Shared Memory-contention-aware Concurrent DNN Execution for Diversely Heterogeneous SoCs

Ismet Dagli
Computer Science Department
Colorado School of Mines
Golden, CO, USA
ismetdagli@mines.edu

Mehmet E. Belviranli
Computer Science Department
Colorado School of Mines
Golden, CO, USA
belviranli@mines.edu

Abstract

Two distinguishing features of state-of-the-art mobile and autonomous systems are: 1) There are often multiple workloads, mainly deep neural network (DNN) inference, running *concurrently* and *continuously*. 2) They operate on shared memory System-on-Chips (SoC) that embed heterogeneous accelerators tailored for specific operations. State-of-the-art systems lack efficient performance and resource management techniques necessary to either maximize total system throughput or minimize end-to-end workload latency. In this work, we propose *HaX-CoNN*, a novel scheme that characterizes and maps layers in concurrently executing DNN inference workloads to a diverse set of accelerators within an SoC. Our scheme uniquely takes per-layer execution characteristics, shared memory (SM) contention, and inter-accelerator transitions into account to find *optimal* schedules. We evaluate *HaX-CoNN* on NVIDIA Orin, NVIDIA Xavier, and Qualcomm Snapdragon 865 SoCs. Our experimental results indicate that *HaX-CoNN* can minimize memory contention by up to 45% and improve total latency and throughput by up to 32% and 29%, respectively, compared to the state-of-the-art.

CCS Concepts: • **Computer systems organization** → **Heterogeneous (hybrid) systems**; • **General and reference** → *Performance*; • **Computing methodologies** → **Shared memory algorithms**; *Neural networks.*

Keywords: Concurrent DNN inference, Shared memory contention, Heterogeneous SoCs

1 Introduction

Modern mobile and autonomous systems —such as cars, drones, and robots— hinge on edge intelligence, which involves running computationally demanding workloads [34, 35, 64]. Notably, a diverse range of applications embed multiple DNNs as subtasks such as object detection and semantic segmentation for autonomous systems [15, 62] or pose estimation and eye-tracking for VR applications [12, 65]. Workloads running *concurrently* and *continuously* in such systems necessitate powerful SoCs that can meet high computational demand and ensure safety [16] and QoS [23] requirements. Thus, such SoCs are often equipped with a CPU, a GPU, and one or more domain-specific accelerators (DSAs), optimized to perform specific types of operations. For example, NVIDIA Xavier AGX and Orin architecture comprise deep learning accelerators (DLA), and a programmable vision accelerator (PVA) in addition to CPU and GPU. Utilizing different types of DSAs for concurrently running workloads enables the flexibility to explore execution strategies [41]. *Leveraging this opportunity, in this work, we focus on mapping layers of parallel DNNs to different types of DSAs so that we can improve computational latency and system throughput.*

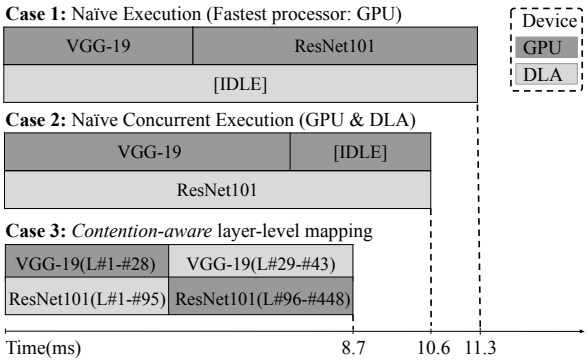

Figure 1. Different ways of executing VGG-19 and ResNet-101 DNNs in parallel on Xavier AGX.

A common feature of such heterogeneous SoCs is the *shared physical main memory* where data is stored for access by all processing units (PUs) in the system. Even though this cost-driven design decision curbs costly data movement, memory subsystems in such architectures are often designed to accommodate the memory demands of a single PU at a time. Consequently, *shared memory contention emerges as one of the primary performance bottlenecks in mobile and autonomous SoCs, when parallel tasks are concurrently mapped to different accelerators* [20, 39, 67].

PPoPP '24, March 2–6, 2024, Edinburgh, United Kingdom

ACM ISBN 979-8-4007-0435-2/24/03.
https://doi.org/10.1145/3627535.3638502

We investigate concurrent execution on shared memory SoCs through a case study. In a typical loop running on autonomous systems, VGG-19 [58] and ResNet101 [18] can be used in tandem for vision (*i.e.*, perception) tasks. Since the remaining tasks in the autonomous loop depend on the completion of these two DNNs, utilizing all computational resources in the SoC for these DNNs is expected to reduce the total latency of the system. Fig. 1 illustrates three different ways of executing two DNNs on NVIDIA Xavier AGX SoC. In **Case 1**, DNNs are *serially* executed on the fastest DSA, which is the GPU, resulting in 11.3ms of cumulative latency. However, this method leaves DLA idle, thereby under-utilizing the system resources. The first approach can be improved by a *naïve concurrent* execution, as shown in **Case 2**. In this scheme, VGG-19 is run on GPU, and ResNet101 is mapped to DLA, resulting in 10.6ms cumulative latency with a slight improvement. However, the speed-up obtained remains limited due to two reasons: (1) DLA takes longer to execute, leaving GPU idle towards the end, and (2) when GPU and DLA operate together, they contend for shared memory and slow down. *For more efficient execution of concurrent DNNs, we need a finer-grained, i.e., layer-level, mapping of the DNNs to DSAs.*

Case 3 depicts an ideal case where layers in both DNNs are divided into two groups after layers #28 and #95, respectively. For each DNN, the execution is switched between two DSAs at the boundary of corresponding layer groups (*i.e.*, transition point). While seemingly non-intuitive, this approach considerably improves the cumulative latency and increases the overall system utilization. This is due to a careful partitioning and mapping of layers to GPU and DLA in a way that: (1) the shared memory contention across concurrently running layers is minimized, (2) neither of the DSAs is left idle, and (3) the overhead of switching between accelerators between two layers is minimized. However, finding such partitioning is not trivial, and the state-of-the-art approaches (detailed in Section 2) fail to provide a holistic approach to perform this partitioning optimally.

In this study, we propose *HaX-CoNN*, a *multi-accelerator and contention-aware execution scheme* for collaboratively and concurrently running DNNs on shared memory SoCs. *HaX-CoNN* is centered around characterizing common layers in DNNs according to their DSA-specific performance and identifying how they are affected by shared memory contention. Leveraging *decoupled* performance and contention characterization *at a layer-level*, *HaX-CoNN* exploits distinct capabilities of each DSA in the system by deciding whether the execution of the next layer in the DNN should *transition* to another DSA or not. *HaX-CoNN* uniquely finds an *optimal* mapping between the layers and DSAs in the system by formulating the problem as a set of constraint-based linear equations and utilizing SAT solvers to find a solution.

Our work makes the following contributions:

- We present *HaX-CoNN*, a contention-aware, multi-accelerator execution scheme that *maximizes compute utilization* and *minimizes the overall latency* of concurrently running DNNs on shared memory SoCs.
- We propose a generalized and formal layer-to-accelerator mapping approach for concurrently running DNNs. We demonstrate that SAT solvers can be utilized to produce *optimal schedules* for *multi-accelerator* execution.
- We build a new contention modeling approach which significantly reduces profiling search space by decoupling performance measurement and the slowdown.
- We present D-*HaX-CoNN*, a dynamic runtime adaptation of SAT solver-based optimal schedule generation for dynamically changing workloads.
- We evaluate *HaX-CoNN* and D-*HaX-CoNN* on NVIDIA AGX Orin, Xavier AGX, and Qualcomm Snapdragon 865 SoCs. Our results show that *HaX-CoNN* can provide latency and throughput improvements up to 32% and 29%, respectively, over greedy-scheduling based approaches.

2 Related Work

Concurrent DNN execution: Several studies [10, 32, 33, 37, 43, 69] propose scheduling techniques for the concurrent execution of multiple DNNs. Two of them focus on multi-DNN inference on SoCs: Herald [37] introduces a mapper to optimize hardware resource utilization across accelerators such as NVDLA [1] and Shi-diannao [14] whereas H2H [69] improves Herald by considering inter-accelerator transition costs.

Multi-accelerator scheduling: Scheduling for systems with more than one type of accelerator has recently been targeted by many studies [4, 6, 24, 28, 29, 63, 66]. Among the most relevant, Gamma [29] and Kang et al. [28] build genetic algorithms to utilize multiple accelerators for a single DNN execution while Wu et al. [66] and Mensa [6] target unique hardware for edge devices. None of these studies address contention and balancing issues with multi-DNN execution. DNN training for large-scale systems [27, 40, 42, 50], on the other hand, is outside the scope of this work.

Optimal schedule generation: Only a couple of studies create optimal schedules for multi-DSA execution. AxoNN [11] maps layers of a single DNN onto heterogeneous accelerators under an energy budget, resulting in a serial

Table 1. Feature comparison between the most related work and *HaX-CoNN*.

