POLITECNICO DI TORINO
Repository ISTITUZIONALE

MATCHA: Efficient Deployment of Deep Neural Networks on Multi-Accelerator Heterogeneous Edge SoCs

Original
MATCHA: Efficient Deployment of Deep Neural Networks on Multi-Accelerator Heterogeneous Edge SoCs / Russo, E., Hamdi, M.A., Ottaviano, A., Conti, F., Garofalo, A., Jahier Pagliari, D., Palesi, M., Benini, L., Burrello, A.. - (In corso di stampa). (63rd ACM/IEEE Design Automation Conference (DAC26) July 26-29, 2026) [10.1145/3770743.3804120].

Availability:
This version is available at: 11583/3016397 since: 2026-10-07T12:31:39Z

Publisher:
ACM/IEEE

Published
DOI:10.1145/3770743.3804120

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
ACM postprint/Author's Accepted Manuscript, con Copyr. autore

(Article begins on next page)

19 November 2026

MATCHA: Efficient Deployment of Deep Neural Networks on Multi-Accelerator Heterogeneous Edge SoCs

Enrico Russo[†], Mohamed Amine Hamdi[‡], Alessandro Ottaviano[⋆], Francesco Conti[§], Angelo Garofalo[⋆§], Daniele Jahier Pagliari[‡], Maurizio Palesi[†], Luca Benini[⋆§], Alessio Burrello[‡]
[†]University of Catania, Italy; [‡]Politecnico di Torino, Italy; [⋆]ETH Zurich, Switzerland; [§]University of Bologna, Italy

arXiv:2604.09124v1 [cs.DC] 10 Apr 2026

Abstract

Deploying DNNs on System-on-Chips (SoC) with multiple heterogeneous acceleration engines is challenging, and the majority of deployment frameworks cannot fully exploit heterogeneity. We present MATCHA, a unified DNN deployment framework that generates highly concurrent schedules for parallel, heterogeneous accelerators and uses constraint programming to optimize L3/L2 memory allocation and scheduling. Using pattern matching, tiling, and mapping across individual HW units enables parallel execution and high accelerator utilization. On the MLPerf Tiny benchmark, using a SoC with two heterogeneous accelerators, MATCHA improves accelerator utilization and reduces inference latency by up to 35% with respect to the the state-of-the-art MATCH compiler.

1 Introduction

Deep Neural Networks (DNNs) are increasingly being deployed at the edge, where they must execute at high performance under tight latency, energy, and cost constraints [11]. To meet these competing objectives, modern edge systems-on-chip (SoCs) are becoming increasingly heterogeneous, integrating multiple specialized accelerators alongside general-purpose host CPUs [7, 9, 27]. Equipping a SoC with more than one kind of DNN accelerators enhances flexibility, as different architectural templates, e.g., Single-Instruction Multiple Data (SIMD) [16], vector units [22], systolic or dataflow processors for General Matrix Mutiplication (GEMM) [10, 27], etc, provide different latency/energy efficiency trade-offs as a function of the accelerated operation(s) and corresponding tensors geometries [7]. However, deployment frameworks for these SoCs often fail to fully exploit this flexibility, resulting in low hardware utilization, as they either offload entire DNNs to a single accelerator, or at most, map each DNN layer (or fused layer sequence) to the most suited compute unit, in a purely sequential fashion, controlled synchronously by the host [5, 12, 21, 25, 28]. As a result, only one accelerator is active at any given time, while the others remain idle.

In this paper, we examine two approaches for optimizing the utilization of available hardware resources. The first one leverages the structure of modern DNN computational graphs, which often include multiple independent branches (e.g., in ResNet-like [13] networks or Attention blocks [29]), that can be potentially offloaded in parallel to different accelerators. However, this approach is limited to multi-branch DNN models, and the optimization space is restricted to one-to-one layer to accelerator mappings, which are often insufficient to eliminate idleness entirely, as branches could be highly unbalanced in workload.

Corresponding authors: enrico.russo@unict.it, mohamed.hamdi@polito.it

This work was accepted at the 63rd ACM/IEEE Design Automation Conference (DAC26).

Thus, at a finer grain, we explore the parallelization opportunities provided by tiling DNN operators and assigning each tile to a different accelerator. While a trivial implementation of this offloading scheme is classic model parallelism, in which layer workloads are split evenly across homogeneous compute units [3], accelerators' heterogeneity in terms of operator support and performance makes that solution either sub-optimal or unfeasible, requiring a more general problem framing.

To fully support both these offloading schemes, we propose a novel DNN optimizing compiler that simultaneously explores layer-level and tile-level parallelization, targeting OS-less, multi-accelerator SoCs for edge inference. Our compiler is based on TVM [6] and on its extension MATCH for heterogeneous accelerators support [12], but we completely redesign both the mapping optimization engine and the runtime to support concurrent, asynchronous execution on multiple compute units. Hence, we name it **MATCHA** (MATCH Asynchronous). Our main contributions are the following:

- We present a unified DNN deployment flow supporting concurrent offloading onto multiple accelerators, possibly with different internal architectures, thanks to a flexible model-based latency cost abstraction. To our knowledge, MATCHA is the first tool to support such kind of mapping for OS-less heterogeneous SoCs.
- We propose a constraint programming-based, tile-centric mapping framework to perform heterogeneous pattern matching and distribute work among accelerators, maximizing utilization.
- Through an experimental campaign on a complex heterogeneous SoC, equipped with a host CPU and two distinct accelerators, and on multiple full networks and layer blocks [1, 13, 29, 30], we show that MATCHA reduces end-to-end latency by up to 35% with respect to the state-of-the-art MATCH compiler.

