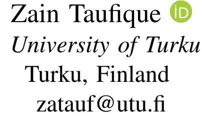
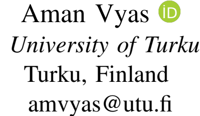
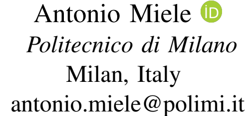
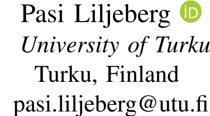
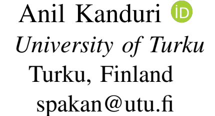
Twill: Scheduling Compound AI Systems on Heterogeneous Mobile Edge Platforms

Zain Taufique 
University of Turku
Turku, Finland
zatauf@utu.fi

Aman Vyas 
University of Turku
Turku, Finland
amvyas@utu.fi

Antonio Miele 
Politecnico di Milano
Milan, Italy
antonio.miele@polimi.it

Pasi Liljeberg 
University of Turku
Turku, Finland
pasi.liljeberg@utu.fi

Anil Kanduri 
University of Turku
Turku, Finland
spakan@utu.fi

***Abstract*—Compound AI (cAI) systems chain multiple AI models to solve complex problems. cAI systems are typically composed of deep neural networks (DNNs), transformers, and large language models (LLMs), exhibiting a high degree of computational diversity and dynamic workload variation. Deploying cAI services on mobile edge platforms poses a significant challenge in scheduling concurrent DNN-transformer inference tasks, which arrive dynamically in an unknown sequence. Existing mobile edge AI inference strategies manage multi-DNN or transformer-only workloads, relying on design-time profiling, and cannot handle concurrent inference of DNNs and transformers required by cAI systems. In this work, we address the challenge of scheduling cAI systems on heterogeneous mobile edge platforms. We present *Twill*, a run-time framework to handle concurrent inference requests of cAI workloads through task affinity-aware cluster mapping and migration, priority-aware task freezing/unfreezing, and Dynamic Voltage/Frequency Scaling (DVFS), while minimizing inference latency within power budgets. We implement and deploy our *Twill* framework on the Nvidia Jetson Orin NX platform. We evaluate *Twill* against state-of-the-art edge AI inference techniques over contemporary DNNs and LLMs, reducing inference latency by 54% on average, while honoring power budgets.**

***Index Terms*—Deep Neural Networks, transformers, Large Language Models, Inference and Compound AI**

I. Introduction

AI applications are rapidly evolving from monolithic models towards Compound Artificial Intelligence (cAI) systems, which integrate multiple task-specific models and components to solve complex problems [1]–[3]. Emerging cAI systems combine Large Language Models (LLMs) with Deep Neural Networks (DNNs) for providing novel services such as conversational language agents [2]–[5], augmented and virtual reality (AR/VR) gear, and interactive autonomous vehicles [6]. cAI systems offer compositional flexibility by selectively chaining multiple transformer (both encoder and generative) and DNN models at run-time [7], [8]. Figure 1(a) shows a conceptual example of a cAI workload designed as a task graph for generating a maintenance report from the input images and text given by the user. In this example, DNN models (`D1`: `VGG-19` and `D2`: `ResNet-152`) are used for image classification, and object detection, transformer models (`T1`: `Bert-base` and `T2`: `Bert-large`) are used for text summarizing and classification, and generative transformers (`T3`: `OPT-350M` and `LLM`: `Deepseek-R1`) are used for reasoning and report generation. Each model is responsible for extracting key features from the given input and sending the output to the subsequent models to perform collaborative tasks. `T1`, `D1`, and `D2` are exclusive inference tasks that can run simultaneously, while `T2`, `T3`, and `LLM` are dependent on the outputs of other models. We deployed the exemplar cAI system on the Nvidia Jetson Orin NX platform. Figure 1(b) shows performance demands (in GFlops) of the exemplar cAI system. In this example, cAI system requires concurrent execution of (i) multiple DNN and a transformer ($t = 0s$ - $0.9s$), (ii) multiple DNN and a generative transformer ($t = 0.9s$ - $1.2s$), and (iii) multiple generative transformers ($t = 1.2s$ - $1.5s$). This demonstrates a high degree

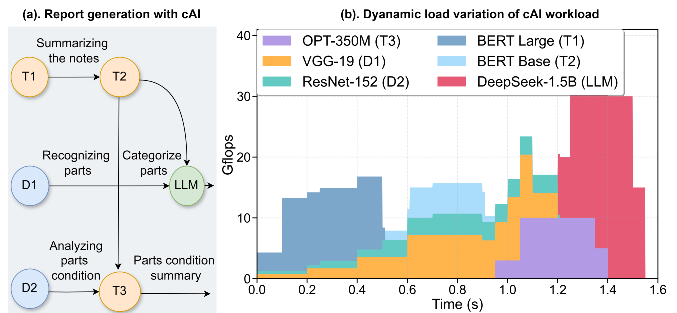


Fig. 1. Exemplar cAI system. (a) Task graph for cAI system chaining multiple models, (b) Run-time workload variation and compute diversity.

of computational diversity and dynamic workload variation with cAI systems. Thus, the primary requirement for implementing cAI systems is concurrent execution of transformers and DNN models, where inference requests are computationally diverse and variable at run-time based on user requirements [7], [8]. On the other hand, there is an increasing demand to deploy cAI inference services on user-end mobile and edge platforms to address the latency, bandwidth, and privacy challenges of the cloud infrastructure [9]. However, running cAI workloads on heterogeneous mobile edge platforms poses significant challenges in scheduling concurrent execution of transformers and DNNs while reducing inference latency within the power constraints [10].

Despite being extremely resource-constrained, mobile edge platforms support AI inference services through powerful mobile GPUs and domain-specific Deep Learning Accelerators (DLAs) [9]–[11]. For example, smart phones [12], smart glasses (e.g. Ray-ban Meta [13]) and AR gear (e.g. Apple Vision Pro [14]) etc., integrate GPUs and DLAs, such as Neural Processing Units (NPUs) or Tensor Processing Units (TPUs) [15]. Existing edge AI orchestration techniques have optimized scheduling of multi-DNN workloads [11], [16]–[19] and transformer models [20], [21] on mobile platforms with GPUs and DLAs. Some of these techniques do not consider transformer models and/or DLAs [16]–[19], [22]. Using these techniques for cAI systems results in significantly higher latency and power consumption, suffering from shared resource contention by mapping both DNN and transformer models on the GPU cluster. Techniques that consider transformer models and DLAs [10], [21], [23] ignore the lack of operation-level support for transformers on DLAs. Existing DLAs are optimized specifically for DNNs and do not support the majority of key operations required for transformer inference [24], [25]. Executing a transformer inference on DLAs leads to frequent GPU *fallbacks*, i.e., moving the execution of operations that are unsupported on the DLA to the GPU. This causes heavy memory access overhead and results in significantly higher inference latency.

With a lack of operator support on DLA, existing techniques execute transformer models inevitably on the GPU [21], [23]. Running transformers on the GPU is a fair choice, as long as a standalone transformer model inference is required. However, cAI workloads require concurrent execution of multiple DNNs and transformers. A trivial solution is to map DNNs on DLA and transformers on GPU; however, this approach is suitable only when the order of inference requests is known at design time. For example, consider a scenario where a transformer inference request arrives when a DNN is already running on the GPU. In this case, the transformer inference request is either queued until the DNN inference task is completed, or both DNN and transformer run concurrently on the GPU, suffering shared resource contention. With no consideration on operator-level support and the need for concurrent DNN-transformer execution, existing edge AI inference techniques are not adaptable for cAI workloads.

