MATCH: Model-Aware TVM-Based Compilation for Heterogeneous Edge Devices

MATCH: Model-Aware TVM-based Compilation for Heterogeneous Edge Devices

Mohamed Amine Hamdi [1], Francesco Daghero [1], Giuseppe Maria Sarda[2,3], Josse Van Delm[2], Arne Symons[2], Luca Benini[4], Marian Verhelst[2,3], Daniele Jahier Pagliari[1], Alessio Burrello[1,4]
[1]*Politecnico di Torino*, Italy, [2]*KU Leuven*, Belgium, [3]*IMEC*, Belgium, [4]*University of Bologna*, Italy

***Abstract*—Streamlining the deployment of Deep Neural Networks (DNNs) on heterogeneous edge platforms, coupling within the same micro-controller unit (MCU) instruction processors and hardware accelerators for tensor computations, is becoming one of the crucial challenges of the TinyML field. The best-performing DNN compilation toolchains are usually deeply customized for a single MCU family, and porting them to a different one implies labor-intensive re-development of almost the entire compiler. On the opposite side, retargetable toolchains, such as TVM, fail to exploit the capabilities of custom accelerators, producing general but unoptimized code. To overcome this duality, we introduce MATCH, a novel TVM-based DNN deployment framework designed for easy agile retargeting across different MCU processors and accelerators, thanks to a customizable model-based hardware abstraction. We show that a general and retargetable mapping framework can compete with, and even outperform custom toolchains on diverse targets while only needing the definition of an abstract hardware cost model and a SoC-specific API. We tested MATCH on two state-of-the-art heterogeneous MCUs, GAP9 and DIANA. On the four DNN models of the MLPerf Tiny suite MATCH reduces inference latency on average by 60.87× on DIANA, compared to using the plain TVM, thanks to the exploitation of the on-board HW accelerator. Compared to HTVM, a fully customized toolchain for DIANA, we still reduce the latency by 16.94%. On GAP9, using the same benchmarks, we improve the latency by 2.15× compared to the dedicated DORY compiler, thanks to our heterogeneous DNN mapping approach that synergically exploits the DNN accelerator and the eight-cores cluster available on board.**

***Index Terms*—AI Compilers, Deep Neural Networks, Heterogeneous Computing, Deep Learning Accelerators.**

I. Introduction

The increasing focus on efficiently executing Deep Neural Networks (DNNs) on low-power MCUs has led to the proliferation of domain-specific Systems-On-Chip (SoCs) [1]–[5]. Three crucial features characterize these architectures: first, being designed for the specific task of accelerating DNNs at the extreme edge, they are usually *OS-less*, in order to minimize both the memory requirements for the software stack and the associated performance overheads. Second, they are usually *heterogeneous platforms* composed of general-purpose cores in charge of simple control tasks (e.g., receiving input data and sending DNN outputs) and a set of accelerators optimized for DNN workloads. As a last point, they commonly avoid HW caches in favor of SW scratchpad memories to reduce the required silicon area, exploiting the regularity of memory accesses in DNN workloads. Therefore, fully exploiting the hardware resources of these platforms when executing a DNN is a non-trivial task, which requires in-depth knowledge of the SoC and its memory hierarchy.

Automated deployment toolchains address these challenges by generating target-dependent code from high-level DNN representations (e.g., Python, ONNX, etc), abstracting hardware details from users [6], [7]. However, existing toolchains for OS-less edge devices are usually either *overly generic* or *overly specific*. Generic toolchains, such as the popular TVM [6] or the MLIR-based TinyIREE [8], have extensive support for diverse DNN operators, but only generate code for common CPU and GPU targets, given that they do not have access to in-depth specifics of the target hardware and, therefore, are not able to exploit SoC-specific features such as HW accelerators, ISA extensions, memory hierarchy dimensions, etc. Autotuning engines [9] can partially address this issue by iteratively generating different versions of the code, profiling it, and using search strategies such as reinforcement learning or genetic algorithms to converge to an optimized implementation. On the other hand, they result in very long compile times for large search spaces [10] and usually require the availability of hardware-in-the-loop for profiling.

On the other side of the spectrum, target-specific toolchains provide top performance on specific SoCs, fully exploiting the target's accelerators and memory hierarchy. However, their support for DNN operators is limited, and adding a new one requires an in-depth knowledge of both the HW target and the full toolchain. Moreover, they are commonly monolithic software stacks that embed hardcoded hardware-dependent heuristics for memory management, operation scheduling, and tiling [7]. Consequently, adding support for a new DNN accelerator or a new SoC requires a labor-intensive overhaul of the toolchain or, in the worst case, the creation of an almost entirely new toolchain, leading to large delivery delays of the final product.

A seminal work that combines these two worlds is HTVM [10], which extends TVM with a customizable plug-in for hardware accelerator support [7]. The extension is responsible for the generation of optimized code for operations that can be executed on the target SoC's accelerator, while TVM can be used to deploy non-accelerable kernels on the main CPU. However, HTVM still relies on a monolithic tool (DORY), in which hardware-specific cost information is intertwined with the code base, thus suffering from the above-mentioned limitations in terms of extensibility to new targets.

In this paper, we describe a step in the direction of solving this *generic toolchains customization problem* propos-

ing **MATCH**, a **M**odel-**A**ware **T**VM-based **C**ompiler for **H**eterogeneous edge devices. We focus specifically on the automatic optimization of tiling and loop ordering, two key steps to maximize throughput and energy efficiency of DNNs layers, which require strongly hardware-dependent decisions (e.g., on the optimal tile size and dimension) [7], [10]. MATCH plugs a DNN Design Space Exploration (DSE) tool in TVM, and to support a new SoC, it requires: i) a high-level HW model to let the DSE tool generate optimized layer-specific scheduling of loops and ii) SoC-specific APIs for the different accelerators, to manage and trigger DNN kernel execution. Our goal is to demonstrate that MATCH can deploy DNNs as efficiently as hardware-specific toolchains while requiring significantly fewer changes to support new hardware. In our experiments, a single compiler engineer, not involved in designing the hardware platforms, has been able to add support for a new SoC in MATCH in less than 1 week, including several refinement iterations for the cost model.

In summary, the main contributions of this work are the following:

- We propose MATCH, a new compiler that extends the TVM compilation flow with a DSE tool for DNN layer scheduling. Although the methodology is in principle orthogonal to the specific DSE engine, this work describes an implementation based on ZigZag [11], an open-source tool that identifies optimal temporal mappings, exploiting an abstraction of the workload and target hardware. To exploit HW heterogeneity, we propose a pattern-matching mechanism that exploits ZigZag's modeling to match each DNN operator with the best hardware module to execute it.
- As ZigZag only yields the temporal mapping, we enhance it with (i) an input interface to read DNN layers workloads from TVM, (ii) a set of easily-modifiable APIs to support new hardware, and (iii) a novel code generation step.
- We benchmark MATCH on two different heterogeneous MCUs: GAP9 [2], featuring an 8 RISC-V cores cluster and a flexible DNN HW accelerator, and DIANA [4], featuring two DNN accelerators working at different bitwidths. On a wide set of Convolutional Neural Network (CNN) layers, MATCH reduces the average latency by up to 119.08× on GAP9 and up to 83.18× on DIANA, compared to the plain TVM solution.
- On end-to-end DNN networks from the MLPerf Tiny benchmark [12], MATCH achieves similar performance compared to the best SoCs-specific open-source toolchains, with 2.15× and 16.94% lower average latency on GAP9 and DIANA, respectively. It also outperforms TVM on both platforms by 67.83× and 60.88×.

We open-source our code at: `https://github.com/eml-eda/match`. The rest of the manuscript is organized as follows: Sec. II and Sec. III discuss the required background and related works respectively. Sec. IV describes MATCH's core components. In Sec. V, we show how to add support for a new SoC in MATCH. Sec. VI reports the experimental results, and Sec. VII concludes the paper.

II. Background

The optimized execution of DNNs at the edge is a complex task that can benefit from both software and hardware optimizations. On the software side, different techniques, such as neural architecture search, pruning, or quantization, have been proposed to reduce DNNs' complexity while preserving accuracy [13]–[18]. On the hardware side, many solutions have been proposed for accelerating DNN workloads, resulting in a plethora of complex heterogeneous SoCs [1], [2], [4], [19]. To fill the gap between high-level DNN models and such complex hardware targets, *AI compilation* frameworks [6]–[10], [20]–[24] are used to generate target-specific code that efficiently exploits the available hardware resources.

Our effort focuses on defining a new AI compiler, MATCH, that not only fulfills its scope of porting DNNs architecture on heterogeneous edge devices but also provides a lightweight interface for compiler engineers that simultaneously allows them to achieve three seemingly contrasting goals: i) *deployment efficiency*, i.e., close-to-optimal performance, by carefully orchestrating the available HW resources; ii) *broad support* for existing and upcoming DNN operators; iii) *easy extensibility* to future hardware targets. In the rest of this section, we first give an overview of existing heterogeneous hardware platforms for DNN execution at the edge and then provide the required background on AI compiler technology.

A. Edge Platforms for DNN Inference

The execution of DNNs in typical general-purpose OS-less edge devices, such as Microcontrollers (MCUs), is not an easy task due to the requirements these devices need to deliver for an acceptable quality of service, e.g. performance, within a limited power envelope. On the other hand, DNN inference is a highly parallel workload, prevalently composed of operations such as convolutions and General Matrix Multiplications (GEMMs), opening the possibility of aggressive hardware optimization. In this direction, modern SoCs for edge DNN inference are becoming increasingly *heterogeneous*; they usually include one or more DNN accelerators, usually built as an array of multiply-and-accumulate (MAC) units to parallelize tensor processing workloads (i.e. convolutions, matrix multiplication, etc), coupled with general-purpose cores to execute less compute-intensive layers. Cores and accelerators typically exchange data through a multi-level, software-managed memory hierarchy to minimize DNNs tensors access time.