2 Background and Related Work

Deploying deep neural networks efficiently on embedded and edge platforms has driven the design of a wide range of DNN accelerators and optimizing compilers. Despite the diversity of hardware and software stacks, most deployment frameworks follow a similar pipeline with four main stages: (i) representing the DNN as a computation graph in an intermediate representation (IR) and applying graph-level transformations, (ii) partitioning the workload into subgraphs or tiles with an appropriate granularity, (iii) mapping each subgraph onto the available execution modules, and (iv) generating target-specific code. The third step, commonly referred to as the *mapping* phase, has been extensively studied for spatial and systolic DNN accelerators [14, 15, 20, 24, 26, 32]. Mapping decisions involve loop tiling (to fit large tensors into small local scratchpad memories), unrolling (to parallelize computation across the available

Table 1: Comparison of state-of-the-art DNN deployment tools.

Work	Open Source	HSoC Support	Operator Extension	Device Extension	Multi ISA	Async. Execution	Pattern Tiling
TFLite [8]	✔	✖	✔	✔	•	✖	•
TelaMalloc [17]	✖	✖	–	–	•	•	✖
Bolt [31]	✔	✖	✔	✖	•	–	–
COMB [33]	✖	✔	✔	✔	✖	–	✖
HTVM [28]	✔	✔	✔	✖	✖	✖	✖
DORY [5]	✔	✖	✖	✖	✖	✖	–
Deeploy [25]	✔	✔	✔	✔	–	–	✖
NEST-C [21]	✔	✔	✔	–	–	✔	✖
Map-and-Conquer [4]	✖	✔	–	–	✔	✔	–
MATCH [12]	✔	✔	✔	✔	✖	✖	✖
Ours	✔	✔	–	✔	✔	✔	✔

✔ Yes ✖ No – Limited • Not Applicable

processing elements), and ordering (to maximize data reuse and minimize data movement across the memory hierarchy).

Modern edge platforms increasingly adopt heterogeneous systems-on-chip (HSoCs), where a general-purpose host core is coupled with multiple specialized accelerators exposing different ISAs, data types, and memory organizations. Examples range from multi-processor SoCs (MPSoCs) that integrate CPU cores, GPUs, and NPUs on a shared DRAM (e.g., NVIDIA Xavier, Apple and Qualcomm SoCs) [7], to extreme edge-oriented SoCs that combine RISC-V control cores with tightly coupled accelerator clusters [9, 27]. In this work we target edge HSoCs that offer several accelerators and processor clusters connected through a software-managed, multi-level scratchpad memory hierarchy.

Table 1 compares representative DNN deployment frameworks for these systems along key dimensions: open-source availability, explicit support for heterogeneous SoCs, extensibility of operator and device backends, support for multi-ISA systems, asynchronous execution capability, and tile-level inter-device parallelization. TFLite [8] provides portability and pluggable kernels but offers limited control over mapping and tiling. DORY [5] optimizes memory allocation for single accelerators using constraint programming, while TelaMalloc [17] and Bolt [31] target specific subproblems such as memory allocation or GPU tuning without full heterogeneous support. More recent tools, including Deeploy [25], HTVM [28], NEST-C [21], COMB [33], and Map-and-Conquer [4], extend compiler frameworks to heterogeneous SoCs, but differ in their granularity and scheduling scope. Deeploy and HTVM combine memory allocation and tiling, yet primarily target coarse-grained offloading and *sequential* execution. COMB focuses on mapping multiple DNN models simultaneously to leverage graph-level parallelism and does not explore asynchronous tile-centric execution. Map-and-Conquer focuses on collaborative execution on *OS-equipped* MPSoCs, which removes the need for explicit memory management and simplifies layer tiling. Furthermore, it considers a trivial workload offloading based on accelerators' peak performance. MATCH [12], the closest to our work, extends TVM with hardware-aware design-space exploration for HSoCs to assign layers or fused layer groups (patterns) to accelerators. Nonetheless, MATCH still maps at layer/pattern granularity and executes kernels sequentially.

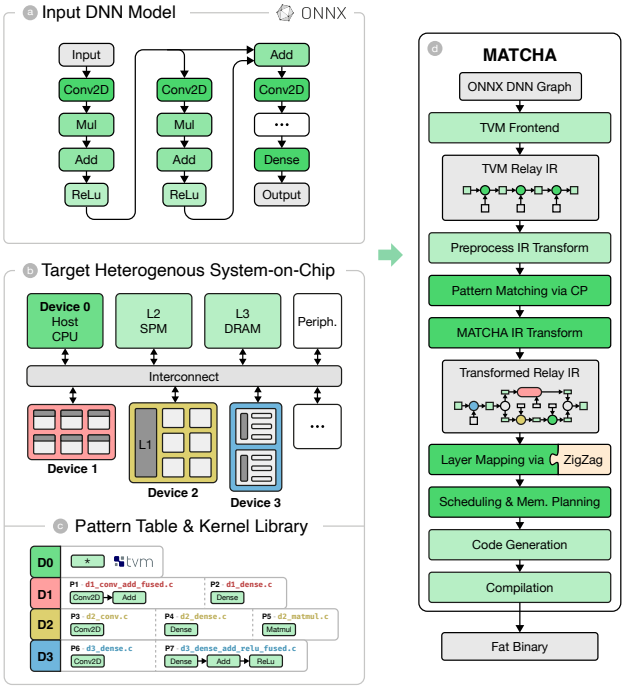


Figure 1: MATCHA deployment framework: inputs (left) and pipeline stages (right).

