Niagara: Scheduling DNN Inference Services on Heterogeneous Edge Processors

Daliang Xu[1][0000−0002−6775−0688], Qing Li[1][0000−0002−1772−9194], Mengwei Xu[2⋆][0000−0001−6271−6993], Kang Huang[3][0000−0002−7476−0665], Gang Huang[1,5][0000−0002−4686−3181], Shangguang Wang[2][0000−0001−7245−1298], Xin Jin[1⋆][0000−0001−8741−5847], Yun Ma[4⋆][0000−0001−7866−4075], and Xuanzhe Liu[1][0000−0002−7908−8484]

[1] Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China
{xudaliang, liqingpostdoc, hg, xinjinpku, liuxuanzhe}@pku.edu.cn
[2] State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, China;
{mwx, sgwang}@bupt.edu.cn
[3] Linggui Tech Company, Beijing, China
kang.huang@nlptech.com
[4] Institute for Artificial Intelligence, Peking University, Beijing, China
mayun@pku.edu.cn
[5] National Key Laboratory of Data Space Technology and System, China

Abstract. Intelligent applications heavily rely on deep neural network (DNN) inference services executed on edge devices to fulfill functional prerequisites while safeguarding user data privacy. However, the execution of such DNN services on resource-constrained edge devices poses a significant challenge: low throughput of inference tasks. To this end, this paper proposes Niagara, a novel system designed to maximize system throughput by judiciously scheduling DNN inference services on heterogeneous processors available on edge devices. Niagara faces two critical challenges: uncertain workload dynamics and high scheduling complexity. To effectively address these challenges, Niagara employs a predictive model to anticipate incoming workload patterns and orchestrates the allocation of services across heterogeneous processors through a combination of offline scheduling optimization and online service dispatching strategies. We have implemented Niagara and conducted thorough experiments. The results demonstrate that Niagara surpasses state-of-the-art approaches by elevating DNN inference throughput by up to 4.67×, all while satisfying the same stringent inference latency requirements. Furthermore, Niagara has been successfully deployed in real-world power supply substations to detect violations, ensuring uninterrupted, accident-free operation during its six-month deployment period.

Keywords: Edge Computing, Heterogeneous Processors, DNN Inference Service

⋆ Corresponding authors.

1 Introduction

Recent years have witnessed various intelligent edge applications (e.g., healthcare, entertainment, and smart home applications) becoming integral components of our daily lives [13]. These applications often rely on deep neural networks (DNNs) for sophisticated sensory interpretation, such as user context and physical surroundings. To ensure a seamless user experience, these edge applications, such as violation operation detection [39], immersive online shopping [41] and AR emoji [38], typically prefer to employ a set of flexible and reliable edge DNN inference services [22,28,35,38]. For instance, violation operation detection, which determines whether operators in a state grid corporation wear wear valid helmets and gloves during operations, necessitates at least four DNN inference services: human detection, pose estimation, and helmet/gloves classification.

However, executing DNN inference services on resource-constraint edge devices often encounters the low throughput problem [22,35,38,39]. Previous studies have primarily focused on optimizing the execution of *individual* DNN services [23, 30, 31, 37], which limits their effectiveness in addressing performance bottlenecks within a multi-service environment.

To tackle this issue, we have observed that various types of heterogeneous processors on edge devices [3,5–7,12] (e.g., ARM A57 cores [1] and the NVIDIA Pascal GPU on Jetson TX2 [3]) can be harnessed to deliver high-throughput DNN services. To this end, we present `Niagara`, the first scheduling engine for DNN inference services on edge devices. The core idea behind `Niagara` is to monitor processor status, predict incoming workload dynamics, and efficiently schedule DNN inference services across heterogeneous processors. `Niagara` faces two primary challenges:

• **High complexity in scheduling design.** As will be elaborated in §2, optimizing DNN-inference-service-to-processor affinity, enabling parallel execution, and efficiently batching inputs have the potential to significantly enhance DNN inference services execution. However, the multiple interdependent optimization choices render the scheduling of DNN services to processors a challenging task.

• **Unknown and mutative DNN inference service workload.** The design of `Niagara` grapples with a dilemma between the need for global knowledge and timely decision-making. Theoretically, having advanced knowledge of upcoming requests could offer more scheduling opportunities. However, services depend on future input, which is only accessible when the corresponding DNN inference service (e.g., person detection) has been executed.

To address the above two challenges, we incorporate two novel techniques: (1) *Offline optimizer and online service scheduler.* We have identified that DNN service request patterns can be abstracted into several typical **service graph templates**. Based on that, `Niagara` optimizes the service-to-processor scheduling strategy for each service graph template offline, caches the strategy, and matches the appropriate strategy to user requests belonging to specific templates online. The offline optimizer accounts for inter-service dependency, batch/parallel execution, and resource constraints. (2) *Dynamic input predictor.* We have found that the service graph tends to be more stable than the content, providing an opportunity for prediction. Consequently, we construct a time series model [16, 33] of the DNN service graph based on the latest and global historical data and employ a combined prediction algorithm to forecast the future DNN service graphs.

Implementation and evaluation We implement an end-to-end prototype of `Niagara` on the Android OS. Our evaluation comprises 8 types of DNN inference service combinations, including 11 distinct DNN services, 3 real-world video stream requests, and 3 different edge devices. These experiments have been conducted in real-world settings. Compared to the state-of-the-art baselines, `Niagara` can enhance overall processing throughput by up to 4.67× while maintaining the same response requirements on identical hardware.

In-the-wild deployment `Niagara` has been integrated into a custom-made IP camera on a Snapdragon 865 development board and deployed in several power supply substations that serve millions of people in a large Chinese city. This deployment aims to enhance the safety of operators working on electric switching operations. During the 6-month pilot run, which included over 18,000 maintenance jobs, zero accidents were reported, representing a significant improvement over traditional human-based supervision. In the near future, `Niagara` will be extended to thousands of substations, showcasing how edge intelligence can contribute to society.

The key contributions of this paper are summarized as follows.

