Real-Time Multitasking of Deep Neural Networks With Nvidia TensorRT

Federico Aromolo*, Andrea Stevanato*†, Alessandro Biondi*, and Giorgio Buttazzo*
*Department of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Pisa, Italy
†Evidence s.r.l., Pisa, Italy

***Abstract*—Graphics processing units (GPUs) are often employed to accelerate the inference of deep neural networks (DNNs) in cyber-physical systems to implement advanced perception and control functionalities. Frameworks for GPU-accelerated DNN inference typically aim at maximizing the processing throughput rather than focusing on providing a predictable timing behavior, which is crucial for time-sensitive cyber-physical systems. This work proposes a framework for GPU-accelerated inference of DNNs on GPU-based embedded platforms in multitasking scenarios, which provides enhanced timing predictability using a design-time optimization procedure of the DNN workload and a specialized method to schedule the GPU acceleration requests of the DNNs at runtime based on fixed-priority limited-preemptive scheduling. Fine-grained control of the inference is achieved by splitting the DNNs into smaller chunks, which are then scheduled using a specialized real-time scheduling mechanism. Experimental results on commercial embedded platforms report significant improvements in terms of schedulability.**

***Index Terms*—Deep neural networks, graphics processing units, real-time systems, schedulability analysis, multitasking.**

I. Introduction

Deep neural networks (DNNs) have become a prevalent technology for enabling advanced perception and control functions in cyber-physical systems (CPSs). In particular, convolutional neural networks proved to be particularly effective at augmenting detection capabilities in tasks such as object detection, classification, tracking, and image segmentation. Consequently, DNNs can provide enhanced perception performance in several domains, including advanced driving assistance systems, autonomous driving, industrial robotics, medical devices. DNN models consist of numerous sequential layers of artificial neurons, where each neuron connects to neurons in preceding and subsequent layers through multiple weighted connections, establishing a highly parallel computational structure. This structure exhibits substantial data and instruction parallelism, making DNN execution (or *inference*) well-suited for acceleration on graphics processing units (GPUs). Given the high performance and safety requirements in CPS applications, DNN inference is often subject to stringent timing constraints, typically expressed as deadlines by which inference must terminate to ensure a safe operation. Furthermore, typical cyber-physical applications consist of multiple tasks that request acceleration for different DNN models. A representative example is autonomous vehicles, which rely on multiple DNNs to handle different simultaneous tasks such as object detection, semantic segmentation, object tracking, and DNN-enhanced localization and mapping. Typical sensor setups include four to eight cameras, along with lidars and radars [1]–[3]. Each sensor stream is processed by one or more specialized DNNs to extract relevant information. For instance, deployments based on the Autoware framework may involve multiple DNNs per sensor stream, leading to a substantial number of concurrent DNN tasks [4], [5]. These tasks often have different priorities and timing requirements, including different periods and deadlines [3], [4]. However, leveraging existing GPU acceleration frameworks to run multiple DNNs concurrently (which is a typical requirement in cyber-physical applications) results in a timing behavior that existing scheduling models and worst-case timing analysis techniques for real-time systems cannot accurately capture.

Contributions. This paper investigates the problem of accelerating the inference of concurrent DNN workloads on GPU-based heterogeneous Systems-on-a-Chip (SoCs) while ensuring enhanced timing predictability, with reference to commercial Nvidia GPU-based platforms. A specialized framework for GPU-accelerated inference of DNNs on embedded platforms is presented, combining a design-time optimization procedure with a runtime scheduling and resource management system based on fixed-priority limited-preemptive scheduling. Building upon Nvidia TensorRT [6] to enable high-performance inference of DNNs on embedded GPUs, the framework works around the uncertainties in the timing behavior of the GPU scheduler and of the default concurrent inference pattern of TensorRT by treating the TensorRT runtime system and the GPU driver as a black box and assigning the GPU resource to tasks requesting DNN acceleration based on careful resource management decisions. The framework further leverages *DNN splitting*, where each DNN to be accelerated is divided into multiple sequential chunks, each composed of consecutive layers of the original DNN, thus enabling finer control granularity in the GPU inference. The paper makes the following contributions: **(i)** it provides a characterization of the TensorRT inference framework, focusing on its timing behavior during concurrent inference on Nvidia Jetson GPU-based SoCs (Sec. IV); **(ii)** it presents a specialized framework enabling real-time, concurrent accelerated inference of DNN workloads that leverages TensorRT, incorporates predictable resource management and supports DNN models split into multiple chunks (Sec. V); **(iii)** it presents a schedulability analysis based on response-time analysis to capture the worst-case timing behavior of the system with periodic workloads (Sec. VI);

(iv) it presents a profiling-based method to determine an optimal DNN splitting configuration that guarantees schedulability while minimizing splitting overhead, accompanied by a heuristic strategy providing high schedulability with reduced profiling efforts (Sec. VII); and **(v)** it reports an experimental evaluation of the proposed framework and DNN splitting methods in terms of analytical and observed schedulability on Nvidia Jetson platforms, including a comparison with the baseline TensorRT concurrent inference approach (Sec. VIII).

II. Background

This section presents the reference platform architecture and application model, and provides an overview of machine learning frameworks for GPU-accelerated inference.

Platform architecture. GPU-based SoC architectures combine scalar multi-core processors with an integrated GPU device. Typically, the CPU processor combines cores of different capability, while the GPU contains a large number of processing elements specialized for graphics and parallel computation. In embedded GPU-based SoCs, the memory controller fabric is shared between the CPU and GPU, both connected to a single off-chip DRAM module. This configuration enables sharing memory buffers between the CPU and GPU, eliminating the data copy overheads typical of discrete GPU configurations, in what is known as a *zero-copy* configuration. Both CPU and GPU feature a dedicated set of cache memory layers to exploit data locality. The resulting platform model is representative of commercial SoC heterogeneous architectures, such as the Nvidia Jetson family of devices [7]. For practical evaluations, we considered three Jetson models: **(i)** the Jetson TX2 [8] (GPU with 256 CUDA cores, six-core CPU, 8 GB of RAM); **(ii)** the Jetson AGX Xavier [9] (GPU with 512 cores, eight-core CPU, 32 GB of RAM); and **(iii)** the Jetson AGX Orin [7] (GPU with 2048 cores, twelve-core CPU, 32 GB of RAM).

Application model. As the focus of this work is on multitasking in cyber-physical applications involving concurrent GPU-accelerated inference of DNNs, we model an application as a task set $\tau = \{\tau_1, \ldots, \tau_n\}$ of n sporadic tasks that execute on the cores of a multiprocessor platform according to a preemptive fixed-priority scheduling policy. The tasks in τ utilize a GPU device integrated on the SoC to accelerate the inference of DNN models on the GPU. Specifically, each job of a given task τ_i can perform a single inference request for a specific DNN model DNN_i, by means of a blocking call to an underlying inference framework. Each task τ_i releases a potentially infinite sequence of jobs with a minimum inter-arrival time T_i and is subject to a deadline D_i, with $D_i \leq T_i$ (constrained deadlines). Each task is assigned a fixed priority level π_i, for instance, according to the Deadline Monotonic (DM) priority assignment, where tasks with smaller deadlines have higher priority. The priority of a task τ_i is considered higher than that of another task τ_j if $i < j$.

DNN inference with TensorRT. Numerous machine learning frameworks are available to support training and inference of DNN models. PyTorch [10], TensorFlow [11], and Caffe [12] are widely adopted open-source frameworks in both academic research and industrial applications. Their popularity stems from their ability to simplify the development and deployment of DNNs, thanks to high-level abstractions and user-friendly APIs. All three frameworks support GPU-accelerated inference; however, their focus is more on flexibility and ease of use rather than on performance. Nvidia TensorRT [6] is a specialized framework for model optimization and high performance inference on Nvidia GPUs and embedded SoCs. TensorRT applies optimizations such as layer fusion, profiling-based kernel tuning, and precision calibration, also leveraging specific knowledge of the internals of the target GPU architecture, to obtain an optimized DNN inference engine providing enhanced inference speed and resource efficiency. While TensorRT does not support the DNN training phase, it accepts standard modeling formats such as ONNX, enabling interoperability with other frameworks, including PyTorch, TensorFlow, and Caffe.

Input models to be accelerated using TensorRT can be either converted from existing modeling frameworks or manually prepared using the TensorRT API. The TensorRT engine builder takes as input a trained DNN model definition and a selected numerical precision (e.g., floating point or integer), and generates an optimized TensorRT inference engine for that precision. To minimize the engine execution time on the specific platform on which the engine is built, the engine builder collects timing and profiling data over multiple runs and selects the fastest available compute kernel for each layer in the DNN model. The resulting engine can be utilized for inference using the TensorRT inference library, available for C++ and Python. While the inference engine is optimized for high-performance inference on the specific target platform on which it is built, it remains compatible with any system that supports TensorRT. Given an optimized TensorRT engine, profiling of the DNN inference times from the point of view of the tasks on the CPU side can be performed using wall-clock timers, whereas profiling from the GPU side can be performed using CUDA event monitoring or the built-in TensorRT profiling tool.

TensorRT engines are obtained as a set of CUDA compute kernels, each accelerating in parallel form one or more layers and operations of the DNN model. As with general kernels in Nvidia GPUs, each kernel composing an engine is executed non-preemptively on the GPU resources it utilizes, meaning that it cannot be suspended to yield the corresponding resources to other kernels executing concurrently. Different kernels composing TensorRT engines may request different GPU resources in different amounts; as a result, when switching from one kernel to the next in one engine, the type and amount of resources requested by the same engine might change. Examples of compute resources that are often requested by such kernels include execution threads within a compute core in a Streaming Multiprocessor (SM), floating point units in each SM, or memory buffers of different types. Note that the internal structure of each TensorRT engine (in terms of type, size, and number of kernels) is not known a priori, as it depends on the results of the optimization performed while building the engine, which in turn depends on multiple factors, including the target GPU architecture and the current occupancy of GPU resources at the time of optimization.