3 Methodology

In the following, we refer to each execution module (host or accelerator) of the HSoC, capable of running a DNN kernel, as a *device*. The overall MATCHA[1] pipeline is depicted in Fig. 1. MATCHA takes as input: a DNN model in ONNX format (Fig. 1a), a description of the target HSoC (Fig. 1b), including available devices and memory capacities, and a catalogue of supported layer patterns together with the kernels provided by each device (Fig. 1c)

The pipeline begins by importing the ONNX model into the TVM [6] Relay intermediate representation (IR), a directed graph in which nodes represent tensors or primitive operators and edges denote data flow. A pre-processing pass applies useful graph transformations to the Relay IR, such as constant folding or dead nodes removal.

In MATCHA, the pattern-matching optimization is then performed by a constraint-programming (CP) optimizer that maps each layer tile to the most appropriate computing device; details are discussed in Section 3.1. Based on the optimizer's output, MATCHA rewrites the IR: operators are split and tiled according to the chosen tiling strategy, fused kernel supernodes, and auxiliary operators (e.g., tensor slicing and concatenation) are added to the IR, and the graph is partitioned to map operators to devices. The transformed graph is subsequently subject to scheduling and memory planning (Section 3.2). In the scheduling phase, the mapped tiles to each device are scheduled to meet local multi-level memory requirements: loop tiling, unrolling, and ordering are optimized using the ZigZag DNN layer mapper [18]. Finally, as described in Section 3.3,

[1]MATCHA is available open-source at: https://github.com/eml-eda/match.

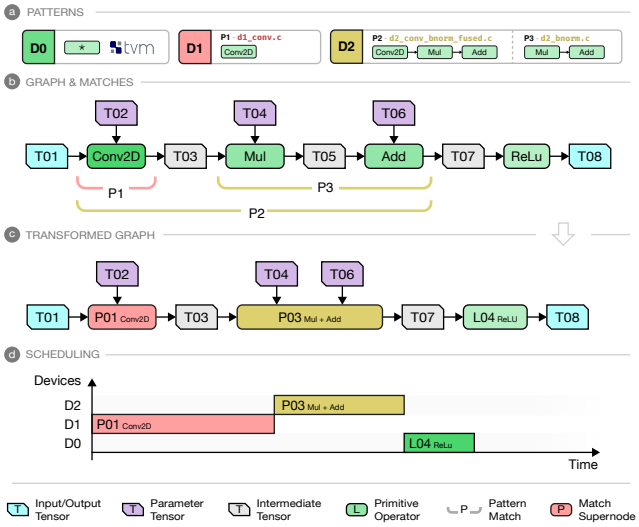


Figure 2: Conventional pattern matching in a heterogeneous system resulting in sequential execution across devices.

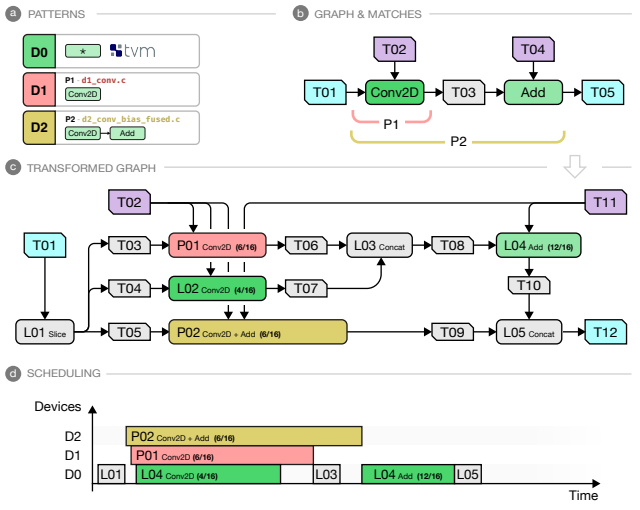


Figure 3: MATCHA's tile-centric pattern matching and tiling with asynchronous execution across heterogeneous devices.

MATCHA emits C code for the host and accelerators and produces a multi-architecture binary image using target-specific compilation tools.

3.1 Pattern Matching and Layer Tiling

We model the Relay IR as a directed graph $\mathcal{G}_{\text{IR}} = (\mathcal{V}, \mathcal{E})$, where nodes (vertices) $\mathcal{V}$ are either tensors or primitive operators, and edges $\mathcal{E}$ represent data dependencies. An operator pattern is modeled as a path (chain) graph $p = (\mathcal{V}_p, \mathcal{E}_p)$ of length ℓ_p with node-level constraints (e.g., operator types, tensor shapes and layouts, data type and precision). In conventional deployment frameworks, pattern matching seeks injective graph homomorphisms $h : \mathcal{V}_p \to \mathcal{V}$ that satisfy these constraints; then, each selected match replaces the corresponding subgraph with a fused supernode. Figure 2b illustrates an IR example and highlights possible pattern matches; Figure 2c shows the resulting IR after matching decisions and graph transformation. As shown in Figure 2d, when the original graph has no branches, this approach leads to sequential execution across devices as each fused supernode is mapped to a single accelerator.

MATCHA generalizes this paradigm to maximize utilization on heterogeneous SoCs by introducing a tile-centric formulation that (i) allows pattern matches to cover integer numbers of tiles of underlying operators, and (ii) enables asynchronous and parallel execution of tiles across devices. The optimizer jointly selects pattern matches, tile allocations, and device mappings to minimize end-to-end latency.

