

# Partition-Aware Scheduling for Mobile Heterogeneous Inference Co-Execution

Zhuojin Li<sup>a</sup>, Marco Paolieri<sup>a</sup>, Leana Golubchik<sup>a</sup>

<sup>a</sup>University of Southern California, Los Angeles, CA, 90089, USA

---

## Abstract

Modern mobile inference runs on heterogeneous platforms combining mobile GPUs with multiple CPU core clusters. Existing optimizations typically exploit either inter-operator parallelism, by assigning entire operators to CPU cores or to the GPU, or intra-operator parallelism, by partitioning each operator for CPU-GPU co-execution. We consider these two forms of parallelism together, to improve inference latency of tasks that can be represented by a static DAG of operators with predefined input/output tensor shapes (e.g., CNNs or vision transformers). We define the problem of *partition-aware DAG scheduling* for mobile heterogeneous inference, illustrating that the best strategy depends on the structure of the inference DAG, thus motivating a joint formulation capturing operator partition choices, device assignment, and execution order. We propose an online iterative search framework, which decomposes large DAGs into stages, focuses search on critical operators, and uses latency predictors to estimate partitioned execution without exhaustive profiling. Across representative mobile inference workloads, our approach achieves latency close to an offline solution while keeping scheduling overhead to a fraction of the model initialization cost, allowing platform-specific scheduling at deployment time.

*Keywords:* Mobile Inference, Scheduling, Workload Partitioning, Latency Optimization

---

## 1. Introduction

Deep learning has achieved significant breakthroughs across a wide range of applications, such as image understanding [1], speech recognition [2], and augmented reality [3]. As neural networks increasingly support interactive applications, there is a growing need to execute inference directly on mobile platforms. On-device inference improves offline availability, preserves user privacy by keeping data local, and enables low-latency responses without relying on cloud connectivity [4]. At the same time, state-of-the-art neural networks continue to exhibit increasing computational demands, requiring mobile platforms to incorporate more powerful compute resources for inference execution. A modern mobile system typically contains multiple computing resources, including heterogeneous CPU clusters and a mobile GPU. This hardware organization creates an important opportunity: rather than executing inference on a single accelerator, we can coordinate multiple compute resources at runtime to execute inference collaboratively.

For inference tasks where tensor shapes are fixed (e.g., convolutional neural networks and vision transformers), computation can be represented as a *static* directed acyclic graph (DAG), where each node is a tensor operator (e.g., convolution) performing one computation, and each edge represents a data dependency. This representation naturally leads to an optimization problem: assigning operators to heterogeneous devices and ordering their execution with respect to dependency constraints. Classical heterogeneous DAG schedulers such as HEFT [5] prioritize tasks and place each task on the device that gives the earliest finish time. Such schedule-only methods exploit *inter-operator parallelism*, where different operators can run concurrently on different devices (once their dependencies are satisfied).

However, DAG scheduling alone is limited in exploiting parallelism when the graph exposes little parallel structure. For instance, many convolutional neural networks (CNNs) contain long chain-like regions, where

---

*Email addresses:* zhuojin1@usc.edu (Zhuojin Li), paolieri@usc.edu (Marco Paolieri), leana@usc.edu (Leana Golubchik)

most latency-dominant operators depend on the immediately preceding operator such that only one operator at a time is ready for computation. In this case, even if the platform has idle compute resources, a schedule-only method cannot use them unless the operator itself can be partitioned. For example, on ResNet16 [6], HEFT takes 6.8 ms and leaves the CPU idle for a significant fraction of the time because the model exposes limited inter-operator parallelism and the schedule is constrained by the critical path (Section 3.2).

A complementary line of recent work uses CPU-GPU co-execution to split each operator across multiple devices [7, 8, 9]. This approach exploits *intra-operator parallelism*, where the workload of one operator is split into subtasks, and different devices compute different portions of the corresponding output. Mobile platforms are well-suited to this approach because the CPU and GPU *share unified memory*, allowing devices to access common tensor buffers without the overhead of extra copies; accordingly, in contrast to distributed computation on cloud GPUs [10], we do not model explicit CPU-GPU tensor-copy communication costs in this work because of our focus on mobile platforms with unified memory (see Section 6.6). Partition-only methods [9] typically make local decisions for each operator, i.e., choosing a workload split that minimizes the latency of that operator across compute units. For chain-like graphs, such operator partitioning can create useful parallelism even when the DAG exposes few parallel operators. In the ResNet16 example, this partition-only approach effectively utilizes the CPU resources and reduces latency to 4.9 ms.

However, partitioning every computationally expensive operator is not always beneficial. In branch-heavy networks, such as Inception [11] and HRNet [12], several operators may be ready at the same time. These operators can have different device affinity, meaning that some run more efficiently on the GPU while others run competitively on CPU clusters. In this setting, using all devices to accelerate one operator can delay another ready operator on the end-to-end critical path. Thus, an operator partition that is locally beneficial can be globally harmful. For example, on HRNet, partition-only execution takes 7.6 ms, which is worse than schedule-only HEFT at 6.7 ms, while jointly considering partitioning and scheduling reduces latency to 6.0 ms.

These observations motivate *partition-aware DAG scheduling*. The scheduler needs to decide not only where to run each operator (and in what order), but also whether an operator should be split, how its workload should be divided, which devices should participate in the split, and how the resulting subtasks should be ordered under DAG dependencies. This formulation generalizes both schedule-only and partition-only methods. The key insight is that partitioning and scheduling are coupled in this formulation, as partitioning determines the tasks exposed to the scheduler, while scheduling determines whether using a partition improves end-to-end latency.

However, solving this joint problem is challenging because each partitioning decision changes the scheduling decision. Even classical heterogeneous DAG scheduling is NP-hard [5]. Partition-aware scheduling adds another dimension of choices. For each partitionable operator, the scheduler should further decide how to partition the operator and how to execute the partitions across available devices. For a full DAG, these choices result in many possible task graphs expanded through node partitioning; each expanded graph must still be scheduled under device-availability and dependency constraints. A mixed-integer optimizer such as Gurobi [13] can solve small instances offline, but such solvers are unsuitable for deployment-time scheduling on mobile devices.

To make partition-aware DAG scheduling practical, we propose an iterative search framework designed for an online setting. The framework starts from a feasible schedule and progressively improves it by exploring partitioning decisions that are likely to reduce end-to-end latency. Three ideas make this search efficient. First, *staging* decomposes a large inference DAG into smaller sub-DAGs (or stages) and limits the search for the best schedule-partition to individual stages, thus reducing the optimization scope while preserving important dependencies at the boundary. Second, *criticality-aware sampling* focuses the search effort on operators that are likely to affect the current makespan, rather than spending equal effort on all nodes. Third, *latency prediction* estimates the cost of unseen partition plans, avoiding exhaustive on-device profiling for every candidate split. Together, these techniques reduce the search space enough for online use, while retaining the main benefits of joint partitioning and scheduling.

This paper makes the following contributions:

- We empirically characterize the limitations of schedule-only and partition-only methods for mobile

heterogeneous inference (Section 3). Chain-like models need intra-operator partitioning because they expose little DAG-level parallelism, while branch-heavy models require schedule-aware partitioning because locally beneficial splits can create global device contention. These results motivate joint decisions over partitioning and scheduling, rather than relying on either approach alone.

- We formulate partition-aware DAG scheduling as a unified optimization problem that captures operator partitioning, device assignment, and execution order (Section 4). This formulation provides a view of schedule-only and partition-only methods as restricted cases and gives an offline reference for evaluating practical algorithms. Based on this formulation, we develop and analyze practical baselines including expanded-DAG scheduling and partition-aware list scheduling, showing that simply exposing additional parallel subtasks is insufficient to obtain the best end-to-end execution plan.
- We design an iterative search framework for partition-aware scheduling (Section 5) that can be used online, at deployment time. To develop such an effective search for high-quality schedules, the framework uses staging to reduce scheduling scope, criticality-aware sampling to focus on operators that affect makespan, and latency prediction to evaluate unseen partition plans without exhaustive profiling.
- We comprehensively evaluate the approach across 18 inference workloads and 4 mobile platforms (Section 6). The iterative search achieves  $1.01\times$ – $1.04\times$  average normalized latency relative to Gurobi-based joint optimization (with a five-minute timeout for each stage) in a simulation setting and  $1.00\times$ – $1.04\times$  in real end-to-end measurements across devices. The total deployment-time overhead, including latency prediction and scheduling, is only 37.4% of model initialization time on average, justifying its practicality for on-device deployment.

## 2. Background and System Model

This section introduces the workload, hardware, and runtime assumptions used throughout the paper. We first model neural-network inference as a DAG of tensor operators and define the heterogeneous devices available on a mobile platform (Section 2.1). We then introduce workload partitioning, where a single operator is divided into subtasks that can execute collaboratively on multiple devices, and illustrate the partitioning strategies used for convolution operators (Section 2.2). Lastly, we discuss why our scheduler is designed for deployment-time use during model initialization (Section 2.3).

### 2.1. Mobile Heterogeneous Inference

A neural-network inference executes a trained model for a given input. The model consists of a graph of *operators*, where each operator performs one tensor computation, such as convolution, linear (matrix multiplication), or element-wise operators. The input and output of each operator are *tensors*, i.e., multi-dimensional arrays. We represent an inference computation as a directed acyclic graph (DAG)  $G = (V, E)$ . Each node  $i \in V$  denotes an operator, and each edge  $(i, j) \in E$  denotes a data dependency, indicating that operator  $j$  can execute only after the required output of operator  $i$  becomes available. In this work, we consider fixed-shape inference workloads with a static operator graph and batch size of 1 (i.e., latency-oriented); batching and dynamic graphs are briefly discussed in Section 6.6.

Modern mobile systems execute these workloads on heterogeneous compute resources. Following the scheduling literature, we refer to each schedulable compute resource as a *device*. Our formulation and scheduling algorithms are defined over a device set  $\mathcal{D}$ , but for ease of presentation and to match the mobile platforms used in our evaluation, we instantiate  $\mathcal{D}$  with three logical devices: GPU, CPU (L), and CPU (M). Here, CPU (L) and CPU (M) denote the large-core and medium-core CPU clusters, respectively, as detailed in Table 1. This abstraction follows the clustered organization of modern mobile CPUs, in which cores within the same cluster have similar frequency and performance characteristics, while different clusters can exhibit substantially different energy efficiency and throughput. Operators can exhibit different device affinity, meaning that an operator may execute faster on one device than another; this affinity depends on factors such as memory behavior, arithmetic intensity and device-specific operator implementation. For example,

| Platform                | Processor          | GPU        | CPU (L)              | CPU (M)                |
|-------------------------|--------------------|------------|----------------------|------------------------|
| OnePlus 11              | Snapdragon 8 Gen 2 | Adreno 740 | 1× 3.2 GHz Cortex-X3 | 2× 2.8 GHz Cortex-A715 |
| Motorola Edge Plus 2022 | Snapdragon 8 Gen 1 | Adreno 730 | 1× 3.0 GHz Cortex-X2 | 2× 2.5 GHz Cortex-A710 |
| Pixel 5                 | Snapdragon 765G    | Adreno 620 | 1× 2.4 GHz Kryo 475  | 1× 2.2 GHz Kryo 475    |
| Pixel 4                 | Snapdragon 855     | Adreno 640 | 1× 2.84 GHz Kryo 485 | 2× 2.42 GHz Kryo 485   |

Table 1: Mobile platforms used in the evaluation. CPU (L) and CPU (M) denote the large-core and medium-core CPU clusters exposed as logical scheduler devices in our experiments.

large convolutions often benefit from GPU parallelism, while small or memory-bound operators may run competitively on CPU cores because they do not fully utilize the GPU. Thus, the best device assignment can vary across operators and also depends on which other operators are ready to execute at the same time. This consideration is crucial to our problem, as a device useful for accelerating one operator may be even more valuable for executing another *ready operator* (i.e., an operator ready to execute) on the DAG critical path.

The inference DAG exposes two forms of parallelism. *Inter-operator parallelism* executes multiple ready operators concurrently after their dependencies have been satisfied; for example, two independent branches of a network can run on different devices in parallel. *Intra-operator parallelism* divides one operator into multiple subtasks and executes them on multiple devices collaboratively, and we refer to this division as *workload partitioning*; for example, a convolution can be partitioned so that the GPU computes part of the output tensor while a CPU cluster computes another part.

A key hardware property that makes this type of fine-grained co-execution practical on mobile platforms is *unified memory*, where the CPU and GPU share the same physical memory. Unlike systems with discrete GPUs, unified memory avoids large explicit data copies when CPU and GPU access the same tensors. Moreover, following prior work [9], our runtime uses OpenCL fine-grained shared virtual memory (SVM), which allows CPU and GPU to access shared allocations with hardware-supported cache coherence and avoids explicit data-mapping operations for maintaining coherence. Therefore, partitioned operators can *directly* read from and write to shared input and output tensors (using atomic operations). Accordingly, in our scheduling model, we do *not* include explicit CPU-GPU communication costs between operators.

Although unified memory removes the need to model explicit CPU-GPU communication costs, executing a schedule still requires lightweight coordination at dependency boundaries; e.g., a successor operator can start only after the required CPU and GPU work has completed. Our runtime handles this coordination by inserting small GPU-side synchronization *kernels* (i.e., small GPU programs) into the GPU command queue to update or wait on dependency progress shared with the CPU; the full runtime implementation is detailed in the Appendix (Section A). These kernels perform little computation, but launching GPU work still incurs overhead because GPU commands are submitted by the CPU and executed asynchronously. Accordingly, our scheduler accounts for GPU-side synchronization by adding a small constant to the GPU execution time; this constant is set to 10  $\mu$ s, based on the average overhead measured in our experiments.

## 2.2. Workload Partitioning Strategies

Workload partitioning splits one operator into multiple subtasks, and each subtask computes a portion of the original operator’s work and is assigned to one device. The goal is to reduce operator latency (particularly for operators that contribute substantially to overall latency) by using otherwise idle devices. In this paper, we focus on partitioning *latency-dominant* operators (e.g., convolution and linear), i.e., those operators that dominate the latency of many mobile CNN workloads [14]. Our framework is not limited to these operator types; in general, any operator can be treated as partitionable if its computation can be partitioned into subtasks. Other operators, such as pooling and element-wise operators, are typically shorter and are treated as indivisible scheduling units.