III. Related work

Existing works on GPU computing for embedded systems focused on characterizing the timing behavior of GPU platforms [13]–[15] and deriving suitable scheduling strategies to enhance timing predictability [16]–[18].

Numerous works more specifically focused on improving the timing behavior of DNN inference on GPU-based heterogeneous platforms. Zhou et al. [19] presented a scheduling framework for multitasked systems in GPU-based heterogeneous platforms, which supports prioritization and scheduling of multiple DNN instances to improve GPU resource utilization. Gujarati et al. [20] proposed another scheduling framework for GPU-based heterogeneous platforms with specialized GPU resource utilization strategies which enhance the resulting timing predictability of the system. Multi-tenant DNN inference on GPUs was investigated by Yu and Bray [21], providing a specialized kernel-level scheduling strategy. Ling et al. [22] presented a framework for real-time scheduling of DNNs focusing on supporting DNNs with largely different size and accuracy requirements. Lee et al. [23] presented an optimization strategy for DNN inference performed by a single periodic task, which dynamically deactivates neurons to speed up the inference. Han et al. [24] described a preemption mechanism for real-time GPU kernels that enables preemption while minimizing associated overhead. In the context of memory management, Jain et al. [25] proposed a strategy to mitigate memory contention on the GPU by partitioning computational and memory resources, with added compatibility for DNN inference. Kang et al. [26] also operated on memory aspects, proposing a framework designed to overcome GPU memory limitations when handling multiple DNNs by dynamically swapping memory between active and inactive DNN models. All of these methods, however, do not provide a dedicated response-time analysis of the resulting timing behavior. Closer to our work, Xiang et al. [3] focused on hybrid CPU-GPU inference of DNN models in a multitasking environment, providing a timing analysis, but does not support TensorRT and performs inference with multiple parallel streams, from which TensorRT does not necessarily benefit (see Sec. IV). Later, Kang et al. [27] extended this approach to provide enhanced schedulability and flexibility in hybrid CPU-GPU inference of DNN models. Both works schedule DNNs at the layer level, effectively applying DNN splitting at the finest granularity. In addition to causing overheads at every layer boundary, layer-level splitting would inhibit key TensorRT optimizations such as layer fusion, kernel auto-tuning, and global memory reuse, degrading the overall inference performance. Furthermore, the optimization goals in these works focus on allocating the DNN layers among the available CPU and GPU nodes. In contrast, this work proposes a coarser-grained splitting scheme that minimizes the number of splits required to meet timing constraints, introducing splits only when necessary to ensure schedulability, and preserving inference performance as much as possible. Furthermore, all of the above works operate at a low level, requiring explicit knowledge of the internal computational structure of the DNN models (e.g., in terms of CUDA kernel configurations and memory access patterns) for scheduling and resource management. Differently, the proposed method operates at the DNN architecture level, treating each TensorRT engine as a black-box component. It requires no internal execution details, relying solely on chunk-level profiling data. DNN splitting (or DNN *partitioning*) has also been widely adopted in distributed computing to determine how to divide DNN inference across multiple nodes (e.g., between edge and cloud devices) [28]–[42]. However, these works focus on throughput or energy efficiency maximization via static or dynamic resource allocation and do not provide worst-case timing guarantees, therefore lacking real-time schedulability analysis.

TABLE I: Overview of related work.

Related work	**Multi-tasking**	**Response-time analysis**	**TensorRT support**	**Transparent integration**
Zhou et al. [19]	✓	✗	✗	✗
Lee et al. [23]	✗	✗	✗	✗
Ling et al. [22]	✓	✗	✗	✗
Han et al. [24]	✓	✗	✗	✗
Gujarati et al. [20]	✓	✗	✗	✗
Jain et al. [25]	✓	✗	✗	✗
Xiang et al. [3]	✓	✓	✗	✓
Kang et al. [26]	✓	✗	✗	✓
Kang et al. [27]	✓	✓	✗	✓
Yu and Bray [21]	✓	✗	✗	✗
This Paper	✓	✓	✓	✓

Table I summarizes the key features of related work on real-time DNN inference compared to our framework. Our framework is the first to manage inference of multiple DNN tasks by leveraging the TensorRT high-performance inference framework, while providing response-time analysis and ensuring seamless integration of existing applications with existing machine learning frameworks and model formats, without requiring any modifications to the underlying software stack, including TensorRT and GPU drivers.

IV. Baseline approach

This section presents a baseline approach for managing concurrent inference of DNN models with TensorRT under the application model described in Sec. II. This approach is then characterized based on empirical observations, highlighting its main limitations.

Baseline approach for concurrent inference. When TensorRT is considered for DNN inference, the application model in Sec. II can be supported by assigning a dedicated CUDA stream (i.e., a container for sequential operations to be executed on the GPU) to each task in the application, on which the task independently performs engine inference, as recommended in the TensorRT documentation [6]. To preserve the correct priority order when managing concurrent inference requests, the priority level of the CUDA stream dedicated to each given task τ_i should correspond to the priority level π_i

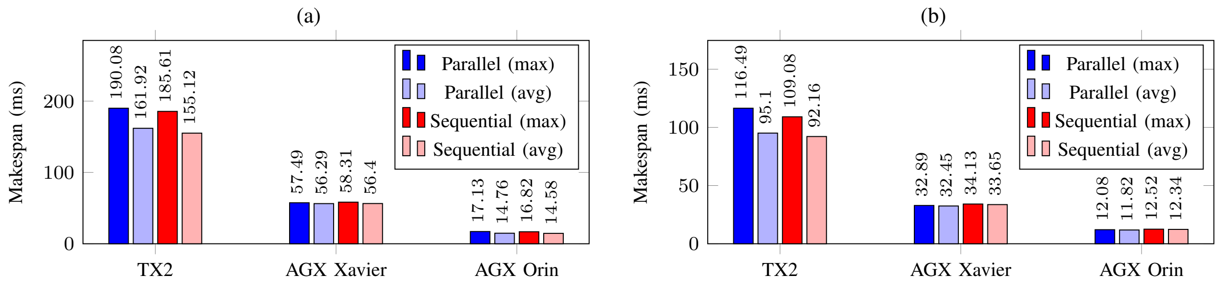

Fig. 1: Comparison of makespan obtained with concurrent and sequential inference on different Jetson platforms. Models considered: (a) VGG19, AlexNet, ResNet18, and InceptionV4; (b) GoogLeNet, MobileNetV2, VGG16, and DenseNet121.

in τ. With this setup, when multiple compute kernels from concurrent TensorRT engines are ready for execution at the same time, then those kernels will execute in parallel on the GPU, provided that sufficient GPU resources are available.

Empirical characterization of the scheduling behavior. Since the internal behavior of TensorRT, Nvidia GPUs and their related drivers is only partially documented in publicly available specifications, it is not possible to precisely validate the scheduling policy governing acceleration of concurrent TensorRT engines. Thus, we characterized the scheduling behavior under the baseline configuration through empirical analysis, using profiling and targeted analysis of execution traces collected on various platforms of the Nvidia Jetson family (namely, Jetson TX2 [8], Jetson AGX Xavier [9], and Jetson AGX Orin [7]) while executing different pre-trained DNN models compatible with TensorRT concurrently under the baseline setup. According to our characterization, the baseline scheduling behavior involves sequential execution of kernels within each DNN engine, while kernels from different engines may partially execute in parallel depending on the availability of different types of GPU resources at any given time, since two concurrent kernels can execute simultaneously only when sufficient GPU resources are available to accommodate both at the same time. Kernels are executed non-preemptively on the resources they are assigned, and a kernel belonging to an engine assigned to a higher-priority stream takes precedence over kernels from lower-priority engines, provided that sufficient resources are available to execute it.

One crucial result of this analysis is that the amount of parallel execution observed for GPU kernels that are released by different concurrent TensorRT engines is generally very low in the evaluated scenarios: in all the recorded execution traces, we observed minimal overlap in the execution of kernels from engines that were running concurrently, with the Nvidia profiling tools reporting very low values for the kernel concurrency metric (as low as 1%). To confirm this observation, we evaluated the makespan for the inference of four TensorRT engines by either **(i)** requesting concurrent inference of the engines on separate CUDA streams, and measuring the maximum completion time for the concurrent requests; or **(ii)** running each engine in isolation and then summing up the execution time of each engine, as in sequential inference[1]. Figure 1(a) reports the resulting statistics for the two strategies, in terms of average and maximum makespan across a total of 500 executions, when considering a set of four pre-trained DNN models by Nvidia [43] (VGG19 [44], AlexNet [45], ResNet18 [46], and InceptionV4 [47]). Interestingly, on both the TX2 and the AGX Orin platforms, concurrent inference resulted in slightly longer makespan than with sequential inference, whereas similar makespans were observed for concurrent and sequential execution on the AGX Xavier. Similar trends are observed in Figure 1(b), where a different set of pre-trained DNN models [43] is considered (GoogLeNet [48], MobileNetV2 [49], VGG16 [44], and DenseNet121 [50]). Overall, these observations show that executing the kernels concurrently, such as in the baseline approach, does not guarantee a gain in the achieved parallelism; in fact, it might even cause a decrease in performance due to the underlying effects of architectural resource contention. This limited observed concurrency is possibly related to the following factors: **(i)** to minimize the expected execution time for an engine, the TensorRT engine builder tends to produce kernels that utilize as many GPU resources as possible in order to speed up the inference, resulting in high resource requirements for each kernel[2]; and **(ii)** the computational resources available on embedded GPUs are quite limited with respect to discrete high-performance GPUs; thus, in practice, the execution of a single TensorRT engine often tends to saturate at least one type of GPU resource.