Orchestrating cAI workloads requires a run-time adaptive scheduling strategy that considers dynamic inference requests and migrates diverse inference tasks among GPU and DLA clusters to minimize overall inference latency. In this work, we address the challenge of scheduling cAI workloads through model affinity and priority-aware task-to-cluster mapping and migration. We present *Twill*, a novel framework for run-time partitioning, distribution, and scheduling of cAI workloads for heterogeneous mobile edge platforms. We analyze each inference request to determine the affinity of a model towards a cluster and schedule concurrent inference requests on feasible clusters to minimize latency. Our approach adaptively migrates inference requests among feasible clusters and/or freezes inference requests based on priority to accommodate other concurrently running inference tasks. Further, we monitor the run-time power consumption and actuate Dynamic Voltage/Frequency Scaling (DVFS) to honor Thermal Design Power (TDP) constraint. We implemented and deployed the *Twill* framework on the Jetson Orin NX embedded platform and evaluated over contemporary cAI workloads comprising of widely used DNNs (`VGG-19`, `ResNet-152`, and `EfficientNet-b4`), transformer (`Bert-base`, `Bert-large`, `ViT-base`, and `ViT-large`) models, and LLMs (`Deepseek-R1`, `Gemma-3`). Our novel contributions are:

- *Twill*, a run-time framework for partitioning, distribution, and scheduling of cAI workloads on heterogeneous mobile edge platforms.
- Online heuristic model for profiling DNN, transformer, and LLM inference requests to determine affinity of the inference task towards a compute cluster (GPU/DLA).
- Inference task-to-cluster mapping and migration, and priority-aware task freezing for concurrent execution of DNN, transformer, and LLM inference requests, and DVFS actuation for power capping.
- Evaluation of *Twill* on commercial edge AI platform Jetson Orin NX using contemporary cAI workloads.

Manuscript Organization: Section II provides background and motivation for our proposed approach, Section III presents an overview of our run-time management framework infrastructure, Section IV presents an evaluation of our proposed solution against other relevant strategies, followed by conclusions in Section V.

II. Background and Motivation

A. Impact of scheduling on cAI workloads

Figure 2 presents a realistic cAI workload example of a multi-linguistic translation application on a Virtual Reality (VR) gear

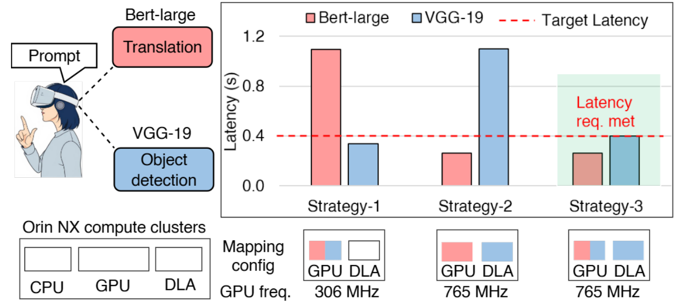


Fig. 2. Inference latency of cAI system across different execution strategies

platform that uses the `VGG-19` [26] model to detect the text and `Bert-large` transformer [27] for translation to a required language. For simplicity of demonstration, we chose cAI workload with two models that have no dependencies; in general, cAI workloads can be composed of several diverse models with data dependencies. In this example, the user generates run-time inference requests based on his/her activity, and the arrival time of these requests is unknown to the baseline platforms at design time. In Figure 2, we compare three workload scheduling strategies for inferring `VGG-19` and `Bert-large` on the Nvidia Jetson Orin NX platform [28] that includes CPU, GPU, and DLA clusters. In this example, we set the batch size of `VGG-19` model to 32 images, the input sequence length of `Bert-large` to 128 tokens, and the minimum inference latency to 400ms, representing a real-world scenario. *Strategy-1* is representative of existing multi-DNN scheduling strategies [16]–[18] that map both applications on GPU. Here, `Bert-large` has to wait for the execution of `VGG-19` to use the GPU resources exceeding the minimum latency requirement. *Strategy-2* maps `VGG-19` to DLA and `Bert-large` to GPU while increasing the GPU frequency. Here, `Bert-large` meets the latency requirements while `VGG-19` is lagging due to the low performance of DLA. *Strategy-2* is representative of state-of-the-art edge AI inference techniques that handle transformer models [10], [22], [23]. Finally, *Strategy-3* (representative of proposed *Twill* approach) successfully meets the latency requirements by (i) partitioning and mapping `VGG-19` on DLA, (ii) mapping `Bert-large` on GPU, and (iii) re-allocating the GPU resources to `VGG-19` once `Bert-large` finishes execution. It should be noted that the scheduling strategies primarily map the inference requests to GPU and DLA clusters, since CPUs have extremely high latency for such computationally-intensive workloads.

B. Dynamics of scheduling DNN-Transformer concurrently

Scheduling DNN and transformer models concurrently requires joint actuation of cluster migration, run-time task freezing/unfreezing, and *DVFS*. We demonstrate the efficacy of each knob actuation setting under different workload scenarios and scheduling strategies. We use DNNs `ResNet-152` and `VGG-19`, and transformer `Bert-base`, which are run on the Jetson Orin NX platform.

Scenario-1. Figure 3(a) shows workload scenario where a DNN model `ResNet-152` is running on GPU and inference request for transformer `Bert-base` arrives at $t = 0.4s$. The SoA approaches [10], [17], [22], [23] lack affinity awareness, and map the incoming `Bert-base` on the available DLA cluster. However, `Bert-base` falls back to the GPU due to a lack of operator support. At this point, if GPU memory is insufficient to run both `ResNet-152` and `Bert-base`, `Bert-base` is queued until `ResNet-152` finishes execution. This results in higher inference latency of `Bert-base`. If sufficient GPU memory is available,

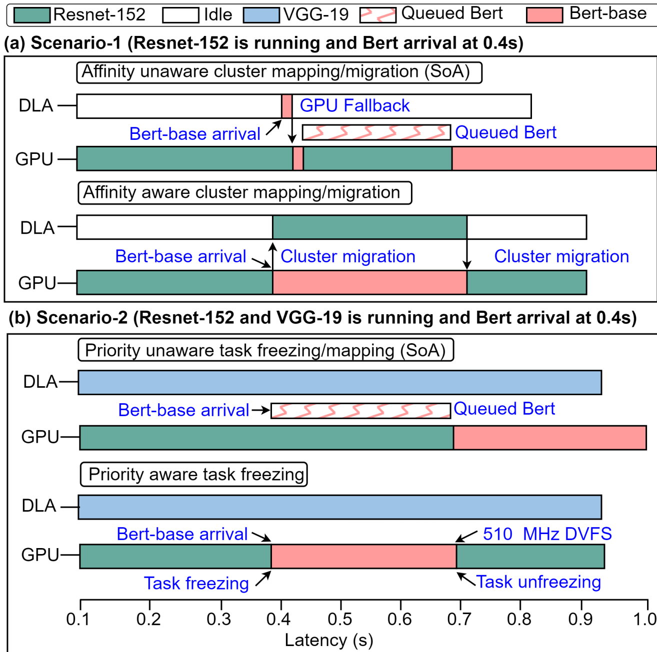


Fig. 3. Various knob actuation scenarios, highlighting lack of priority and affinity adjustments at runtime in SoA strategies.

both inference tasks will run on the GPU; however, this still leads to higher inference latency of both tasks due to shared resource contention. In contrast, our proposed strategy considers the affinity of `Bert-base` towards GPU upon on arrival of the inference request. Consequently, (i) `ResNet-152` is migrated to DLA to accommodate the transformer model on GPU, (ii) `Bert-base` is mapped onto the GPU, (iii) `ResNet-152` is migrated to GPU once `Bert-base` finishes the execution at $t = 0.71s$, freeing up the GPU cluster. This approach minimizes the overall inference latency and waiting time in the execution queue, while maximizing resource utilization.

