Empirical Analysis of Multi-Model Placement for Heterogeneous CPU–NPU Platforms

Abstract—**Placement decisions for multi-model deep learning pipelines on heterogeneous CPU–NPU platforms are often based on single-model profiles or analytical cost models. In practice, these approaches fail to capture contention introduced by CPU-side preprocessing, postprocessing, rendering, and PCIe transfers, resulting in inaccurate predictions under co-execution. We propose an empirical placement methodology that evaluates complete end-to-end pipelines and ranks candidate assignments using a score that balances throughput and frame drops. Our evaluation, conducted on a host with one CPU and two PCIe-attached NPUs connected via asymmetric links, with ResNet50 and YOLOv3 workloads, reveals three key findings: (1) per-model latencies measured in isolation are poor predictors of co-execution performance because CPU and PCIe contention dominate; (2) the optimal placement shifts as the number of concurrently active models increases; and (3) across different input-rate ranges, the identity of the optimal placement remains largely stable even as absolute throughput changes. For four-model workloads, our best placement improves the score by up to 113.5% compared to a baseline derived from single-model profiling.**

Index Terms—**Heterogeneous computing, Model placement, Multi-model inference, Neural processing units**

I. INTRODUCTION

Deep neural network (DNN) inference increasingly runs on heterogeneous platforms with CPUs and NPUs under tight resource and power budgets. Off-the-shelf PCIe-attached accelerators (e.g., Qualcomm Cloud AI 100 [1], Blaize Xplorer X1600P [2], Google Coral Dual Edge TPU [3], and Hailo-8 [4]) allow deployment in commodity host systems with minimal integration effort. However, practical applications rarely consist of a single isolated model. Real workloads often co-run multiple models and include substantial preprocessing, postprocessing, and rendering on the CPU, as well as host–device transfers over PCIe links whose bandwidth and latency may be asymmetric. Under co-execution, these factors cause coupled contention on CPU cores, last-level caches, and PCIe, which makes placement decisions more complex than suggested by per-model profiles.

Frameworks based on cost or communication models (e.g., MOSAIC [5], FlexFlow [6], TVM Ansor [7]) and service-level schedulers (e.g., InferLine [8], Clockwork [9]) have shown strong results in cloud and GPU settings. However, they typically assume predictable workloads and homogeneous accelerators, and therefore do not capture host–device transfer costs, CPU-side stages, or interference among co-running models. As a result, placements that appear favorable in isolation can degrade sharply once models contend for CPU time and PCIe bandwidth. This gap underscores the need for empirical evaluation of co-executing pipelines end-to-end.

In this paper, our contributions are threefold: i) we present an empirical study of multi-model placement on heterogeneous CPU–NPU platforms and propose a score-based method that ranks candidate placements using full end-to-end pipeline measurements. ii) we design and implement a partitioning/compilation toolchain and a runtime system for placement exploration. iii) we evaluate the approach on a CPU with two asymmetric PCIe NPUs using ResNet50 and YOLOv3 workloads, demonstrating consistent gains under co-execution.

Our study targets single-host CPU–NPU systems with PCIe-attached accelerators, where unsupported operators or even entire models may run on the host CPU. Placement therefore determines not only which subgraphs use the NPU but also how much work remains on the host. We compare candidate placements with a score that balances throughput and dropped frames. In evaluation, the best-scoring placement remained stable across different drop-rate penalties.

Key insights from our study

- Per-model latencies did not reliably predict co-execution behavior. Placements derived from isolated profiles often failed under CPU and PCIe contention.
- The best placement shifted with the number of concurrently running models, indicating that model count is a critical factor.
- Across the tested input frequency ranges, the best-scoring placement was largely consistent. Although frequency affected absolute throughput, contention remained the dominant factor.

These findings indicate that empirical end-to-end evaluation provides more accurate guidance for placement than isolated profiling, particularly as workloads increase in scale and complexity.

II. BACKGROUND AND MOTIVATION

A. Characteristics of NPUs

NPUs are designed for energy-efficient execution of operators such as convolutions and matrix multiplications. Unlike CPUs or GPUs, they support only a limited set of DNN operators, have restricted memory, and connect to the host via PCIe. As a result, models must be partitioned into NPU-supported and CPU subgraphs, NPU subgraphs need to be precompiled and preloaded, and PCIe transfers and initialization can dominate end-to-end runtime—especially with asymmetric links. An isolated model may run faster on the CPU (e.g., ResNet50 in Table II), but under co-execution NPUs reduce CPU contention and enable pipeline parallelism, improving overall throughput.

