Under this assumption, the Galaxy Watch 4 achieves an energy efficiency of 27 mJ per token, 55 × lower than ours. Normalizing for operations per token, our energy efficiency is 13.4 × greater, even though the Exynos W920 is implemented in an advanced Samsung 5 nm technology node.

VII. CONCLUSION

In this work, we presented Deeploy, a novel compiler for DNNs allowing broad customizability of deployment flows. We presented the integration of Siracusa, a heterogeneous RISC-V SoC featuring an octa-core compute cluster and an NPU. We demonstrate the deployment of a SLM trained on the TinyStories dataset on Siracusa, achieving a state-of-the-art throughput of 340 Token/s at an average energy cost of 490 µJ per token in autoregressive inference mode by efficiently leveraging on-chip KV caching.

We further analyzed the efficiency of our generated code via microbenchmarks, achieving data marshaling overheads of only 9 % on Transformer encoder layers, even when fully utilizing both cluster cores and NPU collaboratively.

Lastly, we demonstrated that while data marshaling overheads are significant in the autoregressive inference mode, the energy savings compared to executing the generation phase of SLM in parallel mode outweigh this drawback, reducing the energy cost per token by 26 × while increasing throughput by 23 ×.

In future work, we plan to leverage Deeploy's flexibility to support emerging computer architecture innovations, such as multi-accelerator SoCs integrating Compute-In Memory (CIM) macros.

REFERENCES

[1] H. Touvron *et al.*, "Llama 2: Open Foundation and Fine-Tuned Chat Models," *arXiv*, no. arXiv:2307.09288, Jul. 2023.

[2] A. Q. Jiang *et al.*, "Mixtral of Experts," *arXiv*, no. arXiv:2401.04088, Jan. 2024.

[3] G. Team *et al.*, "Gemini: A Family of Highly Capable Multimodal Models," *arXiv*, no. arXiv:2312.11805, Dec. 2023.

[4] Y. Chen *et al.*, "EEGFormer: Towards Transferable and Interpretable Large-Scale EEG Foundation Model," *arXiv*, no. arXiv:2401.10278, Jan. 2024.

[5] C. Wang *et al.*, "BrainBERT: Self-supervised representation learning for intracranial recordings," in *The Eleventh International Conference on Learning Representations*, Sep. 2022.

[6] M. Ahn *et al.*, "Do As I Can, Not As I Say: Grounding Language in Robotic Affordances," in *Conference on Robot Learning*, Apr. 2022.

[7] D. Driess *et al.*, "PaLM-E: An Embodied Multimodal Language Model," in *Proceedings of the 40th International Conference on Machine Learning*, ser. ICML'23, vol. 202. JMLR.org, Jul. 2023, pp. 8469–8488.

[8] R. Eldan *et al.*, "TinyStories: How Small Can Language Models Be and Still Speak Coherent English?" *arXiv*, no. arXiv:2305.07759, May 2023.

[9] P. Zhang *et al.*, "TinyLlama: An Open-Source Small Language Model," *arXiv*, no. arXiv:2401.02385, Jan. 2024.

[10] Y. LeCun, "Deep Learning Hardware: Past, Present, and Future," in *2019 IEEE International Solid-State Circuits Conference - (ISSCC)*, Feb. 2019, pp. 12–19.

[11] J. Choquette, "NVIDIA Hopper H100 GPU: Scaling Performance," *IEEE Micro*, vol. 43, no. 3, pp. 9–17, May 2023.

[12] N. Jouppi *et al.*, "TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, Jun. 2023, pp. 1–14.

[13] A. S. Prasad *et al.*, "Siracusa: A 16 nm Heterogenous RISC-V SoC for Extended Reality with At-MRAM Neural Engine," *arXiv*, no. arXiv:2312.14750, Dec. 2023.

[14] K. Ueyoshi *et al.*, "DIANA: An End-to-End Energy-Efficient Digital and ANAlog Hybrid Neural Network SoC," in *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 65, Feb. 2022, pp. 1–3.

[15] C. Lattner *et al.*, "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb. 2021, pp. 2–14.

[16] T. Chen *et al.*, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, Oct. 2018, pp. 579–594.

[17] S. Han *et al.*, "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, Conference Track Proceedings*, Y. Bengio *et al.*, Eds., 2016.

[18] R. Firoozi *et al.*, "Foundation Models in Robotics: Applications, Challenges, and the Future," *arXiv*, no. arXiv:2312.07843, Dec. 2023.

[19] N. Tekin *et al.*, "A Review of On-device Machine Learning for IoT: An Energy Perspective," *Ad Hoc Networks*, vol. 153, p. 103348, Feb. 2024.

[20] A. Gholami *et al.*, "A Survey of Quantization Methods for Efficient Neural Network Inference," in *Low-Power Computer Vision*. Chapman and Hall/CRC, 2022.