- We quantitatively analyze the challenges and opportunities of DNN inference service execution on edge devices.
- We propose `Niagara`, the first DNN inference services scheduling engine on heterogeneous edge processors. It incorporates two key techniques, including a service graph predictor and a template-based optimizer that judiciously schedules DNN services across processors.
- We evaluate our scheduling strategy and system on popular CNN services with real-world datasets. The results show that `Niagara` and its scheduling solution can effectively improve the overall processing throughput.

2 Background and Related Work

To enhance the quality of edge services, numerous prior studies [14, 19, 21, 24–26, 34, 36, 40] have centered their efforts on augmenting the scheduling efficiency of offloading tasks in the realm of mobile-edge computing, considering factors such as service caching, service dependencies, and multiple application scenarios. For instance, some of these investigations have concentrated on scheduling offloading tasks while simultaneously taking service caching into account [14, 24, 34, 40]. Their objective is to harness caching mechanisms for storing and retrieving frequently used services at the edge, thereby diminishing the necessity for task offloading and mitigating latency. Other studies have underscored the scheduling of dependent services on fog or edge nodes, considering service priorities or catering to multiple applications [19, 25, 26]. These works meticulously address the dependencies between services and prioritize their execution to meet application requirements and bolster overall performance. However, our work, `Niagara`,

Table 1. Latency and utilization of DNN services on SnapDragon 865 SoC.

DNN service	DNN model	Latency			Utilization		
		CPU	GPU	DSP	CPU	GPU	DSP
Person detection	SSD-quant	112.1ms	**79.9ms**	103.1ms	361%	56%	77%
Pose estimation	CenterNet	**22.9ms**	31.7ms	-	287%	30%	-
Helmet detection	SSD-helmet-quant	25.6ms	8.4ms	**5.9ms**	195%	58%	85%
Gloves detection	pole-gloves	6.7ms	**3.2ms**	-	198%	34%	-
Text recognition	OCR-recognition	**30.8ms**	38.1ms	-	295%	35%	-

focuses specifically on maximizing the utilization of heterogeneous processors available on edge devices for efficient and high-throughput service scheduling.

Another critical issue in DNN services scheduling pertains to the unanticipated dynamic inputs. Several studies have endeavored to forecast future requests by harnessing deep learning methodologies [20, 32]. Meanwhile, other research endeavors [18, 27] have taken it a step further by jointly addressing scheduling challenges alongside input prediction. However, these undertakings often prove excessively intricate for practical application in online DNN services prediction scenarios.

In summary, the distinctive hardware specifications of edge devices and the unique computing paradigm associated with DNN model inference render the scheduling of edge services notably distinct from conventional web services and offloading tasks. For instance, the Snapdragon 865 SoC, commonly deployed as the main board for IP cameras [5], includes CPU, GPU, and DSP, whereas other edge devices may feature an Edge TPU or NPU instead. Typically, different DNN models executed on such heterogeneous processors exhibit divergent behaviors. To gain a comprehensive understanding of these distinctive features, we conducted preliminary offline experiments on the Snapdragon 865 SoC, as summarized in Table 1.

• **Service-processor affinity and hardware support.** Our preliminary offline experiments (Table 1) have yielded a crucial insight: a discernible service-processor affinity exists. In other words, there is no one-size-fits-all processor to which all services can be indiscriminately scheduled. For instance, the person detection service achieves its optimal performance on the GPU, while the pose estimation service exhibits superior execution on the CPU. This affinity arises from the highly varied characteristics inherent in modern DNNs, including network architecture, layer shapes, and input sizes [29]. Additionally, certain processors, such as the Hexagon DSP, lack support for floating-point arithmetic, thereby rendering services reliant on quantized models, like helmet detection, more compatible with specific hardware compared to their floating-point counterparts, which can be executed on a wider array of hardware platforms.

• **Parallel or sequence execution.** Different processors boast distinct capabilities when concurrently executing multiple services (parallel execution), thus maximizing hardware utilization. For instance, the CPU can achieve a maximum utilization of 400% in the Snapdragon 865 SoC, while the GPU and DSP

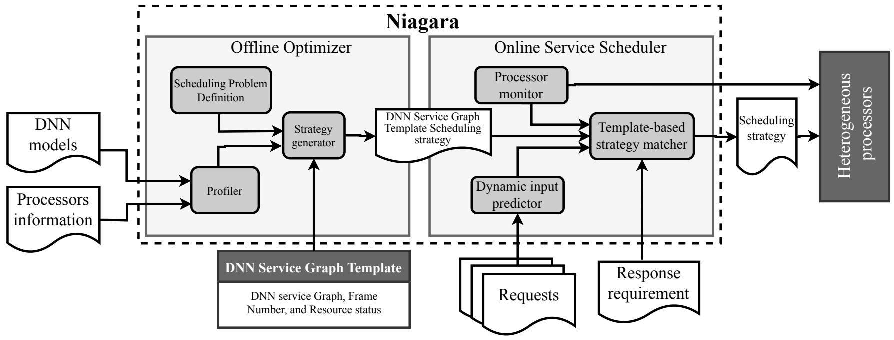


Fig. 1. System workflow of `Niagara`.

are capped at 100%. To ensure optimal performance, the resource consumption by parallel execution on the same processor must not exceed the processor's capacity. Otherwise, processor contention can severely hamper inference performance. Edge CPUs and GPUs typically support parallel execution, whereas edge DSPs/NPUs do not.

• **Batch execution** is a common strategy to group several identical services for simultaneous execution. This approach yields a longer instruction queue and greater instruction parallelism, mitigating stalls in memory access. However, since all these batched services end simultaneously, their output cannot be obtained until all services have completed their execution. Consequently, batch execution can bolster processor utilization and throughput while simultaneously introducing longer per-service latency. For instance, in the case of pose estimation and helmet detection, employing batching can achieve a throughput improvement by 33–68%, albeit at the cost of incurring a 45–51% increase in latency on the GPU. To that end, `Niagara` should meticulously design batch execution strategies to mitigate these drawbacks.