Related Work	Mensa [6]	AxoNN [11]	Pipeline [24]	OmniBoost [32]	MoCA [33]	Herald [37]	H2H [69]	***HaX-CoNN***
Concurrent DNNs	✗	✗	✗	✓	✓	✓	✓	✓
Multi-accelerator	✓	✓	✓	✗	✗	✓	✓	✓
Transition cost	✓	✓	✓	✓	✓	✗	✓	✓
Memory contention	✗	✗	✗	✗	✓	✗	✗	✓
Dynamic scheduling	✓	✗	✓	✗	✓	✗	✗	✓
Optimal schedules	✗	✓	✗	✓	✗	✗	✗	✓

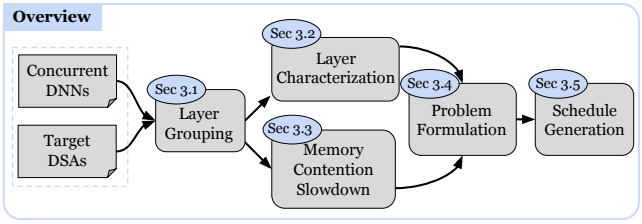

Figure 2. Overview of *HaX-CoNN*.

execution. OmniBoost [32] uses Monte Carlo tree search by exhaustively profiling layers on CPU and GPU. Both approaches only create static schedules.

Shared memory contention: None of the studies mentioned so far addresses shared memory contention. MoCA [33] designs a multitenant DSA architecture with dynamic memory resource management. FAST [68] uses an integer linear programming based operator fusion technique to remedy the memory bottlenecks whereas ParDNN [51] partitions DNNs under a memory limit. However, these approaches are not adaptable to off-the-shelf multi-DSA shared memory SoCs.

Table 1 provides a snapshot of what the most relevant works offer and how they compare against *HaX-CoNN*. Achieving the ideal execution scenario depicted in **Case 3** of Fig. 1 requires holistic consideration of several factors given in the table: (i, ii) interaction and mapping opportunities created by running *concurrent DNNs* on *different types of DSAs*, (iii) the *transition overhead* when the execution within a DNN switches across accelerators, (iv) the slowdown caused by the *shared memory contention* as layers run concurrently – our analysis shows that shared memory contention-unaware decisions can reduce system performance by up to 70%, as detailed in Section 5.2–, (v) support for dynamic schedules, and (vi) optimal schedule creation. The efficient and safe operation of performance critical mobile and autonomous workloads on shared memory SoCs depends on the *holistic* consideration of all these factors. Our experiments demonstrate that the lack of such consideration results in mispredicted performance, which in turn results in inefficient execution.

3 HaX-CoNN: Heterogeneity-aware Execution of Concurrent Deep Neural Networks

An overview of our proposed methodology is given in Fig. 2. *HaX-CoNN* takes the *DNNs* to be scheduled and the target *DSAs* as input and produces the optimal schedule as output.

3.1 Layer grouping

The first step involves identifying minimal layer groups to serve as atomic assignment units for DSAs. This grouping considers several factors:

1) Preserving layer optimizations: Layer/operator fusion [2, 7, 44] merges multiple layers into a single layer. *Transition points* during DNN execution, where we switch execution from one DSA to another, should not impede operator fusion. Therefore, we ensure that fusible layers are grouped together and mapped to the same accelerator.

2) Input/output reformatting: DSAs typically operate in an internal hardware (HW) pipeline. If a transition from a layer mapped to such a DSA disrupts its pipeline, then an additional output reformatting operation is inserted by the execution framework. Similarly, input reformatting might be required after transitioning to that DSA. Layer groupings can be structured to avoid such formatting overheads.

3) Accelerator and software limitations: DSAs are often limited by the layer types, parameters, and batch sizes they support. DNN execution frameworks, such as NVIDIA TensorRT [48] and Qualcomm SNPE [52], ensure such constraints are followed. We identify such limitations via vendor specific API calls and our framework considers these limitations when locating valid transitions between accelerators.

In this step, we group layers as follows: If transitioning to another DSA after a layer is prohibited or leads to increased overhead, the layer is grouped with subsequent layers. Otherwise, the layer is marked as a potential *transition point*.

3.2 Per-layer performance and transition characterization

After we identify all feasible layer groupings, hence the *transition points*, the next step is to characterize each layer's (or layer group's) performance and the overhead if an inter-DSA transition occurs after that particular layer (or layer group).

Layer characterization: Strategically assigning layers to the DSAs where they will run most efficiently has the potential to increase performance. Previous studies [6, 19, 30, 31, 37] have detailed various parameters that affect the efficiency of deep learning accelerators, such as layer type, input size, kernel size, etc. Different layers within a DNN yield varying performance speed-ups when run on a specific DSA. To illustrate and analyze this further, we conduct an experiment where we profile layer groups in GoogleNet on GPU and DLA. The results given in Table 2 show that while the DLA performs slower than GPU for all layers, the speed reduction is less severe for some layers. The fourth column lists the ratio of execution time on DLA over GPU, which varies from 1.40x to 2.02x among different layer groups. Larger performance discrepancies primarily arise because compared to DLAs, GPUs are heavily optimized for large-size matrix operations and they are capable of more effectively exploiting performance on convolution operations with larger inputs. Conversely, smaller kernels, such as those in groups 95-109 and 124-140, are better fits for the DLA's internal on-chip buffer.

Prior studies [3, 6, 11, 25, 32] show that it is feasible to characterize DNNs via a *layer-centric* profiling approach where commonly used layer types are profiled beforehand for different input and filter sizes. Following a similar methodology, we profile each layer or layer group on the DSAs in the system. We utilize IProfiler interface of TensorRT on NVIDIA devices [49], which reports per layer time. Profiled execution

Table 2. Execution (E) and transition (T) time of layer groups in GoogleNet

Layer Group	GPU (ms)	DLA (ms)	D/G E. Time Ratio	T. Time G to D (ms)	T. Time D to G (ms)	Memory Thr. (%)
0-9	0.45	0.75	1.65	0.056	0.15	41.97
10-24	0.19	0.34	1.80	0.075	0.13	62.21
25-38	0.31	0.45	1.44	0.062	0.08	78.49
39-52	0.18	0.37	2.02	0.011	0.03	53.41
53-66	0.16	0.31	1.98	0.055	0.03	55.70
67-80	0.17	0.33	1.96	0.024	0.04	59.24
81-94	0.21	0.31	1.50	0.058	0.05	62.60
95-109	0.25	0.35	1.40	0.030	0.06	76.12
110-123	0.16	0.27	1.66	0.024	0.07	66.95
124-140	0.24	0.36	1.49	0.007	0.05	47.96

times are then embedded into variable t, which is used in equations 2, 4, 5, and 7 in Section 3.4.

Inter-DSA layer transitions: Despite the potential performance boost offered by multi-DSA execution, transitioning between DSAs comes with a cost. This cost, crucial for accurate performance predictions and optimal scheduling, is contingent on the size of the transient data in private caches of DSAs. The output of the layer preceding the transition is flushed back to the shared memory so that the DSA where the next layer will execute on can access it. The fifth and sixth columns in Table 2 represent the time spent when the execution, after each layer group, switches from GPU to DLA and vice versa. As output data sizes decrease toward the end of layer groups, so does transition time. Notably, our experiments also reveal that some layer groups, such as 39-53 and 95-109, ending with pooling layers result in significantly less transition overhead when switching from GPU to DLA. We empirically derive the transition costs of the layers on our target set of accelerators, following the methodology outlined in [11]. To implement them, we insert *MarkOutput* and *addInput* API calls in TensorRT [49]. We then incorporate them into equations 2 and 3 in Section 3.4.

3.3 Characterizing shared memory contention

One of the core novelties of our work is its ability to account for the slowdown caused by shared memory contention. Since existing multi-DSA schedulers do not consider this when making scheduling decisions, the resulting mappings often leave the system under-utilized. However, estimating this slowdown, especially for multi-DSA systems, is not trivial. Exhaustive and peer-wise runs of all combinations of layers by colocating them are required. This will result in a factorial explosion of profiling search space and require significant profiling time [71].

To prevent this, we follow a *decoupled* two-step approach: we first characterize each layer's requested memory throughput when they are run standalone. Using these throughput values, we then utilize a processor-centric slowdown model, PCCS [67], to estimate the slowdown without relying on layer-specific information. PCCS represents the slowdown

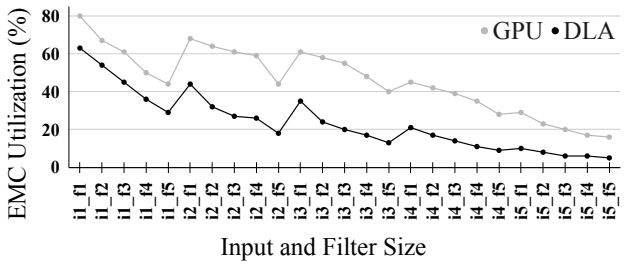

Figure 3. EMC utilization by conv layers on GPU and DLA with varying input (i) and filter (f) sizes

between multiple concurrent workloads as a function of requested memory throughput and external memory traffic, and builds a piece-wise model to predict the slowdown experienced by the accelerator requesting the throughput. Built upon PCCS, our decoupled approach is performed at layer-level and separates the collection of layer-specific standalone performance profiles, as collected in Section 3.2, and the slowdown caused by concurrent execution. The last column in Table 2 lists memory throughput measurements per layer group in GoogleNet. The general pattern we observe in many DNNs is that higher input size results in higher memory throughput. We also observe that, as the filter size in convolution and pooling layers gets larger, there is a decrease in throughput due to the increasing arithmetic intensity of the underlying operation.