We use standard convolution to illustrate the available partitioning strategies. A linear layer can be handled similarly since it can be viewed as a convolution without a spatial window and its output features correspond to output channels. Following common mobile ML framework layouts such as TensorFlow Lite (TFLite) [15], we describe tensor shape using height, width, and channel dimensions; since we consider batch size of 1, we

omit the batch dimension. For a convolution, the input tensor has shape  $H_{\text{in}} \times W_{\text{in}} \times C_{\text{in}}$ , and the output tensor has shape  $H_{\text{out}} \times W_{\text{out}} \times C_{\text{out}}$ . The convolution uses a filter tensor  $W$  of shape  $K_h \times K_w \times C_{\text{in}} \times C_{\text{out}}$ , where each output channel has one corresponding  $K_h \times K_w \times C_{\text{in}}$  filter. Ignoring bias for simplicity, each output element is computed by applying a filter to a local spatial region of the input and accumulating the result across input channels:  $Y[h, w, c_o] = \sum_{r=0}^{K_h-1} \sum_{s=0}^{K_w-1} \sum_{c_i=0}^{C_{\text{in}}-1} X[h+r, w+s, c_i] \cdot W[r, s, c_i, c_o]$ , for  $0 \leq h < H_{\text{out}}, 0 \leq w < W_{\text{out}}, 0 \leq c_o < C_{\text{out}}$ . This formula reveals three natural ways to split the computation: output channels, spatial positions, and input channels.

*Output-Channel Partitioning.* Output-channel partitioning splits the  $C_{\text{out}}$  dimension. Each device computes a subset of output channels using the corresponding filters. Because different output channels are independent, devices compute disjoint regions of the output tensor. Under unified memory, these regions can be written directly into the shared output buffer using atomic operations, without additional copy or aggregation. Output-channel partitioning therefore provides a low-overhead partitioning strategy for convolution and linear operators.

*Spatial Partitioning.* Spatial partitioning splits the output spatial dimensions, such as the width dimension  $W_{\text{out}}$ . Each device computes a contiguous range of output positions. For  $1 \times 1$  convolutions, the split is naturally independent because each output position depends only on the corresponding input position. For larger kernels, however, output elements near a partition boundary require overlapping neighboring input values outside the assigned output range. Unified memory allows these boundary values to be read without explicit CPU-GPU copies.

*Input-Channel Partitioning.* Input-channel partitioning splits the  $C_{\text{in}}$  dimension. Unlike output-channel and spatial partitioning, this split does not directly produce independent final outputs. The devices therefore accumulate their partial sums into the shared output tensor using atomic updates. Under unified memory, the overhead of dispatching a separate aggregation kernel can be avoided by modifying the compute kernel to update the final output tensor directly.

### 2.3. Online Scheduling Budget

The previous subsections define the decisions available to the scheduler; it can exploit inter-operator parallelism by running ready operators on different devices, and intra-operator parallelism by partitioning one operator across multiple devices. However, exploring these decisions introduces scheduling overhead. In this paper, we use *online* scheduling to refer to scheduling determined during model initialization, rather than during an expensive offline tuning phase. Similar to model initialization, the scheduling decisions do not need to be recomputed before every inference request. For a fixed model and fixed input shape, the runtime can determine the execution plan once and reuse it for subsequent inferences. Thus, scheduling overhead is amortized across many requests. From this perspective, a scheduler does not need to be as fast as a single inference, but it should add only modest overhead relative to model initialization. For example, on OnePlus 11, the TFLite benchmark reports 1048 ms to initialize Inception-v3, including weight loading, tensor allocation, graph compilation, and device-specific preparation. A practical online scheduler should execute during this initialization. This requirement prevents the scheduler from exhaustively profiling every partition choice on the device, or solving a large global optimization problem with an offline solver.

## 3. Motivation: Why Partitioning and Scheduling Should Be Joint

As noted above, mobile DNN inference exposes two forms of parallelism: (1) the inference DAG can contain branches that can run concurrently on different devices, i.e., inter-operator parallelism; (2) a latency-dominant operator (e.g., convolution) can be partitioned into subtasks that run collaboratively on multiple devices, i.e., intra-operator parallelism. This section considers (empirically) three representative neural networks to illustrate that exploiting only one form of parallelism can be suboptimal (Sections 3.2 and 3.3), motivating the need to jointly optimize partitioning and scheduling (Section 3.4).

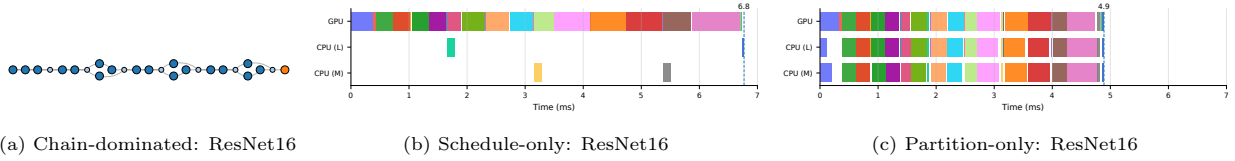


Figure 1: Chain-dominated ResNet16 on OnePlus 11 phone. Schedule-only execution leaves CPU clusters mostly idle because few operators are ready at the same time. Partitioning operators creates intra-operator parallelism.

### 3.1. Representative Model Structures

We study three representative models: ResNet16 [6], Inception-v3 [11], and HRNet [12]; they exhibit different degrees of intra- and inter-operator parallelism. ResNet16 is chain-dominated: as shown in Fig. 1a, its latency-dominant convolutions are primarily organized along a sequential critical path, and at most points in the execution, only one operator is ready to execute. Therefore, the main optimization opportunity is to reduce the latency of each critical operator by partitioning each operator across multiple devices. Inception-v3 has a block-structured DAG; as shown in Fig. 2a, each Inception block (in a shaded rectangular region) creates several parallel branches after a split point and later merges them through concatenation. This structure exposes more inter-operator parallelism than ResNet16, but the parallelism is still limited by block boundaries and by imbalance across branches. Thus, Inception-v3 can benefit from both branch-level scheduling and operator partitioning. HRNet is more branch-heavy; as shown in Fig. 2b, it maintains multiple parallel branches through a large portion of the network. This structure creates a wide middle region in which many operators can be ready at the same time. In this setting, aggressively partitioning every operator can be ineffective because devices used for one partitioned operator may be more suitable for other ready operators from parallel branches.

Figs. 1 and 2 also show execution timelines of different methods on three devices. Each colored block represents the execution interval of an operator or an operator partition (i.e., a subtask). In schedule-only execution, each operator is assigned to one device. In partition-only execution, every partitionable operator is split across all three devices; blocks with the same color correspond to partitions of the same original operator. In joint partition-aware scheduling (Figs. 2g and 2h), the scheduler decides both how each operator is partitioned and how the resulting subtasks should be placed (and ordered) in the DAG schedule.

### 3.2. Schedule-Only Execution Misses Intra-Operator Parallelism

Schedule-only execution treats each operator as an indivisible task. A representative method is HEFT [5], which first prioritizes operators based on an estimate of their remaining critical-path cost and then assigns each operator to the device that gives the earliest finish time (as detailed in Section 4.2). Such methods are effective when the DAG contains enough ready operators to keep multiple devices busy. However, they cannot create additional parallel work when the DAG exposes little inter-operator parallelism. This limitation is clear for chain-dominated ResNet16. As shown in Fig. 1b, HEFT takes 6.8 ms, where the GPU executes most latency-dominant operators, while CPU (L) and CPU (M) remain almost idle (with utilization below 4%); this is mainly because most operators depend on the immediately preceding operator, so there are few ready operators for the CPU clusters to execute. As a result, a schedule-only method cannot use idle devices unless there is another ready operator. On the other hand, partitioning addresses this limitation by creating parallel work inside each operator. As shown in Fig. 1c, partitioned execution splits latency-dominant operators across GPU and CPU clusters, which increases the utilization of all devices to more than 85% and reduces latency to 4.9 ms.

### 3.3. Partition-Only Execution Misses Inter-Operator Parallelism

Partition-only execution typically optimizes each partitionable operator locally. A common objective is to divide one operator across devices so that the participating devices finish their partitions at similar times [8, 9]. This local balancing can reduce the latency of an individual operator, particularly when some devices would otherwise be idle. Inception-v3 illustrates a case where partitioning remains useful. This model



Figure 2: Branch-structured models on OnePlus 11 phone. Inception-v3 has limited branch-level parallelism, so partitioning can use idle devices. HRNet exposes more ready operators, so partition-only execution can block useful inter-operator parallelism. Joint partitioning and scheduling balance these effects.

contains branch-level parallelism, so schedule-only execution can place independent branch operators on different devices, as shown in Fig. 2c. However, the branches inside each block are not always sufficient to fully occupy the slower CPU clusters; the resulting CPU utilizations are only 43% for CPU (L) and 52% for CPU (M). Instead, partition-only execution (Fig. 2e) can utilize these idle devices, reducing latency to 13.8 ms compared with 15.7 ms for schedule-only execution.

However, the local balancing objective can become ineffective when the DAG already exposes abundant inter-operator parallelism. For example, HRNet contains a wide middle region with multiple parallel branches, where many operators can be ready at the same time. Schedule-only execution can exploit this structure; much of the timeline in Fig. 2d is fully utilized within the 2.5–5 ms window, where independent branch operators occupy all the devices. On the other hand, partition-only execution (Fig. 2f) locally partitions each operator across all devices, ignoring whether the devices used for one operator partition could have been used more effectively by other ready operators. As a result, partition-only execution takes 7.6 ms, which is worse than the 6.7 ms achieved by schedule-only execution. Joint partition-aware scheduling avoids this problem by partitioning an operator only when its intra-operator speedup outweighs the benefit of using the same device for other ready operators. For HRNet, jointly optimizing partitioning and scheduling reduces latency to 6.0 ms, as shown in Fig. 2h.

### 3.4. Motivation for Joint Partition-Aware Scheduling

The examples above illustrate that the effectiveness of an execution strategy is closely tied to model topology. In chain-dominated regions, such as ResNet16, execution is primarily constrained by a sequential critical path, so reducing the latency of individual operators through intra-operator partitioning is highly effective. In branch-heavy regions, such as HRNet, many operators can be ready at the same time, and

these operators compete for heterogeneous devices; in this case, partitioning one operator across all devices can reduce its local latency while increasing the end-to-end makespan by delaying other ready operators. Therefore, the scheduler must evaluate each partitioning decision in the context of other ready operators in the graph.

These observations motivate *joint partition-aware DAG scheduling*. Instead of choosing between schedule-only and partition-only execution, the scheduler should jointly decide which operators to partition, how to divide their workload, which devices should execute the resulting subtasks, and when those subtasks should run. The guiding principle is to partition operators when device idleness limits performance, while taking advantage of inter-operator parallelism when ready operators can already keep the devices busy.

#### 4. Problem Formulation and Baselines

This section formalizes partition-aware DAG scheduling (Section 4.1) and defines the baseline methods used in our evaluation. The formulation decomposes the problem into two decisions: *partitioning*, which determines the partition strategy and the workload division for each operator, and *scheduling*, which assigns the resulting subtasks to devices and orders their execution while respecting DAG dependencies. From this perspective, existing approaches can be viewed as restricted versions of the joint problem: schedule-only methods (Section 4.2) optimize device placement and execution order but treat operators as indivisible, while partition-only methods (Section 4.3) optimize local operator partitioning but omit graph-level scheduling.

In addition to these existing baselines, we propose two simple partition-aware baselines to evaluate whether straightforward combinations of partitioning and scheduling are sufficient. The first, *expanded-DAG scheduling* (Section 4.4), fixes a partition plan for each operator, expands the original DAG into subtasks, and then applies a conventional DAG scheduler. The second, *partition-aware list scheduling* (Section 4.5), extends schedule-only list scheduling by greedily choosing a partition plan when each operator is scheduled. Finally, the full joint formulation provides an offline reference problem that can be solved by an optimizer (e.g., Gurobi) for small graphs; this reference allows us to evaluate how close practical methods are to state-of-the-art optimization with a long timeout (5 minutes in our experiments).

##### 4.1. Formulation: Partition-Aware DAG Scheduling

*DAG and Devices.* We model an inference computation as a directed acyclic graph (DAG)  $G = (V, E)$ , where each node  $i \in V$  is an operator and each directed edge  $(i, j) \in E$  denotes that operator  $j$  depends on the output of operator  $i$ . The target mobile platform contains a set of heterogeneous devices  $\mathcal{D}$ , including mobile GPU and CPU core clusters. Each device can execute at most one scheduled task or subtask at a time. Because the CPU and GPU share unified memory, we do not explicitly model data-copy communication cost between dependent operators.

*Partition Strategies.* Unlike traditional DAG scheduling, where each operator is treated as an indivisible task, our formulation allows a subset  $V_p \subseteq V$  of operators to be partitioned. In this paper, partitionable operators primarily include convolution and linear operators, which typically dominate end-to-end latency in mobile inference workloads [14]. For each partitionable operator  $i$ , we consider a set of candidate partition strategies  $A_i = \{\text{none}\} \cup A_i^{\text{part}}$ ,  $A_i^{\text{part}} \subseteq \{c_{\text{out}}, c_{\text{in}}, \text{spatial}\}$ . Here, **none** denotes unpartitioned execution,  $c_{\text{out}}$  denotes output-channel partitioning,  $c_{\text{in}}$  denotes input-channel partitioning, and **spatial** denotes spatial partitioning (as described in Section 2.2). For non-partitionable operators, we set  $A_i = \{\text{none}\}$ .