Scenario-2. Figure 3(b) shows workload scenario where two DNN models viz., `ResNet-152` on GPU and `VGG-19` on DLA are concurrently running, and inference request for transformer `Bert-base` arrives at $t = 0.4s$. In this scenario, both GPU and DLA clusters are busy, and memory utilization is relatively higher. Under this workload, SoA approaches [18], [22], [23], without dynamic mapping/migration capabilities, queue `Bert-base` until `Resnet-152` completes execution on GPU. Inevitably, this leads to higher waiting time in the execution queue and inference latency of `Bert-base`. In this case, our proposed strategy considers both affinity and priority of the `Bert-base` model. We prioritize inference of transformer models that operate on a single contextual text prompt over DNNs that operate on batches of input images. Upon on arrival of the inference request, affinity of `Bert-base` is towards GPU and priority of `Bert-base` is higher. Unlike Scenario-1, `Resnet-152` cannot be migrated to DLA, since DLA is occupied by `VGG-19`. Hence, our approach (i) freezes the execution of `ResNet-152` to accommodate `Bert-base` on GPU, (ii) `Bert-base` is mapped onto the GPU, (iii) `ResNet-152` is unfreezed and resumes execution on GPU once `Bert-base` finishes the execution at $t = 0.71s$, (iv) Frequency of GPU is scaled up within the power constraints to address the performance loss of `ResNet-152` during task freezing. Our approach thus handles dynamic workload variation while minimizing the overall inference latency within the power constraints.

TABLE I
COMPARISON OF RUNTIME TECHNIQUES

Related Work	[21]	[18]	[17]	[16]	[23]	[22]	Twill
Unknown app arrival	✗	✗	✗	✓	✗	✗	✓
Run-time exploration	✓	✓	✗	✓	✗	✗	✓
DLA clusters	✗	✗	✗	✗	✓	✓	✓
Task freezing	✗	✗	✗	✗	✗	✗	✓
Model partitioning	✗	✗	✗	✗	✓	✓	✓
Cluster migration	✗	✗	✗	✗	✗	✗	✓
DVFS tuning	✗	✓	✗	✓	✓	✓	✓
Encoder Transformer	✓	✗	✗	✗	✓	✗	✓
LLMs	✗	✗	✗	✗	✗	✗	✓
DNN workloads	✗	✓	✓	✓	✓	✓	✓

C. Related Work

Widely used strategy for multi-DNN inference on heterogeneous mobile platforms is to partition DNN into convolution blocks and execute them in a pipelined manner among different clusters [19]. A similar approach has also been employed to pipeline transformer models [21], by splitting transformer encoder blocks across different clusters to maximize throughput. However, both these approaches [19], [21] are confined to asymmetric CPUs and do not consider GPUs or DLAs. Edge AI inference techniques such as *Omniboost* [17] extend pipelining across CPU and GPU clusters, while *MOC* [18] proposes a multi-objective deep reinforcement learning agent that controls CPU cores and CPU/GPU frequencies. *Tango* [16] also uses a PPO-based RL agent for multi-DNN workloads to explore accuracy-performance-energy trade-offs to minimize inference latency. Moreover, *Band* [10] demonstrated DNN partitioning across CPU, GPU, and NPU by generating subgraphs for DNNs. However, it only considers basic operators' support for DNNs, overlooking advanced operator support for transformers on DLA. Kim et al. [29] enhance multi-DNN workload scheduling by incorporating ML-based contention prediction, whereas *RankMap* [30] demonstrates a priority-aware multi-DNN manager that prevents DNN starvation under heavy execution loads. The aforementioned strategies primarily target CPU-GPU architectures, focusing on workload partitioning between different clusters. Advanced edge AI inference techniques have used domain-specific accelerators such as DLAs, TPUs, and NPUs. Kim et al. [31] explore CPU, GPU, and NPU resource allocation under varying latency constraints at runtime while minimizing energy consumption of the system. *Axonn* [32] focuses on distributing DNN layers based on energy and performance trade-offs across GPU-DLA clusters. *HaX-CoN* [22] introduces shared memory contention-aware layer grouping and modeling inter-DSA layer transitions using a processor-centric slowdown model, to predict performance degradation. *MapFormer* [23] extends this by incorporating CPU, GPU, and DLA to support multi-DNN workloads, including transformer-based throughput and power estimation with DVFS. Aforementioned multi-DNN scheduling strategies focus on: (i) unimodal DNN workloads for homogeneous tasks such as image classification and text classification, (ii) solutions tailored for fixed design-time systems, and (iii) do not consider run-time arrival of DNNs/Transformers/LLMs workloads. *Twill* addresses these limitations through run-time adaptive scheduling of cAI systems with (i) affinity-aware cluster migration between GPU/DLA, (ii) priority-aware task freezing, and (iii) adaptive DVFS for power capping.

III. TWILL FRAMEWORK

We have designed *Twill* as an online scheduling framework to deploy cAI inference workloads on heterogeneous multi-core platforms hosting CPU, GPU, and DLA. The platform receives run-time inference requests from the user, performs online heuristics

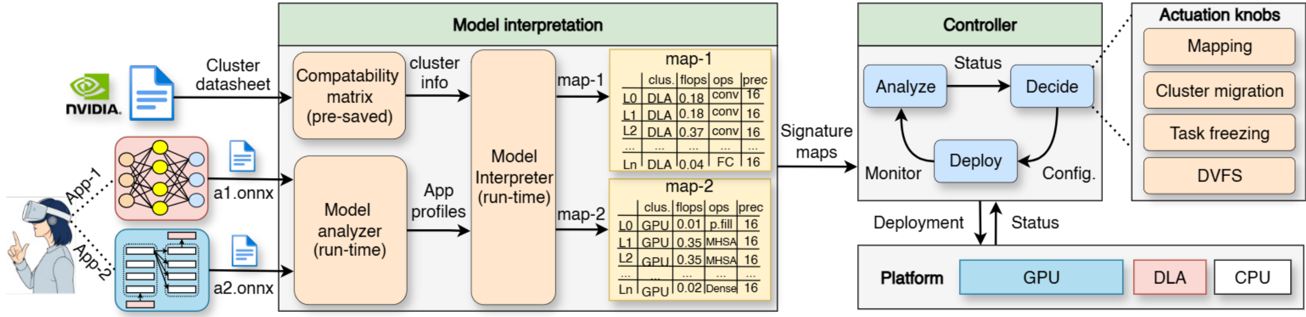


Fig. 4. *Twill* Framework including Model Interpreter to analyze the model-cluster affinity and Controller module performing run-time control knobs actuation.

exploration, and actuates the defined knobs to minimize the overall latency while guaranteeing a given power budget. As shown in Figure 4, the *Twill* framework includes two main modules, namely *Model Interpreter* and *Controller*. The *Model Interpreter* performs online profiling to characterize each inference request. The interpreter includes a *Model analyzer* to extract information from the model files, and a *Compatibility matrix* to check the compatibility of the model layers with the available clusters. For each inference request, the *Model Interpreter* extracts the characterization of the executed model and creates a *Signature map* required for making actuation decisions. The *Controller* monitors the *Signature maps*, and the system status to make actuation decisions based on a set of defined actuation knobs and a heuristic algorithm. Then, the *Controller* applies the workload and system configuration based on the actuation decision and continues monitoring the workload performance. Different modules of the *Twill* framework are detailed in the following.

A. Platform

We consider heterogeneous hardware platforms for edge/embedded computing, including multiple computing clusters; in particular, we focus on the Nvidia Jetson device family hosting CPU, GPU, and DLA. Each cluster is provided with a DVFS knob, with the only exception of the DLA in the platform considered in our experimental results, and the per-board power sensor. The platform runs a standard Operating System (OS) as Linux, providing communication between the software modules to send and receive data and system commands. Specifically, the OS exposes interfaces for hardware control and application execution on various clusters. Platform-specific run-time frameworks are provided for executing applications on GPU and DLA. We restrict the platform's power consumption to the TDP limit, avoiding any physical damage to the board at a high temperature.