Conventional hardware counters to monitor requested memory throughput may not be applicable for some *black-box* DSAs which cannot be profiled with conventional tools. For example, NVIDIA Nsight Compute tool [8] can profile requested memory throughput on GPUs but not on DLAs. As an alternative way to methodologically solve this issue, we develop a four-step approach: 1) We first profile target layers on GPU and analyze the memory throughput for several layer types (*i.e.*, convolution, pooling, and fully connected) and their parameters (*i.e.*, input size filter size). Throughout their execution lifetimes, we observe that many layers individually exhibit homogeneous memory access characteristics as they internally embed homogeneous and dense computations. 2) We then profile external memory controller (EMC) utilization for all layers on both DLA and GPU. In Fig. 3, the input sizes of i1-i5 for the convolution layers correspond to (224,224,64), (224,112,64), (112,112,64), (112,56,64), (56,56,64) and filter sizes of f1-f5 correspond to (1x1), (2x2), (3x3), (4x4), (5x5), respectively. Our analysis reveals that the EMC utilization for DLA and GPU are correlated and proportional. 3) Using this observation, we estimate its memory throughput on black-box DSAs (*e.g.*, DLA in this case) by dividing its GPU-based memory throughput by the ratio of EMC utilization of GPU and DSA for that specific layer. 4) Finally, by utilizing PCCS, we estimate the slowdown of a layer on an accelerator via its requested memory throughput and the external memory throughput requested by the other concurrently running layer on the other DSA.

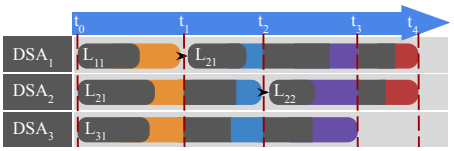

Figure 4. Illustration for a hypothetical execution of five layers from three DNNs running on three different accelerators. Colored regions indicate additional slowdowns each layer experiences for varying external memory pressure.

When multiple layers are run simultaneously on different accelerators, the degree of slowdown throughout their execution is non-uniform and depends on the other layers running concurrently. Fig. 4 illustrates this behavior by depicting the execution timelines of five hypothetical layers belonging to three different DNNs. Each timeline represents the execution of ith layer on jth DNN, labeled as L_{ij} on DSA_k. The black regions in the timeline represent the time that the executions would take if all layers were run separately. Each colored extension to the black regions indicates slowdowns for different sets of layers running together. To address the complexity of handling varying amounts of slowdown during the execution of collocated layers, we introduce a scheduling concept called *contention interval*. Each contention interval (t_i, t_{i+1}) represents a period separated by the start or end of a layer execution, and it is represented by the Eq. 8 explained in Section 3.4. During each contention interval, different rates of slowdowns are observed by each layer, and the slowdown depends on the cumulative external memory pressure demands during that interval.

3.4 Formulating the problem

We integrate layer execution time, inter-accelerator transition time, and memory contention slowdown into a cost function and formulate the scheduling problem as a series of linear equations. Table 3 summarizes the variables and notations we use in the formulation. The primary input to our model is the DNN set for which we explore its mapping to the accelerator set A. $L_{i,n}$ denotes the smallest layer entity that belongs to the layer set of DNN_n. A layer entity is either a single layer or a group of layers, as explained in Section 3.1. Functions $t(L_{i,n}, a)$ and $\tau(L_{i,n}, a, OUT|IN)$ represent the execution time and transition overheads of layer $L_{i,n}$ on accelerator A_a, respectively.

The goal of our formulation is to find the schedule S for all layers across all DNNs. The schedule function, defined in Eq. 1, returns A_a that $L_{i,n}$ should be mapped to. S is assumed to be initially unknown and will be determined by the solver later.

$$S(L_{i,n}) = A_a \quad where \quad 1 \le i \le len(DNN_n), \\ 1 \le n \le len(DNN) \ , \ 1 \le a \le len(A) \tag{1}$$

Total execution time of a DNN is formulated in Eq. 2. Total time comprises *standalone execution time* t of each layer, the *slowdown* C, and and IN and OUT *transition costs*, τ.

Table 3. The notation used by our formulation.

Notation	Explanation
DNN_n	nth DNN in the given DNN set which contains networks to be executed concurrently
$L_{i,n}$	ith layer of the nth DNN in the DNN set
$len(DNN_n)$	Total number (length) of layer groups in DNN_n
A_a	ath accelerator in the given accelerator set A
$S(L_{i,n})$	The schedule, *i.e.*, accelerator mapping, of $L_{i,n}$
$t(L_{i,n}, A_a)$	Total execution time of $L_{i,n}$ on A_a
$st(i,n)$	Execution start time of $L_{i,n}$
$et(i,n)$	Execution end time of $L_{i,n}$
$\tau(L_{i,n}, A_a, OUT\|IN)$	The time required to transition the DNN execution after|before layer l_i executed on accelerator A_a
$TR_{i,n}$	Boolean var if a transition is set after layer $L_{i,n}$
$T(L, S(L))_n$	Total execution time elapsed by the execution of given sets of layer L of the nth DNN
$c_{L_{i,n},S(L),L}$	The slowdown of $L_{i,n}$ due to the contention caused layers running on other accelerators, *i.e.*, $S(L)$
$I_{i,j}$	The length of interval where layers i and j overlap
Int	Interval array holding start and end time of layers

$$T(L, S(L \rightarrow A))_n = \sum_{i=0}^{len(DNN_n)} t(L_{i,n}, S(L_{i,n})) * C_{L_{i,n},S(L),L} \\ + \ TR_{i,n} \ \times \ \tau(L_i, s(L_i), OUT) + \ TR_{i,n} \ \times \ \tau(L_{i+1}, s(L_{i+1}), IN) \tag{2}$$

We encode the decision to make transitions into our formulation via the boolean function given in Eq. 3. This function compares the accelerator assignments of adjacent layers $L_{i,n}$ and $L_{i+1,n}$. If the assignments differ, the transition cost, τ, is subsequently incorporated into Eq. 2.

$$TR_{i,n} = \begin{cases} 1 & , \text{if} \quad S(L_{i,n}) \neq S(L_{i+1,n}) \\ 0 & , \text{if} \quad S(L_{i,n}) = S(L_{i+1,n}) \end{cases} \tag{3}$$

Eq. 4 and 5 compute the execution start and end times, $st()$ and $et()$ respectively, for layer $L_{i,n}$. Int array in Eq. 6 stores the start and end time for layers, facilitating the iterative comparison of the contention intervals across layers.

$$st(i,n) = T(L_{0 \ to \ i-1 \ , \ n}, S(L))_n \tag{4}$$

$$et(i,n) = st(i,n) + t(L_{i,n}, S(L_{i,n})) * C_{i,n} \tag{5}$$

$$\forall L_{i,n} \ , \quad [st(i,n), et(i,n)] \in Int \\ where \quad 1 \le i \le len(DNN_n), \quad 1 \le n \le len(DNN) \tag{6}$$

The contention function C, outlined in Eq. 7, calculates the total slowdown for layer $L_{i,n}$ by taking each time overlapping with that layer and the slowdown ratio corresponding to the interval. The contention model returns an estimated slowdown amount depending on the bandwidth demanded by layer l_i and cumulative external bandwidth demanded by other layers running inside the same interval.

$$C_{L_{i,n},S(L),L} = \sum_{I_k \in Int} \frac{I(L_{i,n} \ , \ L_{j,n}) * cont\_model(L_{i,n} \ , \ L_s)}{t(L_{i,n}, S(L_{i,n})) * len(L_s)} \\ where \ 1 \le j \le len(DNN_n), \ 1 \le n \le len(DNN), \ L_{j,n} \ \in \ L_s \\ Int_k \ \cap \ [st_{i,n}, et_{i,n}] \ \neq \ \emptyset \ , \ Int_k \cap [st_{j,n}, et_{j,n}] \neq \emptyset \tag{7}$$

Eq. 8 details how we determine the duration of contention intervals. If a layer faces no contention, the equation simply returns the layer's execution time, leading to a value of 1 to be returned in Eq. 7, thereby indicating no slowdown effect for a layer running independently in Eq. 2.

$$I(i,j) = \begin{cases} e_j - s_i & if(s_j \le s_i \le e_j \quad \& \quad s_i \le s_j \le e_i) \\ e_j - s_j & if(s_i \le s_j \le e_i \quad \& \quad s_i \le s_j \le e_i) \\ e_i - s_j & if(s_i \le s_j \le e_i \quad \& \quad s_j \le e_i \le e_j) \\ e_i - s_i & if(s_i \le s_j \quad \& \quad e_i \le e_j) \\ e_i - s_i & otherwise \end{cases} \quad (8)$$