*Partition Plan and Scheduling Decision.* For a strategy  $a \in A_i$ , let  $U_{i,a} \in \mathbb{Z}_{>0}$  denote the (integer) workload size of operator  $i$  under that strategy; we model workload size as an integer to reflect actual partition strategies (see Section 2.2). For example,  $U_{i,c_{\text{out}}}$  is the number of output channels of operator  $i$  when output-channel partitioning is used. A specific partition plan for operator  $i$  is  $\pi_i = (a_i, \mathbf{u}_i)$ , where  $a_i \in A_i$  is the selected strategy, and  $\mathbf{u}_i = (u_{i1}, u_{i2}, \dots, u_{iK_i})$  denotes the workload division, satisfying  $u_{iq} \in \mathbb{Z}_{\geq 0}$  and  $\sum_{q=1}^{K_i} u_{iq} = U_{i,a_i}$ . Subtasks with  $u_{iq} = 0$  are inactive and are omitted from the resulting task graph and scheduling decisions. We allow zero-work entries so that the formulation supports partitioning into up to  $K_i$  subtasks. For each partitionable operator  $i$ ,  $\Pi_i$  represents the set of feasible partition plans, i.e., combinations of partition strategy and workload division into at most  $K_i = |\mathcal{D}|$  subtasks. For every operator  $i \in V$ , the

| Symbol                                 | Description                                                                                                  |
|----------------------------------------|--------------------------------------------------------------------------------------------------------------|
| $G = (V, E)$                           | Inference DAG with operators $V$ and dependencies $E$                                                        |
| $\text{pred}(i), \text{succ}(i)$       | Predecessor and successor sets of operator $i$                                                               |
| $\mathcal{D}$                          | Set of devices, e.g., GPU and CPU core clusters                                                              |
| $V_p$                                  | Set of partitionable operators                                                                               |
| $\mathcal{A}_i$                        | Candidate partition strategies for operator $i$                                                              |
| $a_i$                                  | Selected partition strategy of operator $i$ , e.g., none, output-channel, input-channel, or spatial          |
| $K_i$                                  | Maximum number of subtasks for operator $i$                                                                  |
| $U_{i,a}$                              | Integer workload size of operator $i$ under strategy $a$                                                     |
| $\mathbf{u}_i$                         | Workload division vector $(u_{i1}, \dots, u_{iK_i})$ , where $u_{iq}$ is workload size for subtask $(i, q)$  |
| $\pi_i$                                | Specific partition plan of operator $i$ , $\pi_i = (a_i, \mathbf{u}_i)$                                      |
| $\Pi_i$                                | Set of candidate partition plans for operator $i$                                                            |
| $\boldsymbol{\pi}$                     | Partition plans for all operators, $\boldsymbol{\pi} = \{\pi_i\}_{i \in V}$                                  |
| $\mathbf{d}^{\text{OSPD}}$             | One-subtask-per-device mapping used for local partition decisions                                            |
| $\mathbf{d}_i$                         | Device assignment vector $(d_{i1}, \dots, d_{iK_i})$ , where $d_{iq}$ is the assignment for subtask $(i, q)$ |
| $\hat{\tau}_{i,a,d}(u)$                | Estimated latency for workload $u$ on device $d$ under strategy $a$                                          |
| $s_{iq}, e_{iq}$                       | Start and end times of subtask $(i, q)$                                                                      |
| $\ell_{iq}$                            | Duration of subtask $(i, q)$ , $\ell_{iq} = e_{iq} - s_{iq}$                                                 |
| $C_i$                                  | Completion time of operator $i$                                                                              |
| $T$                                    | End-to-end makespan                                                                                          |
| $w_{i,d}$                              | Unpartitioned execution time of operator $i$ on device $d$                                                   |
| $\bar{w}_i$                            | Average unpartitioned execution time of operator $i$ across devices                                          |
| $S$                                    | A constructed schedule, including subtask placement and execution order                                      |
| $T(S)$                                 | Makespan of schedule $S$                                                                                     |
| $\text{rank}_u(i)$                     | HEFT upward rank of operator $i$                                                                             |
| $\text{EST}(\cdot), \text{EFT}(\cdot)$ | Earliest start time and earliest finish time used by list scheduling                                         |

Table 2: Notation used in the optimization model and baseline definitions.

unpartitioned strategy corresponds to the plan  $\pi_i^{\text{none}} = (\text{none}, (U_{i,\text{none}}, 0, \dots, 0))$ . For a non-partitionable operator  $i \notin V_p$ , we set  $\Pi_i = \{\pi_i^{\text{none}}\}$ . Given a specific partition plan  $\pi_i = (a_i, \mathbf{u}_i)$ , the scheduler determines device placement and the execution order of the resulting subtasks. Let  $\mathbf{d}_i = (d_{i1}, d_{i2}, \dots, d_{iK_i})$  denote the device assignment vector for operator  $i$ , where  $d_{iq} \in \mathcal{D}$  is the device assigned to subtask  $(i, q)$ . For a given operator, we require its active subtasks to be assigned to distinct devices; i.e.,  $d_{iq} \neq d_{ir}$  for  $q \neq r$  whenever  $u_{iq} > 0$  and  $u_{ir} > 0$ . The scheduler also orders execution on each device by assigning start and end times  $s_{iq}$  and  $e_{iq}$ , subject to device availability and DAG dependencies. As a result, the final makespan is determined by both the partition plans for all operators and the scheduling decisions that assign tasks to devices and order their execution.

*Execution Time.* For a specific partition plan  $\pi_i = (a_i, \mathbf{u}_i)$  and device assignment  $\mathbf{d}_i$ , the estimated latency of subtask  $q$  is denoted as  $\hat{\tau}_{i,a_i,d_{iq}}(u_{iq}) \geq 0$ . Notably, the estimate includes the device execution time under the selected strategy; for GPU subtasks, it also includes the small constant that accounts for synchronization kernel-dispatch overhead, as described in Section 2.1. The end time of a subtask is  $e_{iq} = s_{iq} + \hat{\tau}_{i,a_i,d_{iq}}(u_{iq})$ . The completion time of the entire operator  $i$  is determined by the slowest subtask  $C_i = \max_{q: u_{iq} > 0} e_{iq}$ . Our formulation enforces non-overlapping execution on each device; i.e., each device can execute at most one subtask at a time. Therefore, if two subtasks are assigned to the same device, their execution intervals cannot overlap:  $[s_{iq}, e_{iq}] \cap [s_{jr}, e_{jr}] = \emptyset$  if  $d_{iq} = d_{jr}$  and  $(i, q) \neq (j, r)$ .

*Full-Barrier Dependency.* We use a full-barrier dependency model, where a successor operator can start only after all subtasks of each predecessor have completed. For each edge  $(i, j) \in E$ , we enforce  $s_{jq} \geq C_i$  for  $q \in \{1, \dots, K_j\}$ . This captures the requirement that the successor operator consumes the full output of each predecessor.

*Number of Subtasks.* In this paper, we set  $K_i = |\mathcal{D}|$ , indicating that each operator creates at most one subtask per device. Under the full-barrier dependency model, creating more subtasks than available devices would require some subtasks of the same operator to execute serially on the same device, which rarely increases parallelism in our setting. We validate this assumption through an ablation study in Section 6.5.

*Objective.* The joint problem optimizes both partitioning plans  $\pi_i = (a_i, \mathbf{u}_i)$  and scheduling decisions  $(\mathbf{d}_i, s_{iq}, e_{iq})$ , with the objective of minimizing the end-to-end inference makespan:  $\min T$  s.t.  $T \geq C_i, \forall i \in V$ .

This formulation captures intra-operator parallelism through partitioning and inter-operator parallelism through scheduling. We implement this joint optimization problem in Gurobi, where partition-plan selection, device assignment, and non-overlap constraints are encoded using binary selection and ordering variables. Because the search space grows rapidly with the number of partitionable operators and candidate partition plans, the optimizer is used only as an offline baseline (with a five-minute solution timeout for each stage).

#### 4.2. Schedule-Only Heuristics

Schedule-only methods exploit inter-operator parallelism by assigning operators to different devices, but each operator remains indivisible. This approach corresponds to fixing the partition plan of every operator  $i$  to  $\pi_i^{\text{none}} = (\text{none}, (U_{i,\text{none}}, 0, \dots, 0))$ . The scheduler then chooses the device placement and execution order of operators. Under this restriction, the problem reduces to classical heterogeneous DAG scheduling.

We use HEFT [5] as a representative schedule-only heuristic. For an operator  $i$ , let  $w_{i,d} = \hat{\tau}_{i,\text{none},d}(U_{i,\text{none}})$  denote the execution time of operator  $i$  on device  $d$ . HEFT first computes an average execution time across devices  $\bar{w}_i = \frac{1}{|\mathcal{D}|} \sum_{d \in \mathcal{D}} w_{i,d}$ . The original HEFT formulation also includes an average communication cost between dependent tasks; we omit this communication term because CPU and GPU share unified memory. The upward rank is defined as  $\text{rank}_u(i) = \bar{w}_i + \max_{j \in \text{succ}(i)} \text{rank}_u(j)$ , where  $\text{succ}(i) = \{j : (i, j) \in E\}$  is the set of successors of operator  $i$  (we set the max to 0 when  $i$  has no successors). HEFT schedules operators in decreasing order of  $\text{rank}_u(i)$ . When operator  $i$  is selected, HEFT assigns it to the device that gives the earliest finish time:  $d_i^* = \arg \min_{d \in \mathcal{D}} \text{EFT}(i, d)$ , where  $\text{EFT}(i, d) = \text{EST}(i, d) + w_{i,d}$ . Here,  $\text{EST}(i, d)$  is the earliest time at which operator  $i$  can start on device  $d$ , considering both predecessor completion times and the current availability of device  $d$ . In summary, HEFT first prioritizes operators that are important to the remaining critical path, and then greedily assigns each operator to the device that achieves the earliest finish time.

We also evaluate several HEFT-like list-scheduling heuristics. Lookahead-HEFT [16] improves the greedy device selection by considering how assigning the current operator to a device affects its immediate successors. PEFT [17] uses an optimistic cost table to estimate the downstream scheduling cost, so that device selection accounts for future critical-path effects rather than only the current earliest finish time. LDCP [18] prioritizes operators using a longest dynamic critical-path criterion, so operators more likely to delay the final makespan are scheduled earlier. CEFT [19] uses constrained critical path information to guide operator prioritization and device selection. PSLS [20] uses a pre-scheduling phase to obtain global scheduling guidance before applying list scheduling. These methods improve different parts of the list-scheduling procedure, but they all keep operators indivisible. We also include a schedule-only Gurobi baseline for small DAGs. This baseline optimizes placement and ordering of indivisible operators, allowing us to separate the benefit of optimal graph-level scheduling.

#### 4.3. Partition-Only Co-Execution

Partition-only methods exploit intra-operator parallelism but do not perform graph-level scheduling. Existing work [8, 9] typically assumes a fixed co-execution device placement, which we refer to as the *one-subtask-per-device* mapping, abbreviated as OSPD. Under OSPD, each subtask of an operator is mapped to a different device:  $\mathbf{d}^{\text{OSPD}} = (d_1^{\text{OSPD}}, \dots, d_{|\mathcal{D}|}^{\text{OSPD}})$ , with  $\cup_{q=1}^{|\mathcal{D}|} \{d_q^{\text{OSPD}}\} = \mathcal{D}$ . Given this fixed device placement, partition-only co-execution chooses the partition plan that locally minimizes the overall latency of each partitionable operator:  $\pi_i^{\text{Local}} = \arg \min_{\pi_i=(a_i, \mathbf{u}_i) \in \Pi_i} \max_{q=1, \dots, K_i} \hat{\tau}_{i,a_i,d_q^{\text{OSPD}}}(u_{iq})$ . The graph then follows the original topological execution order without optimizing operator order or device placement using graph-level information.

A partition-only method can significantly accelerate heavy operators by dividing their work across CPU and GPU. However, this decision is made locally for each operator and does not consider whether keeping a device available would better serve other ready operators. For example, splitting the current operator across both CPU and GPU can minimize the operator latency locally, but it can also delay another ready operator that has stronger affinity to one of the devices. Therefore, partition-only co-execution can be effective for chain-dominated DAGs, but it can underutilize graph-level parallelism in branch-heavy DAGs.

#### 4.4. Proposed Baseline: Expanded-DAG Scheduling

Expanded-DAG scheduling is a simple attempt to combine partitioning with existing DAG schedulers. It decouples the problem into two stages. First, each partitionable operator is assigned a specific partition plan  $\pi_i = (a_i, \mathbf{u}_i)$ . Second, the original DAG is expanded into subtasks (based on the partition plan) preserving the full-barrier dependency. A conventional schedule-only heuristic or optimizer then schedules the expanded graph. We consider two fixed partition rules.

*Equal-Size Expansion.* The equal-size rule divides (integral size) workload evenly (based on the partitioning strategy used) across subtasks, resulting in an equal amount of work per subtask. For each strategy  $a \in \mathcal{A}_i$ , the workload division follows  $u_{iq}^{\text{equal}}(a) \approx \frac{U_{i,a}}{K_i}$ . We then choose the strategy whose equal-size partition has the smallest latency under the OSPD placement:  $a_i^{\text{equal}} = \arg \min_{a \in \mathcal{A}_i} \left( \max_q \hat{\tau}_{i,a,d_q^{\text{OSPD}}} \left( u_{iq}^{\text{equal}}(a) \right) \right)$ . The resulting fixed partition plan is  $\pi_i^{\text{equal}} = \left( a_i^{\text{equal}}, \mathbf{u}_i^{\text{equal}}(a_i^{\text{equal}}) \right)$ .

*Min-Local-Latency Expansion.* Unlike equal-size expansion, which fixes an equal amount of work distributed across devices, this rule searches over workload divisions to choose the one that minimizes the latency of the operator locally. The min-local-latency rule chooses the partition plan that locally minimizes the latency of operator  $i$  under the OSPD placement:  $\pi_i^{\text{Local}} = \arg \min_{\pi_i=(a_i, \mathbf{u}_i) \in \Pi_i} \max_{q=1, \dots, K_i} \hat{\tau}_{i,a_i,d_q^{\text{OSPD}}}(u_{iq})$ . This rule follows the same local objective as partition-only co-execution.