Several examples of heterogeneous MCUs are available both in academia [4], [25], [26] and industry [1], [2], [19], with embedded accelerators that vary from multi-cores Single Instruction Multiple Data (SIMD) / Single Instruction Multiple Threads (SIMT) units to systolic arrays. One notable academic example is the Parallel Ultra-Low Power (PULP) family of devices [27], which is built around a main control core that dispatches digital signal processing (DSP) tasks to a cluster of 8 identical RISC-V cores with specific ISA extensions such as SIMD MACs and memory operations with pointer post-increment. A more aggressively heterogeneous MCU, DIANA, introduced in [4], couples a RISC-V core used for IO-control

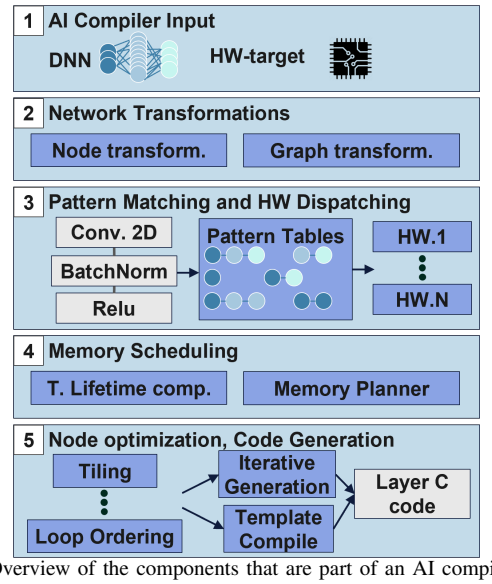


Fig. 1. Overview of the components that are part of an AI compiler.

and workload dispatching with two highly specialized DNN accelerators, a digital SIMD array, and an Analog In-Memory Computing (AIMC) one. Other works, such as [28], [29] propose hardware-software co-design pipelines where both the accelerator architecture (customized from a parametric template), as well as the software to schedule its execution from a host core are co-optimized for a given DNN target. In industry, the MAX78000 [19] by Analog Devices enhances an ARM Cortex-M4 MCU with a CNN accelerator based on a fixed systolic array structure. GAP9, by GreenWaves, is an industrial embodiment of PULP, that couples a main RISC-V controller core with a flexible DNN accelerator (Called NE16) as well as a general-purpose 8-cores cluster [2]. In this work, we target two of these heterogeneous devices to demonstrate the generality of our AI compiler, namely DIANA and GAP9. In Section V, we demonstrate the flexibility of MATCH in supporting these two strongly different HW targets, achieved simply by customizing HW models and DNN operators APIs.

B. AI compilers

The innovations brought by researchers and vendors on both new HW platforms and DNN models are often bottlenecked by the lack of a sufficiently flexible yet performant middleware layer that allows the execution of new models on newly developed HW. Recently, many works have focused on this layer of the stack, usually referred to as the *AI compiler*. In the context of an OS-less heterogeneous SoC, an AI Compiler consists of a toolchain to translate HW-unaware DNN formats, such as ONNX or PyTorch models, into HW-specific compiled code for their optimized execution into CPUs, GPUs, and dedicated accelerators. These tools are particularly important in edge computing, where HW architectures are highly heterogeneous with tight memory constraints. Fig.1 depicts the main AI compiler steps, detailed below.

1) Network Transformations: the compilation of AI models typically starts from a computational graph representing the workload, where nodes represent operations over tensors, abstracted as edges. Network transformations can be categorized into HW-aware and HW-unaware ones. HW-unaware passes include optimizations such as operator fusion. On the other hand, HW-aware transformations consider the target hardware's characteristics to optimize the computational graph further, enhancing performance and efficiency, for instance, by replacing operations with equivalent ones that are better supported by the hardware, or adapting data layouts and quantization formats.

2) Pattern Matching and HW Dispatching: after applying transformations to the network's computational graph, the AI compiler offloads the execution of each node of the optimized graph to a specific hardware module. This step maps high-level operations in the computational graph to low-level hardware-specific operations or kernels. Pattern matching is applied on the graph to identify nodes that are supported by each HW module, to fully leverage SoC-specific capabilities, such as specialized instructions or acceleration units.

3) Memory Scheduling: the last graph-level pass is usually memory scheduling, whose goal is minimizing the peak memory occupation for activations based on tensor lifetimes. Effective memory scheduling ensures that data is efficiently allocated and deallocated, reducing memory overhead and making the model execution feasible on devices with limited volatile memory space.

4) Operator-Specific Optimizations: Finally, operator-specific optimizations are applied to each subgraph matched in step 2, to enhance the performance of individual operations onto the hardware module they have been assigned to. One crucial optimization technique is *loop tiling*, which partitions activation and weights tensors into smaller, more manageable blocks or tiles. This enables more efficient data access patterns, particularly by maximizing data reuse within faster memory levels. Ideally, combining memory tiling with *loop ordering* can enable reaching a performance similar to the one obtained with a theoretically infinite last-level cache. We will call the specific combination of loop tiling and ordering for a layer the *layer schedule*.

After applying all steps, the final code is generated; here, either a template-based approach [7], [9], [10], [21], [22] or a lowering code-generation [6], [8], [23] pipeline can be used. Depending on the strategy, the kernel's code (i.e., the innermost convolution/GEMM operation, after the application of fusion, tiling, loop ordering, etc.) is imported from HW-specific hand-tuned libraries with optimized performance [7] or produced together with the full layer code through auto-tuning pipelines with HW-in-the-loop [9].

III. Related Works

Table I summarizes the main state-of-the-art AI compilers from academia and industry chronologically. We categorize them into three groups based on the following criteria: i) the ease of extending them to a new HW target, ii) the ease of adding a new DNN operator, and iii) the performance achieved.

A first group of toolchains is designed to be highly extensible but often at the cost of generating unoptimized HW-unaware code. For example, TensorFlow Lite for Microcontrollers [23] is a popular framework that allows the conversion

TABLE I
STATE-OF-THE-ART IN AI COMPILATION FOR EDGE DEVICES.

Name	Open Source	Supported Targets	Cost Model	Search Algorithm	Platform Extension	Operator Extension	Perf. Achieved
TVM,2018 [6]	✓	CPU, GPU, MCU	ML-Based	Simulated Annealing			
Ansor,2020 [9]	✓	CPU, GPU, MCU	HW-in-the-loop	Reinforcement Learning			
TFLite,2020 [23]	✓	MCUs	None	None			
DORY,2020 [7]	✓	MCU, Accelerators	Coarse-grain Analytical	Linear Constraint Progr			
NNTool,2020 [21]	✗	GAP8, GAP9	Fine-grain Analytical	Rules-driven			
ROLLER,2022 [20]	✓	CPU, GPU	HW-based LUT	Model-driven			
CubeAI,2022 [22]	✗	STM32 MCUs	None	None			
TinyIREE,2022 [8]	✓	CPU, GPU, MCU	None	None			
HTVM,2023 [10]	✓	MCU, Accelerators	Coarse-grain Analytical	Linear Constraint Progr			
MATCH, Ours	✓	Heterogeneous MCUs	Fine-grain Analytical	Genetic-based			

of TensorFlow models into optimized C++ kernels paired with a minimal runtime. To improve performance, TFLite allows the plugging of hand-written backends. However, it lacks advanced optimization passes, such as loop ordering and tiling, which are essential to fully exploit hardware capabilities. Two of the most popular AI compilers that support edge devices are Apache TVM [6] and the Multi-level intermediate representation (MLIR)-based Tiny Intermediate Representation Execution Environment (TinyIREE) [8]. The first generates bare-metal code for several general-purpose targets, including CPUs, GPUs, and MCUs. Adding new targets or new SW operators is straightforward, given that operators are expressed at a high level, using their algebraic function, without considering HW-specific features or possible optimizations such as ISA extension exploitation, tensor reuse, or data stationarity. Similarly, TinyIREE proposes a generic lowering toolchain that exploits the MLIR paradigm. While offering a flexible and modular progressive lowering infrastructure, it does not natively embed passes for tiling, loop reordering, and other HW-dependent optimizations.

To cope with the lack of specialization of these compilers, HW vendors designed tools that exploit HW-specific information to improve utilization and, therefore, performance. CubeAI [22] focuses on optimizing computation and memory usage for STM32 MCUs, allocating tensors in different DRAM regions, and exploiting different CMISIS-NN-based [30] kernels for different layer geometries. However, both the kernels and the code generation are tailored exclusively for this family of MCUs, not allowing them to be extended to other HW targets. NNTool, the Greenwaves deployment framework for GAP8 [1] and its successor, GAP9, outperforms all the competitors in terms of energy consumption on the four MLPerf Tiny benchmarks. DORY [7] reaches comparable performance on the same HW with an open-source tool based on the PULP-NN kernel library [27]. Both tools exploit knowledge about the memory hierarchy to optimally tile the layer and orchestrate the kernel execution partitioning each tensor appropriately and moving it between the main memory and a SW-managed last-level cache. DORY formalizes tensor tiling as a constraint programming problem, embedding target-specific heuristics in its code. All these tools share common difficulties in supporting upcoming DNN layers and topologies, given that they are constructed monolithically, with corner cases on specific layer geometries, network topologies, or individual operators hard-wired within multiple compiler passes.

Recognizing the need for a balance between extensibility and performance, some tools have begun to address this dichotomy. Ansor [9] exploits TVM and an auto-tuning mechanism to optimize DNN layer scheduling but requires extensive exploration of compilation parameters (e.g., tiling factors), with HW-in-the-loop, making it time-consuming and often impractical for custom DNN hardware accelerators. Specifically, each operator is iteratively compiled with different sets of parameters and benchmarked on the hardware to find the optimal configuration [31]. While this auto-tuning mechanism is effective, executing all possible scheduling and tiling alternatives requires exploring thousands of variants for each kernel. Running them on the target is time-consuming. Therefore, the exploration space is often pruned, leading to sub-optimal final solutions. Furthermore, custom DNN hardware accelerators are currently not natively supported. Another example is ROLLER [20], which employs a deterministic algorithm for optimal loop order and splitting search. The best deployment configuration is found through a Look-Up Table (LUT)-based approach, that is, an offline profiling of microkernels with pre-defined layer geometries is used to construct a LUT of layer performances. However, its extensibility is limited, given that the LUT should contain every possible layer hyperparameter alternative for HW targets characterized by a non-linear dependency between tensor geometry and latency (i.e., most real-world devices), leading to unfeasible compilation times.