We establish a constraint in Eq. 9 that limits two distinct layers from sharing the same accelerator for longer than an ε interval. Ideally, in a flawless model, the estimated execution and slowdown values could yield perfect transitions where accelerator usage periods can be precisely predicted. Variable ε allows us to mitigate the prediction errors, and facilitates more transition points by allowing for a tiny overlap of concurrently assigned layers on the same accelerator at the start or end of their executions.

$$\nexists\, L_{i,nn}, L_{j,n}\ (L_{i,nn} \in DNN_{nn} \ \ and \ \ L_{j,n} \in DNN_n \ \mid$$
$$st_{L_{j,n}} < st_{L_{i,nn}} \mp \varepsilon < et_{L_{i,nn}} \ or \ st_{L_{j,n}} < et_{L_{i,nn}} \mp \varepsilon < et_{L_{j,n}}) \quad (9)$$
$$where\ S(L_{i,nn}) = S(L_{j,n}) \ \ , \ \ nn \neq n$$

Objective functions: Depending on the different scenarios that a user may target, we propose two separate objective functions: Equation 10 maximizes the utilization of the system to increase the total throughput and Equation 11 minimizes the maximum latency among DNNs. The use cases for objective functions are further elaborated in Section 5.

$$\max \sum_{n=1}^{len(DNN)} \frac{1}{T(L,S(L))_n} \quad (10) \qquad \min \max \ T(L,S(L))_n \quad (11)$$

3.5 Optimal and dynamic schedule generation

In our work, we target optimal schedules that satisfy given objectives and constraints because we don't resort to heuristics to find such schedules. We achieve this by representing the entire scheduling problem formulated in Section 3.4 as a constraint-based optimization problem and solving with industry-strength SAT solvers such as Z3 [13], Gurobi [17], and OptiMathSAT [57]. These solvers employ branch & bound techniques to converge towards optimal solutions for many NP-complete problems (*i.e.*, job-shop scheduling) [55]. Considering the relatively small parameter search space of our targeted problem set (*i.e.*, total number of accelerators and tasks in the system), the use of SMT solvers provides optimal schedules in seconds. Depending on the operational requirements of the autonomous system, optimal schedules can be found either statically or dynamically.

Generating optimal schedules beforehand (*i.e.*, *statically*) is feasible for a variety of scenarios, such as in autonomous systems with fixed resolution input devices (like cameras and lidars) and many DNNs designed for a fixed image or a video frame size. Some scenarios, such as a drone switching between *discovery* or *tracking* modes, might require unique control flow graphs (CFGs). Such CFGs (or the path followed in a CFG) and their corresponding schedules can be predetermined statically and toggled during the execution. Thus, users of *HaX-CoNN* can rely on offline profiling to determine the execution costs needed for static scheduling [72].

There are other cases where the static generation of optimal schedules may not be possible. For example, different DNN models may be required for various phases of the autonomous system execution [5, 21, 70], resulting in an unpredictable change in the CFG. For such scenarios, we propose D-*HaX-CoNN*, a runtime-based adaptation of our solution to (1) run SAT solvers on-the-fly, (2) gradually achieve and apply better schedules, and (3) eventually reach an optimal solution as the autonomous system continues to operate. This approach is feasible because autonomous systems often embed long-running loops and once an optimal schedule is found for a recently changed CFG, it will be reused for a while.

4 Experimental Setup

Computing platforms: We use three popular heterogeneous SoCs to evaluate *HaX-CoNN*: NVIDIA AGX Orin [47], Xavier AGX [46], and Qualcomm SnapDragon 865 development kit [53]. All three platforms have a shared memory with multiple accelerators. The technical specifications of these systems are summarized in Table 4. It is essential to note that the maximum number of accelerators we consider in our experiments is limited to two because, to the best of our knowledge, there are no off-the-shelf SoCs that offer more than two types of programmable DSAs for DNN acceleration.

Applications: We use the DNNs that are commonly used in benchmarking DNN inference: Alexnet [36], GoogleNet [61], Inception-V4 [59], ResNet18/52/101/152 [18], VGG-19

Table 4. The HW specifications for targeted architectures.

NVIDIA AGX Orin	
GPU	Ampere arch. 1792 CUDA & 64 Tensor cores
DSA	NVDLA v2.0
CPU	12-core Arm Cortex v8.2 64-bit
Memory	32GB LPDDR5 | **Bandwidth: 204.8 GB/s** with 256-bit
Software	JetPack 5.0.1
NVIDIA Xavier AGX	
GPU	Volta arch. 512 CUDA and 64 Tensor cores
DSA	NVDLA v1.0
CPU	8-core Carmel Arm v8.2 64-bit
Memory	16GB LPDDR4 | **Bandwidth: 136.5 GB/s**, 256-bit
Software	JetPack 4.5
Qualcomm 865 Mobile Development Kit	
GPU	Qualcomm Adreno™ 650 GPU
DSA	Hexagon 698 DSP
CPU	Qualcomm Kryo 585, 8-core, up to 2.84GHz
Memory	6GB LPDDR5 | **Bandwidth: 34.1 GB/s** with 64 bits

Table 5. Standalone runtimes (ms) and relative performance.

Device / DNN	NVIDIA AGX Orin		NVIDIA Xavier AGX	
	GPU (ms)	DLA (ms)	GPU (ms)	DLA (ms)
CaffeNet	0.74	1.79	2.26	5.51
DenseNet	2.19	3.10	7.84	-
GoogleNet	0.99	1.52	1.98	3.68
Inc-res-v2	3.06	5.15	15.12	17.95
Inception	2.49	5.66	8.31	15.94
ResNet18	0.41	0.74	1.37	2.81
ResNet50	0.91	1.67	2.88	6.01
ResNet101	1.56	2.47	5.34	10.6
ResNet152	2.19	3.26	7.7	12.71
VGG19	1.07	2.93	5.95	19.05

[58], FCN-ResNet18, CaffeNet [26], DenseNet [22], and Inc-Res-v2 [60] with datasets from COCO [38], ImageNet ILSVRC [56], and Cityshape [9]. These DNNs could be used for various tasks in autonomous systems, such as object detection, image recognition, semantic segmentation, pose estimation, and depth estimation [15].

Profiling: Profiling duration varies by platforms: Computation, transition, and contention characterizations can take up to 3, 10, and 15 minutes per DNN model, respectively, on NVIDIA Orin, Xavier, and Qualcomm platforms whereas building engines require more time on NVIDIA boards. Since our approach is layer-centric, we performed profiling only once and it is offline.

Neural network synchronization: TensorRT natively does not provide support synchronization between the layers of DNNs concurrently running at different DSAs. To make sure that the inter-accelerator transitions across DNNs are properly performed, we implement a TensorRT plugin that employs inter-DNN synchronization via inter-process shared memory primitives.

Schedule generation: We solve our formulation given in Section 3.4 by using Z3 SMT solver. Z3 has shown superior performance for scheduling problems over popular solvers [55]. It works by determining the satisfiability of the constraints and finding an optimal solution for a given objective and constraints. In most of our experiments, Z3 takes under three seconds to run on a single CPU core of NVIDIA Orin AGX. In some cases, such as for the Inception-ResNet-v2 network which consists of 985 layers, the solver takes around ten seconds to find the optimal schedule.

5 Evaluation

We demonstrate the utility of *HaX-CoNN* via four execution scenarios with different objectives and also via an experiment that exhaustively collocates all the DNNs in our evaluation set. Scenario 1 aims to maximize throughput in concurrent data processing on the same DNN whereas scenarios 2 and 3 target two different DNNs operating in parallel and in a pipeline fashion, respectively. Scenario 4 is a hybrid of scenarios 2 and 3. We benchmark *HaX-CoNN* against five different baselines: (1) GPU only, (2) non-collaborative GPU & DLA, (3) Mensa [6] (which only supports single-DNN execution), (4) Herald [37], and (5) H2H [69] (which both support multi-DNN execution).

5.1 Running multiple instances of the same DNN

Scenario 1 - Concurrent image processing with same DNNs: In systems aiming for high throughput, multiple instances of the same DNN could concurrently process consecutive images. Fig. 5 reports the results of five different experiments designed for this scenario. The experiments are run on NVIDIA Orin and we compare *HaX-CoNN* against two naïve baselines and Mensa [6]. Overall, our experiments for this scenario show that *HaX-CoNN* can boost throughput (*i.e.*, *FPS*) up to 29%. There are several key observations we make in this experiment: (1) In GoogleNet experiment, *HaX-CoNN* maps the middle groups of layers (1-95 and 38-149) to GPU for both DNN instances since GPU executes those layers ∼2x faster than the DLA. (2) Due to shared memory contention, non-collaborative GPU & DLA execution does not always generate a better throughput compared to GPU-only execution. (3) We observe either limited improvements or no improvement by Mensa as it doesn't consider shared memory contention, leading to mismatched layer transitions. Even though Mensa considers transition costs, its greedy strategy fails to account for the transition costs occurring in the future, leading to inaccurate transition decisions.