B. Dynamic Input Rates

In practice, input rates vary over time. Such changes primarily stress CPU-side stages and queues, increasing host overhead and drop risk. Depending on buffering and PCIe feed, NPUs may be intermittently underutilized when the CPU bottlenecks, or they may remain saturated. Conversely, temporary decreases in input rate can leave devices underutilized or idle. A rate change does not by itself imply a different placement is needed. Redeployment should follow end-to-end behavior relative to performance objectives (e.g., throughput and drops).

C. Multi-Model Execution Challenges

Co-running models compete for shared host resources such as CPU cores, last-level cache, memory bandwidth, and runtime services. Delays that build up in one pipeline can propagate to others through these shared resources, increasing the risk of frame drops. For partitioned models, CPU–NPU boundaries add format conversion on the host and tensor transfers over PCIe and DMA, which raise latency and consume bandwidth. As a result, co-execution throughput departs from single-model predictions, and the bottleneck can shift among CPU, PCIe, and NPU as the number or size of models and their input rates change. Single-model profiles fail to capture co-execution effects. Placement should be validated under co-execution with end-to-end metrics that capture throughput and drops.

III. Problem Formulation

We consider a heterogeneous system with one CPU and multiple NPUs. Let $\mathcal{M} = \{m_1, m_2, \dots, m_K\}$ denote the set of models in a workload. A placement $p \in \mathcal{P}$ maps each model $m \in \mathcal{M}$ to an execution domain (CPU, CPU+NPU0, or CPU+NPU1). For a placement p observed over a window of length T seconds, we define:

- Total throughput: $\text{TotalFPS}(p)$, the average number of processed frames per second (FPS; aggregated across models).
- Drop rate: $\text{DropRate}(p) = \text{Dropped}(p; T)/T$, the average number of frames dropped per second.

Our objective is to select the placement p that maximizes a throughput–stability score for the current input rates and active model set:

$$\max_{p \in \mathcal{P}} \text{TotalFPS}(p) - \lambda \cdot \text{DropRate}(p), \tag{1}$$

where λ controls the drop-rate penalty (default λ=0.2). Each score is computed from short co-execution measurements at the current input rates.

IV. Proposed Approach

Our approach evaluates the entire application pipeline, from sensor I/O to postprocessing and rendering, under realistic input rates to capture CPU contention and PCIe asymmetry that isolated profiling misses. Figure 1 shows the three-stage workflow.

A. Stage I: Partitioning & Compilation (Preparation)

Target device configuration. We enumerate the available devices and assign stable identifiers. Our prototype system has one CPU and two NPUs attached via PCIe slots with asymmetric bandwidth. PCIe transfer and initialization costs are captured directly by the end-to-end pipeline measurements used for placement evaluation.

Model partitioning and NPU Compilation Toolchain. Input models are provided in ONNX format, with offline UINT8 quantization applied only to the subgraphs executed on the NPU. Regions intended for the NPU are expressed with quantized operators. The graph includes `QuantizeLinear` and `DequantizeLinear` nodes that mark CPU–NPU boundaries. These conversions run on the host, and no runtime calibration is applied.

Supported operators are fused into the largest contiguous NPU subgraph and compiled into device-specific binaries, while unsupported operators remain on the CPU. Figure 2 shows the flow from the ONNX model to a CPU subgraph and a pre-quantized NPU subgraph, followed by device-specific compilation. Partitioning is driven by a *Partition Specification* that records operator-to-device mappings. The compiler currently generates NPU code for `QLinearConv`, `QLinearAdd`, `QLinearLeakyRelu`, `QLinearRescale`, `QLinearConcat`, `MaxPool`, `Resize`, and `AntaraFusedConv`, a custom convolution operator for our NPU.

B. Stage II: Placement Generation & Evaluation (Search)

Placement Plan Generator. Given the device configuration and the compiled partitions, we enumerate feasible mappings from model partitions to devices. If a model runs entirely on the CPU, we regard it as a single partition.