A. *Analytical limitations*

In addition to the above observations, the baseline scheduling approach presents several limitations in terms of real-time performance, related to scheduling and timing analysis aspects.

Limitation 1. *Analytical challenges.* The baseline approach produces a highly complex and unpredictable timing behavior

[1]In both cases, inference was carried out with the TensorRT C++ runtime API, and wall-clock time was measured on the CPU using high-resolution monotonic C++ timers in order to capture the total inference latency, including actual GPU inference time and any additional synchronization overheads.

[2]The TensorRT documentation notes that resource usage can be bounded by occupying GPU resources during engine optimization. However, since the resource usage of each kernel is statically determined, this strategy may increase kernel execution times and result in underutilized GPU resources.

that is difficult to properly capture with a worst-case timing analysis. Indeed, an accurate analysis would need to: **(i)** model the internal structure of each TensorRT engine in terms of types of kernels and corresponding GPU resource requirements; **(ii)** keep track of the available GPU resources to determine whether pending kernels from different engines can execute in parallel at any given time; **(iii)** consider potentially alternating intervals of parallel execution and interleaving of concurrent engines; **(iv)** account for the blocking delays caused on higher-priority engines by non-preemptive execution of lower-priority kernels when not enough GPU resources are available to execute a high-priority kernel; and **(v)** consider multiple preemption points for each lower-priority engine, as determined by the subdivision of the engines in multiple kernels. To the best of our knowledge, even when assuming the availability of complete specifications regarding the scheduling system of Nvidia GPU devices and TensorRT engines, there is no suitable analysis method that can be applied to derive precise and reliable schedulability guarantees under this baseline approach.

Limitation 2. *Measurement uncertainty.* Concurrent execution of kernels introduces significant expected variability in terms of execution times: reliable estimations of execution times would thus require extensive profiling to properly capture the effects of resource contention inherent to concurrent execution.

Limitation 3. *Limited priority levels.* Only a limited number of priority levels are available for CUDA streams in the GPUs onboard the Jetson platforms (up to 2 priority levels on the TX2, 6 on the AGX Xavier, and 6 on the more recent AGX Orin). This limits the number of task priorities that can be properly supported with the baseline approach, meaning that, if more priority levels are required, engines utilized by tasks with different priority might be forced to share the same priority level on the GPU.

Overall, from a purely empirical perspective, our experiments on the schedulability performance of concurrent inference (detailed later in Sec. VIII) report a large number of deadline misses for the baseline approach, further highlighting the need for a specialized approach for time-critical systems.

V. Scheduling framework

This section provides an overview of the architecture and the main components of the proposed framework, which overcomes the limitations of the baseline approach by guaranteeing a predictable and analyzable timing behavior in concurrent DNN inference on GPU-based SoCs.

The proposed framework integrates an optimization approach driven by schedulability analysis with a specialized runtime mechanism for optimized DNN inference engines. Specifically, the workflow of the framework is illustrated in Figure 2. The optimization module takes as inputs the scheduling parameters of the real-time tasks and the set of trained DNN models utilized by the tasks, and produces a set of optimized DNN models to be executed by a specialized scheduling system. Accurate profiling of engine execution

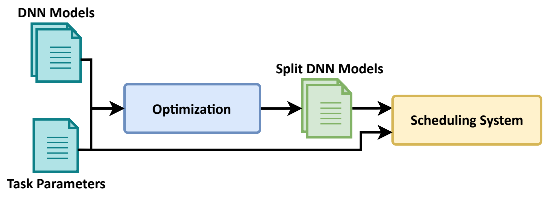

Fig. 2: Overview of the proposed framework.

times is integrated in the optimization module. Inference of the optimized DNN models can then be requested by the tasks in the cyber-physical application by leveraging a specialized scheduling interface provided by the framework.

Given that both the TensorRT framework and the Nvidia GPU driver stack are closed-source, it is not possible to modify their internals to suit a desired behavior. As a result, one crucial design choice of the proposed framework is to adopt a black-box approach: the framework is built as an additional layer on top of the standard TensorRT framework distribution for Nvidia GPU SoCs, without any modification to TensorRT or the GPU device drivers, with the main objective of overcoming the limitations of TensorRT in terms of timing predictability. Specifically, the proposed framework leverages the TensorRT engine builder in the design phase and the TensorRT inference library (which internally utilizes the GPU scheduler) in the runtime phase. From the point of view of the system designer, integrating the proposed framework simply requires applying the optimization tool to the target task set and then invoking inference by means of the framework rather than by leveraging TensorRT directly as in the baseline approach. This guarantees a lightweight and transparent approach that is portable across all platforms compatible with TensorRT. In the following, we describe the two main modules composing the proposed framework: the scheduling system and the optimization module.

A. Scheduling system

Time-predictable execution of concurrent TensorRT engines is obtained in the runtime phase by means of a specialized scheduler, which builds on top of the TensorRT inference library. Specifically, in the proposed framework, concurrent execution of TensorRT engines at runtime is governed according to the following rules:

Rule 1. *At most one engine can be in execution (or pending for execution) on the GPU at any time.* To achieve precise control over GPU scheduling, the framework treats the GPU as a shared resource, allowing at most one pending TensorRT engine inference request to be pending at a time across all GPU streams. This effectively bypasses the default scheduling policy for concurrent inference on different streams (which is only partially documented) and enables a custom scheduling policy to be enacted.

Rule 2. *Concurrent engine inference requests are served according to a priority-based policy.* To select which engine is executing on the GPU at any given time, the framework utilizes a fixed-priority scheduling policy, where each inference request is assigned the priority of the requesting task, and requests are served in order of decreasing priority.

Rule 3. *DNN engines are scheduled non-preemptively.* The TensorRT library does not provide mechanisms for preempting or pausing the execution of an engine. Thus, TensorRT engines are scheduled non-preemptively in the scheduling system: once a task acquires the GPU shared resource (as per Rule 1), it retains it until engine execution completes.

Overall, the above rules enforce a predictable scheduling behavior of concurrent TensorRT engine inference requests on the GPU, effectively corresponding to a non-preemptive fixed-priority scheduling policy on a single-core CPU. Given a worst-case execution time (WCET) C_i for the TensorRT engine of each task τ_i, worst-case response times (WCRTs) for the inference of each engine can be derived with standard response-time analysis techniques [51], [52] by considering the engine inference requests towards the GPU as tasks executing on a single-core processor. This analysis bounds the response time for each task by accounting for the interference from higher-priority workload and the blocking caused by the non-preemptive execution of lower-priority tasks.

As shown in the results from Sec. IV, sequential inference does not necessarily lead to a performance loss with respect to concurrent inference. Therefore, although Rule 1 disallows parallel execution of concurrent engines on the GPU, it is not expected to significantly reduce effective parallelism, at least in embedded GPUs with limited capabilities, such as those in the Jetson platforms. Most importantly, this restriction introduces favorable predictability properties that enable timing analysis, thus addressing Limitation 1 of the baseline approach. In addition, enforcing that at most one engine can be in execution or pending on the GPU at any given time facilitates obtaining reliable estimations of engine execution times by profiling each engine in isolation, given that architectural interference effects are mitigated with mutually exclusive inference. This addresses Limitation 2. Finally, since priority levels are managed explicitly by the scheduling system, this approach overcomes the limitation on the number of priority levels available for CUDA streams (Limitation 3).

B. Optimization phase

Although TensorRT provides optimized inference for state-of-the-art DNN models, their execution times can still be significant (e.g., in the order of milliseconds or tens of milliseconds) relative to the tight timing requirements of cyber-physical applications. Thus, non-preemptive execution of lower-priority inference requests can cause substantial blocking times on higher-priority tasks, which can jeopardize the schedulability of the application. To achieve fine control of DNN scheduling while reducing the blocking times due to non-preemptive execution, the proposed framework introduces a specialized optimization strategy for the DNN models in the application, based on the concept of *DNN splitting*. This technique divides each DNN model into multiple sequential *chunks*, each representing one or more successive layers, and then deploys each chunk as a separate TensorRT engine for inference in the scheduler system. Splitting DNNs into smaller chunks allows to control and limit the blocking time introduced by non-preemptive execution. However, splitting the DNN model into multiple chunks can introduce inference overhead, as multiple TensorRT engines must be executed sequentially to produce the final output. Thus, the proposed optimization techniques used to determine the splitting configuration for each DNN model, detailed in Sec. VII, explore the tradeoff between inference times and blocking by limiting the degree of splitting while ensuring system schedulability. These techniques leverage online profiling to derive reliable worst-case execution times for the execution of each chunk on the scheduling framework, extracted by considering that each chunk executes in isolation on the GPU resources.

Extension of the scheduling system. The following rules are provided to support DNN models composed of multiple chunks (i.e., multiple TensorRT engines), enabling preemption at the frontier of successive chunks.

Rule 4. *Chunks of a split DNN are executed sequentially.* The scheduling system is extended to request the execution of the engines composing a split DNN sequentially, transferring the output of each engine to the next to reproduce the full model inference. The execution of each individual engine composing the DNN is performed in accordance with Rules 1-3.

Rule 5. *The completion of a chunk within DNN inference corresponds to a preemption point for inference on the GPU resource.* The shared GPU resource, accessed exclusively as per Rule 1, is retained by a task after completing a DNN chunk if both of the following conditions hold: **(i)** additional chunks of the DNN remain to be executed, and **(ii)** no higher-priority task is waiting to access the GPU. In all other cases, the task releases the shared resource.

Given these rules, if a higher-priority task requires the inference of one of its chunks, it must wait until the current inference completes. Before executing the next chunk of the currently executing task, the framework checks whether other higher-priority tasks requested the acceleration of a chunk. If there is such a request pending, then the framework starts executing the chunks of the highest-priority pending task, thus effectively implementing limited-preemptive scheduling.