5.2 Concurrently running different type of DNNs

Table 6 lists the results of the experiments we performed by comparing *HaX-CoNN* to naïve and state-of-the-art multi-DNN concurrent execution schemes. Experiments 1-5 are on Xavier AGX, 6-8 are on AGX Orin, and 9-10 are on Qualcomm 865. The second to fourth columns describe the experiment designs and the corresponding scenarios. There are four baselines we compare our work against: (1) *GPU-only*, (2) *GPU & DSA*, (3) *Herald* [37], (4) *H2H* [69]. The last three columns list the optimal schedules found by *HaX-CoNN*, the latency and throughput (*i.e.*, FPS) for *HaX-CoNN*, and the improvement over the best-performing baseline.

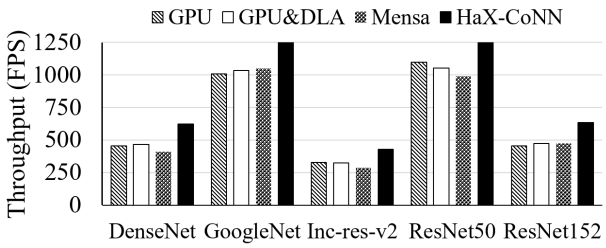

Figure 5. Throughput (FPS) comparison for Scenario 1: Multiple instances of the same DNN is run concurrently on NVIDIA AGX Orin.

Table 6. Experiments run for Scenarios 2, 3, and 4. We compare these scenarios against baselines when run on NVIDIA Xavier AGX (in experiments 1-5), NVIDIA AGX Orin (in experiments 6-8), and Qualcomm 865 (in experiments 9-10). DSA refers to DLA for NVIDIA platforms and to the Hexagon DSP for the Qualcomm platform.

Exp #	Goal	DNN-1	DNN-2	(1) GPU only		(2) GPU & DSA		(3) Herald		(4) H2H		Optimal schedule by HaX-CoNN		Runtime of *HaX-CoNN* schedule		Improvement over the best baseline (%)	
				Lat.	FPS	Lat.	FPS	Lat.	FPS	Lat.	FPS	TR	Dir.	Lat.	FPS	Lat.	FPS
1	Min Latency	VGG-19	ResNet152	17.05	58	**16.05**	62	19.73	50	16.55	60	29 89	DtoG GtoD	**13.01**	77	23	22
2	Min Latency	ResNet152	Inception	16.23	61	15.96	62	15.81	63	**15.75**	64	188 72	DtoG GtoD	**13.11**	76	20	18
3	Max FPS	Alexnet	ResNet101	11.04	90	10.97	**93**	12.10	82	11.49	87	11 161	GtoD DtoG	8.7	**115**	26	23
4	Max FPS	ResNet101	GoogleNet	7.02	**143**	7.37	140	8.95	111	9.10	109	0 0	DtoG DtoG	7.02	**143**	0	0
5	Min Latency	GoogleNet ResNet152	FC_ResN18	**15.41**	77	18.88	61	23.68	47	20.90	54	38 235	DtoG GtoD	**12.09**	85	22	21
6	Min Latency	VGG-19	ResNet152	**3.95**	267	4.58	218	5.76	174	4.90	204	27 95	DtoG GtoD	**3.21**	311	23	22
7	Max FPS	GoogleNet	ResNet101	4.12	378	4.24	364	4.44	340	4.13	**380**	38 128	DtoG GtoD	3.4	**426**	19	18
8	Min Latency	ResNet101 GoogleNet	Inception	5.06	197	4.97	201	5.56	180	**4.91**	203	31 88	DtoG GtoD	**4.41**	226	13	12
9	Max FPS	GoogleNet	ResNet101	98.3	10.1	79.1	**12.6**	95.9	10.4	113.8	8.8	52 148	DtoG GtoD	71.08	**14.1**	11	10
10	Min Latency	Inception	ResNet152	219.6	4.5	**178.2**	5.6	223.1	4.5	202.3	5.2	17 135	DtoG GtoD	**155.3**	6.4	15	15

Scenario 2 - Two different DNNs operating on the same data: This scenario illustrates a case where different DNNs, such as object detection and image segmentation, process the same input in parallel, and they synchronize afterwards. The results are assumed to be passed on to subsequent tasks, such as motion planning [45], and then the loop is started over. Experiments 1, 2, 6, and 10 in Table 6 are run to demonstrate this scenario on three different target architectures. Our results show that *HaX-CoNN* improves both latency and throughput up to 23% in all four experiments of this scenario. We also observe that both H2H and Herald make inaccurate latency estimations that are wrong by up to 75% since neither of them considers shared memory contention. Experiments 1 and 6 show that *HaX-CoNN* results in different schedules for the same scenario running on different SoCs. For experiment 10, GPU & DSP is the best performing baseline for Qualcomm platform since GPU & DSP are more balanced on this platform in terms of their computation capability. Even though the schedule found by *HaX-CoNN* in experiment 10 on Qualcomm has a relatively higher transition cost among other transition candidates, the improvement primarily comes from minimizing the memory contention and effectively distributing the layers to DSAs.

Scenario 3 - Two different DNNs operating on streaming data: This scenario examines a common autonomous system setup where the input (*e.g.*, camera stream) is available as a data stream and multiple tasks, such as object detection followed by object tracking [54], are executed in a pipelined manner. This scenario is covered by experiments 3, 4, 7, and 9. To establish the dependency among DNNs, we connect the last layer of the DNN_1 to the first layer of DNN_2 as an input. Interestingly, *HaX-CoNN* opts not to use DLA for none of the layers in experiment 4 since running two images sequentially on the GPU yields a higher throughput. Particularly, the performance of DLA on ResNet18 is less than the slowdown imposed on the GPU. Overall, if there are cases where layer-level mapping does not foster any benefits, *HaX-CoNN* is capable of identifying these cases and utilizing the baseline solution instead. Our scheme guarantees that no worse results are obtained than the naïve baselines.

Scenario 4 - Multiple DNNs with concurrent and streaming data: In this scenario, two DNNs (DNN_1 and DNN_2) have a serial dependency in between and another DNN (DNN_3) runs in parallel with the former two [54]. Experiments 5 and 8 belong to this scenario and the objective function is set to minimize the combined latency. *HaX-CoNN* is able to provide latency and throughput improvements up to 22%. Best performing baselines run DNN_3 mostly on GPU since unbalanced workloads among accelerators and shared memory contention alleviate the advantages of concurrent utilization. In experiment 5, the schedules that use both DSAs concurrently perform worse than serialized GPU executions, since DLA is generally less effective in running fully-connected layers. In experiment 8, H2H provides the fastest baseline performance since they are capable of exploiting heterogeneity of DSAs for appropriate layers (*e.g.*, such

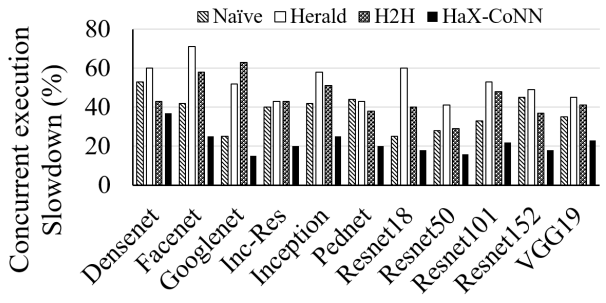

Figure 6. Slowdown of concurrently executing GoogleNet on GPU with different DNNs on DLA.

as running DLA-efficient small layers on DLA and assigning others to the GPU). However, the schedules proposed by H2H lead to an over-subscribed DLA execution. On the other hand, *HaX-CoNN* finds the right transition points where no accelerators are overloaded and the transition cost is lower. In some schedules generated by H2H, such as experiment 3, while the transition points that are identified by H2H prevent accelerator over-subscription, the assignments for the remaining layers on both DNNs lead to workload imbalance.

Throughout the experiments, we generally observe that the benefits of architectural heterogeneity exploited by the state-of-the-art remain limited. The primary reason for the subpar performance of H2H and Herald compared to naive baselines is that their cost functions ignore shared memory contention. This, in turn, causes the timings to be mispredicted and eventually results in being unable to generate optimal schedules. Certain layers end up being assigned to the same accelerator (either GPU or DSA) at the same time, and this is due to poor (*i.e.*, non-optimal) handling of constraints triggered by mispredicted execution times. For example, on the DLA, two layer groups that are supposed to execute at different times are scheduled together, but they end up waiting for each other. During this time, the other accelerator (*e.g.*, the GPU) is left idle.

In experiments 4 and 9 of Table 6, the latency of schedules generated by Herald is better than H2H because H2H makes optimizations to reduce the transition costs, yet leading to worse inter-DSA contention. We also observe that some of the optimizations performed by H2H are already performed by TensorRT. Therefore, such optimizations are already covered by our baselines, and this may hinder the benefits of H2H over our baselines. On the other hand, this situation does not affect the benefits demonstrated by *HaX-CoNN* over H2H and Herald. Also, it is worth noting it takes more time to generate schedules with H2H or Herald (*i.e.*, more than 10 seconds in most cases) than with HaX-CoNN.