Summary & implications. All the above factors must be carefully considered when optimizing DNN inference service scheduling for heterogeneous edge processors. Furthermore, the dynamic nature of hardware contexts in multi-tenant devices necessitates continuous monitoring and real-time adaptation of scheduling decisions by our system.

3 System Design

3.1 System Overview

Design goal The primary objective of `Niagara` is to achieve a high throughput of DNN inference services by fully harnessing the computational capacity of heterogeneous processors on edge devices, including CPUs, GPUs, and DSPs.

Workflow Figure 1 provides an overview of the workflow of `Niagara`. The fundamental concept underlying `Niagara` is the utilization of *DNN inference service*

Algorithm 1: Online service scheduling algorithm

```
Input  : Cached template strategies cached_strategies_map
Output: Scheduling strategy strategy
Current_service_graph_template template
while True do
    Input data = user.request.Get() // Receive input data from users
    service_graph = Dynamic_input_predictor(data) // Section 3.5
    states = Processor_monitor() // Section 3.6
    if temple == NULL or Euclidean_distance(service_graph, template)
      < threshold then
        /* Section 3.4 Template-based strategy matcher */
        for t, s ∈ cached_strategies_map do
            if states < t.S and Euclidean_distance(service_graph, t.G) >
              Euclidean_distance(service_graph, template.G) then
                template = t, strategy = s
            end
        end
        strategy = Strategy_adapter(strategy) // Section 3.4
        return strategy
    end
end
```

graph templates. These templates consist of a set of elements: <a service graph $\mathcal{G}$, the number of requested services $\mathcal{RN}$, the resource status $\mathcal{S}$, and a maximum latency requirement Lat^{RQ}_{max}>. Specifically, it signifies that each of the $\mathcal{RN}$ subsequent service requests will follow the same service graph $\mathcal{G}$. Furthermore, these service graphs are executed on heterogeneous processors, taking into account the current processor status $\mathcal{S}$, which could indicate the availability of idle CPUs or the utilization of busy GPUs. Each inference service within $\mathcal{G}$ must respond within the latency requirement $lat^{R}max$.

`Niagara` employs these service graph templates to generate feasible strategies offline for various scenarios. These strategies subsequently schedule real-time online services onto the heterogeneous processors.

The input to `Niagara` aconsist of user-initiated service requests and the corresponding response requirements. Once deployed on an edge device, `Niagara` operates in two distinct stages:

– *Offline optimizer (Section 3.3)* In the offline stage, `Niagara` formulates the DNN inference services serving problem as a scheduling problem. The inputs for this scheduling problem encompass the service graph template and profiling data related to the services and the heterogeneous processors. A solver is employed to identify feasible solutions for each template.

– *Online service scheduler.* When the request data is received, *Dynamic Input Predictor* (Section 3.5) predicts the service graph within the data frame, while the *Processor Monitor* (Section 3.6) continuously monitors the status of the processors. Based on response requirements, processors status, and service graph, the *Template-based strategy matcher* (Section 3.4) selects the most suitable strategy from the precomputed offline strategies and adapts it to accommodate the real service graph. This allows services to be dispatched effectively to heterogeneous processors. The scheduling algorithm is illustrated in Algorithm 1.

3.2 Problem Formulation

Preliminaries. `Niagara` considers how to schedule various DNN inference services onto heterogeneous processors. Notably, `Niagara` does not modify the structural aspects of the DNN models within these services, in order to maintain accuracy and performance. As a result, it is incumbent upon the developers of each DNN inference service to provide configurations that specify essential details about the DNN model and the processors. These configurations include information about the processors on which the DNN models can potentially execute and the utilization of processors by each model. Users, in turn, are only required to invoke the DNN inference services and supply their input data.
DNN inference service graph model. Within `Niagara`, it is assumed that an edge device needs to process RN continuous requests, producing a total of N services, denoted by $\mathcal{V} = \{v_1, v_2, \cdots, v_N\}$ which belong to L ($L \leq N$) types. `Niagara` employs a directed acyclic graph (DAG) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ to represent the dependent relationships among DNN inference services, where $\mathcal{V}$ signifies the service set and $\mathcal{E}$ represents the set of edges symbolizing the dependencies among these services. If there exists an edge $e_{i,k}$ between any two services i and k, it implies that the output of service i serves as input to service k, signifying that service k cannot commence execution until service i has completed its task.s.
Batch Execution Latency Model. `Niagara` employs a linear model [28] to characterize batch execution latency:

$$batch\_lat(b) = a(b-1) + lat\_single \tag{1}$$

where b signifies the batch size and a represents the additional latency incurred when a new service input is appended to an existing batch execution. Notably, due to the diversity in services and processors, the parameter a is a two-dimensional matrix with dimensions equal to the number of service types and the number of processor types. Determining these parameters can be accomplished through linear fitting, utilizing profiling results.
Heterogeneous processors execution model. `Niagara` posits the existence of M types of heterogeneous processors, denoted as $\mathcal{R} = \{r_1, r_2, \cdots, r_M\}$. Each processor $r_j \in \mathcal{R}$ possesses its unique processing capacity denoted by E_j. Service v_i has the flexibility to execute on any processor $r_j \in \mathcal{R}i$ in various modes such as sequential, batch, or parallel. However, it is essential to emphasize that service execution is exclusive to a single processor at any given time. Regardless of the execution mode, services must not be interrupted or preempted, and they must complete their execution within the user-defined real-time threshold $Lat^{RQ}max$. When multiple services execute in parallel, their combined hardware utilization must not exceed the capacity of the processor.

Table 2. Notation table of problem definition in `Niagara`.