Each IR operator $v \in \mathcal{V}$ is partitioned into an integer number T_v of tiles (e.g., feature map rows for convolutional layer or output neurons in dense layers). Let Ops_v denote the total arithmetic operation count of operator v. The available kernel pattern library is denoted by $\mathcal{P}$ (Fig. 3a) and each pattern $p \in \mathcal{P}$ is associated with device d_p, an efficiency factor $\eta_p \in (0, 1]$, and a fixed per-invocation time overhead δ_p. In addition to accelerator-specific kernels, MATCHA always includes a *wildcard* pattern that covers any individual operator with a TVM-generated kernel ensuring that unmatched tiles can be executed by the host.

Each match m of the pattern p defines a mapping $h_m : \mathcal{V}_p \to \mathcal{V}$ that identifies which IR nodes are involved. For each match, MATCHA introduces a nonnegative integer decision variable $t_{p,m}$ that represents the number of tiles assigned to it. The same $t_{p,m}$ applies to every IR operator v in the image of h_m. Tile conservation across all instantiated matches is enforced as:

$$\sum_{p \in \mathcal{P}} \sum_{m \in \mathcal{M}_p} \mathbb{I}_{v,p,m}\, t_{p,m} \;=\; T_v \qquad \forall v \in \mathcal{V}, \tag{1}$$

where $\mathbb{I}_{v,p,m} = 1$ if $v \in h_m(\mathcal{V}_p)$ and 0 otherwise and $\mathcal{M}_p$ is the set of matches of p.

The optimizer estimates the latency of each instantiated match using a lightweight analytical model. For a fused match m of pattern p on device d_p to which $t_{p,m}$ tiles are assigned, latency is calculated as:

$$L_{p,m}(t_{p,m}) \;=\; t_{p,m} \cdot \left(\sum_{u \in \mathcal{V}_p} \frac{\text{Ops}_{h_m(u)}}{T_{h_m(u)}} \right) \frac{\alpha_{d_p}}{\eta_p} \;+\; \delta_p, \tag{2}$$

where α_d denotes the device speed parameter (time per arithmetic operation, i.e., the inverse of peak operations per time unit). The inner sum yields the per-tile arithmetic work of the fused pattern under match m. Multiplying by $t_{p,m}$ produces the total arithmetic work executed by the match supernode. The device speed, kernel efficiency, and fixed overhead translate assigned work into execution time. A more advanced model that takes into account tile splitting overhead and specific operator geometry is currently not used, but can be explored in future work.

Because the tile variables $t_{p,m}$ simultaneously determine which supernodes are instantiated, how much work they perform, and what work remains for other devices, MATCHA frames joint pattern-matching, device-allocation, and *platform-scheduling* (as tiling-to-device assignment) as a constrained optimization problem whose

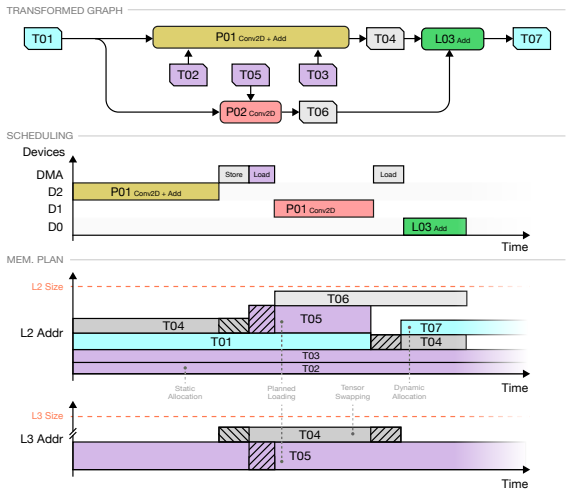


Figure 4: Scheduling and memory plan example with different types of tensor allocation.

objective is the overall makespan. The linear dependence of the latency formulas on the tile variables makes the cost evaluation tractable within CP. Additional constraints encode device memory capacities and concurrency feasibility. Fig. 3c shows a representative pattern matching solution: assume a *Conv2D* kernel on device 1 and a fused *Conv2D+Add* kernel on device 2; both *Conv2D* and the subsequent *Add* are partitioned into $T = 16$ tiles. Two match instances of the fused pattern are selected: one match (pattern *Conv2D*) assigned 6 tiles to device 1, and one match (pattern *Conv2D+Add*) assigned 6 tiles to device 2; the remaining 4 *Conv2D* tiles and the remaining 12 *Add* tiles are handled by host.

After the CP optimizer selects a tiling and matching configuration, MATCHA instantiates the corresponding fused supernodes and helper operators in the IR (e.g., slice, concat, etc.) and proceeds to a final *device-specific scheduling* and memory-planning stage (Section 3.2). Notice that layer-device assignment and asynchronous execution becomes a corner case of this optimization, in which each tile of the layers is assigned to the same device.

3.2 Mapping, Scheduling and Memory Planning

This stage refines the device-specific scheduling, computes tensor placements and lifetimes, and produces a temporally ordered execution plan that respects data dependencies and memory constraints.

The optimizer models the joint scheduling and memory-allocation problem as a two-dimensional bin-packing instance (time × address) on the on-chip main memory (L2) and the larger off-chip memory (L3). Tensor lifetimes induce temporal occupancy intervals; the planner may choose among allocation strategies: (i) static allocation: assign a persistent address in L2 and keep the tensors resident throughout execution (always alive); (ii) dynamic allocation with optional swapping: evict intermediate variable tensors to L3 after production and reload them later when required; (iii) planned loading: load a large parameter tensor from L3 on demand. When swapping or planned loading is selected, data movement between L2 and L3 is modeled and accounted for in the makespan: transfers are performed by DMA engines and are serialized in the current model (i.e., DMA transfers do not overlap with computation). Fig. 4 shows a simple memory plan and scheduling example where each node is assigned to a different device: even if enabled, asynchronous execution can not happen, given that the optimizer forces a sequential scheduling to fit the constrained memory (T04, T05, and T06 do not need to be stored simultaneously).