The most promising approach is currently HTVM [10], a framework that integrates DORY into TVM, aiming to combine the best of both worlds by offloading DNN layers that can be accelerated to DORY. The tool exploits the Bring Your Own Codegen (BYOC) infrastructure [32] of TVM, which allows offloading pre-defined patterns of the network's computational graph to external user-defined routines optimized for specific targets such as accelerators. While this approach improves flexibility maintaining very high performance, it retains the limitations of DORY regarding extensibility to new hardware.

Given the limitations of existing tools, there is a clear need for an AI compiler that combines ease of extensibility

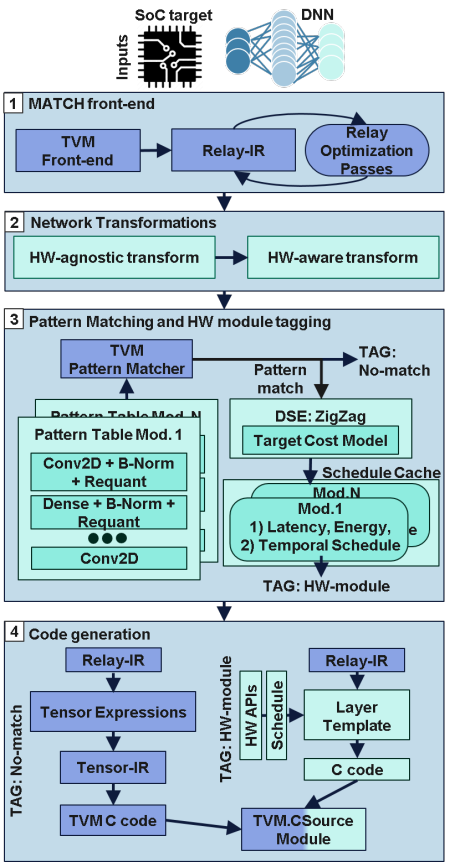


Fig. 2. MATCH flow. TVM default components are colored in dark blue.

with high performance. MATCH aims to fulfill this need by providing a framework that is: i) flexible and easily extensible to both new HW targets, by only providing limited HW-specific information, and to new unsupported operators, by exploiting TVM general routines; ii) capable of generating optimized code for diverse hardware targets, maximizing the utilization of the on-board compute units and accelerators.

IV. MATCH

To address the dualism between flexible-yet-inefficient and optimized-yet-inflexible toolchains, we introduce MATCH, a novel framework that enhances TVM with HW-aware deployment capabilities using the BYOC framework. Starting from a Python-level DNN model, MATCH generates optimized HW-specific C code to deploy the DNN on OS-less heterogeneous devices. MATCH exploits TVM-specific features, i.e., the BYOC interface and the Patter Matcher, to be able to simultaneously deploy optimized operators with external user codebases, while falling back to a default unoptimized solution for un-matched operators. MATCH is written in Python $>$3.8.

Fig. 2 shows a high-level overview of the deployment flow using MATCH, which is composed of four main stages. In the rest of this section, we describe the Framework Frontend and Network Transformations in Sec. IV-A, the Pattern Matching and HW-aware Dispatching in Sec IV-B, and the Code Generation in Sec IV-C. In Sec. V, we further detail how to add a new HW target to MATCH, with two specific examples: DIANA and GAP9.

MATCH currently targets Convolutional Neural Networks (CNNs) since these models are very popular in extreme-edge applications. On the other hand, its extension to emerging AI operators, such as the attention layers of transformers, is facilitated by our approach's modularity, which exploits either the broad support of TVM as a fallback, or the BYOC Pattern Matcher combined with template-based code generation to quickly produce platform-specific code for a new operator, as detailed in the rest of this section. Hereinafter, we use the following notation for the hyperparameters of a 2D Convolutional layer: IX/IY/C for the horizontal, vertical, and channel dimensions of the input tensor; OX/OY/K for the respective output dimensions; FX/FY for the spatial dimensions of the convolutional weights kernel.

A. Framework Front-End and Network Transformations

MATCH receives two main inputs: the NN model and the HW target definition. The target definition encompasses a list of **hardware execution modules**, i.e., units that can be used to execute one or more DNN operators. Each hardware execution module contains the memory description (i.e., hierarchy, dimensions, and connections), a list of offloadable patterns, the set of specific APIs used, and a cost model for each operator. In Sec. V, we detail all the information needed to support a new target.

The first step executed by the compiler is the NN model reading: we rely on the TVM frontend, which accepts a wide range of formats, from quantized Keras models to ONNX. The input format is translated to Relay, the intermediate representation used internally by TVM to describe and manipulate the model graph.

After this first step, standard HW-agnostic transformations are applied to the network nodes using the TVM *ExprVisitor*, such as dead node elimination and constants folding. Starting from the intermediate representation generated after the HW-agnostic transformations, the graph is further manipulated by **HW-aware** transformations, i.e., ones that consider HW-specific features such as the number of processing elements (PE) or the presence of specific HW components. HW-aware transformations can belong to two families: the first one encompasses patterns rewriting, which exploits the TVM *DFPatternCallback* to match a pattern in the graph and transform it into a new one. The second family, similarly to the HW-agnostic passes, exploits the TVM *ExprVisitor* operator to traverse the full graph and apply transformations to individual nodes, such as adding parameters, changing data type or layout, to allow the mapping of operators to specific accelerators.

As an example, we reported in Tab. II the transformations applied when deploying a network to the GAP9 platform: first, constants are folded, and dead nodes are eliminated; then, integerization is applied to transform all tensors to int8 data type. A layout transformation is applied to store all activation tensors in NHWC data format, as required by the backend kernel libraries that we adopt for this target, namely PULP-NN [27] for the RISC-V cluster, and a custom library

TABLE II
HW-AGNOSTIC AND HW-AWARE GRAPH TRANSFORMATIONS APPLIED IN MATCH WHEN DEPLOYING A NETWORK TO GAP9.

Network Transformations	Description	Type
HW-agnostic		
Dead Node Removal and Constant Folding	Merging of contiguous dead/constant nodes	Graph Visitor
Integerization	Quantize operations and weights	Graph Visitor
Layout transformation	Switch between NCHW/NHWC	Graph Visitor
HW-aware		
Requant sequence	Transform division into a right shift operation	Pattern Matching
Padding and slicing	Padding and slicing before/after matched sequence	Graph Visitor
Weights Transformation	Storage of weights in custom data layout	Graph Visitor

for the NE16 accelerator[1]. Then, each mul-add-div sequence, used for re-quantizing a layer's output, is transformed into a re-quantization node implementing the arithmetic function $f(x) = (x * M + B) >> S$, removing the div operator, which is not supported by the NE16 and slow on the cluster. For the NE16 accelerator, padding is further applied to match the input tensor dimensions to the accelerator's spatial parallelism, and weights are re-ordered to match the custom data layout needed. Note that the HW-agnostic transformations can be selected from a large pool of pre-defined passes, while each different HW clearly has its own set of HW-aware passes.

B. Pattern Matching and Accelerator-aware Dispatching

After applying the network graph transformation pipeline, MATCH assigns the execution of each part of the DNN to a selected HW module through its Pattern Matcher module. The Pattern Matcher analyzes the intermediate NN graph and tags groups of nodes (patterns) with reference to the HW module that will execute them. More specifically, MATCH can associate to each pattern either an HW module or a *not-matched* tag; in the latter case, the TVM default code generation pipeline is invoked, and the layer will be executed on the main CPU of the SoC. In the extreme case in which neither any HW module nor TVM supports the generation of code for a specific operator, compilation fails with an error. To solve it, the support for such operator should be added first in TVM's Relay IR, thus allowing its addition in the Pattern Tables of HW modules that support it.

If an HW module supporting the layer is found, MATCH exploits a DSE tool to generate an optimized schedule for it. MATCH builds on top of the TVM's internal pattern matching system to facilitate the definition of new SoC-specific patterns. Each HW module definition includes a Pattern Table that lists all patterns that can be offloaded to it. In detail, each pattern comprises the set of nodes to be matched, the replacement (one or more Relay nodes), and an additional Pattern Constraint that defines a set of rules that must be fulfilled for the match to be valid. These rules can verify if the input/output layouts, the quantization formats, the layer hyper-parameters (e.g., the convolutional kernel dimensions), etc, in the candidate graph pattern are compatible with those supported by the HW module.

Note that compared to other State-of-the-Art (SoA) AI compilers, such as DORY or HTVM, which *separately* support accelerators and MCUs but do not allow the orchestration of multiple HW modules on the same SoC, MATCH implements an iterative exploration algorithm to offload patterns to the HW execution unit with the minimum expected latency/energy among those supporting a given pattern, thus enabling the combined usage of multiple HW modules *on the same SoC*. To do so, MATCH sequentially explores the Pattern Tables of each individual HW module and invokes the DSE tool for each positive match. For each combination of (pattern, node parameters, HW module), the DSE tool provides a twofold output: the best schedule and its predicted latency/energy. If multiple HW modules match a pattern, MATCH selects the one that minimizes one of these metrics and tags the pattern accordingly. To cope with patterns contained within others (e.g., Conv-only versus Conv + Batch Norm + ReLU), we heuristically select the largest one, assuming that node fusion is always convenient to reduce latency and energy.

Currently, MATCH only supports the usage of the different HW modules alternatively: each layer is assigned to a single module, and different layers are executed sequentially, activating one module at a time. We plan to add support for concurrent execution on multiple HW modules in our future work.