Reloading models on PCIe-attached NPUs incurs significant overhead as shown in the loading times of Table II. Therefore, each NPU is restricted to one preloaded partition per deployment. We also prioritize the largest contiguous NPU-supported subgraph to reduce CPU–NPU boundaries and transfers. Our placement policy is subject to two constraints:

- The CPU may execute multiple partitions concurrently.
- Each NPU executes exactly one preloaded partition.

These constraints are enforced in all experiments, and in every reported result table each of CPU+NPU0 and CPU+NPU1 hosts exactly one model per deployment.

Placement Selector. The Placement Selector receives the set of candidate placements from the Placement Plan Generator. It sends each candidate to the Runtime Deployer, which installs the mapping and runs a short trial at the current input rates. During each trial, the Runtime Monitor reports throughput,

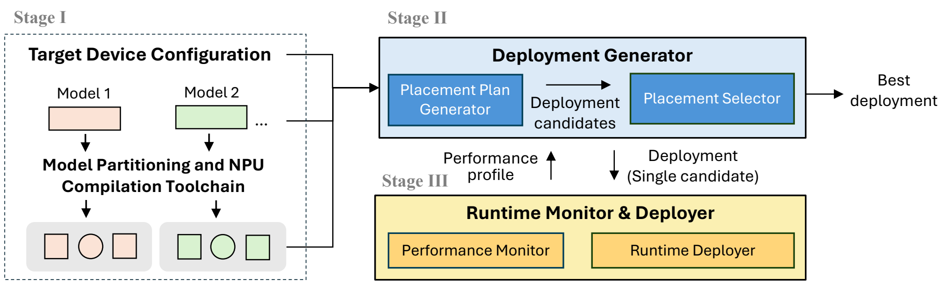

Fig. 1. Architecture overview with three stages: (I) preparation, (II) search, (III) control.

TABLE I
On-disk binary sizes of benchmark models. CPU models are FP32. NPU partitions are statically quantized to UINT8.

Single Model	Size (Mbytes) CPU(fp32)	Partitioned Model Variant	Size (Mbytes) CPU (fp32) + NPU (uint8)
ResNet50	102.6	**ResNet50-small (R50-s)**	0.2 + 28.3
		ResNet50-big (R50-b)	0.2 + 38.4
YOLOv3	247.9	**YOLOv3-small (Y3-s)**	0.6 + 151.1
		YOLOv3-big (Y3-b)	0.6 + 166.8

TABLE II
Measured CPU and NPU loading and inference times for model partitions.

Single Model	Inference Time (ms)	Partitioned	Loading Time (ms)		Inference Time (ms)	
Model	**CPU**	**Model Variant**	**NPU0**	**NPU1**	**CPU+NPU0**	**CPU+NPU1**
R50	12.6	**R50-s**	15.8	38.8	0 + 39.0	0 + 53.1
		R50-b	38.6	77.1	0 + 42.8	0 + 57.2
Y3	164.9	**Y3-s**	87.4	110.4	15.2 + 60.9	15.2 + 83.1
		Y3-b	107.0	467.5	15.2 + 88.2	15.2 + 96.8

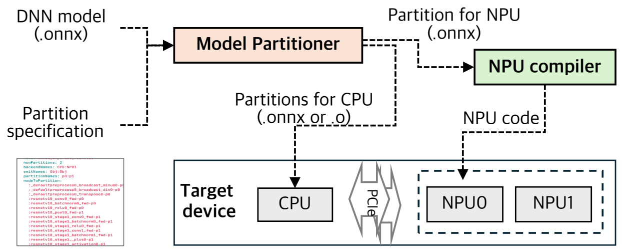

Fig. 2. Partition and compilation pipeline from ONNX model to CPU subgraph and pre-quantized NPU subgraphs.

drop rate, and transfer costs. The selector computes the score defined in Eq. (1) from these measurements, and after all candidates have been tried, it chooses the placement with the highest score as the best.

C. Stage III: Runtime Monitoring & Deployment (Control)

Performance Monitor. The monitor has two roles. First, it executes candidate deployments end-to-end to obtain total throughput and drop rate, as well as per-model throughput and drop statistics. Second, it continuously collects moving averages of FPS and drop rate, and records auxiliary statistics (e.g., queueing delays, device utilization, PCIe overheads).

Runtime Deployer. The deployer receives the placement plan selected by the Placement Selector from Stage II and redeploys the system. It first stops the current deployment and unloads active partitions. It then loads the specified CPU or NPU partitions, initializes the required binaries and bindings, and restarts execution under the new mapping.