Based on what we observe in Table 6, we further analyze the slowdown caused by memory contention. Fig. 6 depicts the amount of slowdown experienced by GoogleNet running on the GPU when other DNNs are concurrently run on the DLA of Xavier AGX. The slowdown is calculated based on the standalone GPU execution of GoogleNet where there

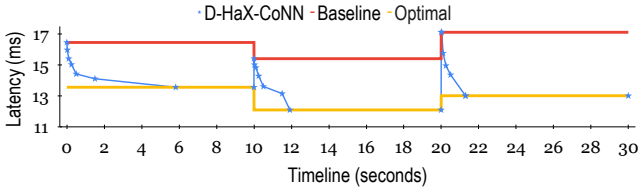

Figure 7. A dynamic execution scenario where the target CFG (*i.e.*, DNN pairs) changes every 10 seconds. D-*HaX-CoNN* is shown to gradually improve the execution time as Z3 is asked to update schedules at 25ms, 100ms, 250ms, 500ms, and 1.5s. Blue stars show the update intervals.

are no other concurrently running DNNs. *HaX-CoNN* significantly reduces the shared memory contention slowdown in all experiments.

5.3 Adapting optimal scheduling to dynamically changing workloads

As discussed in Section 3.5, we propose D-*HaX-CoNN* to handle dynamic changes to the autonomous CFGs. Its operation is as follows: (1) It starts with an initial best naïve schedule.[1] (2) As the autonomous loop starts executing with the initial schedule, we periodically replace the initial schedule with a better schedule as Z3 progresses. (3) We continue running Z3 until no further improvement is possible.

To demonstrate the effectiveness of D-*HaX-CoNN*, we perform an experiment where dynamic changes in the CFG are simulated by changing three DNN pairs being executed every 10 seconds. DNN pairs are the same within experiments 2, 5, and 1 in Table 6, respectively. Fig. 7 depicts the concurrent execution time of the DNN pairs (*i.e.*, latency per image) as they change. In this experiment, D-*HaX-CoNN* is run on a single CPU core with an initial schedule given by baseline. We update the schedules at 25ms, 100ms, 250ms, 500ms, and 1.5s after starting Z3. The blue lines correspond to the execution time of the updated schedule. The optimal schedule for each pair (represented by a yellow line) is calculated to denote the *oracle* solution that D-*HaX-CoNN* is expected to reach.

Our results show that D-*HaX-CoNN* quickly converges to the optimal solution. In particular, D-*HaX-CoNN* reaches an optimal solution faster for the second and third DNN pairs (1.9s and 1.3s), compared to the first pair (5.8s), since the latter pair has three DNNs and more layer groups. As explained before, a larger number of layer groups results in potentially more transition points to explore, which then increases the time required to explore all transition candidates for the optimized objective function.

To evaluate the overhead of running Z3 solver along with the concurrent DNN execution, we conduct another experiment where we run AlexNet on DLA along with various DNNs on GPU while Z3 solver runs on a single CPU core of

[1]We do not start with a Herald or H2H schedule since they also take seconds to return a schedule.

NVIDIA AGX Orin. The results, presented in Table 7, show that running the solver on the fly slows down the DNN execution time by no more than 2%. This is attributed to Z3's low memory footprint and Z3 successfully reduces the size of the parameter search space for our targeted problem into a smaller set.

5.4 Exhaustive evaluation with all DNN pairs

The DNNs that are run concurrently in the experiments presented in Section 5.2 were handpicked to reflect the importance of the use cases in each scenario. In this subsection, we conduct a comprehensive evaluation of *HaX-CoNN*, by running every possible DNN pair in our entire DNN set. Since we test every possible pair, the execution times for two concurrent DNNs can significantly differ. We first check the execution time on DLA and GPU for DNN-1 and compare it to DNN-2. Then, to balance out the discrepancy, we increase the number of iterations for the faster DNN. Such scenarios are quite common in multi-sensor systems where two independent sensor data (*i.e.*, camera and radar) are processed concurrently at different frequencies, or where multiple iterations over consecutive data are required to maintain the system's overall accuracy above a threshold. Results of this experiment are given in Table 8 as a lower triangular matrix –The upper triangular matrix is symmetric because we are running DNN pairs. The first row of each cell shows the accelerator(s) where the baseline is the fastest for the corresponding objectives. The second row of each cell shows the percentage of improvements that *HaX-CoNN* was able to achieve over the baseline. In this experiment, due to the complexity of the scheduling and because of similar reasons explained in Section 5.2, both H2H and Herald mostly result in worse runtimes than the naïve baselines. Key observations from this experiment include:

1. Any pair involving GoogleNet shows improvement since GPU's performance is close to DLA's performance on GoogleNet and *HaX-CoNN* can exploit different transition points where both accelerators are efficient.
2. Overall, *HaX-CoNN* improves the throughput on 35 pairs out of 45 and identifies that GPU-only execution should be applied to the remaining 10 pairs (which are marked as *x* in Table 8), ensuring that *HaX-CoNN* does not underperform. However, experiments involving VGG19 show improvement only in three pairs. The fastest baselines for this DNN are all GPU-only and the execution of VGG19 on DLA is substantially slower than on GPU. Running another DNN on the DLA slows down the entire execution due to high memory contention. When DenseNet or GoogleNet are paired with VGG-19, *HaX-CoNN* shows a slight speed-up since DLA is proportionally faster than the average on the last layer groups in DenseNet and GoogleNet, and in the initial groups of VGG-19.
3. Despite the execution of CaffeNet on DLA being slower compared to GPU; favoring a GPU-only baseline, *HaX-CoNN* is still able to improve performance since CaffeNet is a compute-intensive DNN and does not cause too much contention when paired with other DNNs.

Table 7. The scheduling overhead (%) of dynamically running the Z3 solver on a CPU core while AlexNet on the DLA is concurrently executed along with other DNNs on the GPU of Xavier Orin.

CaffeNet	DenseNet	GoogleNet	Inc-res-v2	Inception	MobileNet
0.45%	0.89%	1.64%	0.69%	1.64%	1.31%
ResNet18	ResNet52	ResNet101	ResNet152	VGG16	VGG19
0.16%	%0.23%	0.38%	0.71%	1.12%	1.59%

Table 8. Comparison among *HaX-CoNN* and the best baseline for DNN pairs running on AGX Orin.

DNNs	1	2	3	4	5	6	7	8	9	10
1-CaffeNet	GPU **1.13**									
2-DenseNet	GPU **1.14**	H2H **1.18**								
3-GoogleNet	GPU **1.06**	D/G **1.18**	GPU **1.22**							
4-Inc-res-v2	GPU **1.08**	GPU **x**	D/G **1.25**	D/G **1.18**						
5-Inception	GPU **1.10**	GPU **1.15**	GPU **1.15**	D/G **1.06**	H2H **1.05**					
6-ResNet18	GPU **x**	D/G **1.14**	G/D **1.13**	D/G **1.32**	GPU **1.19**	GPU **1.23**				
7-ResNet50	GPU **x**	D/G **1.21**	H2H **1.06**	GPU **1.16**	GPU **1.11**	GPU **1.06**	GPU **1.17**			
8-ResNet101	GPU **1.11**	G/D **1.05**	G/D **1.08**	D/G **1.19**	GPU **1.08**	D/G **1.24**	GPU **1.11**	GPU **1.09**		
9-ResNet152	GPU **1.09**	G/D **1.08**	G/D **1.17**	GPU **1.14**	Her. **1.07**	D/G **1.18**	H2H **1.09**	GPU **1.08**	GPU **1.18**	
10-VGG19	GPU **x**	GPU **1.11**	GPU **1.04**	GPU **x**	GPU **x**	GPU **1.08**	GPU **x**	GPU **x**	GPU **x**	GPU **x**

6 Conclusion

We propose *HaX-CoNN*, a scheme that maps layers in concurrently executing DNN inference workloads to the accelerators of a heterogeneous SoC. *HaX-CoNN* holistically considers per-layer execution characteristics, shared memory contention, and inter-accelerator transitions while finding optimal schedules. Our experimental results show that *HaX-CoNN* can improve latency up to 32%.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable feedback. This material is based upon work supported by National Science Foundation (NSF) under Grants No. CCF-2124010 and CHE-2235143, and National Institute for Occupational Safety and Health (NIOSH) Contract number 75D30119C05413. Any opinions, findings, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or NIOSH.

References

[1] NVIDIA Deep Learning Accelerator. 2023. http://nvdla.org/ (accessed on 08/04/2023).

[2] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783725

[3] Laith Alzubaidi, Jinglan Zhang, Amjad J Humaidi, Ayad Al-Dujaili, Ye Duan, Omran Al-Shamma, José Santamaría, Mohammed A Fadhel, Muthana Al-Amidie, and Laith Farhan. 2021. Review of deep learning: Concepts, CNN architectures, challenges, applications, future directions. *Journal of big Data* 8 (2021), 1–74.