1) Model-based DSE Engine: As a DSE tool, MATCH employs the optimizer included in ZigZag [11], which can search both for the optimal *spatial mapping* of a DNN layer onto a PE array (optimizing the number of PEs and their parallelism axes) and for the optimal *temporal mapping* or *schedule* (i.e., operator's loop ordering and loop tiling). Currently, in MATCH, we focus only on temporal mapping search. Spatial mapping is fixed for each targeted HW module, since we consider already manufactured accelerators, either with a fixed dataflow, or with pre-existing backend kernels using a defined spatial parallelization strategy. To fix the spatial mapping, we freeze the corresponding innermost tile sizes to maximally utilize HW resources, and let our tool optimize the remaining loops, executed over time: to do so, we explore the search space using the LOMA [33] engine. The latter generates all valid and non-equivalent scheduling candidates using the Loop Prime Factors approach [33]. It then allocates each loop's operands to the lowest non-full memory level. An analytical hardware performance cost model guides the selection of the optimal schedule among the generated ones. Importantly, LOMA supports *uneven* mapping, i.e., with different tensors tiled in different memory levels. We extend ZigZag and LOMA to take into account double-buffering. Further, we extend them to support a more general accelerator model, including systolic arrays and more generic multi-core accelerators.

Notice that to support a new hardware target, users of MATCH do not need to modify the DSE tool's internals; they

[1]In the NHWC layout for activation tensors, two nearby memory locations store pixels relative to the same spatial position, but successive channels (C), as opposed to NCHW, the default layout in PyTorch, in which two contiguous cells store pixels relative to the same channel and two neighboring spatial locations along the horizontal dimension (W) [27], [30].

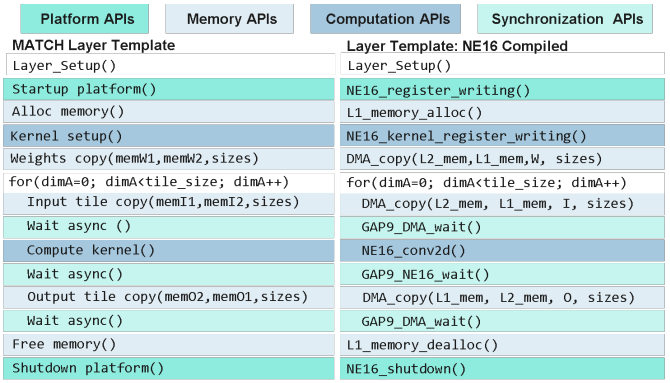


Fig. 3. Overview of the layer template and conversion to target-specific code.

just need to provide the corresponding operators' cost models. A detailed example of how these analytical cost models can be easily and quickly defined is provided in Sec. V.

C. Code Generation

MATCH includes two code generation "branches". For unmatched graph patterns, the *fall-back codegen branch* is taken, and MATCH generates C code exploiting one of TVM's default targets. The NN node in Relay IR is lowered to Tensor Expressions and then into TIR, a lower-level IR that allows representing information such as loop unrolling and layer tiling. Then, TVM produces generic C code for predefined targets such as x86/ARM/RISC-V CPUs. Note that during this lowering step, TVM is unaware of any specific ISA instructions or dedicated HW blocks.

On the other hand, if a pattern has been tagged with a specific HW module, the *specialized codegen branch* is executed. To generate the optimized C code, we provide a generic *Layer Template*, which is compiled with the i) pattern hyperparameters, ii) the spatial/temporal scheduling provided by the DSE, and iii) platform-specific APIs. We use the Python Mako template library for this step.

To generate the network's main file, MATCH uses the default TVM generator that receives the layer function prototypes and the tensor dimensions to allocate the memory. We used TVM's default hill-climbing algorithm for memory allocation.

1) Layer Template Compilation: First, default patterns hyperparameters such as layer geometry, padding, or stride are added to the template. Then, the cached DSE spatio-temporal scheduling is used to define the layer loops order, the loop splitting, and the tensor tiles dimensions. The memory transfers are also correctly placed to minimize the latency overhead, following the DSE scheduling and supporting both single- and double-buffering, depending on the provided DSE output. Finally, the pattern's computation kernel (i.e., a wrapper to the target's backend library) is inserted in the innermost loop, where all the input data have already been transferred to the innermost memory level. In the code generated using the template, all function calls are still platform-agnostic: for instance, a memory-copy does not call the SoC-specific DMA function, but a generic `MATCH_memory_copy`.

In the last code generation step, MATCH embeds HW-specific user-defined APIs in the template. These APIs are

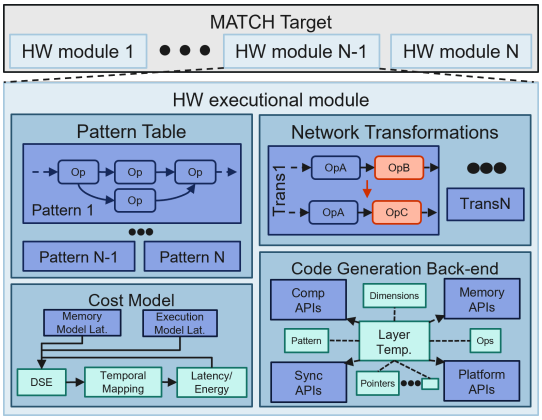


Fig. 4. Example of a generic MatchTarget, detailing how an HW execution module is composed.

defined in the HW model for each target, by specifying their name in an API object. We categorized APIs in 4 families:

- **Platform APIs**: they are used to allocate, deallocate, or configure the corresponding HW modules. For instance, for an HW accelerator, these APIs could be used to write on memory-mapped registers in order to configure the module for executing a specific operation.
- **Memory APIs**: these APIs facilitate memory management by enabling the allocation and deallocation of different memory levels, handling data transfers between them, for instance utilizing DMA calls for efficiency, and computing pointer offsets when employing specific data layouts.
- **Synchronization APIs**: MATCH allows for asynchronous and synchronous data management. Therefore, it exposes APIs to synchronize after memory transfers, kernel computation, or both. Thanks to these APIs, we provide the possibility to perform both single- and double-buffered memory transfers.
- **Computational APIs**: these are the backend library calls mentioned above. They receive the whole pattern context to set up and execute the correct computational kernel with its corresponding hyperparameters.

Fig. 3 reports the layer template on the left and the corresponding compilation with GAP9 NE16 APIs for a convolutional kernel on the right.

V. Adding a new MATCH HW Target

To extend MATCH to a new HW target, we provide the **MatchTarget** class, which can encompass one or more *HW Execution Modules*. Each HW Execution Module contains four key components, as shown in Figure 4.

First, the above-mentioned **Pattern Table** lists the supported patterns for the module. It essentially answers to the question: "which operators can be accelerated by this HW module?". Each entry in the table defines a specific sequence of DNN graph nodes that the hardware module can execute, as well as a series of constraints, as boolean functions of the nodes' hyper-parameters, whose logical conjunction must be true for the sequence to be accelerable. For instance, a pattern might include a 2D convolution node followed by a batch

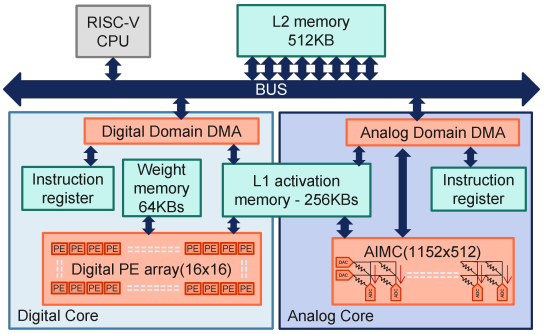


Fig. 5. DIANA architecture. We report the computational units in orange and the memory units in light green.

normalization and a re-quantization step, with an additional constraint of supporting only a 3x3 filter size.

The **Cost Model** is used for generating the correct schedule for each supported operator pattern during the DSE process. The cost model is defined analytically as a generic Python function taking information on the matched pattern, including its composing nodes, their hyper-parameters, data layouts, quantization formats, etc, and returning a scalar (e.g., the pattern's latency or energy). MATCH can support models with different levels of detail, ranging from rough to almost-cycle-accurate estimates. Those can take into account information such as memory bandwidth, HW spatial parallelism, memory hierarchy, ISA extensions, etc. The most important property of a cost model is *rank preservation* between schedules, i.e., if the estimated latency for schedule A is $<$ than the one for schedule B, then the same should hold for the actual on-device latencies. Further, as previously mentioned, notice that the cost model is crucial to discern between multiple `MatchTarget` instances that support the same operations, if available, and select the most efficient one. Therefore, rank preservation across different HW modules is also crucial.

A set of **Network Transformations** is defined to be applied to the neural network both before and after graph partitioning. As mentioned in Sec. IV-A, these transformations manipulate the graph for the specific hardware, modifying properties such as input tensor layout or replacing individual operations with equivalent ones, if the HW Module expects them in a peculiar way.

Finally, a **Code Generation Backend** is present for each HW module: this includes the specialization of the above-mentioned APIs for the target hardware, which replace the generic MATCH APIs in the final generated C code. Notice that APIs can be either HW module-specific or SoC-specific, providing a flexible and extensible framework for supporting new hardware targets.

Below, we provide two examples of HW targets, DIANA (only the digital accelerator) and GAP9, including its cluster of 8 general-purpose cores and its AI accelerator, NE16. Adding a different HW target would require the same steps.

A. DIANA Model Customization

The first platform we target is DIANA [4], shown in Fig. 5. It features a RISC-V MCU, and two accelerators: a digital 8-bit accelerator and an Analog-in-Memory Computing (AIMC) ternary one. Our work targets only the digital one since we only target 8-bit integer networks. However, the analog accelerator can be supported as an additional HW Execution Module of the DIANA `MatchTarget`. The digital accelerator is a 2D SIMD array of 16×16 PEs, that can achieve up to 256 8-bit MAC operations per cycle, allowing execution of convolutions / fully-connected layers with re-quantization, ReLU, and pooling operations directly at the output. The DIANA accelerator implements 2D Convolutions by spatially unrolling the output channels (K) and output spatial width (OX) in the two physical dimensions of the array. Instead, fully connected (FC) layers are spatially unrolled along input and output neurons. We stored the corresponding patterns, including the geometrical constraints on supported kernel sizes (i.e., all odd-sized and square kernels up to 7x7) in the module's Pattern Table. The accelerator has a 256kB L1 activation memory and a 64kB private weight memory, while the SoC L2 main memory is 512kB. Transfers between different memory levels are handled through DMA blocking calls. In order to correctly and fully utilize the PE array, DIANA layer primitives require spatially unrolled dimensions (K and OX) to be multiple of 16 and a custom NCHW-derived data layout. Both these requirements are handled through Network Transformation passes, exploiting spatial padding and static weights reshapes (not adding overhead at runtime).