V. Experimental Evaluation

A. Experimental Setup

Hardware Platform. We evaluate on a desktop system with an Intel Core i7-12700 (12 cores, 20 threads), 64 GB DDR4 memory, and two PCIe-attached NPUs. The PCIe links are asymmetric:

- NPU0 operates at PCIe Gen4 $\times 8$ (higher bandwidth).
- NPU1 operates at PCIe Gen2 $\times 1$ (lower bandwidth).

Each NPU integrates an embedded U74 RISC-V core. Operators not supported by the NPU run on the host CPU.
Benchmark Models. All NPU runs use the pre-quantized (UINT8) models prepared in Stage I. Table I lists the base models and their partitioned variants. We use R50 (ResNet50) with R50-s and R50-b, and Y3 (YOLOv3) with Y3-s and Y3-b. The “s” and “b” variants differ only in compilation options that change binary size and memory layout.

TABLE III
BASELINE PLACEMENTS FROM MAKESPAN MINIMIZATION USING SINGLE-MODEL LATENCIES.

	Deployment		
# Models	**CPU-only**	**CPU + NPU0**	**CPU + NPU1**
2	R50	Y3-s	-
3	R50	Y3-b	Y3-s
4	R50, R50	Y3-b	Y3-s

Evaluation Methodology. To evaluate multi-model placements, we execute the entire application pipeline, which includes video or image file decoding, preprocessing, inference, postprocessing, and display update, for each input-frequency setting. We measure the total throughput (TotalFPS) and the number of dropped frames, and use these to compute the drop rate (DropRate).

In our workload setup, the input arrival rates are systematically varied. For R50 and its variants, the input rate is increased from 2 to 10 images per second in steps of 2. For Y3 and its variants, the input video stream rate is increased from 20 to 60 frames per second in steps of 10. Each input-frequency configuration is executed for 30 s to obtain stable estimates of throughput and drops, from which we compute the score in Eq. (1).

When determining placements based on the score in Eq. (1), we run a full model on the CPU if none of its partitioned variants are assigned to an NPU. In particular, if neither Y3-s nor Y3-b is placed on an NPU, both are executed as the full Y3 model on the CPU; likewise, if neither R50-s nor R50-b is mapped to an NPU, both are executed as the full R50 model.

B. Results and Analysis

We evaluate workloads with two, three, and four models. For each case, we compare the placement predicted from single-model profiling with the best placement measured under co-execution. Tables present representative results. The columns labeled CPU, CPU+NPU0, and CPU+NPU1 show the models executed entirely on the CPU, jointly on the CPU with NPU0, and jointly on the CPU with NPU1, respectively.

Single-Model Profiling (Makespan Minimization). As our baseline, we adopt a simple makespan heuristic. We define a placement as a mapping from models to execution domains:

$$p : \mathcal{M} \to \mathcal{E}, \quad \mathcal{E} = \{\text{CPU, CPU+NPU0, CPU+NPU1}\}.$$

If a model m runs entirely on the CPU, its latency is $T_{m,\mathrm{CPU}}$. If it is partitioned across the CPU and an NPU, we use $T_{m,\mathrm{CPU+NPU0}}$ or $T_{m,\mathrm{CPU+NPU1}}$, depending on the target domain. Since each NPU can host only one preloaded partition per deployment, a placement may assign at most one model to CPU+NPU0 and to CPU+NPU1. For a placement p, we define the CPU critical-path time as

$$L_{\mathrm{CPU}}(p) = \max_{m:\, p(m)=\mathrm{CPU}} T_{m,\mathrm{CPU}}. \tag{2}$$

The heuristic then selects the placement that minimizes the longest among (i) the slowest CPU-only model (if any), (ii) the single CPU+NPU0 model (if any), and (iii) the single CPU+NPU1 model (if any):

$$p^{\star} = \arg\min_{p\in\mathcal{P}} \max\Big\{ L_{\mathrm{CPU}}(p),\ T_{m,\mathrm{CPU+NPU0}},\ T_{m,\mathrm{CPU+NPU1}} \Big\}. \tag{3}$$

All latency values are taken from Table II. Partitioned inference times are shown as the sum of CPU-side work and device execution (e.g., 15.2 + 60.9 ms for Y3-s on CPU+NPU0). This heuristic yields placements that appear balanced when models are profiled in isolation. However, as shown in later sections, these baseline placements often mispredict the best mapping once models co-execute. Scores degrade quickly as input rates increase and frame drops accumulate, showing that latencies measured in isolation cannot capture performance when multiple models run together.