[21] S. Kim *et al.*, "I-BERT: Integer-only BERT Quantization," in *Proceedings of the 38th International Conference on Machine Learning*. PMLR, Jul. 2021, pp. 5506–5518.

[22] S. R. Jain *et al.*, "Trained Quantization Thresholds for Accurate and Efficient Fixed-Point Inference of Deep Neural Networks," in *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, I. S. Dhillon *et al.*, Eds. mlsys.org, 2020.

[23] Y. Li *et al.*, "BRECQ: Pushing the Limit of Post-Training Quantization by Block Reconstruction," in *International Conference on Learning Representations*, Oct. 2020.

[24] G. Xiao *et al.*, "SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models," in *Proceedings of the 40th International Conference on Machine Learning*. PMLR, Jul. 2023, pp. 38 087–38 099.

[25] A. Howard *et al.*, "Searching for MobileNetV3," in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE Computer Society, Oct. 2019, pp. 1314–1324.

[26] J. Lin *et al.*, "Memory-efficient Patch-based Inference for Tiny Deep Learning," in *Advances in Neural Information Processing Systems*, vol. 34. Curran Associates, Inc., 2021, pp. 2346–2358.

[27] E. Flamand *et al.*, "GAP-8: A RISC-V SoC for AI at the Edge of the IoT," in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Jul. 2018, pp. 1–4.

[28] A. Burrello *et al.*, "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, Aug. 2021.

[29] Q. Huang *et al.*, "CoSA: Scheduling by Constrained Optimization forSpatial Accelerators," in *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ser. ISCA '21. Virtual Event, Spain: IEEE Press, Nov. 2021, pp. 554–566.

[30] M. Maas *et al.*, "TelaMalloc: Efficient On-Chip Memory Allocation for Production Machine Learning Accelerators," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, Dec. 2022, pp. 123–137.

[31] M. D. Moffitt, "MiniMalloc: A Lightweight Memory Allocator for Hardware-Accelerated Machine Learning," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, ser. ASPLOS '23. New York, NY, USA: Association for Computing Machinery, Feb. 2024, pp. 238–252.

[32] R. David *et al.*, "TensorFlow Lite Micro: Embedded Machine Learning for TinyML Systems," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 800–811, Mar. 2021.

[33] A. Garofalo *et al.*, "XpulpNN: Accelerating Quantized Neural Networks on RISC-V Processors Through ISA Extensions," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Mar. 2020, pp. 186–191.

[34] L. Lai *et al.*, "CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs," *arXiv*, no. arXiv:1801.06601, Jan. 2018.

[35] M. Spallanzani *et al.*, "QuantLab: A Modular Framework for Training and Deploying Mixed-Precision NNs," TinyML Summit, 2022.

[36] F. Angiolini *et al.*, "An Efficient Profile-based Algorithm for Scratchpad Memory Partitioning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 11, pp. 1660–1676, Nov. 2005.

[37] A. Tumanov *et al.*, "TetriSched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: Association for Computing Machinery, Apr. 2016, pp. 1–16.

[38] "Mako Templates for Python," https://github.com/sqlalchemy/mako/releases.

[39] A. Vaswani *et al.*, "Attention is All you Need," in *Advances in Neural Information Processing Systems*, vol. 30. Curran Associates, Inc., 2017.

[40] B. Zhang *et al.*, "Root Mean Square Layer Normalization," in *Advances in Neural Information Processing Systems*, vol. 32. Curran Associates, Inc., 2019.

[41] C. Banbury *et al.*, "MLPerf Tiny Benchmark," Aug. 2021.

[42] E. Ruedas, "LLM Pipelines: Seamless Integration on Embedded Devices," Virtual Event, Mar. 2024.

[43] X. Chu *et al.*, "MobileVLM : A Fast, Strong and Open Vision Language Assistant for Mobile Devices," *arXiv*, no. arXiv:2312.16886, Dec. 2023.

[44] Z. Liu *et al.*, "MobileLLM: Optimizing Sub-billion Parameter Language Models for On-Device Use Cases," *arXiv*, no. arXiv:2402.14905, Feb. 2024.

[45] A. Karpathy, "Llama2.c," Mar. 2024.

[46] Joey (e/λ) [@shxf0072], "@karpathy llama2.c running on galaxy watch 4 https://t.co/sMPCZM3WE4," Dec. 2023.

[47] iFixit, "Samsung Galaxy Watch4 and Watch4 Classic Teardown," https://url.zip/4befa14, Sep. 2021.

[48] A. Frumusanu, "The Snapdragon 888 vs The Exynos 2100: Cortex-X1 & 5nm - Who Does It Better?" https://www.anandtech.com/show/16463/snapdragon-888-vs-exynos-2100-galaxy-s21-ultra, Feb. 2021.