Note that, since each DNN chunk is executed in isolation on the GPU, a single CUDA stream is sufficient to support TensorRT inference of the DNN chunks of all tasks in τ.

C. Implementation details

The runtime system is implemented as a C++ module that uses the TensorRT C++ runtime. At system initialization, the TensorRT engines produced by the optimization module, each representing a DNN chunk, are loaded into main memory and initialized, in order to eliminate runtime overhead from deserializing large data segments. At runtime, DNN scheduling follows Rules 1-5 and is coordinated through a shared resource manager accessed by all DNN tasks. When a task obtains exclusive access to the GPU to perform inference of a DNN chunk, it executes the corresponding TensorRT engine on a dedicated CUDA stream and synchronizes on the stream to wait for inference completion. Each chunk is treated as an independent TensorRT engine, and inference proceeds sequentially across chunks in each DNN by forwarding the output of one chunk as the input to the next. On Jetson platforms, the CPU and GPU share the main memory, which the runtime system leverages to implement *zero-copy* memory management. This method maps each shared buffer required for DNN inference to the same virtual memory region in both the CPU and the GPU address spaces and pins the corresponding pages to prevent swapping. It eliminates the need to copy inputs and outputs between CPU and GPU buffers before and after inference, avoiding the memory copies typical of systems without shared CPU-GPU memory. Furthermore, when executing DNN chunks sequentially, the output of one chunk must be passed as input to the next. To avoid the overhead of copying data between separate buffers, the runtime system allocates a single pair of buffers per task, alternating their roles as input and output across chunks. Specifically, the input and output memory bindings of the TensorRT engine of each chunk are assigned to one of the two buffers in the pair, following this alternation strategy. This approach eliminates intermediate memory copies, rendering the memory management overhead of DNN splitting negligible.

VI. Schedulability analysis

The proposed framework is explicitly designed to produce a timing behavior that is amenable to real-time analysis. This section presents a schedulability analysis to derive worst-case response times for inference activities within the GPU scheduling framework, that is, the worst-case waiting time that each task may experience when requesting DNN inference.

Extended model. When concurrent DNN inference is managed according to the proposed approach, we extend the application model in Sec. II to consider that the inference of the DNN model DNN$_i$ requested by each task is realized with the sequential inference of c_i DNN chunks $\{\tau_{i,1}, \tau_{i,2}, \ldots, \tau_{i,c_i}\}$, each corresponding to the inference of a separate TensorRT engine. By Rules 1-5, execution on the scheduling framework corresponds to *limited-preemptive* fixed-priority scheduling on a single-core processor considering *fixed preemption points* for each inference request, corresponding to the termination instant of each chunk. The chunks of each task τ_i are scheduled on the framework according to the priority level π_i. Each DNN chunk $\tau_{i,j}$ is characterized by a WCET $C_{i,j}$. To accurately account for additional overheads, such as inference setup and CPU-GPU communication, the WCET of each chunk should be extracted by considering the total latency as observed from the perspective of the CPU (e.g., by leveraging wall-clock profiling on the CPU). The overall WCET C_i of τ_i on the scheduling framework is computed as $C_i = \sum_{j=1}^{c_i} C_{i,j}$.

Response-time analysis. Under the fixed preemption points (FPP) model for limited-preemptive scheduling [52], [53], each task τ_i is split into c_i non preemptive chunks (or subjobs), obtained by considering $c_i - 1$ preemption points in the corresponding program code. When considering split DNN inference, these preemption points correspond to the completion of one DNN chunk and the start of the next chunk. The response-time analysis for single-core FPP limited-preemptive fixed-priority scheduling by Buttazzo et al. [52] can be directly applied to derive a WCRT for each inference request of the system. The analysis is as follows. For each task τ_i, let $C_i^{\max} = \max_{k=1,\ldots,c_i}\{C_{i,k}\}$ and $C_i^{\text{last}} = C_{i,c_i}$. The blocking factor for each task is $B_i = \max_{\tau_j | j>i}\{C_j^{\max} - 1\}$. Let τ_i represent the task under analysis. The length I_i of the relevant busy period (specifically, the *level-i active period*) for τ_i is given by the smallest positive fixed point of the following recurrent relation:

$$\begin{cases} I_i^{(0)} = B_i + C_i \\ I_i^{(s)} = B_i + \sum_{\tau_h | h \le i} \left\lceil \frac{I_i^{(s-1)}}{T_h} \right\rceil C_h \end{cases} \tag{1}$$

A response time of τ_i must be computed for each job $\tau_{i,k}$ of τ_i in the busy interval, i.e., for $\tau_{i,1} \ldots \tau_{i,K_i}$, where $K_i = \left\lceil \frac{I_i}{T_i} \right\rceil$. The start time $s_{i,k}$ of $\tau_{i,k}$ is given by the smallest positive fixed point of the following recurrent relation:

$$\begin{cases} s_{i,k}^{(0)} = B_i + C_i - C_i^{\text{last}} + \sum_{\tau_h | h<i} C_h \\ s_{i,k}^{(\ell)} = B_i + kC_i - C_i^{\text{last}} + \sum_{\tau_h | h<i} \left(\left\lfloor \frac{s_{i,k}^{(\ell-1)}}{T_h} \right\rfloor + 1 \right) C_h. \end{cases} \tag{2}$$

The finishing time $f_{i,k}$ of $\tau_{i,k}$ in the busy period is $f_{i,k} = s_{i,k} + C_i^{last}$. The response time of τ_i is $R_i = \max_{k=1,\ldots,K_i}\{f_{i,k} - (k-1)T_i\}$. The task set is schedulable if $R_i \le D_i$ for all τ_i in τ.

The proposed optimization approach also leverages the concept of *blocking tolerance*, which, for a given task τ_i, is denoted by β_i and is defined as the maximum amount of non-preemptive blocking that τ_i can tolerate without missing any of its deadlines [52]. This value is useful in the optimization to determine how to split the DNN models to obtain a schedulable task set, which corresponds to the problem of selecting the preemption points in a set of limited-preepmtive tasks to achieve schedulability. To compute β_i, the blocking tolerance $\beta_{i,k}$ of $\tau_{i,k}$ is first computed as:

$$\beta_{i,k} = \max_{t \in \Pi_{i,k}} \left\{ t - kC_i + C_i^{\text{last}} - \sum_{\tau_h | h>i} \left(\left\lfloor \frac{t}{T_h} \right\rfloor + 1 \right) C_h \right\}, \tag{3}$$

where

$$\begin{aligned} \Pi_{i,k} = {} & \left((k-1)T_i, (k-1)T_i + D_i - C_i^{\text{last}}\right] \cap \\ & \{hT_{j-1}, \forall h \in \mathbb{N}, j \le i\} \cup \left\{(k-1)T_i + D_i - C_i^{\text{last}}\right\}. \end{aligned} \tag{4}$$

Then, the blocking tolerance of τ_i is given by $\beta_i = \min_{k=1,\ldots,K_i}\{\beta_{i,k}\}$.

VII. Optimization algorithms

This section describes the splitting algorithms adopted in the design-time optimization phase.

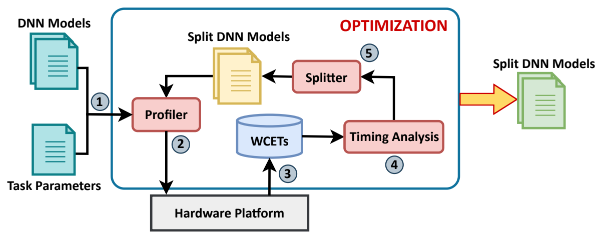

Fig. 3: Structure of the optimization module.

Figure 3 presents an overview of the proposed multi-stage optimization procedure. The process begins with a preliminary stage ① that builds and profiles ② the TensorRT engines for the non-split versions of the DNN models requested by the tasks in τ. The resulting WCETs, obtained by profiling the configuration on the target platform ③, are then used in the schedulability analysis described in Sec. VI ④. If the analysis deems that the initial configuration is schedulable, then it is returned as the design result, and DNN splitting is not needed. Otherwise, an iterative optimization procedure is triggered, where DNN models are progressively split into smaller chunks until an optimal configuration is found that ensures system schedulability while minimizing the overheads due to split inference. In each iteration, further splitting ⑤ is applied to DNN models that hinder schedulability by causing excessive blocking on higher-priority tasks; then, the corresponding TensorRT engines for the candidate configurations are profiled on the target platform, and schedulability analysis is applied again to determine whether the configuration meets the timing constraints. Among multiple valid configurations, the algorithm favors the one with the lowest total execution time, based on the profiled WCETs.

In the following, we provide a detailed specification of the proposed optimization algorithm and of the related DNN splitting scheme, followed by a heuristic algorithm which mitigates the computational complexity of the optimal approach. Additionally, we provide implementation details related to the optimization module of our framework, which supports both PyTorch and Caffe models.

DNN splitting. Splitting a DNN involves identifying a set of candidate *split points*, each representing locations within the model where it can be divided into sequential chunks (i.e., subsets of layers), such that the output of the network can be reconstructed with sequential inference of the intermediate chunks. A *splitting configuration* is obtained by activating a subset of the candidate split points. In the framework, split points of each DNN are automatically identified during the preparation of the optimization phase by exploring the graph structure of the DNN model, where nodes represent layer execution and edges define the execution dependencies. This graph is extracted in PyTorch using the FX analysis and transformation toolkit, which is natively integrated as the `torch.fx` module, or by parsing the `.prototxt` model definition file in the case of Caffe, which explicitly encodes the neural architecture. Given the graph representation of a DNN, candidate split points are identified by locating the set of nodes that must execute sequentially according to the graph topology, that is, nodes with which no other node can run in parallel. A candidate split point can then be placed immediately before or after each such node. Note that, although multiple tasks may share the same DNN model, splitting is applied independently for each task. Furthermore, a more fine-grained splitting scheme could be achieved by analyzing the full computational graph of the DNN, considering split points not only along its main sequential structure but also across parallel branches, accounting for nodes that can execute in parallel with other nodes. However, this approach requires managing the DNN graph topology and the implied data dependencies at runtime, rather than simply executing a linear sequence of chunks. To implement and automate DNN splitting, given a trained model and a splitting configuration, the general approach is to define smaller DNN modules corresponding to each chunk of the original model, and then map the appropriate weights to the layers within each chunk. The specific strategy depends on the modeling framework used to define the DNNs[3]. Each resulting chunk can then be treated as an independent DNN and individually optimized into a TensorRT engine and stored as a serialized engine file.