Variable	Notation	Description
Placement	$x_{i,j}$	Whether service $v_i \in \mathcal{V}$ executes on the processor $r_j \in \mathcal{R}$.
Batch	$B_{i,k,j}$	Whether services $v_i, v_k \in \mathcal{V}$ are batched on processor $r_j \in \mathcal{R}$.
Parallel	$PL_{i,k,j}$	Whether services $v_i, v_k \in \mathcal{V}$ execute in parallel on processor $r_j \in \mathcal{R}$.
Starting time	t_i	Starting time of service i.
Execution time intermediate variable	$T_{i,j}$	The i-th service's latency when running on the j-th type processor. If the i-th service executes separately, the value equals $L_{i,j}$. When the i-th service executes in batch, based on *Eq.* (1), the value is formulated as $T_{i,j} = a_{i,j}^T \sum_{k=1}^{N} B_{i,k,j} + L_{i,j}$

Scheduling problem definition. Given the service graph $\mathcal{G}$ and associated profiler information, including latency ($L_{i,j}$) and hardware utilization ($U_{i,j}$) for each service, processor capacity (E_j), and user-defined response requirement (Lat_{max}^{RQ}), `Niagara` DNN service-to-processor selection, batch execution, and parallel execution simultaneously. This entails the introduction of four primary decision variables and one intermediate variable are summarized in Table 2.

Our solution should satisfy the following constraints:

• *DNN service-to-processor selection constraint.* Any service should execute on exactly one supported processor. `Niagara` does not allow multiple processors to cooperate to complete single DNN service inference.

$$\sum_{j \in \mathcal{R}_i} x_{i,j} = 1, \forall i \in N \tag{2}$$

• *Dependency constraint*: Any service can start iff all precedent services are completed, formulated as for any edge $< i, k > \in \mathcal{E}$, v_k can start iff v_i finishes.

$$t_i + \sum_{j=1}^{M} x_{i,j} T_{i,j} \leq t_k \tag{3}$$

• *Sequence execution constraint*: Any service's execution cannot be interrupted. For any service i and k, if they execute on the same processor j in sequence and $t_i < t_k$, then the service v_k must wait until v_i completes.

$$\frac{t_k - t_i}{x_{i,j} x_{k,j} (1 - PL_{i,k,j})(1 - B_{i,k,j})} \geq T_{k,j} \tag{4}$$

• *Parallel constraint*: Paralleling services' execution times must overlap, meaning when services i and j both execute on the resource j and execute in parallel, their start time distance must be less than or equal to their execution time.

$$x_{i,j} * x_{k,j} * PL_{i,k,j} * abs(t_k - t_i) \leq min(T_k, T_i) \tag{5}$$

• *Batch constraint*: Batch services must begin simultaneously, meaning when services i and j execute on the resource j and are batched, their start time distance must be zero.

$$x_{i,j} * x_{k,j} * B_{i,k,j} * (t_k - t_i) \leq 0 \tag{6}$$

• *Request real-time constraint*: Any services within a request $\mathcal{RQ}$ should complete before users' requirement Lat_{max}^{RQ} to guarantee a real-time response.

$$\forall v_i, v_k \in \mathcal{RQ}, t_k - t_i \leq Lat_{max}^{RQ} \tag{7}$$

• *Capacity constraint*: When several services execute in parallel, their hardware utilization cannot exceed the processor's capacity. The overall hardware utilization will be nearly equal to the combined hardware utilization of individual services running independently.

$$\sum_{k=1}^{N} PL_{i,k,j} U_{k,j} + U_{i,j} \leq E_j \tag{8}$$

• *Objective and optimization model.* Our goal is to find a feasible solution with a maximum throughput which is denoted by $C = 1/\max\{t_i + \sum_{j=1}^{M} x_{i,j} T_{i,j}, \forall i \in N, \forall j \in M\}$. Thus, the problem can be formulated as the following model:

$$\max C \quad s.t.\ Eq.\ (3) - (9) \tag{9}$$

NP-Hard problem. It is important to note that the scheduling problem within `Niagara` is an instance of a classical NP-Hard problem, the Traveling Salesman Problem (TSP) [17]. Consequently, determining the optimal scheduling strategy for this problem is also NP-hard.

3.3 Template-Based Scheduling Strategy Generator

In addressing our scheduling problem, we have found success in leveraging the cutting-edge GUROBI solver [2]. This solver yields solutions with an optimality loss of less than 10% since our service decision variables remain relatively small, numbering around 100. Nevertheless, it is imperative to acknowledge that obtaining an approximately optimal solution through this method may entail several hours of computational effort, rendering it impractical for online scheduling.

To circumvent this challenge, `Niagara` introduces an innovative offline-online hybrid heuristic algorithm. Our insight stems from the observation that the majority of service request patterns exhibit remarkable stability over time. For instance, tasks such as face recognition consistently involve sub-tasks such as person detection, face detection, and facial recognition. In response to this observation, we introduce the concept of *service graph templates*, which encapsulate common service patterns frequently encountered in real-world scenarios. For exceptional and unexpected cases, we also offer an adaptation mechanism designed to modify the scheduling strategy in real-time, aligning it with the specific requirements of dispatching online DNN cascades to heterogeneous processors (as detailed in Section 3.4).

Each service graph template comprises four essential components: service graph $\mathcal{G}$, request number $\mathcal{RN}$, resource status $\mathcal{S}$, latency requirement lat_{max}^{RQ}. Through the analysis of existing request data, we endeavor to identify as many request patterns for services within a single frame as possible. For the second parameter, request number, the range is 1-N, with N representing the maximum number of frames that can be processed within a single second. In addition, `Niagara` conducts a comprehensive exploration of the status of heterogeneous processors, as elaborated in Section 3.6. Taking the Snapdragon 865 SoC