B. Workloads

Twill targets the execution of cAI workloads representing a diverse set of DNN and transformer models collaborating to compute a global output. These models execute different cognitive image and text processing tasks, including classification, detection, and generation. Moreover, the inference tasks may be (i) continuous inferences on streaming input data, such as sequences of images taken from a camera, (ii) single generation requests, e.g., of text outputs, or (iii) hybrid. Models in a single cAI workload may have data/precedence dependencies among each other that can be represented as a task graph; in fact, inference of some models is possibly triggered based on the outputs of a preceding model. Even if the overall task graph of the cAI workload is known at design time, the actual execution presents several aspects that are dependent on the specific workload run and are unpredictable beforehand. In particular, the actual inference request to each model is specified at run-time based

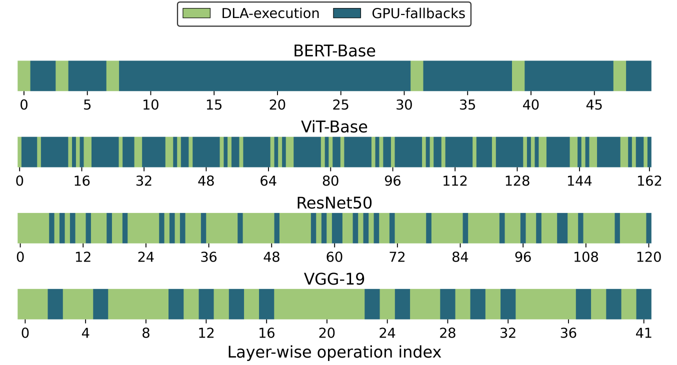


Fig. 5. Supported operations on GPU and DLA clusters of Orin NX platform.

on the user requests or the results of the previous models in the task graph. This means that the execution time of each executed model is highly variable, for instance, based on the length of the input data stream to be processed by a DNN or on the complexity of a generative request. As a consequence, the timing of all dependent models is affected. Therefore, the system experiences an unknown workload composed of multiple running models mixing DNNs, transformers, and LLMs, each one arriving asynchronously.

C. Model Interpretation

We design a model interpretation strategy as an online application profiling mechanism that creates a *Signature map* of the given inference requests for the *Controller* to schedule the cAI workload.

Model analyzer. We use a *Model analyzer* that parses the input *ONNX* file describing the deep learning model using the *ONNX-runtime* library [33] to extract the layer information and metadata of the model. The *Model Analyzer* also receives a user-defined *application priority*, enabling the *Controller* to select suitable candidates for making scheduling decisions while facing resource contention scenarios. *Application priority* is a user-defined parameter that can be configured based on cAI workload requirements. For example, typical cAI systems prioritize user-driven prompt-based LLMs and transformers over DNNs that run on streaming batches of inputs. We designed the *Twill* framework to maneuver the workload scheduling decisions based on the changing application priorities. The *Model analyzer* formulates an application profiling table (`App_profile`) including the extracted model information and application priority. The `App_profile` table includes elaborate model information, including model 1) name, 2) total parameters, 3) total layers, 4) total floating point operations, 5) layer types, 6) layer operation types, 7) layer-level input and 8) output sizes, and 9) activation functions.

Comparability Matrix. We formulate a DLA compatibility matrix (`dla_matrix`) from the official reference manual of Jetson Orin

NX [25]. The matrix includes information about the supported and unsupported operations, precision level, layer types, spatial dimension, batch size, kernel size, and padding and stride ranges. *Twill* requires the cluster compatibility information because the available cluster can have dedicated support for a limited number of model operations. We demonstrate the compatibility of inferring DNN and transformer models on the DLA cluster of Jetson Orin NX in Figure 5. We separately ran inferences of 2 DNN models (`ResNet50` and `VGG-19`) and 2 transformer models (`Bert-base` and `ViT-base`) on the DLA cluster. Figure 5 presents layer-wise operators supported by the DLA and GPU fallback instances for unsupported operators. Most DNN inference operations of `VGG-19` and `ResNet-50` are supported on DLA. For transformer models `Bert-base` and `ViT-base`, DLA supports only the pre-processing and convolution operations, while most of the operations fallback to the GPU cluster. Therefore, the DLA cluster is more suitable to run DNN workloads as compared to the transformers.

The DLA supports fixed-function layers required for DNN inference, including convolution, pooling, activation, and fully connected layers. The DLA does not support the majority of computational operations in transformers, including multi-head attention, layer normalization, and advanced activation such as Gelu (Gaussian Error Linear Unit) [34] [24], [25]. For a model inference, DLAs executes only the supported operators, and *fallbacks* to the GPU for running unsupported operations.

Model Interpreter. The *Model Interpreter* finds the model-cluster affinity based on the knowledge extracted from the application profile and the comparability matrix. The *Model Interpreter* checks the layer-to-cluster affinity of each model through a mechanism shown in Algorithm 1. The algorithm takes as input the application signature (`App_profile`) and DLA compatibility profile (`dla_matrix`). At the start, the *Model Interpreter* extracts the layer information from the application signature table (Line 2) and initializes a compatibility map (Line 3). The algorithm assigns GPU compatibility by default (Line 5) for each layer since GPUs can execute all operation types. The `DLACompatible` function (Lines 6–8) determines if a layer can also execute on DLA by checking three key conditions, (i) the model's precision must match DLA-supported precisions (Lines 11–13), (ii) the operation type must not be in the unsupported layer list (Lines 14–15), and (iii) for convolution and fully-connected layers, additional parameter constraints must be satisfied including kernel size, stride, and padding ranges (Lines 15-19). Finally, the *Model Interpreter* creates a *Signature map* for each model, including the per-layer cluster affinity, layer type, number of floating point operations, precision, and input-output shapes.

D. Controller

The *Twill Controller* is the central scheduling mechanism that monitors the workload and system status, makes actuation decisions on various system-wide settings, and executes the model inferences. Conceptually, the controller is a middleware between the running workload and the hardware architecture. It accesses OS interfaces to monitor system status, in particular the power consumption and the current status of the various clusters. Then, it exploits the heartbeats sent by each running model to monitor their progress.

Control knobs. We defined an advanced set of actuation knobs to allow the controller to regulate workload deployment and execution; they are obtained by acting on the OS interface, opportunistically exploited to override the default OS scheduling and DVFS governors. The knobs include *Affinity-cluster mapping*, *Cluster migration*, *Task freezing*, and *DVFS*. *Affinity-cluster mapping* is used to bind the inference request to a selected cluster; we map at most one model per cluster per time. In *Cluster migration*, the controller splits the model layers and allocates the partitioned workload to a new cluster. *Task freezing* suspends the execution of a running application to free resources for another application. DVFS enables frequency level setting on a selected cluster.