Both equal-size expansion and min-local-latency expansion fix the partition plan of each operator; graph-level scheduling is performed after the expanded graph is constructed. Expanded-DAG scheduling evaluates candidate solutions using a decoupled design, where partitioning is decided locally first, and scheduling is performed afterward. However, because the partition plans are fixed before graph-level scheduling, they do not account for how devices will be assigned to other ready operators. As a result, a partition that is locally beneficial for one operator may still be globally suboptimal.

#### 4.5. Proposed Baseline: Partition-Aware List Scheduling

Partition-aware list scheduling incorporates partitioning decisions during scheduling. Unlike expanded-DAG scheduling, it does not fix all partition plans before scheduling begins. Instead, when an operator is selected by the list scheduler, the heuristic greedily chooses both a partition plan and a device placement for that operator. We present a partition-aware list scheduling extension of HEFT as an example.

*Partition-Aware Priority.* Standard HEFT computes an upward rank using the average unpartitioned execution time of each operator across devices. However, this cost estimate does not reflect the fact that a partitionable operator may execute faster when its work is divided across multiple devices. To account for partitionable execution, we estimate the cost of each operator using its minimized local latency under the OSPD mapping:  $\bar{w}_i^{\text{PA}} = \min_{\pi_i=(a_i, \mathbf{u}_i) \in \Pi_i} \max_q \hat{\tau}_{i,a_i,d_q^{\text{OSPD}}}(u_{iq})$ . This is the same local cost used by the partition-only baseline. Using this effective cost, the partition-aware (PA) upward rank follows the same recursive structure as HEFT:  $\text{rank}_u^{\text{PA}}(i) = \bar{w}_i^{\text{PA}} + \max_{j:(i,j) \in E} \text{rank}_u^{\text{PA}}(j)$ . This rank can be interpreted as an estimate of the remaining critical-path length when allowing partitions.

*Partition-Aware Placement.* The scheduler processes operators in decreasing order of partition-aware upward rank. For each operator  $i$ , we evaluate each candidate partition plan  $\pi_i$  and device assignment  $\mathbf{d}_i$ . For a given pair  $(\pi_i, \mathbf{d}_i)$ , let  $\text{EST}(i, q, \pi_i, \mathbf{d}_i)$  denote the earliest feasible start time of subtask  $q$ , considering both predecessor completion times and current device availability. Since active subtasks of the same operator are assigned to distinct devices, these start times can be computed independently without introducing intra-operator device conflicts. The corresponding earliest finish time is  $\text{EFT}(i, \pi_i, \mathbf{d}_i) = \max_q [\text{EST}(i, q, \pi_i, \mathbf{d}_i) + \hat{\tau}_{i,a_i,d_{iq}}(u_{iq})]$ . The heuristic greedily chooses partition plans and device assignment that minimize this earliest finish time:  $(\pi_i^*, \mathbf{d}_i^*) = \arg \min_{\pi_i \in \Pi_i, \mathbf{d}_i \in \mathcal{D}^{K_i}: d_{iq} \neq d_{ir}, \forall q \neq r} \text{EFT}(i, \pi_i, \mathbf{d}_i)$ . This generalizes device selection in schedule-only list scheduling. Instead of assigning an indivisible operator to one device, the heuristic selects both the workload division and the subtask-device placement for the current operator.

Compared with expanded-DAG scheduling, partition-aware list scheduling can account for current device availability and predecessor completion times when making a partition decision. However, the decision remains greedy; once an operator is scheduled, its partition plan and placement are fixed, and later decisions

cannot revise it. Thus, partition-aware list scheduling is a stronger practical baseline, but it still does not fully search the joint partition-scheduling space.

## 5. Proposed Iterative Search

The formulation in Section 4.1 captures both intra-operator parallelism through partition plans and inter-operator parallelism through DAG scheduling. However, directly optimizing over all partition plans and schedules is too expensive. We therefore propose an iterative search framework that improves the schedule through local updates to partition plans (Section 5.1). In particular, three techniques facilitate a practical online search process. First, staging decomposes a large inference DAG into smaller scheduling regions by exploiting block structure in neural networks (Section 5.2). Second, criticality-aware sampling focuses candidate updates on operators that are likely to affect the makespan (Section 5.3). Third, latency prediction estimates the cost of partition plans without exhaustive on-device profiling (Section 5.4).

### 5.1. Framework Overview

The full iterative search algorithm is summarized in the Appendix (Section B, Algorithm 1). The key idea is to separate partition-plan updates from schedule construction. For each operator  $i$ , the search maintains a selected partition plan  $\pi_i \in \Pi_i$ , where  $\pi_i = (a_i, \mathbf{u}_i)$  follows the notation in Section 4.1. We denote the partition plan at iteration  $t$  by  $\boldsymbol{\pi}^{(t)} = \{\pi_i^{(t)}\}_{i \in V}$ . Then, the schedule-construction algorithm builds a schedule  $S^{(t)} = \text{SCHEDULE}(G, \boldsymbol{\pi}^{(t)}, \hat{\tau}, \mathcal{D})$ . The produced schedule  $S^{(t)}$  contains device assignments and execution order for the resulting subtasks; let  $T(S^{(t)})$  denote its makespan. Since SCHEDULE receives fixed partition plans as input and does not need to solve the partitioning problem, it can be instantiated with an existing schedule-only heuristic, such as the list schedulers described in Section 4.2. The iterative search algorithm acts as the top-level search, which proposes partition-plan updates, uses SCHEDULE to construct the corresponding schedule, and accepts an update that reduces the makespan.

At each iteration, the search algorithm first samples a set of partitionable operators; in this work, we propose the criticality-aware sampling rule described in Section 5.3. For each sampled operator  $i$ , the search algorithm generates a set of candidate updates; a candidate update is denoted by  $m = (i, \tilde{\pi}_i)$ , where  $\tilde{\pi}_i \in \Pi_i$  is a candidate replacement for the current partition plan  $\pi_i^{(t)}$  of operator  $i$ . In the default setting, the replacement plan  $\tilde{\pi}_i$  is sampled by first choosing a partition strategy and then choosing a workload division uniformly at random. Let  $\boldsymbol{\pi}_m^{(t)}$  denote the partition-plan vector obtained after applying candidate update  $m$  to the current plan vector  $\boldsymbol{\pi}^{(t)}$ :  $\pi_{m,i}^{(t)} = \tilde{\pi}_i$  and  $\pi_{m,j}^{(t)} = \pi_j^{(t)}$  for all  $j \neq i$ . The corresponding candidate schedule is  $S_m^{(t)} = \text{SCHEDULE}(G, \boldsymbol{\pi}_m^{(t)}, \hat{\tau}, \mathcal{D})$ , and the makespan improvement is  $\Delta_m^{(t)} = T(S^{(t)}) - T(S_m^{(t)})$ . Thus,  $\Delta_m^{(t)} > 0$  means that the candidate update  $m$  reduces the current makespan. At each iteration, the algorithm accepts the candidate update with the largest positive improvement, and it stops when none of the sampled candidate updates improves the schedule. The resulting procedure is a heuristic local search. Although every accepted update reduces the estimated makespan, the use of sampled candidate updates means that the method provides neither a global optimality guarantee nor a worst-case approximation bound. We therefore evaluate its solution quality empirically against offline joint optimization in Section 6.

### 5.2. Staging

Neural networks are commonly organized as a sequence of blocks, where each block can contain parallel branches. For example, an Inception block is a fork-join structure where branches are merged before the next block begins, as illustrated in Fig. 2a. In such structures, many scheduling decisions are local to a block, since later blocks cannot begin until the block output has been produced. Staging exploits this observation by optimizing one region at a time instead of searching over the entire DAG at once.

A natural stage boundary is a *global join point*. We define a global join point as an operator that separates the DAG into a prefix and a suffix; i.e., every other operator is either an ancestor or a descendant of this operator. Intuitively, when execution reaches such an operator, all computation in the prefix has joined, and the remaining suffix depends only on the joined output. Closing a stage at a global join therefore gives a clean

separation between the two regions. Formally, let  $\text{Anc}(i)$  and  $\text{Desc}(i)$  denote the ancestors and descendants of operator  $i$ . If  $|\text{Anc}(i)| + |\text{Desc}(i)| = |V| - 1$ , then every other operator is comparable with  $i$  in the DAG, and we treat  $i$  as a global join point. We use this criterion to identify stage boundaries when the model contains clear block-level joins.

However, some neural networks do not exhibit frequent global join points. For example, HRNet (Fig. 2b) contains long branch-heavy regions in which multiple branches progress independently before being merged much later. If staging only depends on global joins, these regions can form large stages, making the search expensive. To bound the search cost, we additionally enforce a maximum stage size (set to 20 operators in our experiments).

The way we explore the DAG determines which operators are included in the bounded stage: since iterative search constructs schedules using only the operators inside the current stage, the DAG exploration affects the schedule if a global join is not encountered before reaching the size limit. In a branch-heavy region with multiple parallel branches, a naive exploration order may follow one branch too deeply, causing the bounded stage to contain mostly a dependent sequence of operators from the same branch, thus exposing little inter-operator parallelism, resulting in a search that relies too heavily on intra-operator partitioning, potentially missing opportunities to schedule operators from different branches concurrently. To avoid this, we explore the DAG according to decreasing HEFT upward rank  $\text{rank}_u(i)$ , which estimates the remaining distance from operator  $i$  to the graph exit, as defined in Section 4.2. Starting from the current stage, we repeatedly explore the available operator with the largest upward rank  $\text{rank}_u(i)$ . If this exploration reaches a global join point, the current stage is closed at that join. If no global join is reached before the stage contains the maximum allowed number of operators, the stage is closed at the size limit. This rank-guided exploration tends to select operators from parallel branches with similar remaining latency, rather than following a single branch for many consecutive operators. As a result, bounded stages are more likely to preserve useful inter-operator parallelism.

After the DAG is divided into a list of stages, iterative search is run separately on each stage. For each stage, predecessors that were scheduled in earlier stages provide completion times for operators in the current stage. This preserves correctness with respect to the original dependencies, but can potentially restrict some cross-stage scheduling flexibility (in the case of reaching maximum stage size). Thus, staging substantially reduces search cost while retaining most local scheduling opportunities inside neural-network blocks. Without staging, searching over the entire model can hurt the optimization process and lead to a lower-quality schedule, as shown in the ablation study in Section 6.5.

### 5.3. Criticality-Aware Sampling

The main cost of the iterative search is update evaluation. Each candidate update changes the partition plan of one operator and requires rebuilding the schedule to evaluate the makespan improvement. Therefore, the search should spend most of its budget on operators whose partition plans can affect the current makespan. Our sampling policy is motivated by a standard idea in critical-path scheduling: tasks on the critical path are more important because reducing their execution time can directly reduce the final completion time. Specifically, we use *slack* to quantify how close a scheduled subtask is to the critical path [21]. Slack is the amount of time by which a scheduled subtask can be delayed without increasing the makespan. A task with zero slack lies on at least one critical path of the realized schedule, while a task with large slack can move to a later point in the schedule without affecting end-to-end latency. Therefore, changing a low-slack operator is more likely to reduce the makespan than changing a high-slack operator.

Because the current schedule may contain partitioned operators, we compute slack on the scheduled subtasks rather than on the original DAG. Given the current schedule  $S$ , we construct a scheduled-subtask graph  $\mathcal{G}_S = (\mathcal{V}_S, \mathcal{E}_S)$ , where each node  $(i, q) \in \mathcal{V}_S$  is a scheduled subtask with duration  $\ell_{iq} = e_{iq} - s_{iq}$ . The edge set  $\mathcal{E}_S$  contains two types of constraints: (1) dependency edges representing the full-barrier dependencies over subtasks from the original DAG, and (2) device-order edges connecting consecutive subtasks executed on the same device. Criticality-aware sampling is then performed on  $\mathcal{G}_S$ .

We compute slack through a reverse pass over  $\mathcal{G}_S$ . Let  $T(S)$  be the makespan of the current schedule. For each scheduled subtask  $(i, q)$ , let  $LF_{iq}$  be the latest finish time that does not increase  $T(S)$ , and let

$LS_{iq}$  be the corresponding latest start time. If  $(i, q)$  has no successor in  $\mathcal{G}_S$ , then  $LF_{iq} = T(S)$ ; otherwise,  $LF_{iq} = \min_{(i,q) \rightarrow (j,r) \in \mathcal{E}_S} LS_{jr}$ . The latest start time is  $LS_{iq} = LF_{iq} - \ell_{iq}$ , and the resulting subtask slack is therefore  $\sigma_{iq} = LS_{iq} - s_{iq} = LF_{iq} - e_{iq}$ . We aggregate subtask slack into an operator-level score by taking the minimum slack over the operator’s subtasks:  $\sigma_i = \min_{q: u_{iq} > 0} \sigma_{i,q}$ . Thus, an operator is treated as important when any of its scheduled subtasks lie on (or close to) a tight path in the current schedule.

The sampler should focus on low-slack operators, but it should not sample only the current critical path. Changing a non-critical operator can free a device earlier. We therefore use a mixture distribution that combines criticality-biased sampling with uniform sampling. We assign each operator a weight  $\omega_i = \frac{\sum_q \ell_{iq}}{\sigma_i + \epsilon}$  as operators with longer duration (i.e., total subtask durations  $\sum_q \ell_{iq}$ ) are more likely to have a greater impact on the makespan; a small constant  $\epsilon$  prevents division by zero. The combined distribution is  $P(i) = \rho \cdot \frac{\omega_i}{\sum_{j \in V_p} \omega_j} + (1 - \rho) \cdot \frac{1}{|V_p|}$ , where the parameter  $\rho$  controls the choice between two sampling approaches. In our experiments, we use  $\rho = 0.8$  and  $\epsilon = 10^{-12}$ .

Building  $\mathcal{G}_S$  and computing slack are linear in the number of scheduled subtasks and schedule-order edges. This cost is paid once per iteration and is small compared to update evaluation, which reruns the scheduling process for every sampled operator and candidate partition plan.