At this stage, the computation latency estimates are refined to reflect lower-level mapping decisions. Operators assigned to an accelerator frequently cannot place all working data in the accelerator's local scratchpad (L1); therefore additional tiling between L1 and L2 (e.g., loop unrolling, ordering) is applied and evaluated using a device-level mapper and cost model. MATCHA uses the ZigZag LOMA [18, 26] mapper together with its cost model to select L1/L2 tilings and memory access schedules; the resulting per-node latency estimates (including any necessary data movement) are considered to perform the global schedule.

Again, CP optimization is used. It enforces device memory capacity constraints, concurrency limits (e.g., each device can run one kernel at a time), and data-dependency precedence, and minimizes the makespan subject to these constraints. The optimizer outputs (i) an execution schedule that orders kernel invocations and DMA operations, (ii) a memory plan that specifies address assignments and swap points, together with (iii) a detailed layer-to-device mapping along with the L1/L2 tiling choices obtained through ZigZag.

These artifacts are then translated into low-level code generation primitives and passed to the backend (see Section 3.3) to produce the binaries. The final result of this stage is an executable schedule and memory plan that together enable asynchronous, parallel execution of the DNN across the heterogeneous SoC.

3.3 Multi-device Code Generation

The code-generation stage consumes the transformed Relay IR together with the execution schedule, the planned tensor addresses, and the per-layer tiling solution (loop tiling, unrolling and ordering). MATCHA uses a collection of Mako templates [2] to synthesize C implementations for each executable node (layer) in the transformed graph. In the HSoC platform specification, the user additionally provides a small set of platform-level C APIs that the generator may call for device-specific services such as device initialization, DMA submission and completion, inter-core synchronization, and allocation/management of scratchpad buffers.

MATCHA supports targets that expose symmetric multiprocessing (SMP) semantics. When available, platform APIs are used to obtain core identifiers and to implement intra-device synchronization and work distribution across cores. Alongside per-node kernel code, the generator emits a graph runtime that executes on the host. The host-side runtime exposes a single entry point that starts inference for a given input tensor; on invocation it initializes the planned memory pool, writes the tensor base addresses according to the memory plan, and drives execution by issuing kernel invocations and DMA operations in the temporal order produced by the optimizer. The host runtime is responsible for orchestrating cross-device execution, honoring data dependencies, and invoking any host-resident layer implementations.

MATCHA is designed for heterogeneous SoCs in which accelerators may implement a different ISA from the host and therefore require a distinct compiler toolchain and runtime. To accommodate

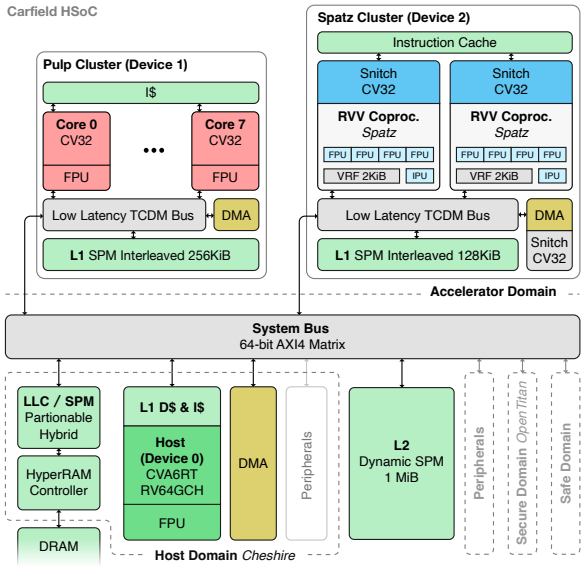


Figure 5: Use case Carfield HSoC considered for evaluation.

such platforms (including bare-metal edge SoCs that lack a full operating system), MATCHA can emit a lightweight device-side runtime for each accelerator. The device runtime implements a simple dispatch loop: it waits for a task descriptor from the host together with input tensor addresses, executes the corresponding layer kernel with the locally mapped tiling and memory plan, and then signals completion back to the host. This separation permits each device image to be compiled with the toolchain appropriate to its ISA and execution environment.

The host and devices communicate and synchronize using one of two supported methods: (i) polling, in which the host and/or device busy-wait on shared status flags or memory locations, or (ii) event/interrupt-based notification, in which the host signals a device via an interrupt or event and the device signals completion likewise. The event/interrupt mechanism enables low-overhead, *asynchronous overlap of each device*; the host runtime uses the optimizer's schedule together with online device availability to correctly dispatch tasks and execute host-resident layers.

After code generation, host and device sources are compiled with the appropriate compilers and toolchains. The compilation outputs are packaged into a single multi-architecture binary image. At startup the host loads and initializes each device runtime; once initialization completes the devices are ready to accept tasks according to MATCHA's generated schedule.

4 Experimental Results

We implement MATCHA in Python and rely on Apache TVM [6] for host-side kernels, the ZigZag LOMA [18, 26] mapper for device-level tiling and mapping, Mako for template-based code generation, and OR-Tools for constraint programming solvers. To evaluate MATCHA, we use the open-source Carfield heterogeneous SoC [9] as our target platform. The design was deployed on a Xilinx VCU118 FPGA and experiments were conducted at 50 MHz clock frequency.