TABLE IV
TWO-MODEL BEST-SCORING MEASURED PLACEMENT PER INPUT-FREQUENCY SCHEDULE (λ=0.2). DROP COUNTS ARE MEASURED OVER $T = 30$ s.

Input Freq.	FPS	# Drops	Score	Deployment		
				CPU-only	CPU+NPU0	CPU+NPU1
(2,20)	37.19	179	36.00	-	Y3-s	R50-s
(4,30)	37.14	337	34.89	-	Y3-s	R50-s
(6,40)	37.14	515	33.71	-	Y3-s	R50-s
(8,50)	37.28	521	33.81	-	Y3-s	R50-s
(10,60)	36.91	522	33.43	-	Y3-s	R50-s

Two-Model Results. Table IV reports results for the two-model workload composed of Y3-s and R50-s. In these experiments, the input frequency of R50-s was varied from 2 to 10 images per second in steps of 2, and the input frequency of Y3-s was varied from 20 to 60 frames per second in steps of 10.

With the default λ=0.2, the consistently optimal mapping executes Y3-s on the higher-bandwidth CPU+NPU0 domain and R50-s on the lower-bandwidth CPU+NPU1 domain, with no model running solely on the CPU. This configuration sustains roughly 36 FPS overall (36.91–37.28) and achieves scores between 33.43 and 36.00 as the input increases from (2,20) to (10,60).

For comparison, the makespan baseline from single-model profiling in Table III executes R50 on the CPU and Y3-s on CPU+NPU0. The placement appears balanced in isolation, yet the score drops as input grows. Figure 3 (a) shows a decline from 32.79 to 23.14. The CPU becomes the bottleneck as it executes the entire R50 and also handles decoding, preprocessing, quantization and dequantization, unsupported operators for Y3-s, YOLO postprocessing, display updates, and the PCIe and driver tasks for NPU0. As the input rate increases, cores and the last-level cache saturate and queues lengthen. Because the input arrival rate exceeds the reduced CPU processing throughput, frames waiting in the host queues are dropped, and NPU0 is intermittently underutilized.

Three-Model Results. Table V reports results for the three-model workload composed of Y3-b, Y3-s, and R50-s. In these experiments, the input frequency of Y3-b was varied from 20

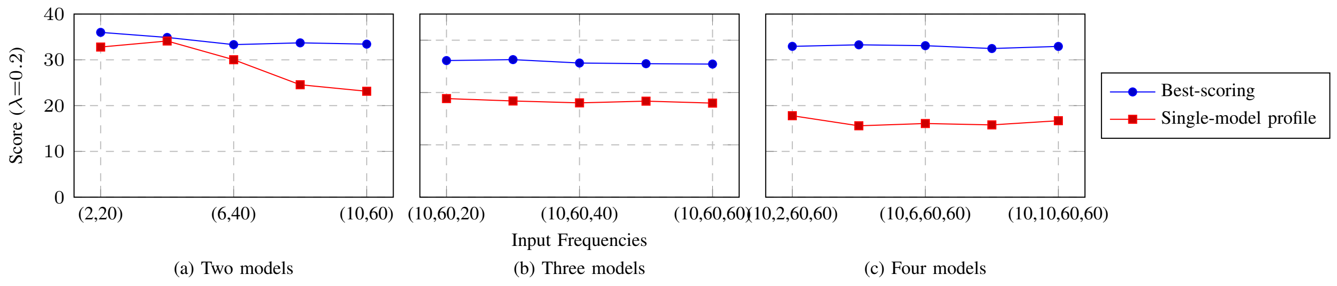

Fig. 3. Score (λ=0.2) under different input-rate schedules. (a) Two-model, (b) three-model, and (c) four-model workloads.