Controller Policy. We designed the *Controller* as a feedback loop that runs in three steps, including *Analyze, Decide*, and *Deploy* phases

Algorithm 1 Layer-Cluster Affinity Mapping

1: **function** GETAPPSIGNATURE($App_profile, dla_matrix$)
2: $layers \leftarrow$ GetLayers($App_profile$)
3: $map \leftarrow \{\}$
4: **for all** $l \in layers$ **do**
5: $map[l] \leftarrow$ {"GPU"} ▷ GPU compatible
6: **if** DLACompatible(l, dla_matrix) **then**
7: $map[l] \leftarrow map[l] \cup$ {"DLA"}
8: **return** $Signature_map$
9: **function** DLACOMPATIBLE(l, dla_matrix)
10: $type \leftarrow$ GetType(l)
11: $prec \leftarrow$ GetPrec($App_profile$)
12: **if** $prec \notin dla_matrix.SupportedPrecs$ **then**
13: **return** `False`
14: **if** $type \in dla_matrix.UnsupportedOps$ **then**
15: **return** `False`
16: **if** $type =$ "Conv2D" $\vee$ $type =$ "FC" **then**
17: **if** $\neg$CheckLimits(l, dla_matrix) **then**
18: **return** `False`
19: **return** `True`

Algorithm 2 Scheduling Policy for *Decide* phase

1: **function** DECIDE($event$)
2: $P_{prec} \leftarrow$ get_power()
3: $done \leftarrow$ false
4: **if** $event =$ new_appl **then**
5: $appl \leftarrow$ get_new_appl()
6: $prio \leftarrow$ get_appl_priority($appl$)
7: $clusters_list \leftarrow$ get_affinity_order($appl$)
8: **else**
9: **if** $freezed_queue \neq \emptyset$ **then**
10: $appl \leftarrow$ get_max_priority_appl($freezed_queue$)
11: $prio \leftarrow$ get_appl_priority($appl$)
12: $clusters_list \leftarrow$ get_affinity_order($appl$)
13: **else**
14: $cluster \leftarrow$ get_freed_cluster()
15: $running_appls \leftarrow$ get_running_appl()
16: $appl \leftarrow$ get_highest_affinity_app($running_appls, cluster$)
17: **if** $appl \neq$ None **then**
18: remap($appl, cluster$)
19: $done \leftarrow$ true
20: **while** !$done$ **and** $clusters_list \neq \emptyset$ **do**
21: $cluster \leftarrow$ pop($clusters_list$)
22: **if** is_free($cluster$) $=$ true **then**
23: map($appl, cluster$)
24: $done \leftarrow$ true
25: **else**
26: $appl_2 \leftarrow$ get_appl_running_on($cluster$)
27: $prio_2 \leftarrow$ get_appl_priority($appl_2$)
28: $cluster_2 \leftarrow$ get_subsequent_best_cluster($appl_2$)
29: **if** is_free($cluster_2$) $=$ true **then**
30: remap($appl_2, cluster_2$)
31: map($appl, cluster$)
32: $done \leftarrow$ true
33: **else if** $prio_2 < prio$ **then**
34: freeze($appl_2, freeze_queue$)
35: map($appl, cluster$)
36: $done \leftarrow$ true
37: **if** !$done$ **then**
38: freeze($appl, freeze_queue$)
39: **else**
40: $P_{curr} \leftarrow$ get_power()
41: $freq_{new} \leftarrow \frac{TDP - P_{prec}}{P_{curr} - P_{prec}} \cdot$ get_freq($cluster$)
42: set_freq($cluster, freq_{new}$)

at every control cycle. In the *Analyze phase*, the *Controller* waits for the events of new inference requests or release of a cluster due to termination or remapping of previously running inferences. The *Controller* receives the *Signature map* of each inference request from the *Model Interpreter* and reads the current status of the available clusters. In the *Decide* phase, the *Controller* creates a mapping configuration for the given workload following a *Scheduling policy*. Finally, in the *Deploy* state, the workload is mapped to the clusters based on the configuration received from the *Decide* state.

Analyze. In this phase, the controller waits for an inference request and transitions to the *Decide* phase when a new request arrives or a previously running application completes execution. In case of a new inference request, the controller receives the *Signature map* of each application from *Model Interpreter*, and the system status, including cluster availability and GPU frequency level. If a previously running application completes execution, the controller reads the system status and transitions to the *Decide* phase to update the system configuration. The same event is notified in case an application is remapped in the previous control cycle.

Decide. In this phase, the *Controller* decides on the knobs actuation settings to achieve the minimum latency under a power budget. We consider TDP as the power budget of the given platform. We have demonstrated the decision strategy of the *Controller* in Algorithm 2. When a new application arrives, the *Controller* extracts the application priority and the list of preferred clusters from the application signature maps acquired from the *Model Interpreter* (Lines 4–7). Otherwise, if the event occurred due to a freed cluster, the *Controller* first checks if any high-priority applications are waiting in the freeze queue and, if so, it gets the application data (Lines 9–12). If the queue is empty, the system evaluates whether an already-running application has higher affinity for the freed cluster (Lines 13–16); if so, such an application is remapped accordingly (Lines 17-19). Finally, if the algorithm entered the last branch since no application entered and the freeze queue is empty, the *Decide* phase is completed.

After identifying the application and its preferred clusters, the *Controller* iterates over the cluster list to attempt application-to-cluster mapping (Line 20–21). If a suitable cluster is available, the *Controller* maps the application and exits the *Decide* phase (Lines 22–24). If the preferred cluster is occupied, the *Controller* checks if the currently-running application can be remapped on a more suitable free cluster (Lines 26–27), and if this condition is true, the latter application is remapped and the newly arrived application is mapped on the freed cluster (Lines 29–32). Alternatively, the *Controller* freezes the currently running lower priority application, and puts it in the freezing queue while making room for the new higher priority application (Lines 33–36). Finally, if no suitable mapping has been found, the *Controller* freezes the new application (Lines 37–38). After successful application-to-cluster mapping, the *Controller* adjusts the frequency of the units provided with DVFS (GPU in our case), accordingly to maximize the power utilization while avoiding TDP violations (Lines 39–42). In particular, power measures are taken before application start (i.e., at the previous control cycle, Line 2) and after the application start (Line 40); then, based to a linear model exploiting the available power budget and the measured power variation the new frequency is computed (Lines 41–42). Do note that in case we also have a remapped $appl_2$, the available power budget is split among the two handled models.

Deploy. Finally, in the *Deploy* phase, the *Controller* deploys the scheduling configuration with knobs actuation decided in the *Decide* phase. The *Controller* enforces the configuration and transitions to the Analyze phase, waiting for a new event.

IV. Experimental Evaluation

In this section, we present the experimental evaluation of our proposed *Twill* framework in comparison with state-of-the-art edge AI inference strategies over relevant cAI workloads.

A. Experimental Setup

Controller prototype. We have implemented the proposed *Twill* framework in the Python programming language. The framework runs as a user-space process in a Linux OS environment. The controller monitors the run-time power using onboard power sensors.

Platform and middleware. For evaluation, we use the Nvidia Jetson Orin NX [28] platform, which is composed of an Ampere GPU, an NVDLA (Nvidia's DLA), and hexa-core ARM A78 CPUs, along with 8 GB of RAM and GPU memory. The platform has onboard power sensors for collecting run-time power consumption of CPU, GPU, and DLA clusters. The platform hosts Linux-20.04 OS with CUDA support to enable GPU operations, and allows run-time actuation of GPU DVFS. We enable the default *Performance* scheduling governor for the GPU cluster to run our experiments. For our experimentation, we set TDP as 10W [28]. The average overhead of running the *Twill* framework is 15ms on 100% utilization of 1 CPU core of the Orin NX platform.

Workloads. For experiments, we considered AI models that can perform image classification, text classification, and text generation tasks, which can be chained to form widely used cAI systems. We use the LangChain [7] tool for orchestrating the cAI workload design. For vision applications, we consider DNNs – `VGG-19`, `ResNet-50`, and `EfficientNet`-B4 [26], [35], [36] and vision transformer models – `ViT-base`, and `ViT-large` [37]. For text classification, we consider encoder-based transformer models of `Bert-base` and `Bert`-large [27]. For LLMs, we consider `DeepSeek R1` [38] with 1.5 billion parameters and `Gemma 3` with 1 billion parameters. We implement the inference mechanism using the PyTorch [39] framework and torchvision [40]. For transformers, we use the Hugging Face [41] library, and for LLM inference, we use Ollama [42].