Carfield is a time-predictable, HSoC intended for mixed-criticality, AI-enhanced sensor-processing and control workloads (e.g., automotive and space applications). The configuration used in our experiments is illustrated in Fig. 5. The SoC comprises a host domain based on the Cheshire platform [19], which integrates a dual-core RV64GCH 64-bit RISC-V CPU, and an accelerator domain containing two heterogeneous accelerator clusters. Conventional peripherals include a system DMA that can be used for L2-L3 transfers; the system interconnect is a 64-bit AXI4 bus. A 1 MiB on-chip, dynamically configurable L2 scratchpad is shared among domains and is exposed to the interconnect with a 128-bit-per-cycle data path.

The accelerator domain contains two accelerators. The first is a PULP cluster [23] composed of eight 32-bit RISC-V (RI5CY) cores with floating-point ISA extensions. The cluster provides a 256 KiB L1 scratchpad and a dedicated DMA engine for L2–L1 transfers.

The second accelerator is a Spatz cluster [22], consisting of two compact scalar RISC-V cores that control two RISC-V Vector Units (RVVUs). Each RVVU implements RVZve64d semantics with a vector length VLEN = 512 bits and supports data formats from FP8 up to FP64, bfloat16, integer types, and mixed-precision primitives (including sum-of-dot-product, sdotp). The Spatz cluster includes a 128 KiB L1 scratchpad and its own DMA engine for L2–L1 transfers.

The Carfield design also exposes a Platform-Level Interrupt Controller (PLIC) for centralized host interrupt handling and per-device interrupt-triggering mailbox units that interface the host to each accelerator cluster and other devices. MATCHA can take advantage of these asynchronous mailbox notifications to implement low-latency task dispatch and completion signaling, enabling parallelism and overlap between host and accelerator execution.

This heterogeneous HW configuration with diverse cores and accelerators, shared on-chip scratchpad, and explicit DMA engines, provides a representative testbed for MATCHA's tile-centric pattern matching, device allocation, and schedule-aware memory planning.

We use MATCH [12] as the state-of-the-art baseline because it can be easily extended to support the target platform, allowing a fair comparison. No other compilers from Table 1 support the Carfield architecture. All the following experiments employ the same set of patterns and kernels across the evaluated toolchains, considering FP16 data precision.

Microbenchmarks. Figure 7 reports floating-point operations per second (FLOPS) for three representative DNN building blocks evaluated on the Carfield HSoC: the first residual block of ResNet-50 (three convolutional layers), the first block of ResNeXt-50 [30], and a Transformer encoder layer (multi-head attention, feed-forward, and normalization) with hidden size 128. Compared to the TVM host-only baseline, MATCHA achieves speedups ranging from 11.04× to 40.34× across these blocks. When only enabling asynchronous execution and leveraging graph-level parallelism, MATCHA yields latency reductions of 18.22%, 7.21%, and 9.47% for the ResNet-50, Transformer, and ResNeXt-50 blocks, respectively, relative to MATCH allocating each layer on the best-performing accelerator. Enabling MATCHA's tile-centric optimization further improves load balancing across devices, reducing latency by 35.02% on the ResNet-50 block, 17.55% on the ResNeXt-50 block, and 23.65% on

Table 2: MLPerf Benchmark results. MATCH employs always the best accelerator for each layer (Spatz / Pulp).

			TVM			MATCH			MATCHA (No Tiling)			MATCHA (Ours)		
Model	MACs	Params	Cycles	Runtime	FLOPS	Cycles	Runtime	FLOPS	Cycles	Runtime	FLOPS	Cycles	Runtime	FLOPS
AutoEncoder	0.27M	268k	5.03M	100.2 ms	5.36M	1.01M	20.1 ms	26.7M	1.01M	20.1 ms	26.7M	0.673M	**13.4 ms**	40.1M
DS-CNN	2.8M	22.6k	30.3M	604.6 ms	9.29M	6.58M	**131.1 ms**	42.9M	6.58M	**131.1 ms**	42.9M	6.58M	**131.1 ms**	42.9M
MobileNet	7.9M	210k	157.4M	3137.8 ms	5.07M	24.4M	**486.7 ms**	32.7M	24.4M	**486.7 ms**	32.7M	24.4M	**486.7 ms**	32.7M
ResNet18	12.8M	78k	200.2M	3991.7 ms	6.43M	22.9M	456.6 ms	56.2M	19.9M	395.9 ms	64.9M	16.3M	**325.1 ms**	79.0M

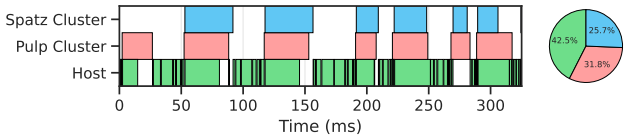


Figure 6: ResNet inference profiling timeline (left) and the execution time breakdown (right) across different devices.

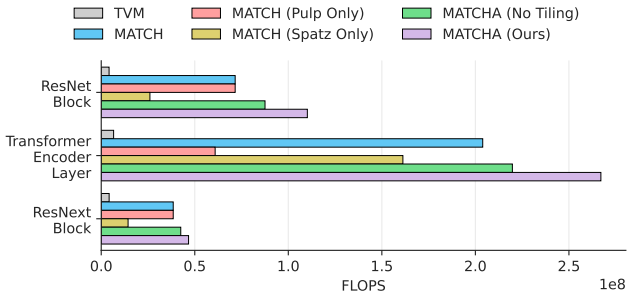


Figure 7: FLOPS comparison for DNN benchmark blocks.

the Transformer encoder layer compared to MATCH. Overall, the achievable speedups are bounded by the performance imbalance between heterogeneous accelerators and by the fraction of operators in each block that are not supported and must execute on the host (with two homogeneous accelerators and with all operators offloaded, the maximum speedup compared to using a single one would be 50%). However, they demonstrate that tile-centric asynchronous execution is crucial to maximize HSoCs HW utilization.