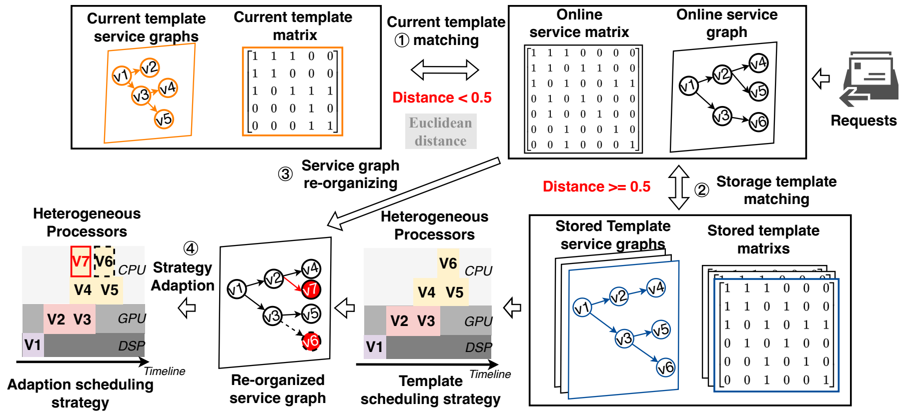


Fig. 2. The workflow of strategy matcher.

as an exemplar, `Niagara` systematically considers all feasible combinations of CPU cores, GPUs, and DSPs, encompassing various resource-status scenarios, thereby ensuring adaptability to the underlying hardware configurations. Regarding the final parameter, the response requirement, `Niagara` endeavors to generate scheduling strategies for all possible scenarios within intervals of 50ms, ranging from 50ms to 1000ms. In practice, `Niagara` has the capacity to generate a multitude of service graph templates and their corresponding feasible scheduling strategies, all of which are stored locally on edge devices. This storage incurs a minimal overhead of less than 10MB.

3.4 Template-Based Strategy Matcher

The matcher takes into account two primary inputs: the real-time service graph and the processor's status. We outline its workflow as illustrated in Figure 2.

The service graph is stored in a two-dimensional matrix format, with a value of 1 indicating the presence of a dependency between services. The Matcher, guided by the processor's status, is responsible for selecting appropriate template strategies under conditions that are no worse than the input circumstances. To achieve this, `Niagara` utilizes the Euclidean distance metric [15] to quantify the disparity between the online service graph (derived from the current image) and the service graph template, ultimately identifying the most suitable strategy.

The matcher includes the following steps:

– Step ①: When the distance between the online service graph and the current service graph template falls below a predefined threshold (e.g., 0.5), `Niagara` continues to employ the current template. This process is depicted in Figure 2①.

– Step ②: If the distance exceeds the threshold, `Niagara` discontinues the current scheduling strategy and selects a new one that closely matches the online service graph. For instance, in Figure 2②, the blue template is chosen due to its minimal distance, and it corresponds to a scheduling strategy.

– Step ③: Recognizing that the online service graph may not always align perfectly with the template, `Niagara` incorporates an adaptation mechanism to

Algorithm 2: Dynamic predictor algorithm

```
Input : First_order_exponential_predictor A, Holt_Winters_predictor B
CSGP = NULL // CSGP: current_service_graph_prediction
while True do
    Input data = user.request.Get() // Receive request data from user
    image_info = Main_DNN_inference(data)
    service_graph = Graph_generator(image_info)
    if CSGP.service_graph != service_graph then
        CSGP.service_graph = service_graph
        if A.history_accuracy > B.history_accuracy then
            CSGP.last_number = A.Predict(service_graph)
        else
            CSGP.last_number = B.Predict(service_graph)
        end
    end
    CSGP.service_graph.Execute()
    A.Update(number), B.Update(number)
end
```

accommodate unexpected variations. As demonstrated in Figure 2③, `Niagara` first reorganizes the current service graph. It endeavors to match online graph services with template services as closely as possible. Services that do not find a match, such as v6 and v7 in Figure 2, are flagged, while the scheduling positions of matching services remain consistent with the template's corresponding strategy. Notably, v6 represents an extra service, while v7 is a newly added service.

– Step ④: `Niagara` selects the first unmatching newly added service (e.g., v7) and places it within the earliest available idle period, as depicted in Figure 2④. In cases where the template scheduling strategy includes extraneous, redundant services, such services are eliminated (e.g., v6). Other services commence as early as possible while adhering to any applicable constraints.

3.5 Dynamic Input Predictor

The predictor is a crucial component in forecasting future service graphs, denoted as pairs of <*service graph, request number*>. Algorithm 2 shows its functionality.

Different scenarios often exhibit distinct recurring patterns in their service graphs. For instance, in the context of a parking system, events such as license plate recognition at an entrance gate may occur at regular intervals, while violation operation detection is more likely to follow a pattern similar to the most recent historical data. To address these diverse scenarios, `Niagara` employs a combined prediction approach, encompassing first-order prediction and triple exponential smoothing (Holt-Winters method), to capture both the latest and global historical patterns. It operates as follows:

– The predictor initiates the first DNN service inference in accordance with the ongoing scheduling strategy or its associated processor, should no active strategy exist. After execution, the predictor obtains essential information such as the count of people or cars, which forms the basis for predicting the service graph within the current request.

– If the newly predicted service graph diverges from the current one, `Niagara` proceeds to compare the historical accuracy of the predictors and selects the more precise one. This selection informs the prediction of how many frames the service graph will remain constant.

3.6 Processor Monitor

In this section, we discuss the processor monitoring mechanism implemented in our system. The monitor leverages system files such as */proc/stat* and */sys/class/kgsl/kgsl-3d0/gpu_ busy_ percentage* to acquire real-time utilization data for the CPU and GPU, and utilizes a benchmarking tool from the Hexagon DSP SDK to obtain information about DSP utilization.

Our monitoring system continuously inspects the status of these processors at intervals of 100 milliseconds. This monitoring frequency is deliberately set to be smaller than the service inference time to ensure the precision of our measurements while avoiding any adverse impact on the quality of service delivery.

4 Implementation and Evaluation

We have developed an end-to-end prototype of our system, comprising over 3,800 lines of code, built on the Android OS 10.0 platform. For DNN inference, we have employed TFLite, a runtime environment capable of supporting on-device CPU, GPU, and DSP inference. To ensure smooth execution of DNN inference while preserving the desired strategy order, we have implemented a ThreadPool and an InferenceFinishListener, enabling asynchronous processing.