#### 5.4. Latency Prediction for Candidate Partition Plans

The iterative search repeatedly evaluates candidate partition plans  $\pi_i = (a_i, \mathbf{u}_i)$ , requiring latency estimation of a subtask with workload size  $\mathbf{u}$  on device  $d$  under partition strategy  $a$ . Exhaustively profiling all such configurations during model initialization is impractical. For example, our search on Inception-v3 considers 2462 operator configurations; measuring all of them on device would add substantial overhead. Simple proxy metrics like FLOPs are insufficient because they do not capture memory behavior or framework-specific implementation choices [14, 22]. This limitation is especially substantial on GPUs, as GPU latency can change discontinuously with workload size because the backend may change the selected kernel implementation or map the workload to different GPU workgroup configurations [9]. Consequently, a simple linear model over workload size cannot accurately predict partition latency. On the other hand, complex ML predictors (e.g., MLP) can introduce too much latency prediction overhead for online search.

To balance prediction accuracy and runtime overhead, we adopt a gradient-boosted decision tree (GBDT) model [23]. For each target device, we train (offline) separate GBDT predictors for each operator type (e.g., convolution, linear and element-wise operators); these predictors can be reused across neural networks. Given operator  $i$ , partition strategy  $a$ , assigned workload size  $u$ , and device  $d$ , the predictor estimates the latency  $\hat{\tau}_{i,a,d}(u)$ . The predictor input includes operator-level configuration features such as input and output tensor shapes, assigned output channels, kernel size, stride, padding, and FLOPs. For GPU predictors, we further add backend-aware dispatch features following [9], such as the GPU workgroup size and number of dispatched workgroups, which expose backend decisions that are hidden from operator configuration but substantially affect GPU execution time. The complete feature table is summarized in the Appendix (Section B, Table B.7). The resulting GBDT predictors (compiled into C code) are sufficiently efficient for online use. For instance, for Inception-v3, the average prediction cost is 18.2  $\mu\text{s}$  per operator configuration; evaluating all 2462 candidate configurations across three devices takes 135 ms in total, which is only 12.9% of the 1048 ms model initialization time, as shown in Section 6.4. The predictors provide sufficient accuracy for partition selection; e.g., convolution operators account for 90% of the model’s end-to-end GPU latency, and the corresponding mean absolute percentage error (MAPE) is below 8.2% across all three devices.

## 6. Evaluation

In this section, we evaluate the proposed iterative search against baseline methods in both simulation (Section 6.2) and on-device execution (Section 6.3). We also evaluate its deployment-time overhead (Section 6.4) and conduct additional ablation and sensitivity analysis (Section 6.5).

| Family                       | # Models | # Convolution Range | # Operators Range |
|------------------------------|----------|---------------------|-------------------|
| Inception / Inception-ResNet | 4        | 94–244              | 170–429           |
| SqueezeNet / SqueezeResNet   | 4        | 26                  | 46–51             |
| PeleeNet                     | 1        | 113                 | 144               |
| HRNet                        | 9        | 91–325              | 149–535           |

Table 3: Workload summary. The full per-model list is reported in the Appendix (Table C.8).

### 6.1. Experimental Setup

*Mobile Platforms.* We evaluate 4 mobile platforms, summarized in Table 1, each modeled as three logical scheduling devices: GPU, CPU (L) for the large-core cluster, and CPU (M) for the medium-core cluster. For CPU execution, threads are pinned within the selected cluster so that each measured latency corresponds to one logical device. Considering that mobile measurements are sensitive to thermal throttling and dynamic frequency scaling, we follow prior benchmarking practice [24] to improve measurement stability by enabling performance mode when available, keeping the phone charging, and attaching an external cooling fan.

*Workloads.* We evaluate 18 image-classification models implemented from *imgclsmob* [25]; each model is exported to TensorFlow Lite with input image resolution  $224 \times 224$  with batch size of 1. Table 3 summarizes the workload families, including Inception [11], Inception-ResNet [26], SqueezeNet [27], PeleeNet [28], and HRNet [12]. These models contain branch-heavy modules that expose substantial inter-operator parallelism, while their latency remains dominated by convolution operators for which intra-operator partitioning can be beneficial. This diversity allows us to evaluate whether a scheduler can make effective decisions about both partitioning and scheduling.

*On-Device Runtime.* Our evaluation system has three components: latency predictors, an iterative scheduler, and an execution runtime. First, the latency predictors estimate the execution time of candidate operator configurations. We constructed a training dataset containing 10,000 samples of convolution configurations, 1,000 samples each for linear and element-wise (add/multiply) operators, and 500 samples for other operators, including padding, resizing, pooling, and activations. A 20% subset of the data was used for validation during training. We train gradient-boosted decision tree (GBDT) models using LightGBM [29], with hyperparameter settings following [9]. The trained predictors are compiled into C code using Treelite [30] and TL2cgen [31], enabling efficient prediction on mobile CPUs. This avoids exhaustively measuring every operator-device-partition configuration during model initialization. Second, the iterative scheduler selects partitioning and scheduling decisions. It is implemented in C++ and compiled into a native Android binary with the `-O3` optimization flag. Unless otherwise stated, each stage contains at most 20 nodes; at each iteration, the search samples 5 candidate nodes and 10 candidate partitions per node. Third, the execution runtime runs the generated schedule for end-to-end measurement, which incorporates CPU and GPU kernels from TensorFlow Lite v2.17.0; specifically, GPU execution uses the OpenCL-based TensorFlow Lite GPU delegate [32], while CPU execution uses XNNPACK kernels [33].

*Desktop Offline Evaluation.* We use a desktop machine with an Intel Core i7-14700K CPU and 64 GB of memory for offline simulation and solver-based results (i.e., Gurobi). All simulations are implemented in Python. The Gurobi results are implemented with `gurobipy` [13] and run with 28 CPU threads, with a timeout of 300 s for each stage obtained by dividing the original DAG. These solver-based results are used to evaluate schedule quality, not as deployment-time scheduling methods.

### 6.2. Simulation Results: Scheduling Quality

We first evaluate scheduling quality in simulation using predicted operator-level latencies. For each platform, latency is *normalized* by the *offline* result achieved by JOINT-GUROBI, which solves the joint partition-aware scheduling formulation in Section 4. Notably, some approaches have multiple variants of schedule-construction algorithms. For these approaches, *Best* reports the lowest latency among the evaluated list-scheduling heuristics, and *Gurobi* reports the corresponding offline solver-based result for that *restricted* search space. Table 4 shows that single-device execution leaves substantial performance potential unexploited; GPU is the fastest device, but it is still  $1.61 \times$ – $2.13 \times$  slower than JOINT-GUROBI on average across platforms. In addition, neither partition-only nor schedule-only execution is sufficient. Partition-only execution improves

| Approach         | Variant                  | OnePlus 11 |       | Motorola 2022 |       | Pixel 4 |       | Pixel 5 |       |
|------------------|--------------------------|------------|-------|---------------|-------|---------|-------|---------|-------|
|                  |                          | Avg        | Worst | Avg           | Worst | Avg     | Worst | Avg     | Worst |
| Single-device    | GPU                      | 1.64       | 2.35  | 1.61          | 2.23  | 2.10    | 2.94  | 2.13    | 2.44  |
|                  | CPU (L)                  | 7.57       | 10.05 | 7.71          | 9.46  | 6.31    | 7.27  | 4.87    | 6.30  |
|                  | CPU (M)                  | 6.83       | 8.89  | 6.39          | 7.45  | 3.81    | 4.33  | 5.16    | 6.51  |
| Partition-only   | Min-local-latency        | 1.20       | 1.39  | 1.22          | 1.59  | 1.16    | 1.42  | 1.11    | 1.28  |
| Schedule-only    | Best                     | 1.28       | 1.44  | 1.30          | 1.47  | 1.47    | 1.78  | 1.51    | 1.84  |
|                  | Gurobi                   | 1.22       | 1.42  | 1.24          | 1.45  | 1.44    | 1.77  | 1.49    | 1.84  |
| Expanded-DAG     | Min-local-latency Best   | 1.15       | 1.26  | 1.15          | 1.33  | 1.11    | 1.25  | 1.08    | 1.22  |
|                  | Min-local-latency Gurobi | 1.09       | 1.15  | 1.08          | 1.22  | 1.05    | 1.15  | 1.04    | 1.13  |
|                  | Equal-size Best          | 1.34       | 1.52  | 1.34          | 1.60  | 1.25    | 1.42  | 1.18    | 1.36  |
|                  | Equal-size Gurobi        | 1.30       | 1.45  | 1.28          | 1.51  | 1.20    | 1.36  | 1.14    | 1.32  |
| Partition-aware  | HEFT                     | 1.12       | 1.21  | 1.12          | 1.29  | 1.08    | 1.19  | 1.07    | 1.18  |
| Iterative-search | HEFT                     | 1.04       | 1.08  | 1.03          | 1.11  | 1.01    | 1.06  | 1.02    | 1.05  |

Table 4: Main simulation results. Latency is normalized to JOINT-GUROBI. *Best* denotes the best result among evaluated list-scheduling heuristics within each baseline method, while *Gurobi* denotes the offline solver-based result for the corresponding restricted search space. *Avg* (*Worst*) reports the average (worst) normalized latency across 18 models. Full results are reported in the Appendix (Table C.9).

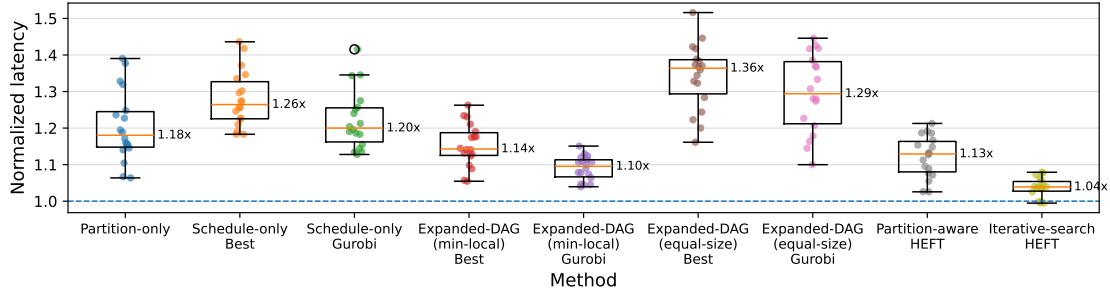


Figure 3: Distribution of normalized simulation latency across 18 models on OnePlus 11 (annotated values report the median). Latency is normalized to JOINT-GUROBI. Additional results on other devices are reported in the Appendix (Fig. C.7).

significantly over GPU-only execution, reducing average normalized latency to  $1.11\times$ – $1.22\times$  across devices. However, it remains slower than JOINT-GUROBI because it chooses each partition using local operator latency rather than end-to-end DAG makespan. Schedule-only methods also exhibit significant limitations due to unpartitioned operators. Even schedule-only Gurobi is  $1.22\times$ – $1.49\times$  slower than JOINT-GUROBI on average, which shows that the performance gap is not simply due to weak list-scheduling heuristics.

The expanded-DAG scheduling results show that exposing subtask partitioning is helpful but not sufficient. With min-local-latency expansion, the best list scheduler reduces average normalized latency to  $1.08\times$ – $1.15\times$ , and offline Gurobi scheduling of the expanded graph further reduces it to  $1.04\times$ – $1.09\times$ . However, these methods still fix the partition plan before scheduling, so they can miss globally better partition choices. Equal-size expansion performs much worse, with average normalized latency between  $1.18\times$  and  $1.34\times$  for the best scheduling heuristic. Partition-aware HEFT improves over expanded-DAG with best heuristics by choosing partition plans during scheduling, reaching  $1.07\times$ – $1.12\times$  average normalized latency. However, it remains limited because its greedy local scheduling decisions can prevent the discovery of globally better schedules.

The proposed iterative search closes most of the gap to JOINT-GUROBI. Across the four devices, it achieves  $1.01\times$ – $1.04\times$  average normalized latency and  $1.05\times$ – $1.11\times$  worst-case normalized latency. Fig. 3 depicts the distribution of normalized latency across models for each method on OnePlus 11; iterative search achieves a median normalized latency of  $1.04\times$  and 90th-percentile latency of  $1.07\times$ , which further confirms

| Approach         | Variant                  | OnePlus 11 |       | Motorola 2022 |       | Pixel 4 |       | Pixel 5 |       |
|------------------|--------------------------|------------|-------|---------------|-------|---------|-------|---------|-------|
|                  |                          | Avg        | Worst | Avg           | Worst | Avg     | Worst | Avg     | Worst |
| Single-device    | GPU                      | 1.62       | 1.95  | 1.60          | 2.01  | 1.94    | 2.46  | 1.93    | 2.10  |
|                  | CPU (L)                  | 7.68       | 11.72 | 7.45          | 10.29 | 6.60    | 9.11  | 4.45    | 5.54  |
|                  | CPU (M)                  | 5.49       | 8.64  | 5.21          | 7.32  | 3.21    | 3.75  | 4.71    | 5.76  |
| Partition-only   | Min-local-latency        | 1.37       | 2.92  | 1.25          | 1.61  | 1.20    | 1.93  | 1.12    | 1.52  |
| Schedule-only    | Best                     | 1.15       | 1.33  | 1.08          | 1.26  | 1.25    | 1.52  | 1.33    | 1.66  |
|                  | Gurobi                   | 1.08       | 1.22  | 1.07          | 1.25  | 1.24    | 1.48  | 1.31    | 1.68  |
| Expanded-DAG     | Min-local-latency Best   | 1.34       | 1.61  | 1.16          | 1.31  | 1.13    | 1.73  | 1.08    | 1.46  |
|                  | Min-local-latency Gurobi | 1.19       | 1.33  | 1.13          | 1.36  | 1.09    | 1.69  | 1.07    | 1.53  |
|                  | Equal-size Best          | 1.68       | 2.08  | 1.43          | 1.80  | 1.22    | 1.90  | 1.20    | 1.82  |
|                  | Equal-size Gurobi        | 1.56       | 1.83  | 1.63          | 2.16  | 1.20    | 1.88  | 1.18    | 1.88  |
| Partition-aware  | HEFT                     | 1.36       | 1.69  | 1.18          | 1.39  | 1.11    | 1.78  | 1.10    | 1.60  |
| Iterative-search | HEFT                     | 1.02       | 1.11  | 1.01          | 1.10  | 1.00    | 1.29  | 1.04    | 1.15  |

Table 5: Main measurement results. Latency is normalized to JOINT-GUROBI. Full results are reported in the Appendix (Table C.10).