MlPerf. Table 2 reports inference cycles, latencies and FLOPS of MLPerf Tiny [1] benchmark models for TVM, MATCH (sequential execution choosing the fastest device for each operator), MATCHA with pattern tiling disabled (asynchronous layer offloading only), and MATCHA with pattern tiling enabled. In these experiments, patterns including convolutional layers are tiled along the rows of the output feature map; this tiling requires inserting input slicing operations and output concatenations, which introduce additional runtime overhead. For compact networks dominated by depthwise convolutions (e.g., DS-CNN and MobileNet) we observe no latency improvement relative to the baseline. Depthwise convolutions exhibit low arithmetic intensity with respect to traditional convolutional layers; therefore, the overheads of slicing and concatenation outweigh any computational benefit from tiling. This result highlights the importance of optimizing slice/concat primitives and automatically identifying the optimal tiling dimension, as discussed in Sec. 5. For ResNet-18 trained on CIFAR-10, MATCHA already benefits from asynchronous cross-device execution of residual branches and yields a 13.3% latency reduction. Enabling MATCHA's intra-layer tiling to increase device utilization produces a larger reduction, reaching 28.8% relative to MATCH. The AutoEncoder model, commonly used for anomaly detection, consists of a chain of fully connected layers, which offer little opportunity for graph-level parallel execution. Nonetheless, MATCHA's pattern-tiling capability achieves a 33.3% latency reduction versus MATCH. We tile fully connected layers across the output-neuron dimension; since the corresponding weight tiling can be folded into the offline weight layout, this strategy incurs essentially zero runtime overhead. Overall, MATCHA outperforms the TVM host-only compilation by factors ranging from 4.61× to 12.28× across the evaluated networks. Fig. 6 shows the ResNet inference profiling timeline and the execution time breakdown showing a balanced workload distribution; concurrent execution is interleaved by unmatched activation operators and helper operators running exclusively on the host.

5 Conclusions

We presented MATCHA, a tile-centric deployment framework that employs constraint programming to perform pattern matching, device allocation, scheduling, and memory planning, enabling asynchronous DNN inference on heterogeneous SoCs. On the representative Carfield SoC, MATCHA yields end-to-end speedups versus prior SoA methods of 35%, demonstrating that tile-aware pattern matching and asynchronous execution are effective approaches to increase utilization in HSoCs.

While already providing latency reduction, MATCHA still presents some limitations which will be addressed in future work to unlock even higher utilization: (i) the flow is split between the coarse pattern matcher and the detailed scheduling per accelerator to reduce the overall search space. However, jointly optimizing pattern matching, per-device layer mapping, and low-level tiling in a single cost model (potentially with hardware-in-the-loop parameter calibration) could unlock further performance improvement. (ii) Helper-operator overheads (slice/concat) are not accurately modeled in latency models. Moreover, these operators can be eliminated by view-based tiling or carefully planned contiguous tensor address placements. Lastly, the assessment of the energy improvement of asynchronous execution is not evaluated.

Acknowledgments

This work was supported by European Commission through the Chips Joint Undertaking under grant agreement number 101139790 (ECS4DRES). Part of this work was carried out while Enrico Russo was visiting the Integrated Systems Laboratory (IIS), ETH Zurich.

References

[1] Colby Banbury, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Kiraly, Pietro Montino, David Kanter, Sebastian Ahmed, Danilo Pau, et al. 2021. MLPerf Tiny Benchmark. *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks* (2021).

[2] Michael Bayer. n.d.. Mako Templates for Python. https://www.makotemplates.org/

[3] Tal Ben-Nun and Torsten Hoefler. 2019. Demystifying Parallel and Distributed Deep Learning: An In-depth Concurrency Analysis. *ACM Comput. Surv.* 52, 4, Article 65 (Aug. 2019), 43 pages. doi:10.1145/3320060

[4] Halima Bouzidi, Mohanad Odema, Hamza Ouarnoughi, Smail Niar, and Mohammad Abdullah Al Faruque. 2023. Map-and-conquer: Energy-efficient mapping of dynamic neural nets onto heterogeneous mpsocs. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[5] Alessio Burrello, Angelo Garofalo, Nazareno Bruschi, Giuseppe Tagliavini, Davide Rossi, and Francesco Conti. 2021. DORY: Automatic end-to-end deployment of real-world DNNs on low-cost IoT MCUs. *IEEE Trans. Comput.* 70, 8 (2021), 1253–1268.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[7] Ismet Dagli, Alexander Cieslewicz, Jedidiah McClurg, and Mehmet E Belviranli. 2022. Axonn: Energy-aware execution of neural network inference on multi-accelerator heterogeneous socs. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 1069–1074.

[8] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, et al. 2021. Tensorflow lite micro: Embedded machine learning for tinyml systems. *Proceedings of machine learning and systems* 3 (2021), 800–811.