4.1 Expriment Settings and Methodology

Hardware and OS. In order to assess the versatility of our scheduling strategy across diverse heterogeneous processor platforms, we executed `Niagara` on three SoCs configurations detailed in Table 4. These SoCs are widely employed in IP cameras, as indicated by [5]. Each of these SoCs encompasses three heterogeneous processors with varying capabilities. To maintain uniformity, all these devices operated on the Android 10 system.

Baselines. To highlight the advantages of our approach, we conducted a comparative analysis of `Niagara` against the following existing methods:

– *TFLite* employs unmodified TensorFlow Lite 2.4.0 [11]. When a service request for a model is received, TFLite immediately invokes a new runtime instance for execution, consistently dispatching the service to its affinity processor.

– *Greedy Algorithm* consistently schedules the service to its affinity processor, ensuring assignment until the processor becomes idle and can accommodate it.

Table 3. Experimental combinations of 3 scenarios and their corresponding datasets.

DNN service combination	Name	Complexity	DNN1	DNN2	DNN3	Video Input	Video Description
Violation Operation Detector (VOD)	VOD	High	SSD-Main	CenterNet-Keypoint	Pole-gloves/ SSD-helmet-quant	Power grid site 1 week, 1 camera	Resolution: 960*540 FPS:30
	VOD-Y	Low	Tiny-yolov3-quant				
	VOD-FH	Middle	SSD-Main		Pole-gloves/ SSD-helmet		
	VOD-FR	Low	Fast-RCNN-quant		Pole-gloves/ SSD-helmet-quant		
	VOD-P	Low	SSD-Main	Posenet			
Vehicle License Plate Detector (LPR)	LPR	Low	Tiny-yolov3-quant	Wpod	OCR-recognizer	Traffic cameras 1 week, 20 cameras [28]	Resolution: : 960*540 FPS:30
Nameplate Identification (NI)	NI	Middle	SSD-Main	Text detector	OCR-recognizer	Power grid site 1 week, 3 cameras	Resolution: 416*416 FPS:30
	NI-FR	Middle	Fast-RCNN-quant				

– *FIFO Algorithm.* Originally designed to optimize the scheduling of a DNN service graph on heterogeneous edge nodes while minimizing total latency under resource constraints, we have modified this algorithm to suit the on-device heterogeneous processors' environment.

– *LSTM-`Niagara` algorithm* uses the LSTM model as the dynamic predictor, and other parts are the same as `Niagara`. We LSTM-Niagara to evaluate our dynamic predictor efficiency.

Evaluation Scenarios. The assessment of `Niagara` encompasses 3 real application (video surveillance) scenarios encompassing 8 distinct service combination patterns, as outlined in Table 3. These scenarios make use of a range of pretrained DNN models, including publicly available models and those developed by the authors, such as SSD-Helmet and pole-gloves.

Table 4. Experimental platforms.

SoC	Description	capacity
SnapDragon 865 [10]	CPU: 4*Kryo 585(A77)	400%
	GPU: Adreno 650	100%
	DSP: Hexagon 698	100%
SnapDragon 855 [9]	CPU: 4*Kryo 485(A76)	400%
	GPU: Adreno 640	100%
	DSP: Hexagon 690	100%
SnapDragon 750G [8]	CPU: 2*Kryo 570(A76)	200%
	GPU: Adreno 619	100%
	DSP: Hexagon 694	100%

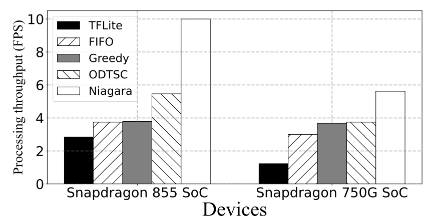


Fig. 3. Processing throughput of VOD atop two different devices.

Evaluation Datasets. The evaluation dataset comprises three video streams, with two of them collected from real-world environments where `Niagara` has been deployed, and one sourced from open repositories commonly utilized in edge service benchmarks, as meticulously delineated in Table 3. All videos have undergone uniform preprocessing to attain a frame rate of 30 frames per second (fps), thus ensuring evaluation consistency.

The complexity classification, as presented in Table 3, elucidates the number of services encompassed within a given request. Here, "high", "middle", and "low" denote the presence of more than 10 services, 7-10 services, and less than 7 services, respectively.

4.2 Experiment Results

Different combinations We evaluate 8 combinations in Table 3 in three real scenarios, as shown in Figure 4. Each pipeline's result is averaged over 100 same

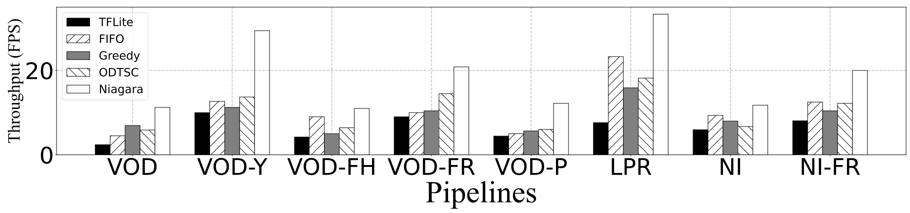
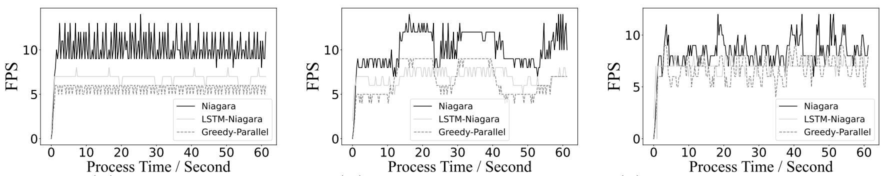


Fig. 4. Processing throughput of all eight experimental combinations.