The ZigZag DIANA performance Cost Model includes two main contributions L_{ops} and $L_{mem,i,j}$, respectively, the latency of inner loop computations in L1 and the latency for memory transfers between the i-th and j-th hierarchy levels. L_{ops} is derived automatically from the spatial parallelism information provided as input, coupled with coefficients that account for the number of computation cycles in L1. For DIANA, this corresponds to the latency for reading inputs, performing MACs, and writing outputs (1 cycle each), plus the application of elementwise operators to the outputs, and their storage into the activation memory (23 cycles). $L_{mem,1,2}$ models the memory transfer cycles and the performance overheads caused by L2-to-L1 DMA calls. This component is derived from the memory bandwidth information and the transferred tensors dimensions. Additionally, the shape of the transfers influences the performance overhead, which we established to be 70-cycles for each chunk of data stored contiguously in memory. If a data block is not stored contiguously, the overhead is multiplied by the number of contiguous sub-blocks. Since DIANA transfers data synchronously, ZigZag computes the overall latency as $L = L_{ops} + L_{mem,1,2}$.

The DIANA code generator includes the specific HW accelerator invocation and DMA APIs. No Synchronization APIs are needed, given the blocking nature of the DMA and the presence of a single active accelerator.

B. GAP9 Model Customization

GAP9 combines a control MCU with a programmable multi-core compute cluster and a dedicated AI accelerator, NE16 (Figure 6). All cores use a RISC-V ISA enhanced with custom DSP-oriented extensions. Noteworthy, GAP9's `MatchTarget` is composed of two different HW Execution Modules (cluster and NE16), thus showcasing the flexibility

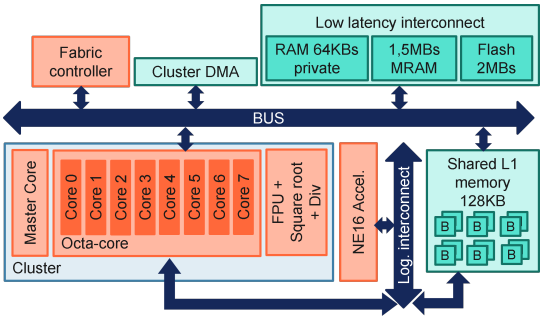


Fig. 6. GAP9 architecture. We report the computational units in orange and the memory units in light green.

of our tool and its management of heterogeneity. Both HW modules communicate with the control MCU through DMA, exploiting double-buffering and interleaving memory transfers with computation. The cluster and NE16 share an L1 multi-bank memory of 128kB for both activation and weights, while the SoC includes an L2 main memory of 1.5 MB. The cluster supports a wide range of operators, including pooling, convolutions, additions, and fully-connected layers, all followed by re-quantization. NE16, instead, only supports convolutions. Thus, the cluster pattern table is a superset of the NE16 one, and MATCH can choose the best HW Module to deploy operators that hit both tables, depending on their hyperparameters. This causes a slight increase in compilation time, since the scheduling optimization has to be repeated for both modules, selecting the best result. Network Transformations for GAP9 have already been shown in Table II.

Two cost models are defined for the two HW modules: for NE16, we rely on the open-source cost model at[2], which allows us to compute the L_{ops} component. Despite the cluster not being an "accelerator" in the proper sense, we still define its *optimal* spatial mapping by considering how the kernels parallelize the innermost loops. For instance, for a convolution, we set the optimal spatial mapping to be equal to the tile dimensions that maximize the parallelization and memory reuse in the internal loop of the adopted kernel library (PULP-NN), i.e., $OX = 2$, $K = 4$, and $OY = 8$ [27]. When this spatial mapping cannot be used, e.g., when the OY dimension is not multiple of 8, MATCH can select between *padding* and *parallelism reduction*. For each spatially unrolled dimension (e.g. OY), its largest divider smaller than the optimal unrolling factor is used as a new unrolling factor (e.g., D, s.t. $OY = nD$, $D < 8$). Then, the number of temporal iterations required to process the layer using this new spatial parallelism is computed. If it is identical to the one obtained with the optimal values on a padded version of the input, then the reduced parallelism mapping is kept, as it does not incur a memory overhead. Otherwise, the input is padded. This flexibility is allowed by the structure of the cluster, which does not have a pre-defined spatial unrolling and is supported for all accelerators of this kind in MATCH. Therefore, it is not part of the required target-specific customizations. L_{ops} is finally computed with a model extrapolated from the compiled back-end library, PULP-NN.

For both NE16 and Cluster, since DMA transfers are asynchronous, the total latency is computed as $L = max(L_{ops}, L_{mem,1,2})$, where L_{ops} is modeled as detailed above. $L_{mem,1,2}$ models again the memory transfer cycles and the performance overheads caused by L2-to-L1 DMA calls. This component is derived from the memory bandwidth information and the transferred tensors dimensions, as in DIANA. In this case, we established a 27-cycles overhead for each chunk of data transferred that is stored contiguously in memory.

The GAP9 code generator includes APIs for both HW modules; further, it also includes the Synchronization APIs to orchestrate NE16 and Cluster execution, and memory transfers. An example of the API functions for NE16 has been previously shown in Fig. 3.

VI. Experimental Results

In this section, we evaluate MATCH by compiling different DNNs targeting both DIANA and GAP9. For both platforms, we set the operating frequency to 260MHz and used the latest available SDKs and open-source kernel libraries. We did not modify or refine the kernel libraries, since we aim to evaluate our AI compiler's performance given a pre-existing backend, and not to maximize absolute performance by digging into backend code optimizations. Latency measurements were obtained using on-board dedicated performance counters. Power numbers for DIANA are extracted from its reference paper [4]; GAP9 power is measured using the Nordic Power Profiler Kit II with a supply voltage of 0.8V[3]. Our tool is compared with TVM (which only generates code for the main MCU of the two systems) and with HTVM (for DIANA) and DORY (GAP9), which utilize the same platform backend libraries and HW modules but different AI compilation steps. Note that we only compare with DORY for deployment on the cluster, given that its support for NE16 is not fully documented and working for every node of the benchmarked networks. Additionally, on GAP9, we also compare MATCH with the NNTool, a proprietary tool from GreenWaves that utilizes a different backend library. Noteworthy, all the following results are measured on the actual HW platforms.

In Sec. VI-A, we first report MATCH's results on single convolutional layers, comparing them with plain TVM. In this comparison, MATCH's speed-ups derive from the utilization of the HW accelerators (the DIANA's digital accelerator, and the GAP9's cluster / NE16 accelerator) and not from additional compilation passes, which are the same for both MATCH and TVM. In Sec. VI-B, we benchmark MATCH and SoA competitors measuring the end-to-end inference time and the energy achieved on the four networks from the *MLPerf Tiny* benchmark suite [12]: a ResNet V1 [34] architecture with a backbone of 8 convolutional layers, trained for image classification on CIFAR10; a MobileNetV1 [35] with a width multiplier of 0.25 for person detection on the Visual Wake Words dataset; a Depthwise Separable CNN (DS-CNN) [36] trained for Keyword Spotting (KWS) on the Speech Commands v2 dataset [37], with 105,829 utterances, to be classified

[2]https://github.com/eml-eda/plinio/blob/main/plinio/cost/ne16_latency.py

[3]https://www.nordicsemi.com/Products/Development-hardware/Power-Profiler-Kit-2

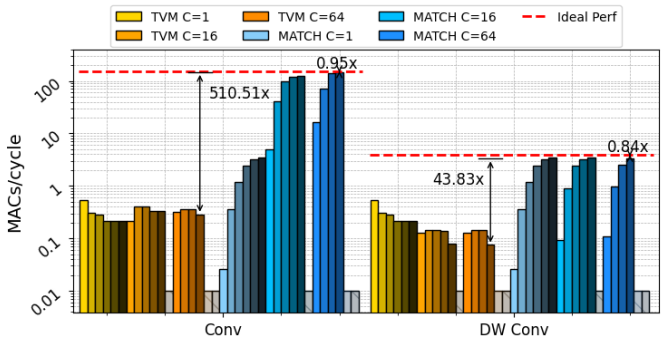


Fig. 7. DIANA micro benchmarking results. Darker colors point to bigger spatial dimensions. Grey bars correspond to kernels not fitting in memory.

into 12 classes; a fully-connected (FC) Autoencoder, which targets the Toy-car data fold included in the DCASE2020 dataset [38], to detect anomalies based on machine operating sounds. Finally, in Sec. VI-C, we perform ablation studies to show the impact of last-level cache memory reduction and demonstrate the performance gain obtained by exploiting SoC heterogeneity.

A. Micro Benchmarking

In this section, we present the results of our micro benchmarking experiments. We benchmarked a series of convolutional blocks, encompassing a 2D Conv followed by bias addition and re-quantization (multiplication, right shift, clip, and cast operators). The hyperparameters used were $IX = IY \in \{2, 8, 16, 32, 64, 128\}$, $C = K \in \{1, 16, 64\}$, with a padding of 1 for all corners, FX = FY = 3, a convolution stride of 1, and a dilation of 0. We tested classical and depthwise (DW) convolutions with the same hyperparameters.

Fig. 7 showcases the results of MATCH on DIANA's digital accelerator. For standard convolutions, MATCH achieves an average speed-up of 4.43× compared to TVM with C = 1. As the number of input channels increases, the speed-up also increases, reaching a maximum of 510.51× for C = 64, IX = IY = 32. With these dimensions, MATCH achieves 146.12 MACs/cycle, which is only 5% lower compared to the ideal performance predicted by our cost model, demonstrating close-to-optimal compiler management of memory and computation orchestration, and perfect spatial utilization of the HW accelerator. For DW convolutions, the digital accelerator achieves lower efficiency, given its limited spatial utilization. Nevertheless, with C = 64 and IX = IY = 32, MATCH is still able to achieve a speed-up of 43.83× compared to TVM, achieving 77% of the estimated MAC/cycle. Note that even if the accelerator has not been originally designed to execute DW Convolutions, using an appropriate cost model MATCH can still find ways to efficiently execute these layers on it, achieving a significant speed-up.