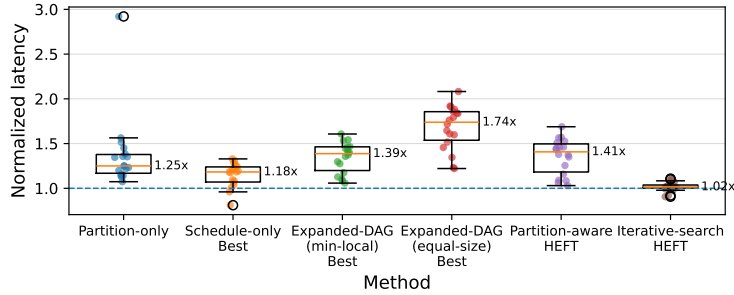


Figure 4: Distribution of measured end-to-end latency across 18 models on OnePlus 11, normalized to JOINT-GUROBI (annotated values report the median). Additional results on other devices are reported in the Appendix (Fig. C.8).

that the improvement of iterative search is consistent across models.

### 6.3. Real Measurements: End-to-End Latency

We next evaluate the generated schedules on the mobile platforms. Unlike the simulation study, this experiment executes the schedules through our runtime implementation and accounts for practical runtime effects such as kernel dispatch and CPU-GPU synchronization. Table 5 shows that the main trends from simulation continue to hold on real devices, but with a clearer distinction between methods. Single-device execution remains far from the joint optimization solution. Schedule-only execution performs relatively well on some platforms because it dispatches fewer kernels and avoids partitioning overhead. However, even offline schedule-only Gurobi remains  $1.07\times$ – $1.31\times$  slower than JOINT-GUROBI on average, with worst-case ranging from  $1.22\times$  to  $1.68\times$ . This confirms that indivisible scheduling still misses intra-operator parallelism. Partition-only execution is less consistent in real measurements. Its average normalized latency ranges from  $1.12\times$  on Pixel 5 to  $1.37\times$  on OnePlus 11, and its worst case reaches  $2.92\times$  on OnePlus 11. This result suggests that excessive partitioning may deliver only marginal improvement because of dispatch and synchronization overhead in practice. Iterative search is less sensitive to such overhead because it starts from a non-partitioning plan, accepting a partition update only if it reduces the overall makespan; thus, it can avoid excessive partitioning when inter-operator parallelism is available, facilitating utilization of devices by other ready operators. The expanded-DAG methods show a similar effect; e.g., equal-size expansion performs poorly, reaching  $1.68\times$  average normalized latency on OnePlus 11. Even min-local-latency expanded-DAG

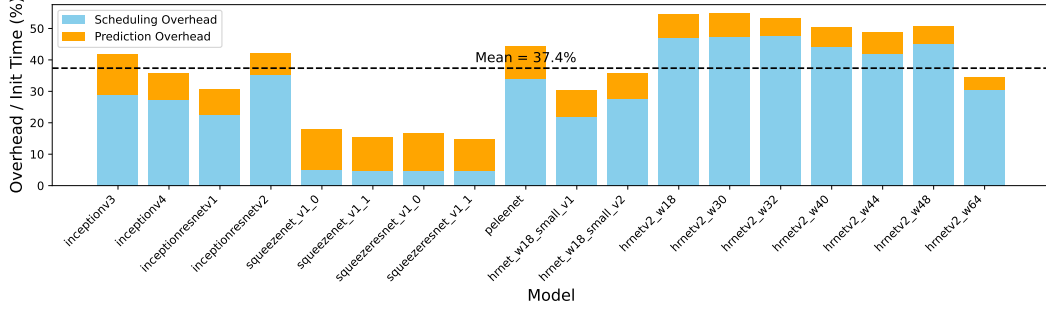


Figure 5: Scheduling overhead compared to TFLite model initialization time, measured on OnePlus 11. Each bar is stacked by prediction overhead and iterative scheduling overhead. Full measurements are reported in the Appendix (Table C.11).

scheduling can be worse than schedule-only execution on some devices, indicating that a partition plan that is locally effective may still be ineffective once runtime overhead and graph-level device availability are accounted for.

Iterative search is the most robust practical method. Across the four phones, it achieves an average normalized latency within  $1.00\times$ – $1.04\times$  of JOINT-GUROBI, with worst-case latency ranging from  $1.10\times$  to  $1.29\times$ . The distribution across models on OnePlus 11 (Fig. 4) further confirms this conclusion, where iterative search has a median normalized latency of  $1.02\times$  and a 90th-percentile latency of  $1.09\times$ . Overall, the real-device results show that iterative search preserves most of the benefit of joint partition-aware scheduling while avoiding the limitations of excessive partitioning, fixed expanded-DAG construction, or purely greedy partition-aware decisions.

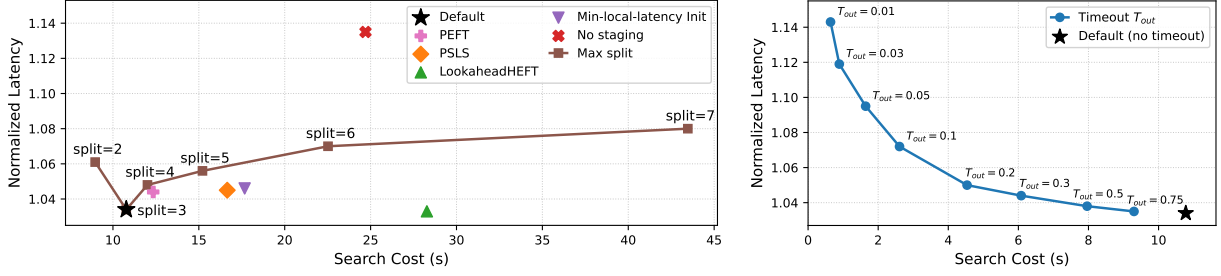
#### 6.4. Runtime Overhead

We next evaluate whether iterative search is practical to use during the model initialization process. As discussed in Section 2.3, the schedule is computed once for a fixed model and input shape, cached, and reused across inference requests. Therefore, the question is whether the overhead due to the iterative search is reasonable with respect to model initialization time rather than with respect to single-inference latency. Fig. 5 reports the runtime overhead on OnePlus 11. We separate two components: latency-prediction overhead (in orange), which constructs the latency table for candidate partition plans, and scheduling overhead (in blue), which is incurred by the iterative search.

The latency-prediction overhead is relatively stable across models, ranging from 51 ms to 185 ms (4.0% to 12.9% of model initialization time). Many configurations repeat within the same network, allowing predicted latencies to be reused. Thus the scheduler can predict latency of candidate partitions without exhaustively measuring each operator-device-partition combination. Scheduling overhead scales more directly with the number of partitionable operators and the size of the search space; it ranges from 24 ms for small SqueezeNet models to up to 1339 ms for the large HRNet models. Consequently, scheduling accounts for a larger fraction of the total overhead of large models. At the same time, model initialization time also grows with graph size and model complexity due to graph construction and memory allocation. The total overhead is 37.4% of model initialization time on average across all models. For the larger models (e.g., HRNets), the overhead is mostly dictated by characteristics of the model family. Regardless of model family, the total overhead is much smaller than exhaustive on-device profiling or solving the global optimization problem with an offline solver, which can take hours or days.

#### 6.5. Ablation and Sensitivity Analysis

The iterative search contains several design choices that trade scheduling quality for search overhead. We study these choices to understand which components are most important and how the method behaves under different search budgets. In this subsection, search cost is measured using the Python implementation on the desktop platform described in Section 6.1. The absolute search costs are therefore not directly comparable to



(a) Ablation and split sensitivity. Numbers next to square markers denote the split value. The default setting uses staged search, HEFT schedule construction, search cost but degrade latency quality. The default and split of 3. (b) Timeout sensitivity. Smaller timeout values reduce search cost but degrade latency quality. The default setting has no timeout constraint.

Figure 6: Sensitivity of scheduling quality to iterative-search cost. Latency is normalized to JOINT-GUROBI. Each point corresponds to one configuration, where lower normalized latency and lower search cost are preferred. Full data are reported in the Appendix (Table C.12).

the on-device overhead in Section 6.4, but the relative trends are useful for comparing design choices. Fig. 6 summarizes the trade-off between normalized latency (on a OnePlus 11 phone) and search cost, as detailed next.

*Staging.* Staging decomposes a large inference DAG into smaller scheduling regions. Without staging, the search has a broader global view, but evaluating each candidate update becomes substantially more expensive, thus performing worse than the staged approach; search cost increases from 10.8 s to 24.7 s, while normalized latency also increases from 1.03 to 1.14, under a 30-second search limit (which is already above the highest per-model search cost observed with staging). Although the unstaged search can consider more of the graph at once, its higher per-update cost means that it evaluates fewer useful alternatives before the time limit. This result suggests that staging is important not only for reducing search overhead but also for guiding the search toward useful local improvements.

*Number of Splits.* By default, we set the maximum number of subtasks per operator (i.e., splits) to the number of devices (*three* in our experiments). As shown in Fig. 6a, this choice provides the best trade-off. Using fewer splits reduces search cost but degrades schedule quality due to limited flexibility; e.g., split of 2 reduces search cost to 9.0 s but increases normalized latency to 1.06. Using more splits does not improve quality and instead increases search cost, since each GPU kernel incurs a dispatch overhead, which may outweigh the benefits of additional flexibility due to finer-granularity partitions; e.g., split of 7 raises the search cost to 43.5 s while producing worse normalized latency than split of 3. This supports our default choice of setting the maximum split count to the number of devices.

*Initialization.* We compare two initial partition plans before search commences: no partitioning and min-local-latency partitioning. The min-local-latency initialization achieves similar normalized latency but with increased search cost, from 10.8 s to 17.7 s. This suggests that the final schedule quality is relatively robust to the initialization strategy, and that a locally optimal initialization does not necessarily lead to a better global schedule.

*Scheduling Algorithms.* We also evaluate alternative scheduling heuristics within the iterative-search framework, including PEFT, PSLS, and Lookahead-HEFT. PEFT and PSLS produce slightly worse normalized latency than the HEFT-based default while increasing search cost. Lookahead-HEFT achieves similar normalized latency, but requires 28.3 s of search, more than  $2.6\times$  the default HEFT search cost. Thus, HEFT provides the best practical trade-off as a schedule-construction algorithm.

*Search Budget.* Lastly, we vary the search budget by enforcing a timeout on the search for each stage. As shown in Fig. 6b, reducing the timeout produces a smooth trade-off between search cost and scheduling quality. For example, a timeout of 0.3 s reduces search cost from 10.8 s to 6.1 s while maintaining similar normalized latency. A more aggressive timeout of 0.2 s further reduces search cost to 4.5 s with normalized latency of 1.05. Very small timeout values continue to reduce search cost, but the latency degradation becomes more significant.

| <b>Scheduling \ Execution</b> | <b>OnePlus 11</b> | <b>Motorola 2022</b> | <b>Pixel 4</b> | <b>Pixel 5</b> |
|-------------------------------|-------------------|----------------------|----------------|----------------|
| <b>OnePlus 11</b>             | 1.02              | 1.06                 | 1.23           | 1.27           |
| <b>Motorola 2022</b>          | 1.05              | 1.01                 | 1.17           | 1.21           |
| <b>Pixel 4</b>                | 1.34              | 1.31                 | 1.00           | 1.13           |
| <b>Pixel 5</b>                | 1.41              | 1.31                 | 1.36           | 1.04           |

Table 6: Cross-platform scheduling sensitivity. Each entry reports normalized latency when the schedule is generated using the row device and executed on the column device.

*Cross-Platform Scheduling.* We now consider whether a schedule generated for one mobile platform can be reused on another platform. In this experiment, the scheduler constructs an execution plan using latency predictions from one mobile platform, and the fixed plan is then executed on another platform. The results in Table 6 show that cross-platform reuse works well only when devices have similar CPU vs GPU performance characteristics. For example, OnePlus 11 and Motorola 2022 transfer well in both directions, with up to 5% increase in latency compared to performing the search on the actual device, because both are recent Snapdragon 8-series platforms with similar CPU-GPU relative performance. Further evidence of such similar performance characteristics is given in Table 5, where single-device (GPU or CPU) executions achieve comparable normalized latency across the pairs of devices. In contrast, schedules transfer poorly between newer Snapdragon 8-series devices and older Pixel devices, reaching up to  $1.41\times$  when a Pixel 5 schedule is executed on OnePlus 11. Thus, cross-platform reuse is possible for closely matched devices, but accurate per-device latency predictions remain important, e.g., when CPU-GPU relative performance changes.

#### 6.6. Discussion and Threats to Validity

Our evaluation demonstrates the effectiveness of partition-aware scheduling under the workload and hardware assumptions considered in this work. We next discuss the scope of these assumptions and factors that may affect the generalizability of our results to other inference workloads and hardware platforms.

*Dynamic Inference Graphs.* Our formulation targets fixed-shape inference workloads represented by a static operator DAG, for which an execution plan can be constructed once during model initialization and reused across inference requests. Although our experimental evaluation focused on CNNs, fixed-shape attention-based models (including most ViTs [34]) would fit the same DAG-level formulation if appropriate attention-specific partition strategies and latency models are provided. However, most autoregressive (e.g., LLMs) and dynamically routed Mixture-of-Experts (MoE) models would have computation and resource demands that vary across tokens or inputs, violating our assumption of a reusable fixed execution plan. Supporting such workloads would require dynamic scheduling, which we leave to future work.

*Dynamic Batching.* We focus on latency-oriented mobile inference and use batch size of 1 throughout our evaluation. For a *fixed* batch size greater than one, the formulation would still keep a static DAG, but the corresponding tensor shapes would need to be incorporated into the latency models and partition plans. Dynamic batching across multiple requests, however, introduces additional decisions, including when to form a batch and which requests to combine; this would also introduce throughput considerations in addition to single-request latency. Joint batch formation and heterogeneous scheduling are outside the scope of the current formulation.