Comparison w.r.t. state-of-the-art approaches. We consider three State-of-the-Art (SoA) edge AI inference strategies for heterogeneous platforms, including *MapFormer* [23], *Tango* [16], and *Band* [10]. *MapFormer* considers multi-DNN workloads for partitioning and allocating a suitable cluster between GPU and DLA while scaling the DVFS to manage power and throughput. This strategy makes design-time mapping decisions, assuming all inference requests arrive simultaneously. We implemented a transformer-based estimator of *MapFormer* that predicts throughput and power consumption distributions for multi-DNN workloads. *Tango* is a run-time multi-DNN workload management strategy on CPU and GPU clusters using reinforcement learning based exploration. We used Gymnasium library [43] to implement *Tango's* latency estimation model. *Band* partitions the multi-DNN workload into subgraphs at design-time, and maps the subgraphs based on the supported DNN operations on GPU and NPU clusters. We implemented the *Band's* model analyzer to generate the subgraphs for cluster mapping based on the DNN operations and FLOP counts.

Evaluation metrics. We measure per-application inference latency, throughput of the entire workload mix, and the platform's power consumption for our experimentation. For DNNs, we measure the latency as the time required to infer a batch of input images. For transformers, we measure the latency as the inference of a text prompt for a classification task. For LLMs, the inference latency varies depending on the number of generated output tokens against a text prompt. We measure throughput as the number of inferences

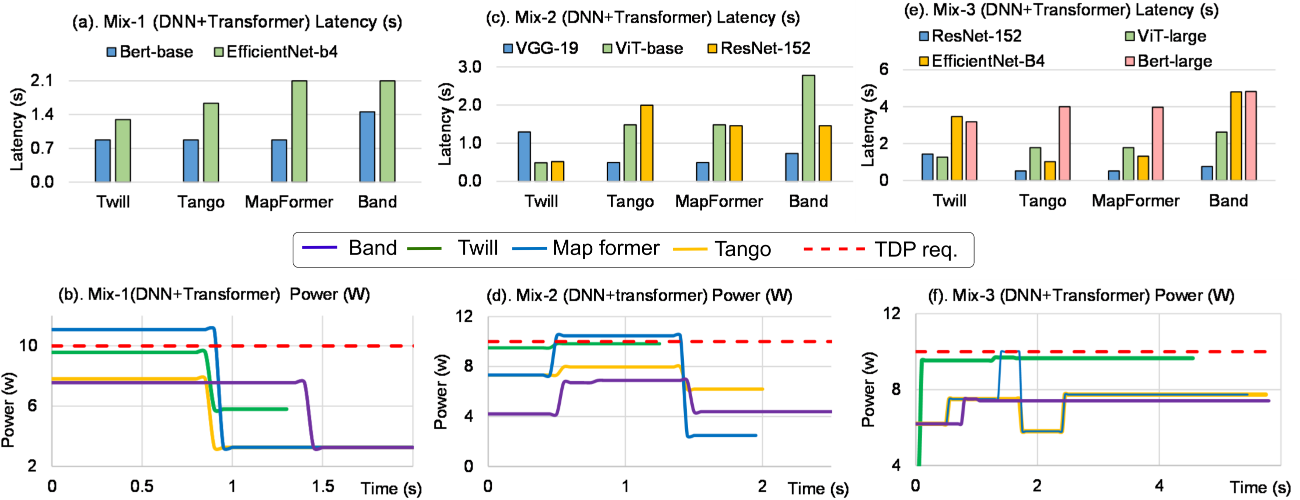


Fig. 6. Inference latency and run-time power consumption of different workload mixes. The first row represents the application latency, and the second row presents the power consumption of each strategy.

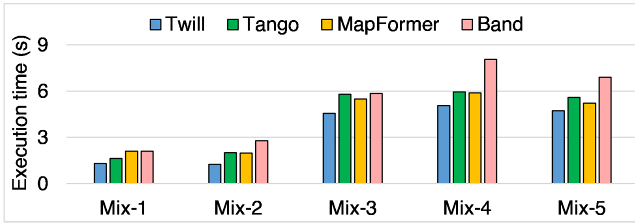


Fig. 7. Overall cAI system execution time for mix 1-5.

per second, reflecting the system's capacity to handle dynamic cAI workloads.

B. Experimental Results

For evaluation, we create different cAI workload combinations (*Mix 1-5*), encompassing various degrees of composability using multiple DNNs, transformers, and LLMs. Workload *Mixes* 1-3 are based on DNN and transformer models, and the *Mixes 4-5* are based on DNNs, transformers, and LLM. Figure 7 shows the execution time of different cAI workload mixes with *Twill* and three SoA edge AI inference strategies. Our proposed *Twill* strategy has the lowest execution time in comparison with other relevant strategies, achieving up to 38%, 54%, 22%, 37%, and 31% lower execution time for *Mix-1* to *Mix-5*, respectively. *Twill* jointly tunes multiple actuation knobs of affinity-cluster mapping, task freezing, cluster migration, and DVFS to minimize the inference latency within the available power budget. This coordinated joint actuation for run-time varying cAI workloads results in significantly lower execution time. Other strategies are confined to offline analysis of workload characteristics and can only handle inference request arrivals that are known at design time. Hence, they minimize the inference latency of the first and/or second arriving applications, while resulting in significantly higher overall inference latency as the number of concurrent inference requests and compute diversity increases. On average, *Twill* has 20%, 19%, and 34% lower execution time than *Tango*, *MapFormer*, and *Band*, respectively. Figure 8 shows the time each inference request spent in the waiting queue due to shared resource contention. *Twill* successfully avoids inference request queuing in *Mix-1* and *Mix-2*. For *Mixes* 3-5, *Twill* reports 83%, 87%, and 88% lesser waiting time on average than other relevant strategies. Evaluation metrics of each workload mix are detailed in the following.

Workload Mix-1. *Mix-1* includes inference requests of `Bert-base` and `EfficientNet-b4` models with `EfficientNet-b4` arriving $20ms$ after `Bert-base`. This creates a scenario of progressively increasing workload with both DNN and transformer applications, such that both applications run concurrently at $t = 20ms$. Figure 6(a) shows the latency of both `Bert-base` and `EfficientNet-b4` against different strategies. *Tango* does not support DLA execution, and puts `EfficientNet-b4` in the waiting queue while executing `Bert-base` on GPU causing a waiting overhead for the DNN model. Similarly, *Mapformer* first allocates GPU to `Bert-base`, and DLA to `EfficientNet` once it arrives. This strategy decides the workload allocation at an application arrival and does not support run-time configuration changes for applications with unknown arrivals. Hence, `EfficientNet` continues executing on a lower performing DLA even when GPU becomes available after the execution `Bert-base`. Finally, *Band* also allocates GPU to *Bert-base*, and DLA to EfficientNet, without DVFS control. The strategy depends on the default Linux governors for DVFS control and is constrained to making workload partitioning decisions upon arrival. Finally, *Twill* maps `EfficientNet` on DLA until `Bert-base` executes on the GPU cluster. *Twill* migrates `EfficientNet-b4` to GPU for faster execution once `Bert-base` terminates, at $t = 874ms$. Moreover, *Twill* scales the GPU frequency following the given workload to maximize the power budget utilization for better performance. Hence, *Twill* capitalizes on run-time knob actuation decisions for unknown workloads scenarios, ensuring faster workload execution while staying within the available power budget. Figure 6(b) shows the power consumption of each strategy during the workload execution. *Tango* has lesser power consumption because it runs one application on GPU for any given instance and does not involve DLA. *Band* does not actively actuates the DVFS which reduces the overall power consumption. *Mapformer* actively tunes DVFS shows TDP violations for 48% of the entire execution time because it does not cater to the power budget. *Twill* ensures the maximum utilization of the available power budget to achieve minimum latency.