[4] Mehmet E Belviranli, Farzad Khorasani, Laxmi N Bhuyan, and Rajiv Gupta. 2016. Cumas: Data transfer aware multi-application scheduling for shared gpus. In *Proceedings of the 2016 International Conference on Supercomputing*. 1–12.

[5] Behzad Boroujerdian, Radhika Ghosal, Jonathan Cruz, Brian Plancher, and Vijay Janapa Reddi. 2021. Roborun: A robot runtime to exploit spatial heterogeneity. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 829–834.

[6] Amirali Boroumand, Saugata Ghose, Berkin Akin, Ravi Narayanaswami, Geraldo F Oliveira, Xiaoyu Ma, Eric Shiu, and Onur Mutlu. 2021. Google neural network models for edge devices: Analyzing and mitigating machine learning inference bottlenecks. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 159–172.

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

[8] NVIDIA Nsight Compute. 2022. https://docs.nvidia.com/nsight-compute/NsightCompute/index.html (accessed on 08/04/2023).

[9] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. 2016. The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3213–3223.

[10] Ismet Dagli and Mehmet E Belviranli. 2021. Multi-accelerator Neural Network Inference in Diversely Heterogeneous Embedded Systems. In *2021 IEEE/ACM Redefining Scalability for Diversely Heterogeneous Architectures Workshop (RSDHA)*. IEEE, 1–7.

[11] Ismet Dagli, Alexander Cieslewicz, Jedidiah McClurg, and Mehmet E. Belviranli. 2022. AxoNN: Energy-Aware Execution of Neural Network Inference on Multi-Accelerator Heterogeneous SoCs. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*.

[12] Ismet Dagli, Andrew Depke, Andrew Mueller, Md Sahil Hassan, Ali Akoglu, and Mehmet Esat Belviranli. 2023. Contention-Aware Performance Modeling for Heterogeneous Edge and Cloud Systems. In *Proceedings of the 3rd Workshop on Flexible Resource and Application Management on the Edge* (Orlando, FL, USA) *(FRAME '23)*. Association for Computing Machinery, New York, NY, USA, 27–31. https://doi.org/10.1145/3589010.3594889

[13] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*. Springer, 337–340.

[14] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture(ISCA)*. 92–104.

[15] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. 2020. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics* 37, 3 (2020), 362–386.

[16] Jérémie Guiochet, Mathilde Machin, and Hélène Waeselynck. 2017. Safety-critical advanced robots: A survey. *Robotics and Autonomous Systems* 94 (2017), 43–52.

[17] Gurobi Optimization, LLC. 2023. Gurobi Optimizer Reference Manual. https://www.gurobi.com

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

[19] Kartik Hegde, Po-An Tsai, Sitao Huang, Vikas Chandra, Angshuman Parashar, and Christopher W Fletcher. 2021. Mind mappings: enabling efficient algorithm-accelerator mapping space search. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 943–958.

[20] Mark Hill and Vijay Janapa Reddi. 2019. Gables: A roofline model for mobile socs. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 317–330.

[21] Yu-Shun Hsiao, Siva Kumar Sastry Hari, Michał Filipiuk, Timothy Tsai, Michael B Sullivan, Vijay Janapa Reddi, Vasu Singh, and Stephen W Keckler. 2022. Zhuyi: perception processing rate estimation for safety in autonomous vehicles. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 289–294.

[22] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q Weinberger. 2017. Densely Connected Convolutional Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition(CVPR)*.

[23] Nathaniel Hudson, Hana Khamfroush, and Daniel E Lucani. 2021. QoS-aware placement of deep learning services on the edge with multiple service implementations. In *2021 International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 1–8.

[24] EunJin Jeong, Jangryul Kim, Samnieng Tan, Jaeseong Lee, and Soonhoi Ha. 2021. Deep learning inference parallelization on heterogeneous processors with tensorrt. *IEEE Embedded Systems Letters* 14, 1 (2021), 15–18.

[25] Fucheng Jia, Deyu Zhang, Ting Cao, Shiqi Jiang, Yunxin Liu, Ju Ren, and Yaoxue Zhang. 2022. CoDL: efficient CPU-GPU co-execution for deep learning inference on mobile devices. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*. Association for Computing Machinery New York, NY, USA, 209–221.

[26] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093* (2014).

[27] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A unified architecture for accelerating distributed {DNN} training in heterogeneous {GPU/CPU} clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 463–479.

[28] Duseok Kang, Jinwoo Oh, Jongwoo Choi, Youngmin Yi, and Soonhoi Ha. 2020. Scheduling of deep learning applications onto heterogeneous processors in an embedded device. *IEEE Access* 8 (2020), 43980–43991.

[29] Sheng-Chun Kao and Tushar Krishna. 2020. Gamma: Automating the hw mapping of dnn models on accelerators via genetic algorithm. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.

[30] Sheng-Chun Kao and Tushar Krishna. 2022. MAGMA: An Optimization Framework for Mapping Multiple DNNs on Multiple Accelerator Cores. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 814–830.

[31] Sheng-Chun Kao, Suvinay Subramanian, Gaurav Agrawal, and Tushar Krishna. 2023. FLAT: An Optimized Dataflow for Mitigating Attention

Performance Bottlenecks. *published in arxiv, will appear in Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2023).

[32] Andreas Karatzas and Iraklis Anagnostopoulos. 2023. OmniBoost: Boosting Throughput of Heterogeneous Embedded Devices under Multi-DNN Workload. *arXiv preprint arXiv:2307.03290* (2023).

[33] Seah Kim, Hasan Genc, Vadim Vadimovich Nikiforov, Krste Asanović, Borivoje Nikolić, and Yakun Sophia Shao. 2023. MoCA: Memory-Centric, Adaptive Execution for Multi-Tenant Deep Neural Networks. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 828–841.

[34] Srivatsan Krishnan, Zishen Wan, Kshitij Bhardwaj, Ninad Jadhav, Aleksandra Faust, and Vijay Janapa Reddi. 2022. Roofline model for uavs: A bottleneck analysis tool for onboard compute characterization of autonomous unmanned aerial vehicles. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 162–174.

[35] Srivatsan Krishnan, Zishen Wan, Kshitij Bhardwaj, Paul Whatmough, Aleksandra Faust, Sabrina Neuman, Gu-Yeon Wei, David Brooks, and Vijay Janapa Reddi. 2022. Automatic Domain-Specific SoC Design for Autonomous Unmanned Aerial Vehicles. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 300–317.

[36] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems(NeurIPS)* 25 (2012).

[37] Hyoukjun Kwon, Liangzhen Lai, Michael Pellauer, Tushar Krishna, Yu-Hsin Chen, and Vikas Chandra. 2021. Heterogeneous dataflow accelerators for multi-DNN workloads. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 71–83.

[38] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. 2014. Microsoft coco: Common objects in context. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*. Springer, 740–755.

[39] Hao Luan, Yu Yao, and Chang Huang. 2022. A Many-Ported and Shared Memory Architecture for High-Performance ADAS SoCs. *IEEE Design & Test* 39, 6 (2022), 5–15.

[40] Pak Markthub, Mehmet E Belviranli, Seyong Lee, Jeffrey S Vetter, and Satoshi Matsuoka. 2018. DRAGON: breaking GPU memory capacity limits with direct NVM access. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 414–426.

[41] Mohammad Alaul Haque Monil, Mehmet E Belviranli, Seyong Lee, Jeffrey S Vetter, and Allen D Malony. 2020. Mephesto: Modeling energy-performance in heterogeneous socs and their trade-offs. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 413–425.

[42] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles(SOSP)*. 1–15.

[43] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. {Heterogeneity-Aware} Cluster Scheduling Policies for Deep Learning Workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 481–498.

[44] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation(PLDI)*. 883–898.

[45] Rihards Novickis, Aleksandrs Levinskis, Roberts Kadikis, Vitalijs Fescenko, and Kaspars Ozols. 2020. Functional architecture for autonomous driving and its implementation. In *2020 17th Biennial Baltic Electronics Conference (BEC)*. IEEE, 1–6.

[46] NVIDIA. 2023. AI-Powered Autonomous Machines at Scale | NVIDIA Jetson AGX Xavier. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/. (accessed on 08/04/2023).

[47] NVIDIA. 2023. Next-level AI performance for next-gen robotics | NVIDIA Jetson Orin AGX. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/. (accessed on 08/04/2023).

[48] NVIDIA. 2023. TensorRT. https://developer.nvidia.com/tensorrt (accessed on 08/04/2023).

[49] NVIDIA. 2023. TensorRT IProfiler. https://docs.nvidia.com/deeplearning/tensorrt/api/c_api/classnvinfer1_1_1_i_profiler.html (accessed on 08/04/2023).