Notation. Consider a task τ_i and the corresponding DNN model utilized by that task. A splitting configuration $S_{i,p}$ for the task τ_i is represented as $S_{i,p} = [s^i_{p,1}, s^i_{p,2}, \dots, s^i_{p,n^i_s}]$, where n^i_s is the number of split points identified by the DNN splitting algorithm for the DNN utilized by τ_i, and each $s^i_{p,q}$ represents the state of the q-th split point in the DNN in the configuration $S_{i,p}$. The state of each split point $s^i_{p,q}$ can be either active (if $s^i_{p,q} = 1$) or inactive (if $s^i_{p,q} = 0$). Given a splitting configuration $S_{i,p}$, we consider the corresponding chunked DNN model $L_{i,p} = [l^i_{p,1}, l^i_{p,2}, \dots, l^i_{p,\tilde{n}^i_s+1}]$, where $\tilde{n}^i_s$ is the number of active split points in $S_{i,p}$. $L_{i,p}$ is constructed from $S_{i,p}$ as follows. The first chunk includes the initial layers up to the first active split point in $S_{i,p}$. Then, an additional chunk is added for every active split point in $S_{i,p}$ after the first. The last chunk includes the remaining layers starting from the last active split point in $S_{i,p}$. In the algorithms below, we consider a function $\textsc{Chunks}(S_{i,p})$ to extract the chunked model $L_{i,p}$ for $S_{i,p}$. Then, the $\textsc{Profile}(l^i_{p,q})$ function extracts the WCETs obtained by profiling a chunk $l^i_{p,q}$.

Profiling. The framework profiles the WCETs of the TensorRT engines of each chunk using an instrumented version of the runtime inference system to measure wall-clock latency from the perspective of the CPU via high-resolution monotonic C++ timers, thus complying with the model assumptions in

[3] With PyTorch models, each chunk is defined as a separate DNN module (`nn.Module`) based on the graph extracted using the FX analyzer, then weights are loaded from the trained non-chunked model (typically stored in a `.pth` file) based on the layer-to-weight mapping integrated in the `.pth` file as a state dictionary (`state_dict`). With Caffe models, the original DNN definition, stored in a `.prototxt` file, is split into multiple chunk definitions, each referencing and loading relevant weights from the `.caffemodel` file, which contains the weights of the original DNN.

Algorithm 1 Optimal approach.

```
1: $\tau_1 \leftarrow$ task obtained without DNN splitting
2: $\beta_1 \leftarrow$ BlockingTolerance$(\tau_1)$
3: for all $i \in \{2, \dots, n\}$ do                          ▷ Decreasing priority
4:     $B \leftarrow \min_{j \in hp(\tau_i)} \{\beta_j\}$      ▷ Minimal blocking tolerance
5:     for all $l^i_{p,q} \in$ Chunks$([1, \dots, 1])$ do        ▷ Full split
6:         if $B <$ Profile$\left(l^i_{p,q}\right)$ then
7:             return not schedulable
8:     $\mathcal{S}_i \leftarrow \{[0, \dots, 0]\}$      ▷ Set with non-split config. only
9:     $\mathcal{S}^\star_i \leftarrow \{\}$                  ▷ List of valid solutions
10:    while $\mathcal{S}_i$ is not empty do
11:        $S_{i,p} \leftarrow$ extract the first element in $\mathcal{S}_i$
12:        $L_{i,p} \leftarrow$ Chunks$(S_{i,p})$
13:        $\mathcal{C}_{i,p} \leftarrow \left\{\text{Profile}\left(l^i_{p,q}\right) \mid l^i_{p,q} \in L_{i,p}\right\}$
14:        if $B < \max_{c^i_{p,q} \in \mathcal{C}_{i,p}} \left\{c^i_{p,q}\right\}$ then       ▷ Split more
15:            $\mathcal{S}_i \leftarrow \mathcal{S}_i \cup$ AddSplitPoints$(S_{i,p})$
16:        else                                              ▷ Valid solution
17:            $P_{i,p} \leftarrow \sum_{c^i_{p,q} \in \mathcal{C}_{i,p}} c^i_{p,q}$          ▷ WCET of $L_{i,p}$
18:            Append $S_{i,p}$ to $\mathcal{S}^\star_i$
19:    $r \leftarrow \arg\min_{S_{i,p} \in \mathcal{S}^\star_i} \{P_{i,p}\}$   ▷ Select the best solution
20:    $\tau_i \leftarrow$ Task$(L_{i,r}, \mathcal{C}_{i,r})$          ▷ Task with $S_{i,r}$ splitting
21:    $\beta_i \leftarrow$ BlockingTolerance$(\tau_i)$
22: return $[\tau_1, \tau_2, \dots, \tau_n]$
```

Sec. VI. As the same chunk may appear in multiple splitting configurations explored in the optimization, the implemented framework employs a profiling cache to avoid redundant profiling. This significantly reduces the runtime of the optimization by ensuring that each unique chunk is profiled at most once.

Optimal approach. Algorithm 1 describes the optimal DNN splitting approach employed in the proposed framework. The algorithm takes as input a task set τ that has already been verified as unschedulable during the preliminary stage of the optimization. It then identifies a schedulable splitting configuration for the DNNs that minimizes the splitting overhead added to the inference times of each DNN by considering the overall WCETs of the chunked DNNs. The splitting configuration determined through the optimization approach specifies the chunks into which the DNN model DNN_i of each task τ_i has to be split.

To avoid exhaustively exploring all the possible splitting configurations for the tasks in τ, the algorithm leverages the assumption that, by activating additional split points in a DNN model, the sum of the WCETs of the chunks composing the DNN does not decrease. Thus, the algorithm operates starting from zero active split points and only activates additional split points if the current splitting configuration hinders schedulability. This assumption was tested with quantitative assessments involving TensorRT engines based on different DNN models, implemented in both split and non-split form, and is reasonably expected to hold in the majority of practical use cases (see Sec. VIII for supporting results). However, even if the assumption is found to not hold for a DNN model which is part of the target application τ, we highlight that the algorithm still produces a schedulable splitting configuration, provided that one exists, although that solution might be suboptimal in terms of splitting overheads. Note that optimality in this context is defined with respect to the set of candidate split points. If this set includes all possible split points, as defined by the sequential chunk inference model, and the assumption holds, then the algorithm finds the absolute optimum.

In the procedure, the highest-priority task τ_1 is never split, since it does not generate any blocking time (Line 1). The optimization process then explores each task τ_i in τ in decreasing priority order (Line 3) to determine an optimal splitting configuration for τ_i that guarantees that the blocking tolerance of all tasks with higher priority than τ_i is satisfied. For a given task τ_i, the algorithm first computes the minimal blocking tolerance B that is admitted by the higher-priority tasks in τ, denoted by $hp(\tau_i)$ (Line 4). Then, the algorithm checks whether activating all the split points of DNN_i would result in a configuration that satisfies the minimal blocking tolerance (Lines 5-7). In fact, under the assumption that every added split point reduces the maximum WCET among the chunks of a DNN, the fully split configuration generates the shortest chunks in terms of WCETs. As a result, if the fully split configuration does not guarantee schedulability, then the algorithm returns a failure, since it is not possible for the algorithm to determine a schedulable configuration (Line 7). Conversely, if the fully split condition guarantees schedulability, then the optimization proceeds by exploring each configuration $S_{i,p}$ in the set $\mathcal{S}_i$ of candidate splitting configurations (Line 10), which initially only contains the non-split configuration of DNN_i (Line 8), with the objective of deriving a set $\mathcal{S}^\star_i$ of configurations that satisfy the blocking tolerance (Line 9). In each iteration, a configuration $S_{i,p}$ is removed from the set $\mathcal{S}_i$ (Line 11), profiled (Line 13), and evaluated (Line 14). The evaluation involves verifying whether the largest WCET among those correspoding to the chunks in $L_{i,p}$ exceeds the minimal blocking tolerance B allowed by the higher-priority tasks (Line 4). If the minimal blocking tolerance is exceeded by that chunk, then a set of new candidate solutions is created by activating an additional split point in $S_{i,p}$ among those that are not already active, and these candidates are added to $\mathcal{S}_i$ (Line 15). In Line 15, the AddSplitPoints function generates all the possible configurations obtained by activating one more split point in addition to those that are already active in $S_{i,p}$. Conversely, if the minimal blocking tolerance is not exceeded, then $S_{i,p}$ represents a valid splitting configuration for τ_i and is thus appended to the set $S^\star_i$ (Line 18).