Workload Mix-2. For workload *Mix-2*, we introduce three inference applications `VGG-19`, `ViT-base`, and `ResNet-152`, representing a different scenario from *Mix-1* in terms of dynamicity and computational load (Figure 6(c)). `VGG-19` arrives first and is mapped to the GPU by *Tango*. At $t = 20ms$, `ViT-base` arrives with high

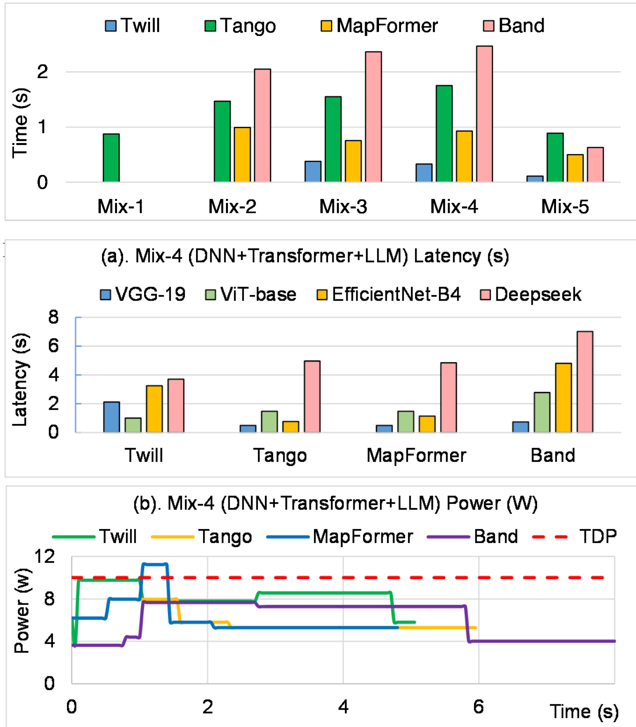


Fig. 9. Inference latency and run-time power consumption of *Mix-4*

GPU affinity, but *Tango* cannot migrate `VGG-19` to DLA, resulting in high wait time. After `VGG-19` finishes at $t = 494ms$, `ViT-base` runs on the GPU until $t = 1484ms$. Meanwhile, `ResNet-152` arrives at $t = 1000ms$ and waits until the GPU is free. *Mapformer* maps `VGG-19` to GPU and queues `ViT-base`, later assigning it to GPU and `ResNet-152` to DLA. *Band* follows a similar mapping but suffers from higher latency due to lacking DVFS control. *Twill* minimizes total execution time by initially assigning `VGG-19` and `ViT-base` to DLA, scaling GPU frequency, and mapping `ResNet-152` to GPU. Once `ResNet-152` completes, `VGG-19` is migrated to GPU. Figure 6(d) shows runtime power usage, where *Mapformer* exceeds the TDP limit for 49% of the execution time.

Workload Mix-3. In this workload mix shown in Figure 6(e), we progressively introduced four applications to the system, including two DNNs (`ResNet-152` and `EfficientNet-b4`), and two transformer models (*ViT-large*, and `Bert-large`). At t=0, `ResNet-152` arrives and *Tango* maps it to the GPU cluster, while `ViT-base` waits in a queue after arriving at $t = 20ms$. When `ViT-base` is executing, `EfficientNet-b4`, `Bert-large` arrive at $t = 1000ms$ and $t = 10020ms$, respectively, and wait in the queue for later execution. *MapFormer* executes `ResNet-152`, and puts `Vit-large` in waiting queue. Later, it maps `ResNet` on DLA, while `Bert-large` again waits in the queue for termination of `ViT-large`. *Band* again has similar mapping as *Mapformer* without DVFS scaling. *Twill* successfully runs `ResNet-152`, and `ViT-large` in parallel by mapping them on DLA, and GPU successfully. Later, it maps `Bert-base` on GPU, while freezing `ResNet` because `Bert-base` has higher GPU affinity. Finally, *ResNet* first runs on DLA after `VGG-19` finishes execution and migrates to GPU once `Bert-base` finishes execution. Figure 6(f) shows the power consumption during the workload execution. *Mapformer* shows TDP violations for 7% of the entire execution.

Workload Mix-4. In *Mix-4*, we considered a highly heterogeneous workload consisting of two DNNs (`VGG-1`, `Efficientnet-b4`), one encoder transformer (`ViT-base`), and an LLM (Deepseek-R1), as shown in Figure 9(a). For quantifiable analysis, we considered the latency of 100 output tokens from the generative model. *VGG-19* arrives at $t = 0$ and *Tango* maps it to GPU, while `ViT-base`, arriving at $t = 20ms$, waits in the queue for the prior application to finish. `EfficientNet` at $t = 1000ms$ and *Deepseek* at $t = 1020ms$ also wait for the preceding tasks to complete, causing cumulative delays. *Mapformer* and *Band* similarly queue `ViT-base` while executing `VGG-19`, and later run `EfficientNet` on DLA, followed by `Deepseek` on GPU. *Twill* maps `VGG-19` on DLA and `ViT-base` on GPU; later `DeepSeek` runs on GPU, given its higher cluster affinity. Figure 9(b) shows that only *Mapformer* exhibits TDP violations for 7% of the execution time.

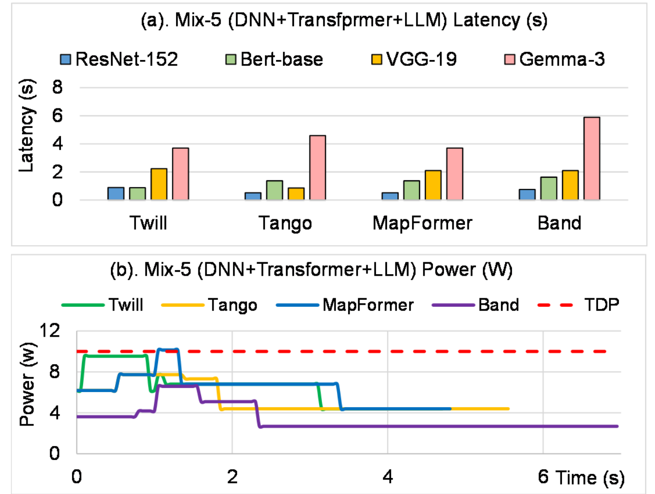


Fig. 10. Inference latency and run-time power consumption of *Mix-5*

Workload Mix-5. In the final workload *Mix-5* shown in Figure 10(a), we considered four workload application including `ResNet-152`, `Bert-base`, `VGG-19`, and `Gemma-3` introduced at $t = 0ms$, $t = 20ms$, $t = 1000ms$, and $t = 1020ms$ progressively to the system. *Tango*, maps each application on GPU, where each application other than the first *ResNet-152* has to wait in the queue for execution. *Mapformer*, and *Band* map `ResNet-152`, `Bert-base`, and `Gemma` on GPU and `VGG-19` on DLA where `Bert-base` waits in the queue while GPU is occupied by `ResNet`. *Twill* intelligently migrates `ResNet` between DLA and GPU while making room for `Bert-base`, and `Gemma` on the GPU cluster. `VGG-19` is mapped on DLA for parallel execution with `Gemma`. Figure 10(b) reports 6% TDP violations of *Mapformer*, while the other strategies remain within the available power budget.

V. Conclusions

We presented *Twill*, an adaptive run-time framework for scheduling cAI workloads on heterogeneous mobile edge platforms. Our proposed approach uses affinity-aware mapping and migration, priority-aware task freezing/unfreezing, and DVFS to handle concurrent inference requests, while minimizing inference latency within power budgets. Experimental evaluation of our strategy over contemporary cAI workloads against relevant edge AI inference techniques demonstrated up to 54% lower inference latency while honoring power budgets. Orchestrating multi-LLM cAI systems on mobile platforms is planned as our future work.