*Accuracy of Latency Predictors.* Our scheduler relies on predicted operator latency to evaluate candidate partition plans; thus, prediction errors can affect partitioning and scheduling decisions. The accuracy of operator latency prediction was evaluated in our earlier work on CNNs [14, 35] and ViTs [36]. While we do not conduct a comprehensive sensitivity analysis of prediction error in this work, the cross-platform results in Table 6 provide indirect evidence of its impact: Schedules transfer well between platforms with similar CPU-GPU performance characteristics ( $1.05\text{--}1.06\times$ ), but can degrade by up to  $1.41\times$  across dissimilar platforms. This highlights the importance of accurately capturing relative device performance.

*Communication Model.* Because our target mobile platforms provide unified CPU-GPU memory, we omit explicit tensor-copy communication costs while retaining synchronization and dispatch overhead. This assumption does not directly hold for systems with non-uniform memory (e.g., discrete GPUs or distributed devices), where tensor copies would need to be incorporated into the cost model.

## 7. Related Work

*Heterogeneous DAG Scheduling.* Static scheduling of DAGs on heterogeneous devices is a classical problem. For instance, classical list-scheduling heuristics, including HEFT [5], Lookahead-HEFT [16], PEFT [17], LDCP [18], CEFT [19] and PSLs [20], assign each DAG task to one device while respecting dependency constraints. These methods differ in how they rank tasks, estimate downstream criticality, or select devices. However, they share the same task abstraction, where each DAG node is an indivisible unit that must execute on one device. This abstraction is appropriate for many task-graph workloads, but it misses an important opportunity in mobile DNN inference: Latency-dominant operators such as convolutions can often be partitioned across CPU and GPU, so treating every operator as an atomic task can leave devices idle in chain-dominated models. Our work extends the scheduling decision space by allowing selected operators to be split into subtasks whose placement and timing are optimized together with the DAG schedule. At the same time, our iterative framework can reuse these list schedulers as scheduling modules once a candidate partition plan is fixed.

*Meta-Heuristic, Learning-Based, and Offline Scheduling Approaches.* Beyond list scheduling, meta-heuristic and learning-based methods explore larger scheduling spaces. Genetic algorithms [37, 38], particle-swarm methods [39], and chemical-reaction optimization methods [40] have been applied to heterogeneous DAG scheduling. Learning-based systems use reinforcement learning [41, 42, 43] or graph neural networks [44] to learn device-placement or scheduling policies. These approaches show that richer search can outperform simple list scheduling, especially when workload structure is complex. However, they typically require substantial offline training, repeated simulation, or many runtime evaluations, and they often target cloud clusters, distributed training graphs, or general DAG workloads. Instead, our goal is to ensure that our scheduler runs during mobile model initialization and fits within a deployment-time budget. Therefore, we use offline optimization (with a long timeout of 5 minutes per stage) only as a notion of an optimal solution and design a lightweight iterative search that reduces the search space through staging, criticality-aware sampling, and latency prediction, while attempting to produce a solution close to that of the offline approach.

*DNN Partitioning for Edge and Collaborative Inference.* Another line of work partitions DNN computation across distributed edge-cloud platforms. Systems such as Neurosurgeon [45] and JointDNN [46] study layer-level partitioning between mobile devices and cloud servers, while DDNN [47] and Edgent [48] consider hierarchical edge-cloud execution, early exits, or adaptive model sizing. These systems primarily decide where *coarse-grained* model subgraphs or distributed inference tasks should run across networked platforms. In that setting, network bandwidth, feature-map transfer, and offloading latency are central optimization factors. In contrast, our work targets execution on heterogeneous hardware units within a single mobile device. The key challenge is not where to offload a subgraph across the network, but how to jointly exploit intra-operator partitioning and inter-operator DAG scheduling across local CPU and GPU resources.

*Mobile Heterogeneous DNN Co-Execution.* Recent systems study fine-grained CPU-GPU co-execution for mobile inference. CoDL [8] shows that a DNN operator can be partitioned across mobile CPU and GPU to improve performance and energy efficiency. Our previous work [9] further improves fine-grained co-execution by reducing synchronization overhead and improving latency prediction for partitioned operators. These works are closely related to ours because they demonstrate that mobile CPU and GPU can effectively cooperate within a single operator. However, they mainly optimize partitioning at the operator level. They do not address the full DAG scheduling problem, where multiple operators may be ready at the same time and where using all devices for one partitioned operator can delay other ready operators. Our work extends this line of research from isolated operator co-execution to end-to-end inference DAG execution. Our results

show that the key question is not only how to partition an operator, but also whether that operator should be partitioned in the current graph context.

*Joint Partitioning and Scheduling.* The closest work is HeSP [49], which studies a scheduling-partitioning problem, incorporating recursive task partitioning as an additional degree of freedom alongside scheduling. Their idea is closely related to our motivation, since both works observe that partitioning and scheduling should not be optimized independently; partitioning changes the task graph seen by the scheduler, while scheduling determines whether the extra parallelism introduced by partitioning is actually useful. However, HeSP is designed as an (offline) *simulation* framework and is evaluated mainly on dense linear-algebra task graphs with GPU servers and CPU-based edge devices. In contrast, our approach targets mobile neural-network inference DAGs, where our scheduler must be sufficiently efficient to be used online during model initialization, rather than as an offline simulation or solver. Given the neural-network setting, we also introduce partitioning strategy (i.e., operator-level workload splits, such as output-channel, input-channel, or spatial partitioning) as an additional search dimension. Given our online (model initialization time) focus, in contrast to [49], we design staging, criticality-aware sampling, and latency prediction to capture most of the benefit of joint optimization but within a limited deployment-time budget.

## 8. Conclusion

Our work shows that efficient mobile inference on devices with heterogeneous accelerators requires jointly considering operator partitioning and DAG scheduling. Schedule-only methods can exploit inter-operator parallelism, but often leave devices underutilized, particularly in chain-dominated models. Partition-only methods improve utilization by greedily co-executing operators across CPU and GPU through intra-operator parallelism, but they commonly miss opportunities to utilize an available device for other ready operators. To exploit both forms of parallelism, we formulate this problem as partition-aware DAG scheduling, which jointly considers partition choices, workload division, device assignment, and execution order. To make our solution practical for deployment time, we design an iterative search framework that uses staged optimization, criticality-aware sampling, and lightweight latency prediction to reduce optimization overhead. As a result, the proposed scheduler achieves latency close to that of offline joint optimization – across representative models and mobile platforms – while keeping overhead within a practical model-initialization budget. These results demonstrate that combining intra-operator co-execution with inter-operator DAG scheduling is an effective approach for accelerating mobile inference. Future directions include extending the framework to specialized on-device accelerators such as NPUs and DSPs, and incorporating additional objectives such as energy consumption and thermal behavior.

## Acknowledgments

This work was supported in part by the NSF CNS-1816887, CCF-1763747, and IIS-1833137 awards.

## References

- [1] A. Krizhevsky, I. Sutskever, G. E. Hinton, ImageNet classification with deep convolutional neural networks, *Proceedings of NIPS* 25 (2012).
- [2] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, et al., Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups, *IEEE Signal Processing Magazine* 29 (6) (2012) 82–97.
- [3] G. Lampropoulos, E. Keramopoulos, K. Diamantaras, Enhancing the functionality of augmented reality using deep learning, semantic web and knowledge graphs: A review, *Visual Informatics* 4 (1) (2020) 32–42.

- [4] J. Chen, X. Ran, Deep learning with edge computing: A review, *Proceedings of the IEEE* 107 (8) (2019) 1655–1674.
- [5] H. Topcuoglu, S. Hariri, M.-Y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, *IEEE Transactions on Parallel and Distributed Systems* 13 (3) (2002) 260–274.
- [6] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: *Proceedings of CVPR*, 2016, pp. 770–778.
- [7] S. Wang, G. Ananthanarayanan, T. Mitra, OPTiC: Optimizing collaborative CPU-GPU computing on mobile devices with thermal constraints, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38 (3) (2019) 393–406.
- [8] F. Jia, D. Zhang, T. Cao, S. Jiang, Y. Liu, J. Ren, Y. Zhang, CoDL: efficient CPU-GPU co-execution for deep learning inference on mobile devices, in: *Proceedings of MobiSys*, Vol. 22, 2022, pp. 209–221.
- [9] Z. Li, M. Paolieri, L. Golubchik, Accelerating mobile inference through fine-grained CPU-GPU co-execution, in: *Selected Papers of EPEW 2025*, Vol. 15657 of LNCS, Springer, 2026, pp. 41–55.
- [10] Z. Li, M. Paolieri, L. Golubchik, S. Lin, W. Yan, Predicting throughput of distributed stochastic gradient descent, *IEEE Trans. Parallel Distributed Syst.* 33 (11) (2022) 2900–2912.
- [11] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, Z. Wojna, Rethinking the inception architecture for computer vision, in: *Proceedings of CVPR*, 2016, pp. 2818–2826.
- [12] J. Wang, K. Sun, T. Cheng, B. Jiang, C. Deng, Y. Zhao, D. Liu, Y. Mu, M. Tan, X. Wang, et al., Deep high-resolution representation learning for visual recognition, *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43 (10) (2021) 3349–3364.
- [13] Gurobi Optimization, LLC, Gurobi Optimizer Reference Manual, <https://www.gurobi.com> (2026).
- [14] Z. Li, M. Paolieri, L. Golubchik, Inference latency prediction for CNNs on heterogeneous mobile devices and ML frameworks, *Performance Evaluation* 165 (2024) 102429:1–102429:26.
- [15] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., TensorFlow: A system for large-scale machine learning, in: *Proceedings of OSDI*, 2016, pp. 265–283.
- [16] L. F. Bittencourt, R. Sakellariou, E. R. Madeira, DAG scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm, in: *Proceedings of Euromicro*, IEEE, 2010, pp. 27–34.
- [17] H. Arabnejad, J. G. Barbosa, List scheduling algorithm for heterogeneous systems by an optimistic cost table, *IEEE Transactions on Parallel and Distributed Systems* 25 (3) (2014) 682–694.
- [18] M. I. Daoud, N. Kharma, A high performance algorithm for static task scheduling in heterogeneous distributed computing systems, *Journal of Parallel and Distributed Computing* 68 (4) (2008) 399–409.
- [19] M. A. Khan, Scheduling for heterogeneous systems using constrained critical paths, *Parallel Computing* 38 (4-5) (2012) 175–193.
- [20] Y. Zhao, S. Cao, L. Yan, List scheduling algorithm based on pre-scheduling for heterogeneous computing, in: *Proceedings of IEEE ISPA/BDCloud/SocialCom/SustainCom*, IEEE, 2019, pp. 588–595.
- [21] J. E. Kelley Jr, M. R. Walker, Critical-path planning and scheduling, in: *Proceedings of the Eastern Joint IRE-AIEE-ACM Computer Conference*, 1959, pp. 160–173.

- [22] X. Tang, S. Han, L. L. Zhang, T. Cao, Y. Liu, To bridge neural network design and real-world performance: A behaviour study for neural networks, *Proceedings of MLSys 3* (2021) 21–37.
- [23] J. H. Friedman, Greedy function approximation: A gradient boosting machine, *The Annals of Statistics* 29 (5) (2001) 1189–1232.
- [24] Z. Li, M. Paolieri, L. Golubchik, A benchmark for ML inference latency on mobile devices, in: *Proceedings of EdgeSys, ACM*, 2024, pp. 31–36.
- [25] Sandbox for training deep learning networks, <https://github.com/osmr/imgclsmob> (2024).
- [26] C. Szegedy, S. Ioffe, V. Vanhoucke, A. A. Alemi, Inception-v4, inception-resnet and the impact of residual connections on learning, in: S. Singh, S. Markovitch (Eds.), *Proceedings of AAAI, AAAI Press*, 2017, pp. 4278–4284.
- [27] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, K. Keutzer, SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and < 0.5 MB model size, *arXiv preprint arXiv:1602.07360* (2016).
- [28] R. J. Wang, X. Li, C. X. Ling, Pelee: A real-time object detection system on mobile devices, *Proceedings of NeurIPS 31* (2018).
- [29] Y. Shi, G. Ke, D. Soukhavong, J. Lamb, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, T.-Y. Liu, N. Titov, D. Cortes, LightGBM: Light gradient boosting machine, <https://github.com/lightgbm-org/LightGBM> (2026).
- [30] H. Cho, M. Li, Treelite: toolbox for decision tree deployment, <https://www.amazon.science/publications/treelite-toolbox-for-decision-tree-deployment> (2018).
- [31] TL2cgen: Model compiler for decision trees, <https://github.com/dmlc/tl2cgen> (2026).
- [32] J. Lee, N. Chirkov, E. Ignasheva, Y. Pisarchyk, M. Shieh, F. Riccardi, R. Sarokin, A. Kulik, M. Grundmann, On-device neural net inference with mobile GPUs, *arXiv preprint arXiv:1907.01989* (2019).
- [33] Google, XNNPACK: High-efficiency floating-point neural network inference operators for mobile, server, and web, <https://github.com/google/XNNPACK> (2026).
- [34] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, et al., An image is worth 16x16 words: Transformers for image recognition at scale, *arXiv preprint arXiv:2010.11929* (2020).
- [35] Z. Li, M. Paolieri, L. Golubchik, Predicting inference latency of neural architectures on mobile devices, in: *Proceedings of ICPE, ACM*, 2023, pp. 99–112.
- [36] Z. Li, M. Paolieri, L. Golubchik, A study on inference latency for vision transformers on mobile devices, in: *Proceedings of VALUETOOLS 2024, Vol. 663 of LNICST, Springer*, 2026, pp. 229–251.
- [37] S. Ding, J. Wu, G. Xie, G. Zeng, A hybrid heuristic-genetic algorithm with adaptive parameters for static task scheduling in heterogeneous computing system, in: *Proceedings of Trustcom/BigDataSE/ICSS, IEEE*, 2017, pp. 761–766.
- [38] M. Sulaiman, Z. Halim, M. Lebbah, M. Waqas, S. Tu, An evolutionary computing-based efficient hybrid task scheduling approach for heterogeneous computing environment, *Journal of Grid Computing* 19 (1) (2021) 11.
- [39] M. H. Shirvani, A hybrid meta-heuristic algorithm for scientific workflow scheduling in heterogeneous distributed computing systems, *Engineering Applications of Artificial Intelligence* 90 (2020) 103501.