Once all the candidate solutions have been explored (i.e., S_i is empty), the configuration that generates the minimum overall WCET in the inference is selected from $S^\star_i$ (Lines 19-20). Then, the blocking tolerance β_i of τ_i is calculated with the method in Sec. VI before proceeding with the optimization of the lower-priority tasks (Line 21). Finally, when all tasks in τ have been optimized, the algorithm returns the optimal configurations selected for each task (Line 22). Note that the blocking tolerance β_i for a task τ_i only depends on the scheduling parameters of τ_i and of higher-priority tasks; thus,

Algorithm 2 Greedy splitting heuristic.

```
1: $\tau_1 \leftarrow$ task obtained without DNN splitting
2: $\beta_1 \leftarrow$ BLOCKINGTOLERANCE$(\tau_1)$
3: for all $i \in \{2, \ldots, n\}$ do                         ▷ Decreasing priority
4:     $B \leftarrow \min_{j \in hp(\tau_i)} \{\beta_j\}$     ▷ Minimal blocking tolerance
5:     for all $l^i_{p,q} \in$ CHUNKS$([1, \ldots, 1])$ do     ▷ Full split
6:         if $B <$ PROFILE $\left(l^i_{p,q}\right)$ then
7:             return not schedulable
8:     $S_i \leftarrow [0, \ldots, 0]$
9:     $L_i \leftarrow$ CHUNKS$(S_i)$                          ▷ Start with no splits
10:    $\mathcal{C}_i \leftarrow \left\{\text{PROFILE}\left(l^i_q\right) \mid l^i_q \in L_i\right\}$
11:    while $B < \max_{c^i_q \in \mathcal{C}_i} \left\{c^i_q\right\}$ do     ▷ Split more
12:        $\mathcal{S}_i \leftarrow$ ADDSPLITPOINTS $(S_i)$
13:        for all $S_{i,p} \in \mathcal{S}_i$ do
14:            $L_{i,p} \leftarrow$ CHUNKS$(S_{i,p})$
15:            $\mathcal{C}_{i,p} \leftarrow \left\{\text{PROFILE}\left(l^i_{p,q}\right) \mid l^i_{p,q} \in L_{i,p}\right\}$
16:        $r \leftarrow \arg\min_{S_{i,p} \in \mathcal{S}_i} \left\{\max_{c^i_{p,q} \in \mathcal{C}_{i,p}} \left\{c^i_{p,q}\right\}\right\}$
17:        $(S_i, L_i, \mathcal{C}_i) \leftarrow (S_{i,r}, L_{i,r}, \mathcal{C}_{i,r})$     ▷ Best solution
18:    $\tau_i \leftarrow$ TASK$(L_i, \mathcal{C}_i)$          ▷ Task with $S_i$ splitting
19:    $\beta_i \leftarrow$ BLOCKINGTOLERANCE$(\tau_i)$
20: return $[\tau_1, \tau_2, \ldots, \tau_n]$
```

it is not necessary to recompute the blocking tolerances of all tasks at every iteration of the loop in Line 3.

In the worst case, the algorithm needs to explore all the possible splitting configurations for each task τ_i (except τ_1), which are given by selecting all possible combinations of the values of each state $s^i_{p,q}$ in $S_{i,p}$. This amounts to a total of $2^{n^i_s}$ configurations for each task τ_i. One way to limit the complexity of the optimal approach and avoid combinatorial explosion is to reduce the maximum number of splitting configurations explored by the algorithm by selecting a subset of split points with a heuristic policy and removing them from the set of valid split points on which the algorithm can act. This heuristic approach can significantly reduce the number of configurations to test, but may also prevent the discovery of schedulable or optimal configurations in some cases.

Greedy heuristic. We now describe a greedy heuristic approach for the splitting which avoids the potential combinatorial explosion of the optimal approach. The main difference is that, for each task τ_i, instead of considering a set of candidate solutions, only one candidate configuration is selected for τ_i and refined in each iteration of the internal loop of the algorithm, while the other candidates are discarded. The first part of the modified algorithm (Algorithm 2) is unchanged (Lines 1-7). For each task τ_i, the algorithm first evaluates the non-split configuration (Line 9), which is profiled (Line 10) to verify whether it satisfies the minimal blocking tolerance B (Line 11). If the blocking tolerance is not satisfied, then the algorithm iterates by activating additional split points in the DNN until it is satisfied (Lines 11-17). At each iteration, a set $\mathcal{S}_i$ of candidate configurations is created from the current configuration S_i, by leveraging the ADDSPLITPOINTS function. Each element of this candidate set is profiled (Lines 13-15); then, the best configuration among those in $\mathcal{S}_i$ is selected as the next value for S_i, whereas the other candidates are discarded. The best configuration is selected as the one that guarantees the minimum amount of blocking time for higher-priority tasks, i.e., the one whose maximum chunk WCET is the smallest (Lines 16-17). Once a solution that satisfies the minimal blocking tolerance B is found (Line 11), the configuration for τ_i is fixed to the current candidate S_i, then its blocking tolerance is computed (Lines 18-19). The algorithm returns the splitting configurations selected for each task (Line 20).

With this approach, the worst-case number of tested configurations for each task τ_i is bounded by $\left(n^i_s\right)^2$, at the cost of selecting a potentially suboptimal configuration.

TABLE II: Number of split points (SPs) identified by the framework and considered in the experiments.

	AlexNet	InceptionV4	ResNet18	VGG19
Total SPs	10	23	11	21
Considered SPs	10	6	11	5

VIII. Experimental results

This section presents experimental results comparing the schedulability of the proposed framework with the baseline approach on different Jetson platforms, also reporting runtimes of the design phases, memory usage, and latency overheads.

A. Experimental setup

The evaluation was performed on the three reference Jetson platforms: TX2, AGX Xavier, and AGX orin, all configured to operate in maximum performance mode. The offline design toolset was implemented in mixed Python and C++, while the runtime system was implemented in C++. We considered a set of pre-trained Caffe DNN models by Nvidia [43]: VGG19 [44], AlexNet [45], ResNet18 [46], and InceptionV4 [47]. For the schedulability evaluation, we considered a set of real-time tasks, each performing inference using one of these models. The framework identified a large number of split points for these models. To limit the runtime of the optimal approach and enable a large-scale evaluation, we statically disabled a subset of split points for InceptionV4 and VGG19, retaining only those located at the boundaries between major architectural blocks (see Table II).

Table III presents the input size (in KB) for each DNN and the output sizes (in KB) of the layers corresponding to the split points considered in the experiments. These sizes were used to configure the buffers used at runtime for inference. Overall, the maximum buffer size for each considered DNN remains within the order of tens of kilobytes, indicating no significant increase in buffer size requirements with the proposed method. Table IV reports the profiled WCETs of each chunk of the DNN models in the fully split configuration, their sum, and the profiled execution time of the non-split DNNs. These results show that the sum of the WCETs of the fully split configuration is greater than the WCET of the non-split configuration for every DNN model, supporting the assumption in Sec. VII.

TABLE III: Input size (KB) of each DNN and output size (KB) of each chunk in the fully split configuration (l_k). The maximum size for each DNN is in bold. Sizes are computed assuming floating point values for the inputs/outputs of the layers.

DNN model	**Input**	l_0	l_1	l_2	l_3	l_4	l_5	l_6	l_7	l_8	l_9	l_{10}	l_{11}
Alexnet	603.855	**1134.375**	**1134.375**	273.375	729.0	729.0	169.0	253.5	253.5	169.0	36.0	3.906	-
InceptionV4	1047.668	2775.125	**3330.625**	1837.5	1156.0	1156.0	384.0	3.906	-	-	-	-	-
ResNet18	588.0	**3136.0**	784.0	784.0	784.0	392.0	392.0	196.0	196.0	98.0	98.0	2.0	3.906
VGG19	588.0	**12544.0**	6272.0	3136.0	1568.0	392.0	3.906	-	-	-	-	-	-

TABLE IV: Profiled execution times on the Nvidia Jetson AGX Orin of each chunk in the fully split configuration (l_k), their sum (**Sum**), and profiled execution times of each non-split DNN model (**Non-split**). Times are in milliseconds.

DNN model	l_0	l_1	l_2	l_3	l_4	l_5	l_6	l_7	l_8	l_9	l_{10}	l_{11}	**Sum**	**Non-split**
Alexnet	0.182	0.135	0.085	0.216	0.288	0.077	0.137	0.192	0.161	0.037	3.292	-	4.802	4.469
InceptionV4	0.163	0.904	1.205	2.088	2.193	1.777	0.799	-	-	-	-	-	9.129	8.67
ResNet18	0.151	0.047	0.318	0.195	0.131	2.08	0.13	0.125	0.153	0.27	0.049	0.101	3.751	2.533
VGG19	0.168	1.203	0.714	1.122	0.976	7.243	-	-	-	-	-	-	11.425	6.615

To evaluate real-time performance, we analyze schedulability under various system configurations using both the baseline approach and the proposed framework, considering optimal and heuristic splitting strategies. Applications are generated using a custom workload generation algorithm that creates applications with a fixed number of tasks, each periodically performing inference of a single DNN model, uniformly selected from the models listed above and fixed for the duration of the experiment. With this approach, the experiments validate a variety of task sets with different periods and different overall GPU workload. The WCET C_i of the inference request of a task τ_i is set to the profiled execution time of the corresponding DNN model in its non-split form. We consider an utilization metric capturing the amount of requested scheduling time on the framework by each task τ_i, computed as $U_i = C_i/T_i$. The total utilization is computed as $U = \sum_{\tau_i \in \tau} U_i$. In the experiments, the total utilization U is varied from 0.6 to 0.9 in increments of 0.1. For each utilization point, we generate 50 task sets, where the utilization U_i of each task is generated with the UUniFast algorithm [54], such that $U = \sum_{\tau_i \in \tau} U_i$, with $D_i = T_i$ and DM priority assignment.