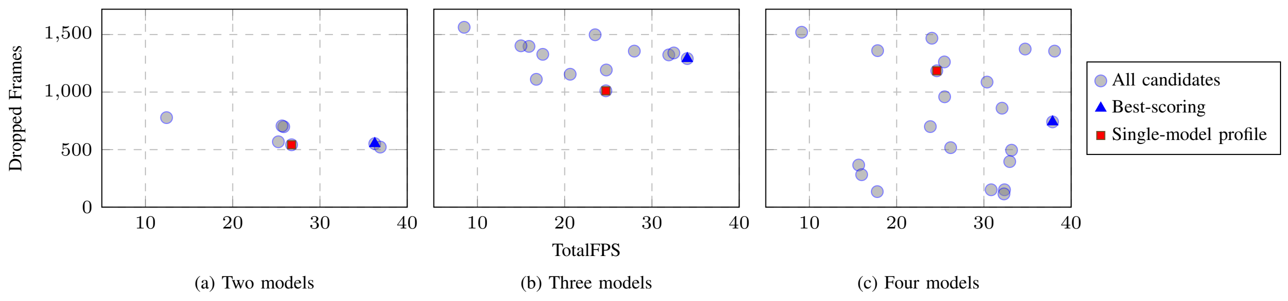

Fig. 4. Pareto plots of total FPS versus dropped frames at the highest input-rate setting. (a) Two-model, (b) three-model, and (c) four-model workloads.

TABLE V
THREE-MODEL BEST-SCORING MEASURED PLACEMENT PER INPUT-FREQUENCY SCHEDULE (λ=0.2). DROP COUNTS ARE MEASURED OVER $T = 30$ s.

Input Freq.	FPS	# Drops	Score	Deployment		
				CPU-only	CPU+NPU0	CPU+NPU1
(10,60,20)	34.42	1247	26.11	Y3	R50-s	Y3-s
(10,60,30)	34.38	1214	26.29	Y3	Y3-s	R50-s
(10,60,40)	34.01	1255	25.64	Y3	Y3-s	R50-s
(10,60,50)	34.09	1287	25.51	Y3	Y3-s	R50-s
(10,60,60)	34.05	1291	25.44	Y3	Y3-s	R50-s

TABLE VI
FOUR-MODEL BEST-SCORING MEASURED PLACEMENT PER INPUT-FREQUENCY SCHEDULE (λ=0.2). DROP COUNTS ARE MEASURED OVER $T = 30$ s.

Input Freq.	FPS	# Drops	Score	Deployment		
				CPU-only	CPU+NPU0	CPU+NPU1
(10,60,60,2)	33.83	133	32.94	R50, Y3	Y3-s	R50-b
(10,60,60,4)	34.09	122	33.28	R50, Y3	Y3-s	R50-b
(10,60,60,6)	38.63	831	33.09	Y3, Y3	R50-b	R50-s
(10,60,60,8)	33.28	122	32.47	R50, Y3	Y3-s	R50-b
(10,60,60,10)	37.87	741	32.93	Y3, Y3	R50-b	R50-s

to 60 frames per second in steps of 10 while keeping the other models fixed.

At the lowest setting $(10, 60, 20)$ the best-scoring placement assigns Y3-b to the CPU (executed as the full Y3), R50-s to the higher-bandwidth CPU+NPU0 domain, and Y3-s to the lower-bandwidth CPU+NPU1 domain. For all higher input frequencies the mapping flips. Y3-b remains on the CPU, while Y3-s is placed on CPU+NPU0 and R50-s on CPU+NPU1. Across these settings the measured placements sustain about 34 FPS overall (34.01–34.42) with scores between 25.44 and 26.29 as the inputs increase from (10,60,20) to (10,60,60).

The baseline placement derived from single-model profiling in Table III assigns R50 to the CPU, Y3-b to CPU+NPU0, and Y3-s to CPU+NPU1. Although this appears balanced in isolation it achieves only about 25 FPS in practice and yields lower scores (17.98–18.84), as shown in Figure 3 (b). The measured placements maintain higher scores and steadier throughput while the baseline degrades as the input rate for Y3-b on the CPU increases.

Four-Model Results. Table VI reports results for the four-model workload composed of Y3-b, Y3-s, R50-b, and R50-s. In these experiments, the input frequency of R50-b was varied in steps of 2 to evaluate performance sensitivity.

Across all input-frequency settings, the best-scoring placements keep Y3-b on the CPU (executed as the full Y3) and use the NPUs in one of two patterns that match Table VI: (i) Y3-s on CPU+NPU0 and R50-b on CPU+NPU1 (rows for 2, 4, and 8), or (ii) R50-b on CPU+NPU0 and R50-s on CPU+NPU1 (rows for 6 and 10). This configuration sustains about 33–39 FPS overall (33.28–38.63) with scores between 32.47 and 33.28.