[9] Angelo Garofalo, Alessandro Ottaviano, Matteo Perotti, Thomas Benz, Yvan Tortorella, Robert Balas, Michael Rogenmoser, Chi Zhang, Luca Bertaccini, Nils Wistoff, Maicol Ciani, Cyril Koenig, Mattia Sinigaglia, Luca Valente, Paul Scheffler, Manuel Eggimann, Matheus Cavalcante, Francesco Restuccia, Alessandro Biondi, Francesco Conti, Frank K. Gurkaynak, Davide Rossi, and Luca Benini. 2025. A Reliable, Time-Predictable Heterogeneous SoC for AI-Enhanced Mixed-Criticality Edge Applications. *IEEE Transactions on Circuits and Systems II: Express Briefs* 72, 11 (2025), 1625–1629. doi:10.1109/TCSII.2025.3591225

[10] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. 2021. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 769–774. doi:10.1109/DAC18074.2021.9586216

[11] Sukhpal Singh Gill, Muhammed Golec, Jianmin Hu, Minxian Xu, Junhui Du, Huaming Wu, Guneet Kaur Walia, Subramaniam Subramanian Murugesan, Babar Ali, Mohit Kumar, Kejiang Ye, Prabal Verma, Surendra Kumar, Felix Cuadrado, and Steve Uhlig. 2024. Edge AI: A Taxonomy, Systematic Review and Future Directions. *Cluster Computing* 28, 1 (Oct. 2024), 18. doi:10.1007/s10586-024-04686-y

[12] Mohamed Amine Hamdi, Francesco Daghero, Giuseppe Maria Sarda, Josse Van Delm, Arne Symons, Luca Benini, Marian Verhelst, Daniele Jahier Pagliari, and Alessio Burrello. 2025. MATCH: Model-Aware TVM-Based Compilation for Heterogeneous Edge Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2025), 1–1. doi:10.1109/TCAD.2025.3556967

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[14] Qijing Huang, Minwoo Kang, Grace Dinh, Thomas Norell, Aravind Kalaiah, James Demmel, John Wawrzynek, and Yakun Sophia Shao. 2021. Cosa: Scheduling by constrained optimization for spatial accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 554–566.

[15] Sheng-Chun Kao and Tushar Krishna. 2020. Gamma: Automating the hw mapping of dnn models on accelerators via genetic algorithm. In *Proceedings of the 39th International Conference on Computer-Aided Design*. 1–9.

[16] Mukul Lokhande, Gopal Raut, and Santosh Kumar Vishvakarma. 2025. Flex-PE: Flexible and SIMD Multiprecision Processing Element for AI Workloads. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 33, 6 (2025), 1610–1623. doi:10.1109/TVLSI.2025.3553069

[17] Martin Maas, Ulysse Beaugnon, Arun Chauhan, and Berkin Ilbeyi. 2022. 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*. 123–137.

[18] Linyan Mei, Pouya Houshmand, Vikram Jain, Sebastian Giraldo, and Marian Verhelst. 2021. ZigZag: Enlarging joint architecture-mapping design space exploration for DNN accelerators. *IEEE Trans. Comput.* 70, 8 (2021), 1160–1174.

[19] Alessandro Ottaviano, Thomas Benz, Paul Scheffler, and Luca Benini. 2023. Cheshire: A lightweight, linux-capable risc-v host platform for domain-specific accelerator plug-in. *IEEE Transactions on Circuits and Systems II: Express Briefs* 70, 10 (2023), 3777–3781.

[20] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. 2019. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 304–315.

[21] Jeman Park, Misun Yu, Jinse Kwon, Junmo Park, Jemin Lee, and Yongin Kwon. 2024. NEST-C: A deep learning compiler framework for heterogeneous computing systems with artificial intelligence accelerators. *ETRI Journal* 46, 5 (2024), 851–864.

[22] Matteo Perotti, Samuel Riedel, Matheus Cavalcante, and Luca Benini. 2025. Spatz: Clustering compact RISC-V-based vector units to maximize computing efficiency. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2025).

[23] Michael Rogenmoser, Yvan Tortorella, Davide Rossi, Francesco Conti, and Luca Benini. 2025. Hybrid modular redundancy: Exploring modular redundancy approaches in RISC-V multi-core computing clusters for reliable processing in space. *ACM Transactions on Cyber-Physical Systems* 9, 1 (2025), 1–29.

[24] Enrico Russo, Maurizio Palesi, Giuseppe Ascia, Davide Patti, Salvatore Monteleone, and Vincenzo Catania. 2023. Memory-aware DNN algorithm-hardware mapping via integer linear programming. In *Proceedings of the 20th ACM International Conference on Computing Frontiers*. 134–143.

[25] Moritz Scherer, Luka Macan, Victor JB Jung, Philip Wiese, Luca Bompani, Alessio Burrello, Francesco Conti, and Luca Benini. 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 (2024), 4009–4020.

[26] Arne Symons, Linyan Mei, and Marian Verhelst. 2021. Loma: Fast auto-scheduling on dnn accelerators through loop-order-based memory allocation. In *2021 IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. IEEE, 1–4.

[27] Kodai Ueyoshi, Ioannis A. Papistas, Pouya Houshmand, Giuseppe M. Sarda, Vikram Jain, Man Shi, Qilin Zheng, Sebastian Giraldo, Peter Vrancx, Jonas Doevenspeck, Debjyoti Bhattacharjee, Stefan Cosemans, Arindam Mallik, Peter Debacker, Diederik Verkest, and Marian Verhelst. 2022. 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. 1–3. doi:10.1109/ISSCC42614.2022.9731716

[28] Josse Van Delm, Maarten Vandersteegen, Alessio Burrello, Giuseppe Maria Sarda, Francesco Conti, Daniele Jahier Pagliari, Luca Benini, and Marian Verhelst. 2023. HTVM: Efficient neural network deployment on heterogeneous TinyML platforms. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[30] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. 2017. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1492–1500.

[31] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. 2022. Bolt: Bridging the gap between auto-tuners and hardware-native performance. *Proceedings of Machine Learning and Systems* 4 (2022), 204–216.

[32] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, et al. 2020. Interstellar: Using halide's scheduling language to analyze dnn accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 369–383.

[33] Size Zheng, Siyuan Chen, and Yun Liang. 2023. Memory and computation coordinated mapping of dnns onto complex heterogeneous soc. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.