We evaluate the following approaches. **Optimal**: proposed framework with optimal splitting. **Heuristic**: proposed framework with heuristic splitting. **Baseline**: baseline approach without stream priority assignment, i.e., where the CUDA streams of all tasks share the same stream priority. **Baseline SP**: baseline approach with stream priority assignment. In **Baseline SP**, each task was assigned a separate stream priority in descending order of the task priority, until the maximum number of available stream priorities was reached. The remaining lower-priority tasks were assigned the same stream priority. For the proposed methods (**Optimal** and **Heuristic**), we report the schedulability metric for each utilization point, defined as the percentage of task sets deemed schedulable by the response-time analysis following the optimization in Sec. VII. Split configurations were profiled on the Jetson platforms whenever required by Algorithms 1 and 2 using CPU wall-clock profiling, using caching to avoid redundant profiling within each optimization (see Sec. VII). In order to ensure practical schedulability of the optimized task sets on the Jetson platforms, the theoretical schedulability results were further validated by running each optimized schedulable application on the implemented runtime system for 10 hyperperiods and ensuring that no deadline misses occurred. Since a schedulability analysis is not available for the baseline methods (**Baseline** and **Baseline SP**), we considered an application schedulable if no deadline misses were observed when running the application for 10 hyperperiods under the corresponding strategy on the Jetson platforms, and non-schedulable otherwise.

B. Experimental results

Figure 4 reports the schedulability obtained when varying the GPU workload U from 0.6 to 0.9 and when considering different combinations of number of tasks per task set and computing platforms (as reported above each plot). Figures 4(a) and 4(b) report the schedulability obtained with applications composed of 3 and 4 DNN tasks, respectively, on the Jetson TX2. As expected, schedulability decreases with higher utilization for all methods. Both the optimal and heuristic methods outperform the baseline scheduler by a significant margin, even when multiple stream priorities are considered. Furthermore, the heuristic achieves a schedulability performance that is very close or equal to that of the optimal approach, while avoiding the worst-case combinatorial explosion that can occur with the optimal strategy. Figures 4(c) and 4(d) report the schedulability for 6 and 8 tasks, respectively, on the Jetson AGX Xavier. The overall trend is similar to the previous results, with an even larger performance gap between the baseline and the proposed approach, arguably due to the increased complexity of the task sets to be scheduled. Finally, Figures 4(e) and 4(f) report the schedulability for 8 and 12 DNN tasks, respectively, on the Jetson AGX Orin. Thanks to the increased computational power of the computing platform, all methods show improved schedulability. Notably, the proposed methods achieve up to 96% schedulability even with 12 tasks and $U = 0.9$, whereas, under the same configuration, the baseline techniques reach only 10% schedulability.

Figure 5 reports the average design phase runtimes for both the optimal and heuristic strategies, considering 8 tasks per task set on the Jetson AGX Orin (same configuration as in Figure 4(e)). These results show that average runtimes re-

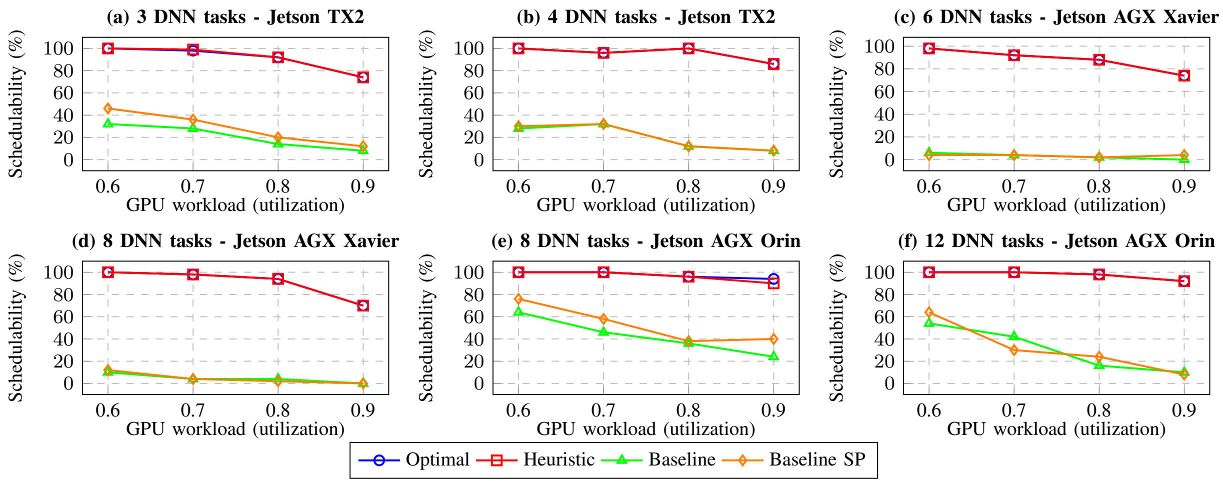

Fig. 4: Schedulability (%) with different combinations of platform and task set sizes under varying GPU workload.

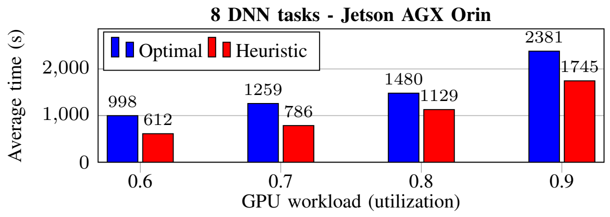

Fig. 5: Comparison of average design phase runtimes for the optimal and the heuristic methods on the AGX Orin (8 tasks).

TABLE V: Average memory usage of the runtime scheduling system (in GB and percentage) across different configurations.

Experiment	Baseline		Baseline SP		Optimal		Heuristic	
	GB	%	GB	%	GB	%	GB	%
TX2, 3 tasks	2.22	28.9	2.23	29.1	2.32	30.3	2.23	29.1
TX2, 4 tasks	3.08	40.2	3.02	39.5	2.93	38.2	3.06	39.9
Xavier, 6 tasks	2.78	9.2	2.75	9.1	2.81	9.2	2.79	9.2
Xavier, 8 tasks	4.33	14.3	4.30	14.2	4.36	14.4	4.33	14.3
Orin, 8 tasks	3.73	12.5	3.93	13.2	4.02	13.5	3.91	13.1
Orin, 12 tasks	4.92	16.5	4.93	16.5	5.00	16.7	4.95	16.6

main within reasonable bounds. Notably, the heuristic strategy reduces average runtimes by up to 38% compared to the optimal approach. When profiling times are excluded (i.e., using pre-optimized TensorRT engines and pre-profiled chunk runtimes), average optimization times drop to 12 s and 4 s for the optimal and heuristic strategies, respectively, across all utilization levels.

Table V reports the average memory usage of each scheduling system on the Jetson platforms, computed over the task sets used in each experiment in Figure 4. Given that Jetson platforms feature a unified CPU-GPU memory, reported values reflect the total shared memory usage. Results show minimal variation from the baseline (-1% to $+2\%$), indicating negligible memory overhead. Memory usage remains moderate across all configurations and stable during operation, as all components are initialized and preloaded at system startup. Startup time, reflecting the one-time system initialization and model preloading, is consistent across all strategies, requiring at most 11.1 s on the TX2 with the proposed method. The average CPU overhead to dispatch a chunk inference request to the GPU with the proposed framework was measured as 13 µs, 9 µs, and 8 µs on the Jetson TX2, AGX Xavier, and AGX Orin, respectively. This overhead was calculated as the difference between CPU-observed wall-clock latency (measured via high-resolution C++ timers) and the time taken by the GPU to complete the inference (measured via CUDA event-based timing). These overheads remain low, possibly due to the fact that at most one chunk is dispatched to the GPU at a time, which reduces contention and ensures stable synchronization costs. These overheads are fully accounted for in the WCETs obtained in the optimization phase, which also capture all relevant delays, including CPU-GPU synchronization and shared resource management.

IX. Conclusions and Future Work

This paper presented a framework for multitasking DNN workloads on GPU-based SoCs, based on TensorRT. The framework leverages DNN splitting to attain finer control of GPU scheduling, whereas the runtime system works around the limitations of concurrent TensorRT inference by requesting the inference of at most one chunk at a time to enforce a priority-based policy and enable limited-preemptive response-time analysis. Experimental results on Jetson platforms showed large gains in schedulability over the baseline approach based on concurrent inference on multiple CUDA streams. Future work includes extending the timing analysis to account for preprocessing and postprocessing overheads on the CPU by leveraging self-suspending task models [55]–[57] and investigating the application of the proposed strategy to other DNN inference frameworks and in FPGA-based SoCs.

Acknowledgment

This work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union-NextGenerationEU and by the Department of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Piazza Martiri della Libertà 33, Pisa 56127, Italy.

References

[1] S.-C. Lin, Y. Zhang, C.-H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars, "The architectural implications of autonomous driving: Constraints and acceleration," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2018)*. ACM, 2018, pp. 751–766.

[2] C. Wang, X. Wang, H. Hu, Y. Liang, and G. Shen, "On the application of cameras used in autonomous vehicles," *Archives of Computational Methods in Engineering*, vol. 29, no. 6, pp. 4319–4339, 2022.

[3] Y. Xiang and H. Kim, "Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference," in *Proceedings of the 40th IEEE Real-Time Systems Symposium (RTSS 2019)*. IEEE, 2019, pp. 392–405.

[4] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, "Autoware on board: Enabling autonomous vehicles with embedded systems," in *Proceedings of the ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS 2018)*. IEEE, 2018, pp. 287–296.

[5] Autoware Foundation, "Autoware documentation," https://www.autoware.org/, 2023, accessed: 2025-05-01.

[6] NVIDIA Corporation, "TensorRT," 2025, accessed: 2025-05-01. [Online]. Available: https://developer.nvidia.com/tensorrt

[7] ——, "NVIDIA Jetson AGX Orin," 2025, accessed: 2025-05-01. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/

[8] ——, "NVIDIA Jetson TX2," 2025, accessed: 2025-05-01. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2/

[9] ——, "NVIDIA Jetson AGX Xavier," 2025, accessed: 2025-05-01. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-series/

[10] PyTorch, "PyTorch," 2025, accessed: 2025-05-01. [Online]. Available: https://pytorch.org/

[11] TensorFlow, "TensorFlow," 2025, accessed: 2025-05-01. [Online]. Available: https://www.tensorflow.org/

[12] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM International Conference on Multimedia (MM 2014)*. ACM, 2014, pp. 675–678.