(a) Stable (b) Slowly-changing (c) Frequently-changing

Fig. 5. Throughput of one-minute real videos in three different situations.

requests. Overall, `Niagara` achieves a 3.0×, 1.9×, 2.0×, and 1.8× throughput improvement compared with TFLite, FIFO, Greedy, and ODTSC on average, respectively. That is because our strategy jointly considers batch and parallel execution with DNN inference service-to-hardware selection. As the scenario is more complex, the benefits `Niagara` obtains are more. VOD-Y is one of the best examples. It uses a tiny-yolov3-quant model for person detection service, which consumes the least hardware utilization. Thus, this service can be parallelized with any other DNN services on the CPU, significantly reducing the critical path length. On the contrary, the nameplate identification (NI) pipeline's performance improvement is not so obvious because the person detection service consumes lots of hardware resources, and no one can be parallelized with it

Different edge devices. We also evaluate `Niagara` on different edge devices, as shown in Table 4. From Figure 3, `Niagara` always achieves the lowest delay compared with the other four baselines. For instance, `Niagara`'s throughput is 10 FPS on Snapdragon 855 SoC development board, while 5.46 FPS, 3.78 FPS, 3.74 FPS, and 2.84 FPS under ODT-SC CP, Greedy, FIFO, and TFLite baselines, respectively. On Snapdragon 750G SoC development board, `Niagara` can achieve 4.56×, 1.87×, 1.52×, and 1.50× higher throughput, respectively.

Besides, comparing the two figures, Snapdragon 855 SoC achieves better performance improvement than Snapdragon 750G SoC. That is because 855 SoC has a higher-performance SoC with a four-core CPU, while 750G SoC only has two, providing more scheduling space for `Niagara` to exploit.

Real deployment. We have successfully deployed `Niagara` in an electric station and conducted evaluations in three typical situations: stable, slowly changing, and frequently changing, with a focus on violation operation detection (VOD in Table 3). Additionally, we also analyzed `Niagara`'s performance over a 20-minute work period to assess its efficiency.

The evaluation results, shown in Figure 5 and 6, demonstrate the effectiveness of `Niagara` compared to state-of-the-art baselines. `Niagara` achieves throughput

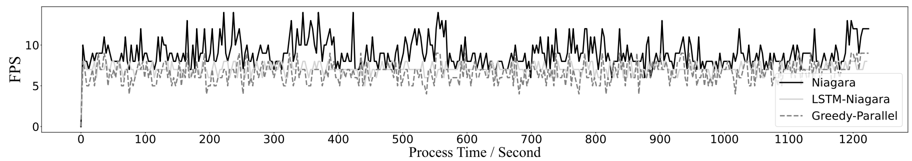


Fig. 6. Throughput comparison of a 20-minute real video.

improvements ranging from 1.26 to 2.33 $\times$. Particularly, in scenarios with more stable content, `Niagara` provides greater benefits, e.g., Figure 5(a) and Figure 6 200-250s. This can be attributed to `Niagara` accurately predicting unforeseen service graphs, providing more scheduling space, which enables better utilization of its offline strategies.

5 Discussion

Applicability of NPU in edge devices. Many contemporary edge devices are furnished with Neural Processing Units (NPUs), such as the Kirin 9000 [4]. Since `Niagara` is a hardware-agnostic framework, the integration of support for new NPUs entails minimal alterations to existing algorithms and system design. This integration process primarily involves the addition of NPU-specific support implementations, encompassing profiling, hardware configurations, and hardware status monitoring. Actually, `Niagara` already extends its support to NPUs, with experimental deployments showcasing its compatibility with a particular NPU architecture (Hexagon DSP) developed by Qualcomm.

6 Conclusion

This work proposed `Niagara` to achieve high throughput for serving DNN inference services on edge devices. `Niagara` proposes an offline algorithm for the on-edge-device DNN inference service scheduling problem. It then applies the template scheduling strategies to the variable unforeseen DNN cascades application with the help of an input predictor, processor monitor, and strategy matcher. We have implemented a prototype of `Niagara` on commodity edge devices and comprehensively evaluate its effectiveness via a set of experiments on typical DNN inference service scenarios.

Acknowledgement

This work was supported by the National Key Research and Development Program of China under the grant number 2022YFB4500700, the National Natural Science Foundation of China under the grant numbers 62325201, 62172008, 62102009, and 62102045, the National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas), the China Postdoctoral Science Foundation 8206300713, the Beijing Outstanding Young Scientist Program under the grant number BJJWZYJH01201910001004, and Center for Data Space Technology and System, Peking University.