Fig. 8 presents the results of MATCH on the GAP9 platform, considering both the cluster and the NE16 accelerator. On the cluster, MATCH outperformed TVM by a maximum factor of 104.01× for standard convolutions and 9.48× for depthwise convolutions on 64-channels layers. Noteworthy, these speed-ups do not only derive from the higher number of cores of the cluster compared to the single CPU utilized by TVM: thanks to the backend library that it exploits, MATCH also utilizes SIMDs operations, HW loops, and specific ISA extensions that TVM can not be aware of. The maximum performance achieved is nearly ideal, reaching 91% and 88% of the cost model estimation. When deployed on the NE16 accelerator, MATCH demonstrated a maximum speed-up of 669.94× relative to TVM for standard convolutions and 39.5× for depthwise convolutions. MATCH reached 83% of the predicted performance for 64-channel convolutions and 77% for depthwise convolutions, indicating a minor discrepancy between the cost model and actual hardware performance. This suggests that further refinement to the template and specific NE16 backend could enhance the overall final performance. However, the scope of this paper and of our proposed tool is to extract the maximum performance by exploiting a pre-defined existing kernel library and using a universal layer template. We will investigate additional HW-specific optimizations and customizations in our future work.

B. Full Network Execution and Comparison with the State-of-the-Art

In Tab. III, we extend our experiments to the MLPerf Tiny networks on GAP9 and DIANA, comparing latency/energy results against plain TVM, HTVM, DORY, and NNTool.

When targeting DIANA, MATCH demonstrates significant improvements over TVM, achieving an average speed-up and energy reduction of 60.87× across the tested networks. Specifically, it achieves 169.4× lower latency/energy on ResNet, 6.7× on DSCNN, and 6.4× on DAE, whereas MobileNet cannot be deployed as it exceeds the total on-chip memory. Similarly, for GAP9, MATCH outperformed TVM substantially, with speed-ups of 47.7× for MobileNet, 159.2× for ResNet, 53.1× for DSCNN, and 11.3× for DAE, and energy reductions of 34.43×, 121.54×, 42.25×, and 8.98×, respectively.

Compared with HTVM, MATCH exhibits a slight performance deficit for DSCNN and DAE, with latencies of 7.3 ms versus 7.26 ms and 0.4 ms versus 0.36 ms, respectively. This slight disadvantage arises because our layer template is designed to be general across multiple platforms, resulting in a marginally slower execution. However, for MobileNet and ResNet, MATCH improves both latency and energy from 6.17 ms / 153.9 uJ and 1.27 ms / 31.65 uJ to 6.08 ms / 151.84 uJ and 0.79 ms / 19.6 uJ, respectively. These gains are attributed to the additional features of MATCH with respect to the tiler included in HTVM, which particularly impacts the performance of layers (both DW and normal conv) with a low number of channels; specifically, uneven tensor mapping shows great advantages in saving memory transactions and therefore latency in cases where loops on channel dimensions are very small. In terms of compilation time, MATCH is, on average, slightly slower than HTVM (up to 2.2×). However, even for the most complex model (MobileNet), compilation for DIANA requires less than 10s.

When comparing MATCH to DORY, MATCH's ability to leverage the full heterogeneous platform comprising the NE16 accelerator results in an average speed-up across the 4 networks of 2.15× and in an average energy reduction

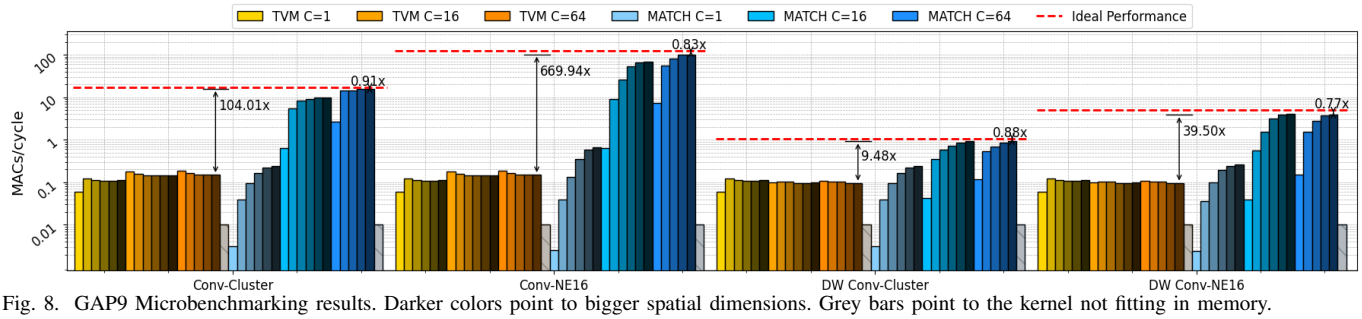


Fig. 8. GAP9 Microbenchmarking results. Darker colors point to bigger spatial dimensions. Grey bars point to the kernel not fitting in memory.

TABLE III
DEPLOYMENT RESULTS: END-TO-END INFERENCE TIME AND ENERGY RESULTS FOR DIFFERENT NETWORKS. IN BOLD IS THE BEST ALTERNATIVE FOR EACH PLATFORM AND NETWORK, AND IN ITALICS, THE SECOND-RANKED ONE. LATENCY IS MEASURED IN MS, ENERGY IN UJ.

Network	DIANA						GAP9							
	TVM		HTVM		MATCH		TVM		DORY		NNTool*		MATCH	
	Lat. [ms]	En. [uJ]	Lat. [ms]	En. [uJ]	Lat. [ms]	En. [uJ]	Lat. [ms]	En. [uJ]	Lat. [ms]	En. [uJ]	Lat. [ms]	En. [uJ]	Lat. [ms]	En. [uJ]
MobileNet	OoM	OoM	*6.17*	*153.9*	**6.08**	**151.84**	236.22	21645.62	11.29	*1398.8*	**1.61**	n.a.	*4.94*	**628.69**
ResNet	133.1	3322.08	*1.27*	*31.65*	**0.79**	**19.6**	342.72	31875.12	5.76	*745.81*	**0.88**	n.a.	*2.15*	**262.27**
DSCNN	49.16	1227.04	**7.26**	**181.24**	*7.30*	*182.32*	83.41	7500.3	4.19	*480.44*	**0.68**	n.a.	*1.57*	**177.51**
DAE	2.58	64.28	**0.36**	**8.98**	*0.40*	*9.98*	6.12	535.95	*0.52*	**56.78**	**0.26**	n.a.	0.54	*59.66*

*NNTool is a proprietary source tool that is also platform-specific. Latency results extracted from [10].

of 2.17×. Still, even when employing just the cluster (precise numbers reported in Tab. IV below), for MobileNet and ResNet, we achieve latencies/energy of 11.2ms/1.39 mJ and 5.48ms/0.71mJ, respectively, outperforming DORY by 1%/5%. The benefit is particularly clear for late layers with small feature maps, where adding the possibility of loop reordering minimizes the memory movements. For DSCNN and DAE, however, MATCH lags behind, with latencies/energies of 4.25 ms/0.49mJ and 0.54ms/0.06mJ, due to the layer template generality, which adds non-negligible overhead in very small layers. MATCH's compilation time is higher for GAP9, and up to 24.7× slower than DORY's, due to the more complex nature of this platform's HW modules, and the need of running DSE twice for operators supported both by the Cluster and by NE16. However, it remains manageable, requiring less than 5 minutes in the worst case.

Finally, compared to NNTool, MATCH produces, on average, 2.5× slower code. This difference is primarily due to the more efficient and optimized back-end kernels used by NNTool, which are tailored to each specific hyperparameter combination (the library contains hundreds of specialized convolutional kernels), and secondarily to template generalization overheads. We do not compare in terms of energy, given that the measuring setup reported by the company is not compatible with ours. Nevertheless, it is important to note that MATCH's framework is designed to be easily extensible, which could allow the integration of alternative back-end kernels, including those used by NNTool. More in general, MATCH is not designed to provide extremely optimized results on a new platform out-of-the-box, but rather, to support deploying workloads to new accelerators quickly, with few easy customizations. Overall, the results of this section demonstrate this objective, as MATCH is superior or in the worst case on-par with all other competitors when using the same backend. While we are outperformed by NNTool, MATCH's modularity that allows replacing the backend library, and even the DSE engine, could be leveraged to progressively reduce this gap, while maintaining full compatibility with the TVM ecosystem.

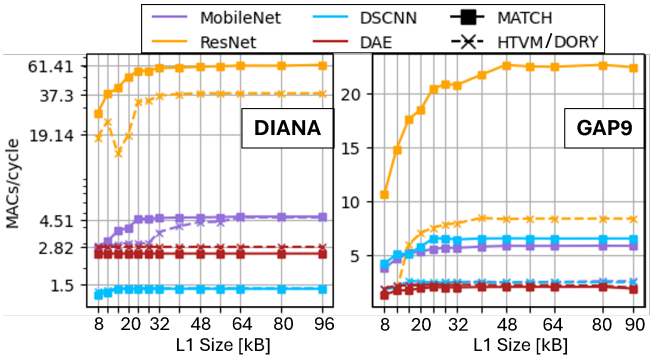


Fig. 9. MACs/Cycle of networks on DIANA/GAP9 with scaling L1 size (kB).

C. Ablation study

1) Scheduling and memory scaling: In this section, we explore the impact of L1 memory size on the achieved MAC/cycle across the four networks—MobileNet, ResNet, DSCNN, and DAE, on the GAP9 and DIANA hardware platforms (Fig. 9). The goal is to demonstrate how MATCH enables better scheduling and improved performance with respect to competitors, particularly when memory is constrained. For the DAE and DSCNN networks, no notable trends are observed across different L1 memory sizes. This is because these networks do not require tiling, allowing for consistent performance regardless of the available memory. The simplicity of these networks allows maintaining relatively stable MAC/cycle performance across varying L1 sizes, for both MATCH and its comparisons.