The baseline placement derived from single-model profiling in Table III assigns two R50 models to the CPU, Y3-b to CPU+NPU0, and Y3-s to CPU+NPU1. Although this looks balanced in isolation, it achieves only about 25 FPS in practice and suffers from frequent frame drops, resulting in scores around 15.6–17.8 as shown in Figure 3 (c) and Figure 4 (c). At the input-frequency setting $(10, 60, 60, 6)$ the best-scoring

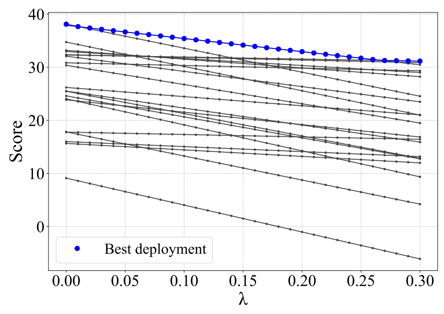

Fig. 5. Score vs. λ for the four-model workload at the highest input-rate setting. Each line corresponds to one candidate placement.

placement achieves a score of 33.09, compared to the baseline score of 15.59, an improvement of $+113.5\%$. The measured placements consistently achieve higher throughput and fewer frame drops, whereas the baseline degrades as the input load increases.

Pareto Analysis of Candidate Placements. Figure 4 plots total throughput versus dropped frames at the highest input rate. Best-scoring placements lie near the lower-right frontier whereas single-model baselines sit farther inside and indicate inferior trade-offs. As the number of models increases from two to four, the candidate placements spread out more widely and the Pareto frontier becomes steeper. This indicates that performance differences between placements grow larger, and that even small gains in throughput require sacrificing many more frames.

In the three-model case, the best placements offer higher throughput than the profiling baselines but at the cost of slightly more frame drops. In contrast, in the four-model case they deliver both higher throughput and comparable or lower drop levels, representing a clear improvement.

Overall, the Pareto view complements the per-schedule tables by showing that runtime evaluation consistently identifies near-frontier mappings, while placements derived from isolated profiles remain far from optimal under co-execution.

Sensitivity to λ (Drop-Rate Penalty). As in the Pareto analysis, we conduct the λ–sensitivity study at the highest input-rate setting. We vary λ from 0 to 0.30 in increments of 0.01 and recompute the score in Eq. (1) for all candidate deployments. Each curve in Fig. 5 corresponds to a single placement candidate and the blue circles mark the candidate with the highest score for each λ.

For the two-model workload the best-scoring deployment remains the same across the entire sweep. The same holds for the three-model workload where the optimal mapping does not change with λ, confirming strong stability. For the four-model workload, the optimal deployment also remains unchanged, but the score margins between the top candidates shrink as λ increases, as illustrated in Fig. 5. Overall, this indicates that while absolute scores vary with λ, the ranking of high-quality placements is robust, with noticeable sensitivity only in the four-model case.

Takeaways. Single-model profiles are poor proxies under co-execution, because host CPU work and PCIe transfers dominate performance. The best-scoring placement also changes with the number of models that run at the same time. Finally, although throughput rises or falls as input rates vary, the same placements tend to remain best. Short end-to-end measurements are sufficient to identify near-Pareto mappings.

VI. Related Work

Runtime-Aware Scheduling. Systems such as *InferLine* [8] and *Clockwork* [9] focus on predictive scaling and fine-grained scheduling to provide latency and service-level guarantees. These frameworks are effective in cloud and GPU environments with homogeneous accelerators, but they do not consider the CPU contention, host-side stages, or PCIe asymmetry that arise on CPU–NPU platforms.

Multi-Model Placement. Frameworks such as *FlexFlow* [6], *TVM Ansor* [7], and *PartitionTuner* [10] explore large partition spaces using simulators or cost-model-based search to optimize throughput. Other work such as *Band* [11] studies multi-model coordination on mobile processors with an emphasis on energy efficiency. These approaches highlight the importance of placement decisions, yet most rely on isolated profiling or assume predictable workloads rather than empirically measuring co-execution effects on host–device systems.