References

1. Cortex a57, https://en.wikipedia.org/wiki/ARM_Cortex-A57
2. Gurobi solver, http://www.gurobi.com
3. Jetson tx2, https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2/
4. Kirin 9000, https://www.hisilicon.com/cn/products/Kirin/Kirin-flagship-chips/Kirin-9000
5. Powerful 64-bit heterogeneous processing, advanced analytics and 4g lte redefine the ip camera, https://www.edge-ai-vision.com/2015/11/qualcomm-announces-ip-camera-reference-platform-with-high-end-processing-imaging-and-analytics-capabilities-to-advance-security-cameras/
6. Qualcomm snapdragon 625 ip camera, https://anyconnect.com/recommended-sbcs/thundercomm/thundercomm-qualcomm-snapdragon-625-ip-camera
7. Snapdragon 650 ip camera brings consciousness to camera security, https://www.qualcomm.com/news/onq/2016/02/snapdragon-650-ip-camera-brings-consciousness-camera-security
8. Snapdragon 750g soc, https://www.qualcomm.com/products/mobile/snapdragon/smartphones/snapdragon-7-series-mobile-platforms/snapdragon-750g-5g-mobile-platform
9. Snapdragon 855 soc, https://www.qualcomm.com/products/mobile/snapdragon/smartphones/snapdragon-8-series-mobile-platforms/snapdragon-855-mobile-platform
10. Snapdragon 865 soc, https://www.qualcomm.com/products/mobile/snapdragon/smartphones/snapdragon-8-series-mobile-platforms/snapdragon-865-plus-5g-mobile-platform
11. Tflite., https://www.tensorflow.org/lite/
12. Edge tpu. https://github.com/XiaoMi/mace (2021)
13. Almeida, M., Laskaridis, S., Mehrotra, A., Dudziak, L., Leontiadis, I., Lane, N.D.: Smart at what cost? characterising mobile deep neural networks in the wild. In: ACM IMC. pp. 658–672 (2021)
14. Chai, F., Zhang, Q., Yao, H., Xin, X., Gao, R., Guizani, M.: Joint multi-task offloading and resource allocation for mobile edge computing systems in satellite iot. IEEE Trans. Veh. Technol. **72**(6), 7783–7795 (2023)
15. Danielsson, P.E.: Euclidean distance mapping. Computer Graphics and image processing **14**(3), 227–248 (1980)
16. Diggle, P., Al-Wasel, I.: Time series. (1990)
17. Dorigo, M., Gambardella, L.M.: Ant colonies for the travelling salesman problem. biosystems **43**(2), 73–81 (1997)
18. Eshraghi, N., Liang, B.: Joint offloading decision and resource allocation with uncertain task computing requirement. In: IEEE INFOCOM. pp. 1414–1422 (2019)
19. Fu, X., Tang, B., Guo, F., Kang, L.: Priority and dependency-based dag tasks offloading in fog/edge collaborative environment. In: CSCWD. pp. 440–445 (2021)
20. Hu, S., Zou, G., Zhang, B., Wu, S., Lin, S., Gan, Y., Chen, Y.: Temporal-aware qos prediction via dynamic graph neural collaborative learning. In: ICSOC. pp. 125–133. Springer (2022)
21. Huang, V., Wang, C., Ma, H., Chen, G., Christopher, K.: Cost-aware dynamic multi-workflow scheduling in cloud data center using evolutionary reinforcement learning. In: ICSOC. pp. 449–464. Springer (2022)

22. Jeong, J.S., Lee, J., Kim, D., Jeon, C., Jeong, C., Lee, Y., Chun, B.G.: Band: coordinated multi-dnn inference on heterogeneous mobile processors. In: ACM MobiSys. pp. 235–247 (2022)
23. Kim, Y., Kim, J., Chae, D., Kim, D., Kim, J.: μlayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization. In: EuroSys. pp. 1–15 (2019)
24. Li, Z., Yang, C., Huang, X., Zeng, W., Xie, S.: Coor: Collaborative task offloading and service caching replacement for vehicular edge computing networks. IEEE Trans. Veh. Technol. pp. 1–6 (2023)
25. Liao, H., Li, X., Guo, D., Kang, W., Li, J.: Dependency-aware application assigning and scheduling in edge computing. IEEE IoT (2021)
26. Liu, J., Ren, J., Zhang, Y., Peng, X., Zhang, Y., Yang, Y.: Efficient dependent task offloading for multiple applications in mec-cloud system. IEEE TMC (2021)
27. Meng, Z., Xu, H., Huang, L., Xi, P., Yang, S.: Achieving energy efficiency through dynamic computing offloading in mobile edge-clouds. In: IEEE MASS. pp. 175–183. IEEE (2018)
28. Shen, H., Chen, L., Jin, Y., Zhao, L., Kong, B., Philipose, M., Krishnamurthy, A., Sundaram, R.: Nexus: a gpu cluster engine for accelerating dnn-based video analysis. In: ACM SOSP. pp. 322–337 (2019)
29. Sze, V., Chen, Y.H., Yang, T.J., Emer, J.S.: Efficient processing of deep neural networks: A tutorial and survey. IEEE **105**(12), 2295–2329 (2017)
30. Tan, T., Cao, G.: Fastva: Deep learning video analytics through edge processing and npu in mobile. In: IEEE INFOCOM. pp. 1947–1956. IEEE (2020)
31. Wang, M., Ding, S., Cao, T., Liu, Y., Xu, F.: Asymo: scalable and efficient deep-learning inference on asymmetric mobile cpus. In: ACM MobiCom. pp. 215–228 (2021)
32. Wei, T., Zhang, P., Dong, H., Jin, H., Bouguettaya, A.: Mobility-aware proactive qos monitoring for mobile edge computing. In: ICSOC. pp. 134–142 (2022)
33. Wei, W.W.: Time series analysis. In: The Oxford Handbook of Quantitative Methods in Psychology: Vol. 2 (2006)
34. Xiao, H., Xu, C., Ma, Y., Yang, S., Zhong, L., Muntean, G.M.: Edge intelligence: A computational task offloading scheme for dependent iot application. IEEE Wirel Commun **21**(9), 7222–7237 (2022)
35. Xu, M., Zhang, X., Liu, Y., Huang, G., Liu, X., Lin, F.X.: Approximate query service on autonomous iot cameras. In: ACM MobiSys. pp. 191–205 (2020)
36. Yang, Y., Chen, G., Ma, H., Zhang, M.: Dual-tree genetic programming for deadline-constrained dynamic workflow scheduling in cloud. In: ICSOC. pp. 433–448. Springer (2022)
37. Yeo, H., Chong, C.J., Jung, Y., Ye, J., Han, D.: Nemo: enabling neural-enhanced video streaming on commodity mobile devices. In: ACM MobiCom. pp. 1–14 (2020)
38. Yi, J., Lee, Y.: Heimdall: mobile gpu coordination platform for augmented reality applications. In: ACM MobiCom. pp. 1–14 (2020)
39. Zhang, J., Zhang, D., Xu, X., Jia, F., Liu, Y., Liu, X., Ren, J., Zhang, Y.: Mobipose: Real-time multi-person pose estimation on mobile devices. In: ACM SenSys. pp. 136–149 (2020)
40. Zhao, G., Xu, H., Zhao, Y., Qiao, C., Huang, L.: Offloading tasks with dependency and service caching in mobile edge computing. IEEE Transactions on Parallel and Distributed Systems **32**(11), 2777–2792 (2021)
41. Zhao, Z., Luo, H., Chu, S.C., Shang, Y., Wu, X.: An immersive online shopping system based on virtual reality. J. Netw. Intell. **3**(4), 235–246 (2018)