In contrast, on the MobileNet, at a constrained L1 memory size of 24 kB, on DIANA, we observe that MATCH can still achieve near-full performance, maintaining high MACs per cycle. In this scenario, HTVM's performance drops significantly, falling to 3 MAC/cycle, which results in MATCH achieving a speed-up of 1.5× over HTVM (versus 1% at full L1 size). This highlights MATCH's ability to optimize layer scheduling effectively, even with limited memory resources.

For ResNet, the performance of competing tools declines sharply as L1 memory size decreases on both the DIANA

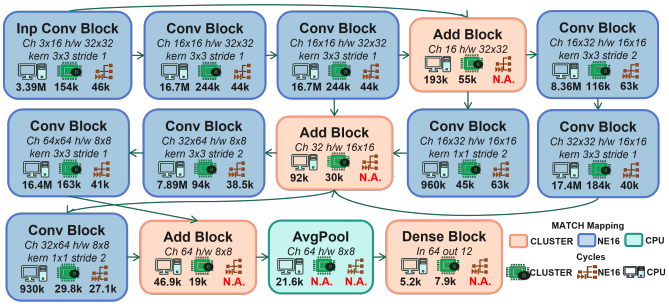


Fig. 10. MATCH mapping for the ResNet, displaying for each block the latency, in number of cycles, for each single HW module.

and GAP9 platforms. For example, at 16 kB of L1 memory, HTVM becomes 3.01× slower than MATCH, a stark contrast to the 1.61× difference observed when using the full L1 size. Similarly, on GAP9, DORY experiences a significant slowdown, being 8.85× slower than MATCH at 12 kB, compared to a 2.68× slow-down at full memory capacity. Notably, DORY fails to deploy ResNet at 8 kB of L1 memory on GAP9 because the last layers with high C dimension do not fit, even with the smallest tile size, given the additional buffers that PULP-NN requires. MATCH, on the other hand, successfully deploys ResNet by offloading these layers to the CPU, demonstrating its flexibility and robustness in handling low-memory environments.

TABLE IV
GAP9 NETWORK DEPLOYMENT USING DIFFERENT HW MODULES.

Network	CPU Only	Cluster+CPU	NE16+CPU	Full
		Latency		
MobileNet	236.22 ms	11.2 ms	5.02 ms	4.94 ms
ResNet	342.72 ms	5.48 ms	2.90 ms	2.15 ms
DSCNN	83.41 ms	4.25 ms	14.46 ms	1.57 ms
DAE	6.12 ms	0.54 ms	6.12 ms	0.54 ms
		Energy		
MobileNet	21.65 mJ	1.39 mJ	0.58 mJ	0.63 mJ
ResNet	31.88 mJ	0.71 mJ	0.31 mJ	0.26 mJ
DSCNN	7.50 mJ	0.49 mJ	1.30 mJ	0.18 mJ
DAE	0.54 mJ	0.06 mJ	0.54 mJ	0.06 mJ

2) Heterogeneity impact: Tab. IV summarizes the performance in terms of latency and energy, obtained by enabling different HW modules on GAP9. This analysis highlights the flexibility of MATCH in handling multi-accelerator systems, demonstrating the effectiveness of our partitioning mechanism, which groups nodes and assigns them to the HW module that the scheduler predicts will deliver the lowest latency.

Only enabling the Cluster HW module reduces the latency of the CPU-only solution by an average of 28.64×, with the ResNet network achieving a peak speed-up of 62.58×. Enabling the NE16 HW module without the 8-core cluster achieves an average speed-up of 43× over the CPU-only baseline, with a maximum speed-up of 118.32× for the ResNet network. It is important to note that for the DAE network, the NE16+CPU configuration yields identical results to the CPU-only setup because the DAE model is composed entirely of fully connected layers, which the NE16 library does not support. Similarly, for the DSCNN network, the NE16+CPU configuration results in a lower performance than the Cluster+CPU one since the first layer, featuring a 4x10 rectangular filter, cannot be offloaded to the accelerator. Fully leveraging MATCH's heterogeneous deployment capabilities proves to be highly effective, outperforming all other configurations, being on average 67.83× better than the CPU baseline, 2.13× better than the Cluster+CPU solution, and 5.7× better than the NE16+CPU one. For the MobileNet network, the multi-target configuration shows a 1.56% improvement over the NE16+CPU target by offloading the first layer to the cluster, which results in lower latency. In the DSCNN network, the cluster processes the first layer due to the unsupported filter shape by the NE16, while the accelerator efficiently handles the remaining convolutional layers. The full configuration yields results identical to the cluster+CPU setup for the DAE. In the case of the ResNet model, the full configuration achieves a 34.55% improvement over the NE16+CPU target and a 2.54× speed-up over the cluster+CPU configuration. Fig. 10 provides a detailed breakdown of the effective latency, in cycles, for each HW module across all ResNet layers, and illustrates MATCH's decision-making process in offloading each computation. NE16 processes every convolutional layer; the 8-core cluster manages addition operations and the final dense block, while the average pooling operation is handled by the CPU.

Similar trends to those observed with latency appear in the energy results. The CPU-only configuration consistently requires the highest energy, with MobileNet consuming over 21 mJ and ResNet exceeding 31 mJ. Both the Cluster+CPU and NE16+CPU options significantly reduce overall energy, with one or the other excelling depending on NE16's coverage of a particular DNN's layers. In terms of energy, using NE16 is even more beneficial compared to the Cluster, due to a lower average power consumption, reducing, for instance, the energy consumption on MobileNet and ResNet by 2.4× and 2.3×. Combining both HW modules leads to the lowest energy usage overall, with a single exception on MobileNet. For this DNN, MATCH offloads two layers to the Cluster to maximize performance (since its DSE used a latency cost model), which however results in 1.1× more energy.

TABLE V
GAP9 FULL-SOC RESULTS COMPARING MATCH'S ANALYTICAL COST MODELS WITH AN ORACLE POLICY THAT SELECTS THE BEST HW MODULE BASED ON MEASURED LATENCY.

	MobileNet	ResNet	DSCNN	DAE
MATCH Models	4.94 ms	2.15 ms	1.57 ms	0.54 ms
MATCH Oracle	4.73 ms	2.07 ms	1.56 ms	0.54 ms

3) HW cost models influence: To analyze MATCH's capability of selecting the correct HW module, Table V compares the results of Sec. VI-B (first row) with an "Oracle" policy (second row). The Oracle always chooses the fastest HW module based on the *actual* measured latency for each layer, whereas "MATCH Models" relies on the *predicted* latency from analytical cost models. Overall, the discrepancies are minimal, at most 0.21 ms (4.4%) on MobileNet and 0.08 ms (3.8%) on ResNet.

Specifically for MobileNet, MATCH wrongly assigns one convolutional block to the cluster, predicting that it would run faster than on NE16, while in reality, NE16 would have terminated 0.21 ms earlier. Similarly, on ResNet, MATCH optimally

deploys all layers except two (mapping shown in Fig. 10). The first error is on a convolutional layer with parameters $C = 16$, $K = 32$, $OY = OX = 16$, $FY = FX = 1$, and stride = 2 (second box from the right in the middle row of the figure). This layer is offloaded to NE16, although the Cluster would have been slightly more efficient. This is due to the discrepancy between the estimated and real latency on NE16, caused by inefficiencies in the associated kernels. The second exception occurs in the final dense block, where the cluster slightly underperforms compared to the TVM fallback solution. Although these discrepancies are minor, MATCH's robustness could be further enhanced by adding more refined cost models. The only exception is when the optimal alternative is the host CPU, since TVM's performance metrics are not integrated into MATCH's decision-making process, implicitly assuming the host to be always slower. While this does not hold for very small layers, their impact on overall latency is negligible (correctly allocating the FC layer in the ResNet would lead to a 0.4% speed-up).

VII. Conclusions

We introduced MATCH, a model-aware toolchain for DNN deployment on accelerators. Differently from other target-specific toolchains, MATCH does not embed hardware-dependent optimizations or heuristics in the code but rather exposes an API to define high-level model-based hardware abstractions, fed to a generic and flexible optimization engine. As a consequence, adding support for a new HW module becomes significantly easier, avoiding complex optimization pass re-implementations. A new HW target can be added in less than 1 week of work. Similarly, adding support for a new operator is even faster, either by adding an entry in an HW module's Pattern Table or relying on the TVM fallback solution. With experiments on several deep learning workloads and end-to-end networks, we have shown that MATCH can achieve performances comparable to highly optimized and HW-specific toolchains on two highly heterogeneous platforms, proving the tools' flexibility and the capability of handling heterogeneous targets. Compared to the SoA open-source solution, on the four MLPerf Tiny benchmarks, we obtained 16.94% lower latency on DIANA and 2.15× lower latency on GAP9. In our future work, we plan to extend MATCH to new hardware targets. Moreover, we also plan to support dynamic and conditional NN graphs, as required by more advanced AI topologies such as generative transformers. Lastly, we also plan to extend our tool to support the concurrent utilization of multiple HW modules at the same time (e.g. for independent layers in a multi-branch DNN).

VIII. Acknowledgments

This work has received funding from the Key Digital Technologies Joint Undertaking (KDT-JU) under grant agreement No 101095947 and grant agreement No 101112274. This project has been partly funded by the European Research Council (ERC) under grant agreement No. 101088865, and the Flanders AI Research Program (FAIR)." The authors acknowledge the support of the Italian Ministry of University and Research (MUR) and the Sustainable Mobility Center (MOST) through the project PNRR - M4C2 - CNMS - Spoke 2, funded under the scheme CN00000023 - PNRR – M4C2 Inv. 1.4 with grant agreement no. 55_PRR22_1112_22_AT002129.