[50] Jay H Park, Gyeongchan Yun, Chang M Yi, Nguyen T Nguyen, Seungmin Lee, Jaesik Choi, Sam H Noh, and Young-ri Choi. 2020. Hetpipe: Enabling large DNN training on (whimpy) heterogeneous GPU clusters through integration of pipelined model parallelism and data parallelism. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. 307–321.

[51] Fareed Qararyah, Mohamed Wahib, Doğa Dikbayır, Mehmet Esat Belviranli, and Didem Unat. 2021. A computational-graph partitioning method for training memory-constrained DNNs. *Parallel computing* 104 (2021), 102792.

[52] Qualcomm. 2023. Neural Processing SDK for AI. https://developer.qualcomm.com/software/qualcomm-neural-processing-sdk (accessed on 08/04/2023).

[53] Qualcomm. 2023. Snapdragon 865 Mobile Hardware Development Kit. https://stage.developer.qualcomm.com/hardware/snapdragon-865-hdk. (accessed on 08/04/2023).

[54] Ratheesh Ravindran, Michael J Santora, and Mohsin M Jamali. 2020. Multi-object detection and tracking, based on DNN, for autonomous vehicles: A review. *IEEE Sensors Journal* 21, 5 (2020), 5668–5677.

[55] Sabino Francesco Roselli, Kristofer Bengtsson, and Knut Åkesson. 2018. SMT solvers for job-shop scheduling problems: Models comparison and performance evaluation. In *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*. IEEE, 547–552.

[56] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. Imagenet large scale visual recognition challenge. *International journal of computer vision* 115 (2015), 211–252.

[57] Roberto Sebastiani and Patrick Trentin. 2015. OptiMathSAT: A tool for optimization modulo theories. In *International conference on computer aided verification*. Springer, 447–454.

[58] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations(ICLR)*.

[59] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. 2017. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-first AAAI conference on artificial intelligence*.

[60] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A. Alemi. 2017. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. In *AAAI*.

[61] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.

[62] Tesla. 2023. Tesla Autopilot AI. https://www.tesla.com/AI

[63] Stavros Tzilis, Pedro Trancoso, and Ioannis Sourdis. 2019. Energy-efficient runtime management of heterogeneous multicores using online projection. *ACM Transactions on Architecture and Code Optimization (TACO)* 15, 4 (2019), 1–26.

[64] Zishen Wan, Karthik Swaminathan, Pin-Yu Chen, Nandhini Chandramoorthy, and Arijit Raychowdhury. 2022. Analyzing and Improving Resilience and Robustness of Autonomous Systems. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. 1–9.

[65] Miao Wang, Xu-Quan Lyu, Yi-Jun Li, and Fang-Lue Zhang. 2020. VR content creation and exploration with deep learning: A survey. *Computational Visual Media* 6 (2020), 3–28.

[66] Hsin-I Wu, Da-Yi Guo, Hsu-Hsun Chin, and Ren-Song Tsay. 2020. A pipeline-based scheduler for optimizing latency of convolution neural network inference over heterogeneous multicore systems. In *IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. IEEE, 46–49.

[67] Yuanchao Xu, Mehmet Esat Belviranli, Xipeng Shen, and Jeffrey Vetter. 2021. PCCS: Processor-Centric Contention-Aware Slowdown Model for Heterogeneous System-on-Chips. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 1282–1295. https://doi.org/10.1145/3466752.3480101

[68] Dan Zhang, Safeen Huda, Ebrahim Songhori, Kartik Prabhu, Quoc Le, Anna Goldie, and Azalia Mirhoseini. 2022. A full-stack search technique for domain optimized deep learning accelerators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 27–42.

[69] Xinyi Zhang, Cong Hao, Peipei Zhou, Alex Jones, and Jingtong Hu. 2022. H2H: Heterogeneous Model to Heterogeneous System Mapping with Computation and Communication Awareness. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*.

[70] Hengyu Zhao, Yubo Zhang, Pingfan Meng, Hui Shi, Li Erran Li, Tiancheng Lou, and Jishen Zhao. 2020. Safety score: A quantitative approach to guiding safety-aware autonomous vehicle computing system design. In *2020 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 1479–1485.

[71] Qi Zhu, Bo Wu, Xipeng Shen, Li Shen, and Zhiying Wang. 2017. Co-run scheduling with power cap on integrated cpu-gpu systems. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 967–977.

[72] Jie Zou, Xiaotian Dai, and John A McDermid. 2022. Resilience-Aware Mixed-Criticality DAG Scheduling on Multi-cores for Autonomous Systems. *ACM SIGAda Ada Letters* 42, 1 (2022), 81–85.

A Artifact Appendix

A.1 Abstract

The artifact described here includes the source code for HaX-CoNN, DNN profiling logs, runtimes, and the sources for the applications used in our evaluation.

A.2 Description

Check-list (artifact meta information):

- **Hardware:** NVIDIA Jetson Xavier AGX 32 GB and NVIDIA Jetson Orin AGX 32 GB
- **Software:** Default installation of Jetpacks by NVIDIA Jetpack 4.5.1 on Xavier AGX and JetPack 5.1.1 on Orin AGX
- **Architecture:** aarch64
- **Software details needed:** Xavier AGX uses Python 3.6.9, TensorRT 7.1.3, CUDA 10.2.89 and Orin AGX uses Python 3.8.10, TensorRT 8.4.0, CUDA 11.2
- **Binary:** Binary files are large, so generating them by using scripts in this artifact is necessary.
- **Output:** Profiling data (execution time, transition time, memory use) for both layers and neural networks. The end result is the improved total execution time/throughput.
- **Experiment workflow:** Makefile, Python and bash scripts
- **Disk space required (approximately):** 8GB

A.3 Public Availability

We maintain the most updated version of the code in the GitHub repository given below. Please refer to this repository for the most updated version.

`https://github.com/ismetdagli/HaX-CoNN`

As stated in the requirements of green badge definition, this code is publicly available under Zenodo as well.

`https://zenodo.org/records/10225025`

A.4 Hardware dependencies

We performed our experiments on an NVIDIA Jetson Xavier AGX 32 GB and NVIDIA Jetson AGX Orin 32 GB. While HaX-CoNN is compatible with any architectures using TensorRT with NVIDIA GPUs, we also use DLA which only exists in NVIDIA Jetson Families. So, reproducibility of the code requires Xavier AGX or AGX Orin whereas the methodology can be applied to other heterogeneous shared memory SoCs (i.e., Qualcomm 865 Development Kit).

A.5 Software dependencies

The easiest way to follow our dependencies is to use Jetpack 4.5.1 on Xavier AGX and Jetpack 5.1.1 on Orin AGX. We mainly use TensorRT as ML framework in our implementation, as DLA can only be programmed via TensorRT. Xavier AGX has TensorRT 7.1.3 and Orin AGX uses TensorRT 8.4.0. It is important to note that manually installing TensorRT/CUDA on Jetson boards etc. is not suggested and all software-related installations are available through JetPack.

A.6 Installation

We assume installation through JetPack is followed. Upon it, run the script below to install python3 dependencies.

```
$ apt install -y python3-pip
$ pip3 install -r requirements.txt
```

If you are using a different Python 3 version than the default one that comes with JetPack, please modify the default version as 3.6.9 on Xavier AGX and 3.8.10 on Orin AGX by using update-alternatives

A.7 Starter guide

This is a starter guide for a DNN pair (Resnet101 and GoogleNet). The system can either run both DNNs on GPU, or select the map among GPU and DLA. We propose to distribute the layers among GPU and DLA.

```
$ ./starter_guide.sh
```

A.8 Experiment Flow

All of the applications's input files we discussed in Sec. 3/4/5. Their input file can be found under $< main - directory > /prototxt\_input\_files$. Under src, we provide buildEngine.py. By using input files, TensorRT generates the engines to run on GPU or DLA by following the given arguments. Running *make* generates all the necessary engines to analyze GoogleNet, which is given as an example in Table 2. We provide three major profiling scripts/ under script folder, which are layer_analysis, transition_analysis, and emc_analysis. First, the layer analysis step explains how layers are profiled. A summary of comprehensive profiling results can be running the script under it as explained in the first subsection of Section 3.2. The second step explains how transition times are collected as explained in the second subsection of Section 3.2. The last step explains how EMC utilizations are collected as explained in Section 3.3. Also, the Nsight Compute profiling script is run for this step. We modified TensorRT's sampleInference.cpp source file. If another version is targeted, the same logic needs to be applied to synchronize DNNs. If different devices are targeted, DL frameworks need to be modified in a way that DNNs can simultaneously start.

```
$make
$python3 scripts/layer_analysis/layer_all_util.py
$python3 scripts/transition_time_analysis/
transition_util.py
```

Using profiling results, we generate a schedule and run the experiments. Under src, we run z3 solver. Following the schedules found, the corresponding engines are generated. Multi-DNN experiment script is given below and targets to generate some experiments given in Table 6. The overhead experiment given in Table 7 can be regenerated by using the overhead script given below.

```
$ python3 src/z3_solver_multi_dnn.py
$ ./collect_data_multidnn_experiment.sh
$ ./scripts/run_all_plan.sh build/overhead_gpu
```