[13] F. Wurst, D. Dasari, A. Hamann, D. Ziegenbein, I. Sanudo, N. Capodieci, M. Bertogna, and P. Burgio, "System performance modelling of heterogeneous HW platforms: An automated driving case study," in *Proceedings of the 22nd Euromicro Conference on Digital System Design (DSD 2019)*. IEEE, 2019, pp. 365–372.

[14] I. S. Olmedo, N. Capodieci, J. L. Martinez, A. Marongiu, and M. Bertogna, "Dissecting the CUDA scheduling hierarchy: a performance and predictability perspective," in *Proceedings of the 26th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2020)*. IEEE, 2020, pp. 213–225.

[15] J. Bakita and J. H. Anderson, "Demystifying nvidia GPU internals to enable reliable GPU management," in *Proceedings of the 30th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2024)*. IEEE, 2024, pp. 294–305.

[16] N. Capodieci, R. Cavicchioli, M. Bertogna, and A. Paramakuru, "Deadline-based scheduling for GPU with preemption support," in *Proceedings of the 39th IEEE Real-Time Systems Symposium (RTSS 2018)*. IEEE, 2018, pp. 119–130.

[17] S. Bateni, Z. Wang, Y. Zhu, Y. Hu, and C. Liu, "Co-optimizing performance and memory footprint via integrated CPU/GPU memory management, an implementation on autonomous driving platform," in *Proceedings of the 26th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2020)*. IEEE, 2020, pp. 310–323.

[18] S. Liu, S. Yao, X. Fu, R. Tabish, S. Yu, A. Bansal, H. Yun, L. Sha, and T. Abdelzaher, "On removing algorithmic priority inversion from mission-critical machine inference pipelines," in *Proceedings of the 41st IEEE Real-Time Systems Symposium (RTSS 2020)*. IEEE, 2020, pp. 319–332.

[19] H. Zhou, S. Bateni, and C. Liu, "Sˆ 3dnn: Supervised streaming and scheduling for GPU-accelerated real-time DNN workloads," in *Proceedings of the 24th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2018)*. IEEE, 2018, pp. 190–201.

[20] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace, "Serving DNNs like clockwork: Performance predictability from the bottom up," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020)*. USENIX, 2020, pp. 443–462.

[21] F. Yu and S. Bray, "Automated runtime-aware scheduling for multi-tenant DNN inference on GPU," in *Proceedings of the 40th IEEE/ACM International Conference On Computer Aided Design (ICCAD 2021)*. IEEE, 2021, pp. 1–9.

[22] N. Ling, K. Wang, Y. He, G. Xing, and D. Xie, "Rt-mdl: Supporting real-time mixed deep learning tasks on edge platforms," in *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems (SenSys 2021)*. ACM, 2021, pp. 1–14.

[23] S. Lee and S. Nirjon, "Subflow: A dynamic induced-subgraph strategy toward real-time DNN inference and training," in *Proceedings of the 26th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2020)*. IEEE, 2020, pp. 15–29.

[24] M. Han, H. Zhang, R. Chen, and H. Chen, "Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences," in *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022)*. USENIX, 2022, pp. 539–558.

[25] S. Jain, I. Baek, S. Wang, and R. Rajkumar, "Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs," in *Proceedings of the 25th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2019)*. IEEE, 2019, pp. 29–41.

[26] W. Kang, J. Lee, Y. Lee, S. Oh, K. Lee, and H. S. Chwa, "Rt-swap: Addressing GPU memory bottlenecks for real-time multi-DNN inference," in *Proceedings of the 30th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2024)*. IEEE, 2024, pp. 373–385.

[27] W. Kang, K. Lee, J. Lee, I. Shin, and H. S. Chwa, "Lalarand: Flexible layer-by-layer CPU/GPU scheduling for real-time DNN tasks," in *Proceedings of the 42nd IEEE Real-Time Systems Symposium (RTSS 2021)*. IEEE, 2021, pp. 329–341.

[28] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 615–629, 2017.

[29] J. Kim, Y. Park, G. Kim, and S. J. Hwang, "Splitnet: Learning to semantically split deep networks for parameter reduction and model parallelization," in *Proceedings of the 34th International Conference on Machine Learning (ICML 2017)*. PMLR, 2017, pp. 1866–1874.

[30] F. M. C. de Oliveira and E. Borin, "Partitioning convolutional neural networks for inference on constrained internet-of-things devices," in *30th IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2018)*. IEEE, 2018, pp. 266–273.

[31] U. U. Hafeez, X. Sun, A. Gandhi, and Z. Liu, "Towards optimal placement and scheduling of DNN operations with pesto," in *Proceedings of the 22nd ACM International Middleware Conference (Middleware 2021)*. ACM, 2021, pp. 39–51.

[32] Y. Duan and J. Wu, "Joint optimization of DNN partition and scheduling for mobile cloud computing," in *Proceedings of the 50th ACM International Conference on Parallel Processing (ICPP 2021)*. ACM, 2021, pp. 1–10.

[33] S. Tuli, G. Casale, and N. R. Jennings, "Splitplace: Ai augmented splitting and placement of large-scale neural networks in mobile edge environments," *IEEE Transactions on Mobile Computing*, vol. 22, no. 9, pp. 5539–5554, 2022.

[34] A. Parthasarathy and B. Krishnamachari, "Partitioning and placement of deep neural networks on distributed edge devices to maximize inference throughput," in *Proceedings of the IEEE 32nd International Telecommunication Networks and Applications Conference (ITNAC 2022)*. IEEE, 2022, pp. 239–246.

[35] Z. Zhang, Q. Li, L. Lu, D. Guo, and Y. Zhang, "Joint optimization of the partition and scheduling of DNN tasks in computing and network

convergence," *IEEE Networking Letters*, vol. 5, no. 2, pp. 130–134, 2023.

[36] I. Sanchez Leal, E. Saqib, I. Shallari, A. Jantsch, S. Krug, and M. O'Nils, "Waist tightening of CNNs: A case study on tiny yolov3 for distributed iot implementations," in *Proceedings of the 5th ACM Cyber-Physical Systems and Internet of Things Week (CPS-IoT Week 2023)*. ACM, 2023, pp. 241–246.

[37] J. Chen, D. Van Le, R. Tan, and D. Ho, "Nnfacet: Splitting neural network for concurrent smart sensors," *IEEE Transactions on Mobile Computing*, vol. 23, no. 2, pp. 1627–1640, 2023.

[38] Y. Duan and J. Wu, "Optimizing job offloading schedule for collaborative DNN inference," *IEEE Transactions on Mobile Computing*, vol. 23, no. 4, pp. 3436–3451, 2023.

[39] S. K. Ghosh, A. Raha, V. Raghunathan, and A. Raghunathan, "Partnner: Platform-agnostic adaptive edge-cloud DNN partitioning for minimizing end-to-end latency," *ACM Transactions on Embedded Computing Systems*, vol. 23, no. 1, pp. 1–38, 2024.

[40] F. Kreß, E. M. El Annabi, T. Hotfilter, J. Hoefer, T. Harbaum, and J. Becker, "Automated deep neural network inference partitioning for distributed embedded systems," in *Proceedings of the 2024 IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2024)*. IEEE, 2024, pp. 39–44.

[41] B. Huang, X. Huang, X. Liu, C. Ding, Y. Yin, and S. Deng, "Adaptive partitioning and efficient scheduling for distributed DNN training in heterogeneous iot environment," *Computer Communications*, vol. 215, pp. 169–179, 2024.

[42] Z. Taufique, A. Vyas, A. Miele, P. Liljeberg, and A. Kanduri, "Hidp: Hierarchical DNN partitioning for distributed inference on heterogeneous edge platforms," in *Proceedings of the 28th IEEE Design, Automation & Test in Europe Conference (DATE 2025)*. IEEE, 2025, pp. 1–7.

[43] NVIDIA Corporation, "Jetson inference," 2025, accessed: 2025-05-01. [Online]. Available: https://github.com/dusty-nv/jetson-inference/releases

[44] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[45] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[46] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the 29th IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2016)*. IEEE, 2016, pp. 770–778.

[47] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Proceedings of the 31st AAAI Conference on Artificial Intelligence*. AAAI, 2017, pp. 4278–4284.

[48] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the 28th IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2015)*. IEEE, 2015, pp. 1–9.

[49] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the 31st IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2018)*. IEEE, 2018, pp. 4510–4520.

[50] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the 30th IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2017)*. IEEE, 2017, pp. 4700–4708.

[51] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, "Controller area network (CAN) schedulability analysis: Refuted, revisited and revised," *Real-Time Systems*, vol. 35, pp. 239–272, 2007.

[52] G. C. Buttazzo, M. Bertogna, and G. Yao, "Limited preemptive scheduling for real-time systems. a survey," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 3–15, 2012.

[53] A. Burns, "Preemptive priority-based scheduling: an appropriate engineering approach," in *Advances in real-time systems*. ACM, 1995, pp. 225–248.

[54] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-time systems*, vol. 30, no. 1, pp. 129–154, 2005.

[55] J.-J. Chen, G. Nelissen, and W.-H. Huang, "A unifying response time analysis framework for dynamic self-suspending tasks," in *Proceedings of the 28th Euromicro Conference on Real-Time Systems (ECRTS 2016)*. IEEE, 2016, pp. 61–71.

[56] F. Aromolo, A. Biondi, G. Nelissen, and G. Buttazzo, "Event-driven delay-induced tasks: Model, analysis, and applications," in *Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2021)*. IEEE, 2021, pp. 53–65.

[57] F. Aromolo, A. Biondi, and G. Nelissen, "Response-time analysis for self-suspending tasks under EDF scheduling," in *Proceedings of the 34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022, p. 13:1–13:18.