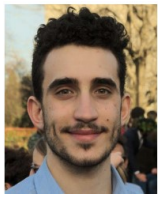
References

[1] E. Flamand et al., "Gap-8: A risc-v soc for ai at the edge of the iot," in IEEE Int. Conf. on App.-specific Sys., Arch. and Proc. (ASAP), 2018.
[2] GreenWaves. Gap9 processor. Accessed Aug. 28, 2024. [Online]. Available: https://greenwaves-technologies.com/gap9_processor/
[3] H. Genc et al., "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in ACM/IEEE Design Autom. Conf. (DAC), 2021.
[4] K. Ueyoshi et al., "DIANA: An End-to-End Energy-Efficient Digital and ANAlog Hybrid Neural Network SoC," in IEEE Int. Solid-State Circ. Conf. (ISSCC), vol. 65, Feb. 2022.
[5] V. Jain et al., "TinyVers: A 0.8-17 TOPS/W, 1.7 μW-20 mW, Tiny Versatile System-on-chip with State-Retentive eMRAM for Machine Learning Inference at the Extreme Edge," in IEEE Symp. on VLSI Tech. and Circ. (VLSI Tech. and Circ.), 2022.
[6] T. Chen et al., "TVM: An automated End-to-End optimizing compiler for deep learning," in Proc. 13th USENIX Conf. on Operating Sys. Design and Impl. (OSDI), 2018.
[7] A. Burrello et al., "DORY: Automatic end-to-end deployment of real-world DNNs on low-cost IoT MCUs," IEEE Trans. on Comput., 2021.
[8] B. Vanik et al., "Tinyiree: An ml execution environment for embedded systems from compilation to deployment," IEEE Micro, vol. 42:5, pp. 9–16, Sep. 2022.
[9] L. Zheng et al., "Ansor: Generating high-performance tensor programs for deep learning," in Proc. 14th USENIX Conf. on Operating Sys. Design and Impl. (OSDI), no. 49, 2020, pp. 863–879.
[10] J. V. Delm et al., "Htvm: Efficient neural network deployment on heterogeneous tinyml platforms," in Proc. 60th ACM/IEEE Design Autom. Conf. (DAC), 2023, pp. 1–6.
[11] L. Mei et al., "Zigzag: Enlarging joint architecture-mapping design space exploration for dnn accelerators," IEEE Trans. on Comput., vol. 70, no. 8, pp. 1160–1174, Feb. 2021.
[12] MLPerf Tiny Benchmark V1.0 Results. Accessed Aug. 28, 2024. [Online]. Available: https://mlcommons.org/en/inference-tiny-10/
[13] A. Kuzmin et al., "Pruning vs quantization: which is better?" in Proc. 37th Int. Conf. on Neural Inf. Proc. Sys.(NIPS), vol. 36, no. 2725, 2024.
[14] K. Wang et al., "Haq: Hardware-aware automated quantization with mixed precision," in IEEE/CVF Conf. on Comp. Vision and Pattern Rec. (CVPR), 2019, pp. 8612–8620.
[15] C. White et al., "Neural architecture search: Insights from 1000 papers," arXiv:2301.08727, 2023.
[16] W. Jiang et al., "Hardware/software co-exploration of neural architectures," IEEE Trans. on Computer-Aided Design of Integrated Circ. and Sys. (TCAD), vol. 39, no. 12, pp. 4805–4815, Apr. 2020.
[17] X. Luo et al., "Lightnas: On lightweight and scalable neural architecture search for embedded platforms," IEEE Trans. on Computer-Aided Design of Integrated Circ. and Sys. (TCAD), vol. 42, no. 6, pp. 1784–1797, Sep. 2023.
[18] J. Lee et al., "Snas: Fast hardware-aware neural architecture search methodology," IEEE Trans. on Computer-Aided Design of Integrated Circ. and Sys. (TCAD), vol. 41, no. 11, pp. 4826–4836, Dec. 2022.
[19] A. Devices. Artificial Intelligence Microcontroller with Ultra-Low-Power Convolutional Neural Network Accelerator. Accessed Aug. 28, 2024. [Online]. Available: https://www.analog.com/en/products/max78000.html
[20] H. Zhu et al., "ROLLER: Fast and efficient tensor compilation for deep learning," in 16th USENIX Symp. on Operating Sys. Design and Impl. (OSDI), Jul. 2022, pp. 233–248.
[21] GreenWaves. NN-Tool. Accessed Aug. 28, 2024. [Online]. Available: https://github.com/GreenWaves-Technologies/gap_sdk/tree/master/tools/nntool
[22] ST Microelectronics. X-CUBE-AI. Accessed Aug. 28, 2024. [Online]. Available: https://www.st.com/en/embedded-software/x-cube-ai.html
[23] R. David et al., "TensorFlow Lite Micro: Embedded Machine Learning for TinyML Systems," in Proc. of Machine Learning and Systems (MLSys), vol. 3, 2021, pp. 800–811.
[24] Y. Xing et al., "Dnnvm: End-to-end compiler leveraging heterogeneous optimizations on fpga-based cnn accelerators," IEEE Trans. on Computer-Aided Design of Integrated Circ. and Sys. (TCAD), vol. 39, no. 10, pp. 2668–2681, Jul. 2020.

[25] H. Jia et al., "A programmable heterogeneous microprocessor based on bit-scalable in-memory computing," IEEE Journal of Solid-State Circuits (JSSC), vol. 55, no. 9, pp. 2609–2621, Apr. 2020.
[26] H. Jia et al., "Scalable and programmable neural network inference accelerator based on in-memory computing," IEEE Journal of Solid-State Circuits (JSSC), vol. 57, no. 1, pp. 198–211, Oct. 2022.
[27] A. Garofalo et al., "Pulp-nn: accelerating quantized neural networks on parallel ultra-low-power risc-v processors," Philos. Trans. R. Soc., vol. 378, no. 2164, Feb. 2020.
[28] H. Ye et al., "Hybriddnn: a framework for high-performance hybrid dnn accelerator design and implementation," in Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference, ser. DAC '20. IEEE Press, 2020.
[29] H. Genc et al., "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in 2021 58th ACM/IEEE Design Automation Conference (DAC), 2021, pp. 769–774.
[30] L. Lai et al., "Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus," arXiv:1801.06601, 2018.
[31] T. Moreau et al., "A hardware–software blueprint for flexible deep learning specialization," IEEE Micro, vol. 39, no. 5, pp. 8–16, Jul. 2019.
[32] Z. Chen et al., "Bring Your Own Codegen to Deep Learning Compiler," arXiv:2105.03215, 2021.
[33] A. Symons et al., "Loma: Fast auto-scheduling on dnn accelerators through loop-order-based memory allocation," in IEEE 3rd Int. Conf. on Art. Int. Circ. and Sys. (AICAS), 2021, pp. 1–4.
[34] K. He et al., "Deep residual learning for image recognition," in IEEE Conf. on Comp. Vision and Pattern Rec. (CVPR), 2016, pp. 770–778.
[35] A. G. Howard et al., "Mobilenets: Efficient convolutional neural networks for mobile vision applications," arXiv:1704.04861, 2017.
[36] Y. Zhang et al., "Hello edge: Keyword spotting on microcontrollers," arXiv:1711.07128, 2017.
[37] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," arXiv:1804.03209, 2018.
[38] Y. Koizumi et al., "Description and discussion on dcase2020 challenge task2: Unsupervised anomalous sound detection for machine condition monitoring," arXiv:2006.05822, 2020.

Mohamed Amine Hamdi is a PhD student at Politecnico di Torino. He received both his B.Sc. degree and M.Sc. degree in computer engineering from Politecnico di Torino, Italy, in 2021 and 2023 respectively. His research interests revolve around AI compilation and efficient machine learning.

Francesco Daghero is a PhD student at Politecnico di Torino. He received a M.Sc. degree in computer engineering from Politecnico di Torino, Italy, in 2019. His research interests concern embedded machine learning and Industry 4.0.



Giuseppe M. Sarda received the B.Sc. and M.Sc. degrees in electrical engineering from the Politecnico di Torino in 2018 and 2020, respectively. He carried out his master's thesis in TU Wien, Austria. In September 2020, he joined imec Leuven and the MICAS Group, KU Leuven, as a Ph.D. Researcher. His current research focuses on in-memory design for efficient embedded machine learning.

Josse Van Delm received the M.Sc. degree in Electrical Engineering from KU Leuven, Belgium, in 2021, and the M.Sc. degree in Electrical Engineering Technology from KU Leuven, Belgium, in 2019. He is currently pursuing a PhD degree in compilers for machine learning (ML) accelerators at MICAS Laboratories. His research interests include multi-level compilers, open-source software development, ML accelerators, Reduced Instruction Set Computer-V (RISC-V) architecture, and hardware/software co-design for heterogeneous multi-core systems.

Arne Symons received the B.Sc. and M.Sc. degrees in electrical engineering from KU Leuven, Leuven, Belgium, in 2018 and 2020, respectively. He is currently working toward the Ph.D. degree in mapping optimization of machine learning workloads on specialized accelerator architectures. In 2022, he was a Visiting Researcher with Robust Systems Group, Stanford University, and interned as a Research Scientist with Meta in 2023.

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

Marian Verhelst (Fellow, IEEE) received the Ph.D. degree from KU Leuven, Leuven, Belgium, in 2008. She is a Full-Time Professor with ESAT-MICAS Laboratories, KU Leuven. Her research focuses on embedded machine learning, hardware accelerators, and low-power edge processing. She is an SSCS Distinguished Lecturer, a member of the Young Academy of Belgium. She currently holds an ERC Grant from the European Union and was a laureate of the Royal Academy of Belgium.

Daniele Jahier Pagliari received the M.Sc. and Ph.D. degrees in computer engineering from the Politecnico di Torino, Turin, Italy, in 2014 and 2018, respectively. He is currently an Assistant Professor with the Politecnico di Torino. His research interests are in the computer-aided design and optimization of digital circuits and systems, with a particular focus on energy-efficiency aspects and on emerging applications, such as machine learning at the edge.

Alessio Burrello received his B.Sc and M.Sc degree in Electronic Engineering at the Politecnico of Turin, Italy, in 2016 and 2018. He is currently working toward his Ph.D. degree at the Department of Electrical, Electronic and Information Technologies Engineering (DEI) of the University of Bologna, Italy. 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.