Dynamic Load Balancing and Adaptive Serving. Systems such as *NestDNN* [12] and *INFaaS* [13] dynamically adjust model variants to meet quality-of-service targets. GPU resource managers such as *Gandiva* [14] and *Salus* [15] enable fine-grained sharing and migration to handle dynamic loads. While these studies demonstrate the utility of runtime adaptation, they do not capture CPU involvement, asymmetric PCIe links, or full pipeline overheads present in CPU–NPU execution.

In contrast, our study empirically analyzes CPU–NPU platforms with asymmetric PCIe links. We evaluate full pipelines under co-execution and rank placements with a score that balances throughput and dropped frames, providing an end-to-end perspective that complements prior cost-model and cloud-focused methods.

VII. Conclusion

We presented an empirical study of multi-model placement on heterogeneous CPU–NPU platforms with asymmetric PCIe links. By measuring full end-to-end pipelines and ranking placements with a score that balances throughput and dropped frames, our approach consistently outperformed single-model profiling baselines on ResNet50 and YOLOv3 workloads. The results show that isolated latencies do not predict co-execution, the best placement shifts with the number of active models, and short measurement trials yield stable rankings even under varying input rates. Future work should extend the study to broader model classes and newer interconnects, explore multi-objective score functions that include latency, fairness, and energy, and develop scalable search or learning-based predictors to handle larger model sets and device counts.

References

[1] "Qualcomm Cloud AI 100 Product Brief," Product brief (PDF), Qualcomm Technologies, Inc., hHHL PCIe accelerator card; accessed: 2025-08-30. [Online]. Available: https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/QualcommCloudAI100_ProductBrief_200608.pdf

[2] "Xplorer x1600p pcie ai accelerator," Product page, Blaize, low-profile PCIe add-in accelerator; accessed: 2025-08-30. [Online]. Available: https://www.blaize.com/products/xplorer-x1600p/

[3] "Mini pcie accelerator — datasheet," Datasheet (PDF), Coral, edge TPU module in Mini PCIe form factor; accessed: 2025-08-30. [Online]. Available: https://coral.ai/docs/_downloads/2fcd85fc2ad6766d8bf494c5b956a75a/accelerator-mpcie-datasheet.pdf

[4] "Hailo unveils hailo-8 century PCIe ai accelerator card line," Newsroom post, Hailo. [Online]. Available: https://hailo.ai/company-overview/newsroom/news/hailo-ai-hailo-8l-entry-level-ai-accelerator-announcement/

[5] M. Han, J. Hyun, S. Park, J. Park, and W. Baek, "Mosaic: Heterogeneity-, communication-, and constraint-aware model slicing and execution for accurate and efficient inference," in *Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2019, pp. 165–177.

[6] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks," in *Proceedings of Machine Learning and Systems*, vol. 1, 2019.

[7] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J. E. Gonzalez, and I. Stoica, "Ansor: Generating high-performance tensor programs for deep learning," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 863–879.

[8] D. Crankshaw, G.-E. Sela, C. Zumar, X. Mo, J. E. Gonzalez, I. Stoica, and A. Tumanov, "Inferline: Latency-aware provisioning and scaling for prediction serving pipelines," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. ACM, 2020, pp. 477–491.

[9] A. Gujarati, R. Karimi *et al.*, "Serving dnns like clockwork: Performance predictability from the bottom up," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[10] M. Yu, Y. Kwon, J. Lee, J. Park, J. Park, and T. Kim, "Partitiontuner: An operator scheduler for deep-learning compilers supporting multiple heterogeneous processing units," *ETRI Journal*, vol. 45, no. 2, pp. 318–328, 2023.

[11] J. S. Jeong, J. Lee, D. Kim, C. Jeon, C. Jeong, Y. Lee, and B.-G. Chun, "Band: Coordinated multi-dnn inference on heterogeneous mobile processors," in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services (MobiSys)*, 2022, pp. 235–247.

[12] B. Fang, X. Zeng, and M. Zhang, "Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision," in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*. ACM, 2018, pp. 115–127.

[13] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "Infaas: Automated model-less inference serving," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 397–411.

[14] W. Xiao, R. Fonseca, G. Ananthanarayanan, J. Qian, Y. He, S. S. Bagaria, and A. Akella, "Gandiva: Introspective cluster scheduling for deep learning," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, 2018, pp. 595–610. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/zhang

[15] P. Yu and M. Zaharia, "Salus: Fine-grained gpu sharing primitives for deep learning applications," in *Proceedings of the ACM EuroSys Conference*. ACM, 2019.