Acknowledgments

This work is funded by the European Union Horizon 2020 Research and Innovation Program (APROPOS) under the Marie Curie grant No. 956090

REFERENCES

[1] M. Zaharia, O. Khattab, L. Chen, J. Q. Davis, H. Miller, C. Potts, J. Zou, M. Carbin, J. Frankle, N. Rao, and A. Ghodsi, "The shift from models to compound ai systems," https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/, 2024.

[2] L. Chen, J. Q. Davis, B. Hanin, P. Bailis, I. Stoica, M. A. Zaharia, and J. Y. Zou, "Are more llm calls all you need? towards the scaling properties of compound ai systems," *Advances in Neural Information Processing Systems*, vol. 37, pp. 45 767–45 790, 2024.

[3] L. Chen, J. Q. Davis, B. Hanin, P. Bailis, M. Zaharia, J. Zou, and I. Stoica, "Optimizing model selection for compound ai systems," *arXiv preprint arXiv:2502.14815*, 2025.

[4] K. Valmeekam, M. Marquez, and S. Kambhampati, "Can large language models really improve by self-critiquing their own plans?" *arXiv preprint arXiv:2310.08118*, 2023.

[5] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu *et al.*, "Autogen: Enabling next-gen llm applications via multi-agent conversation," *arXiv preprint arXiv:2308.08155*, 2023.

[6] C. Cui, Y. Ma, X. Cao, W. Ye, and Z. Wang, "Drive as you speak: Enabling human-like interaction with large language models in autonomous vehicles," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, 2024, pp. 902–909.

[7] H. Chase, "Langchain," 2022, release date: 2022-10-17. [Online]. Available: https://github.com/langchain-ai/langchain

[8] Langbase, "Langbase: AI-Powered Multilingual Database," 2025. [Online]. Available: https://langbase.com/

[9] Y. G. Kim *et al.*, "Autoscale: Energy efficiency optimization for stochastic edge inference using reinforcement learning," *Proc. of Int. Symp. on Microarchitecture, MICRO*, pp. 1082–1096, 2020.

[10] J. Seong *et al.*, "Band: coordinated multi-dnn inference on heterogeneous mobile processors," in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, 2022.

[11] X. Guo *et al.*, "Automated exploration and implementation of distributed cnn inference at the edge," *IEEE Internet of Things Journal*, vol. 10, no. 7, pp. 5843–5858, April 2023.

[12] Apple Inc., "Apple intelligence," 2024, accessed: 2025-04-21. [Online]. Available: https://www.apple.com/apple-intelligence/

[13] M. P. Inc., "Meta ray-ban smart glasses: Next generation smart eyewear," *Meta Technology Review*, 2023. [Online]. Available: https://www.ray-ban.com/usa/ray-ban-meta-ai-glasses

[14] A. Inc., "Apple vision pro: Spatial computing device," 2024. [Online]. Available: https://www.apple.com/apple-vision-pro/

[15] Qualcomm, "Qualcomm launches its next-generation xr and ar platforms," 2023, accessed: 2025-04-08. [Online]. Available: https://www.qualcomm.com/news/releases/2023/09/qualcomm-launches-its-next-generation-xr-and-ar-platforms--enabl

[16] Z. Taufique, A. Vyas, A. Miele, P. Liljeberg, and A. Kanduri, "Tango: Low latency multi-dnn inference on heterogeneous edge platforms," in *2024 IEEE 42nd International Conference on Computer Design (ICCD)*, 2024, pp. 300–307.

[17] A. Karatzas and I. Anagnostopoulos, "OmniBoost: Boosting Throughput of Heterogeneous Embedded Devices under Multi-DNN Workload," in *Proc. of ACM/IEEE Design Automation Conf. (DAC)*, 2023, pp. 1–6.

[18] Y. Wu, Y. Gong, Z. Zhan, G. Yuan, Y. Li, Q. Wang, C. Wu, and Y. Wang, "MOC: Multi-Objective Mobile CPU-GPU Co-Optimization for Power-Efficient DNN Inference," in *Proc. of ACM/IEEE Intl. Conf. on Computer Aided Design (ICCAD)*, 2023, pp. 1–10.

[19] S. Wang, G. Ananthanarayanan, Y. Zeng, N. Goel, A. Pathania, and T. Mitra, "High-Throughput CNN Inference on Embedded ARM Big.LITTLE Multicore Processors," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2254–2267, 2019.

[20] X. Guo, Q. Jiang, Y. Shen, A. D. Pimentel, and T. Stefanov, "Easter: Learning to split transformers at the edge robustly," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 11, pp. 3626–3637, 2024.

[21] H.-Y. Chang, S. H. Mozafari, C. Chen, J. J. Clark, B. H. Meyer, and W. J. Gross, "Pipebert: High-throughput bert inference for arm big.little multi-core processors," *J. Signal Process. Syst.*, vol. 95, p. 877–894, Oct. 2022.

[22] I. Dagli and M. E. Belviranli, "Shared memory-contention-aware concurrent dnn execution for diversely heterogeneous system-on-chips," in *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. ACM, Feb. 2024, p. 243–256.

[23] A. Karatzas and I. Anagnostopoulos, "Mapformer: Attention-based multi-dnn manager for throughout & power co-optimization on embedded devices," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '24. New York, NY, USA: Association for Computing Machinery, 2025.

[24] L. Rockchip Electronics Co., "Rknn toolkit operator support list (v1.7.5)," https://github.com/rockchip-linux/rknn-toolkit/blob/master/docs/RKNN_OP_Support_V1.7.5.md, 2023, accessed: 2025-04-19.

[25] N. Corporation, "Working with dla," 2025. [Online]. Available: https://docs.nvidia.com/deeplearning/tensorrt/latest/inference-library/work-with-dla.html

[26] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[27] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, 2019, pp. 4171–4186.

[28] NVIDIA, "Nvidia jetson Orin," https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/, 2024.

[29] Y. Kim, I. Kim, K. Choi, J. Ahn, J. Park, and J. Huh, "Interference-aware dnn serving on heterogeneous processors in edge systems," in *2024 IEEE 42nd International Conference on Computer Design (ICCD)*, 2024.

[30] A. Karatzas, D. Stamoulis, and I. Anagnostopoulos, "Rankmap: Priority-aware multi-dnn manager for heterogeneous embedded devices," 2024.

[31] J. Kim and S. Ha, "Energy-aware scenario-based mapping of deep learning applications onto heterogeneous processors under real-time constraints," *IEEE Transactions on Computers*, vol. 72, no. 6, pp. 1666–1680, 2023.

[32] I. Dagli, A. Cieslewicz, J. McClurg, and M. E. Belviranli, "Axonn: Energy-aware execution of neural network inference on multi-accelerator heterogeneous socs," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 1069–1074.

[33] O. R. developers, "Onnx runtime," https://onnxruntime.ai/, 2021.

[34] D. Hendrycks and K. Gimpel, "Gaussian error linear units (gelus)," 2023.

[35] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[36] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 97, 2019, pp. 6105–6114.

[37] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[38] DeepSeek-AI, D. Guo, and et al, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," 2025.

[39] P. Team, "Pytorch installation guide," https://pytorch.org/get-started/locally/, 2024.

[40] "Torchvision," https://pytorch.org/vision/stable/index.html, PyTorch Team, 2024.

[41] H. Face, "Hugging face: The ai community building the future of machine learning," 2025, accessed: 2025-04-18. [Online]. Available: https://huggingface.co/

[42] O. Inc., "Ollama: Run large language models locally," 2025, accessed: 2025-04-18. [Online]. Available: https://ollama.com/

[43] M. Towers *et al.*, "Gymnasium," 2023. [Online]. Available: https://zenodo.org/record/8127025