- [40] Y. Xu, K. Li, L. He, T. K. Truong, A DAG scheduling scheme on heterogeneous computing systems using double molecular structure-based chemical reaction optimization, *Journal of Parallel and Distributed Computing* 73 (9) (2013) 1306–1322.
- [41] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, J. Dean, Device placement optimization with reinforcement learning, in: *Proceedings of ICML*, 2017, pp. 2430–2439.
- [42] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, M. Alizadeh, Learning scheduling algorithms for data processing clusters, in: *Proceedings of ACM SIGCOMM*, 2019, pp. 270–288.
- [43] R. Addanki, S. Bojja Venkatakrishnan, S. Gupta, H. Mao, M. Alizadeh, et al., Learning generalizable device placement algorithms for distributed machine learning, *Proceedings of NeurIPS* 32 (2019).
- [44] Y. Zhou, X. Li, J. Luo, M. Yuan, J. Zeng, J. Yao, Learning to optimize DAG scheduling in heterogeneous environment, in: *Proceedings of MDM*, IEEE, 2022, pp. 137–146.
- [45] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, L. Tang, Neurosurgeon: Collaborative intelligence between the cloud and mobile edge, *ACM SIGARCH Computer Architecture News* 45 (1) (2017) 615–629.
- [46] A. E. Eshratifar, M. S. Abrishami, M. Pedram, JointDNN: An efficient training and inference engine for intelligent mobile cloud computing services, *IEEE Transactions on Mobile Computing* 20 (2) (2021) 565–576.
- [47] S. Teerapittayanon, B. McDanel, H.-T. Kung, Distributed deep neural networks over the cloud, the edge and end devices, in: *Proceedings of ICDCS*, IEEE, 2017, pp. 328–339.
- [48] E. Li, L. Zeng, Z. Zhou, X. Chen, Edge AI: On-demand accelerating deep neural network inference via edge computing, *IEEE Transactions on Wireless Communications* 19 (1) (2020) 447–457.
- [49] A. Rey, F. D. Igual, M. Prieto-Matías, HeSP: A simulation framework for solving the task scheduling-partitioning problem on heterogeneous architectures, in: *Proceedings of EuroPar*, Springer, 2016, pp. 183–195.

## Appendix

### A. Runtime Implementation for Scheduled Co-Execution

This appendix describes how the execution plan produced by our scheduler is executed at runtime. After partitioning decisions are fixed, we view the computation as an expanded-DAG. Specifically, each node represents either an original operator or one partitioned piece of an operator, where edges represent dependencies based on the full-barrier dependencies from the original DAG. The scheduler assigns each node to a device and determines the execution order of nodes on each device. The runtime execution follows this plan by maintaining one ordered queue per device. Each device executes its own queue in the order chosen by the scheduler.

The runtime execution maintains DAG dependencies using a shared dependency state. For each node, the runtime execution records how many of its predecessor nodes are still unfinished; a node becomes ready when this count reaches zero. When a node finishes, the runtime execution updates the dependency states of its successor nodes. This simple mechanism also captures the full-barrier dependency model used by the scheduler; i.e., if an operator is partitioned, any successor can start only after all required predecessor pieces have completed.

The CPU implements this dependency directly. After a CPU node finishes, the CPU updates the shared dependency states of its successors using lightweight atomic operations. Before starting the next node in its queue, the CPU checks whether that node is ready and waits only if some predecessor is still unfinished. In

---

**Algorithm 1** Iterative Search for Partition-Aware Scheduling

---

**Inputs:** search budget  $B$ , operator DAG  $G = (V, E)$ , devices  $\mathcal{D}$ , candidate plans  $\{\Pi_i\}_{i \in V}$ , partitionable operators  $V_p = \{i \in V : |\Pi_i| > 1\}$ , latency predictor  $\hat{\tau}$ , scheduling routine SCHEDULE, operator sampler SAMPLEOPERATORS, update generator GENERATEUPDATES

**Outputs:** Final partition-plan assignment and schedule

```
1  $\pi \leftarrow$  initial partition  $\{\pi_i\}_{i \in V}$ 
2  $S \leftarrow$  SCHEDULE( $G, \pi, \hat{\tau}, \mathcal{D}$ )
3 for  $t = 1, \dots, B$ 
4    $V_{\text{samp}} \leftarrow$  SAMPLEOPERATORS( $G, V_p, \pi, S$ )
5    $\pi^* \leftarrow \pi, S^* \leftarrow S, \Delta^* \leftarrow 0$ 
6   for each  $i \in V_{\text{samp}}$ 
7      $\mathcal{M}_i \leftarrow$  GENERATEUPDATES( $i, \Pi_i, \pi_i$ )
8     for each  $m = (i, \tilde{\pi}_i) \in \mathcal{M}_i$ 
9       Construct  $\pi_m$  by replacing  $\pi_i$  with  $\tilde{\pi}_i$ 
10       $S_m \leftarrow$  SCHEDULE( $G, \pi_m, \hat{\tau}, \mathcal{D}$ )
11       $\Delta_m \leftarrow T(S) - T(S_m)$ 
12      if  $\Delta_m > \Delta^*$ 
13         $\pi^* \leftarrow \pi_m, S^* \leftarrow S_m, \Delta^* \leftarrow \Delta_m$ 
14   if  $\Delta^* = 0$ 
15     return  $\pi, S$ 
16    $\pi \leftarrow \pi^*, S \leftarrow S^*$ 
17 return  $\pi, S$ 
```

---

our measurements, this CPU-side synchronization overhead is negligible compared with operator execution time.

On the other hand, the GPU requires an additional step because GPU work is submitted asynchronously through a command queue. The CPU can enqueue GPU computation kernels in the scheduled order, but dependency updates that occur after a GPU computation must also respect this GPU queue order. Therefore, when a GPU node needs to publish its completion or wait for work produced by CPU, the runtime execution inserts a small GPU-side synchronization kernel. This kernel only updates or checks the shared dependency state and is inserted after each GPU operator computation kernel. Although these GPU-side synchronization kernels perform little computation, launching a GPU kernel still incurs dispatch overhead because the CPU must submit the kernel to the GPU command queue. Accordingly, our schedule evaluator accounts for the overhead of dispatching a synchronization kernel by adding a small constant to GPU execution time. In our implementation, this constant is set to 10  $\mu\text{s}$ , based on the average dispatch overhead measured in our experiments.

## B. Supplemental Framework Details

This appendix provides additional details of our iterative search, including (1) pseudo-code corresponding to our iterative search, and (2) a feature table for building our latency predictors. Algorithm 1 summarizes our iterative search framework. Lines 1–2 initialize the partition plan (e.g., no partitioning in our default setting) and schedule. Line 4 selects a subset of partitionable operators on which to spend the search budget of each iteration; our default implementation uses schedule slack to prioritize operators near the critical path, as described in Section 5.3. Lines 7–10 generate candidate replacement plans and reschedule the resulting graph. Lines 11–13 keep the best improved update in the current iteration. If no sampled update improves the makespan, line 15 terminates the search. Otherwise, line 16 accepts the best candidate assignment and continues to the next iteration. We set the search budget  $B$  to a large value that it is never reached in our

| Operator    | Device    | Features                                                                                                                                                                                                                                                              |
|-------------|-----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Convolution | GPU       | Input/output height, input/output width, input/output channels, filter shape, stride, group count, input/output sizes, filter size, FLOPs, grid size ( $x, y, z$ -dims), workgroup size ( $x, y, z$ -dims), workgroup count ( $x, y, z$ -dims), total workgroup count |
|             | CPU       | Input/output height, input/output width, input/output channels, filter shape, stride, group count, input/output sizes, filter size, FLOPs                                                                                                                             |
| Linear      | GPU       | Input/output height, input/output width, input/output channels, input/output sizes, weight size, FLOPs, grid size ( $x, y, z$ -dims), workgroup size ( $x, y, z$ -dims), workgroup count ( $x, y, z$ -dims), total workgroup count                                    |
|             | CPU       | Input/output height, input/output width, input/output channels, input/output sizes, weight size, FLOPs                                                                                                                                                                |
| Pooling     | CPU / GPU | Input/output height, input/output width, input/output channels, input/output sizes, kernel shape, stride                                                                                                                                                              |
| Others      | CPU / GPU | Input/output height, input/output width, input/output channels, input/output sizes                                                                                                                                                                                    |

Table B.7: Features used for training GBDT predictors across different operators and devices.

experiments; it serves only as a safeguard against unusually long searches, while termination is normally determined by the absence of improving candidate updates.

Table B.7 summarizes the operator, partition, and GPU dispatch features used by the latency predictors.

### C. Supplemental Data

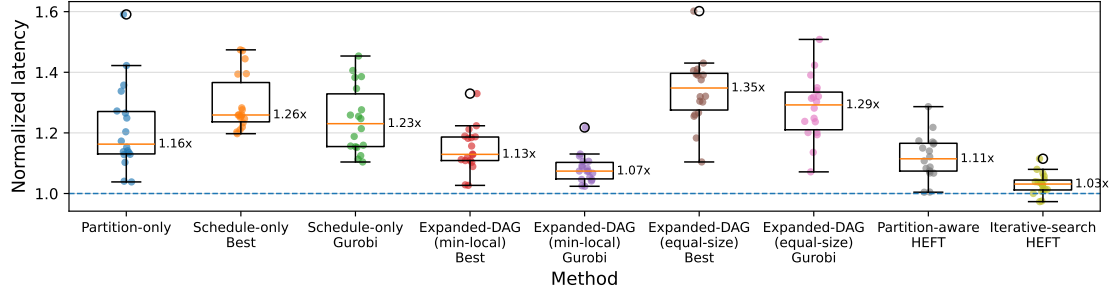
This appendix provides supplemental data for the information and results provided in the main text. Table C.8 summarizes the full specifications of models in our study. Table C.9 reports the complete simulated latency results for all baselines across the four mobile platforms. Fig. C.7 depicts the distributions of simulated latencies across the 18 models on the three platforms not included in the main text. Table C.10 reports the complete real-device latency measurements for all baselines across the four mobile platforms. Fig. C.8 depicts the distributions of measured latencies across the 18 models on the three platforms not included in the main text. Table C.11 compares TensorFlow Lite model initialization time with the latency-prediction and iterative-scheduling overheads. Tables C.12 and C.13 provide the complete ablation results for iterative-search design choices and search-budget sensitivity.

| Model              | # Conv | # Ops |
|--------------------|--------|-------|
| inceptionv3        | 94     | 170   |
| inceptionv4        | 149    | 271   |
| inceptionresnetv1  | 132    | 231   |
| inceptionresnetv2  | 244    | 429   |
| squeezenet_v1_0    | 26     | 47    |
| squeezenet_v1_1    | 26     | 46    |
| squeezeresnet_v1_0 | 26     | 51    |
| squeezeresnet_v1_1 | 26     | 50    |
| peleenet           | 113    | 144   |
| hrnet_w18_small_v1 | 91     | 149   |
| hrnet_w18_small_v2 | 164    | 279   |
| hrnetv2_w18        | 325    | 535   |
| hrnetv2_w30        | 325    | 535   |
| hrnetv2_w32        | 325    | 535   |
| hrnetv2_w40        | 325    | 535   |
| hrnetv2_w44        | 325    | 535   |
| hrnetv2_w48        | 325    | 535   |
| hrnetv2_w64        | 325    | 535   |

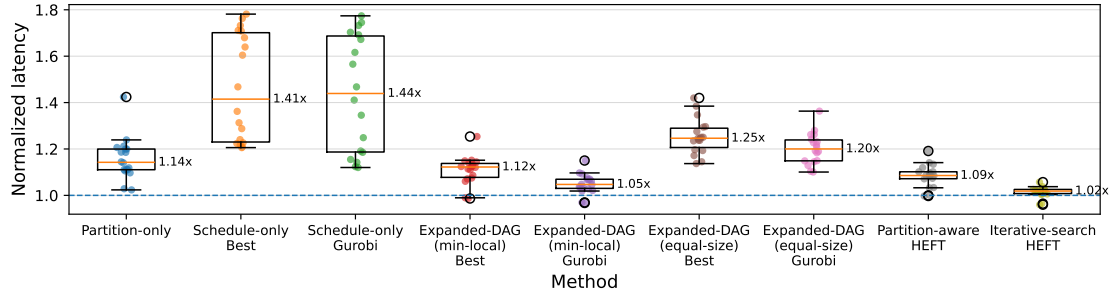
Table C.8: Neural-network workloads. All models are exported to TFLite and evaluated with input resolution 224×224.

| Approach                         | Variant           | OnePlus 11 |       | Motorola 2022 |       | Pixel 4 |       | Pixel 5 |       |
|----------------------------------|-------------------|------------|-------|---------------|-------|---------|-------|---------|-------|
|                                  |                   | Avg        | Worst | Avg           | Worst | Avg     | Worst | Avg     | Worst |
| Joint-search                     | Gurobi            | 1.00       | 1.00  | 1.00          | 1.00  | 1.00    | 1.00  | 1.00    | 1.00  |
| Single-device                    | GPU               | 1.64       | 2.35  | 1.61          | 2.23  | 2.10    | 2.94  | 2.13    | 2.44  |
|                                  | CPU (L)           | 7.57       | 10.05 | 7.71          | 9.46  | 6.31    | 7.27  | 4.87    | 6.30  |
|                                  | CPU (M)           | 6.83       | 8.89  | 6.39          | 7.45  | 3.81    | 4.33  | 5.16    | 6.51  |
| Partition-only                   | Min-local-latency | 1.20       | 1.39  | 1.22          | 1.59  | 1.16    | 1.42  | 1.11    | 1.28  |
| Schedule-only                    | HEFT              | 1.29       | 1.42  | 1.31          | 1.47  | 1.47    | 1.78  | 1.51    | 1.84  |
|                                  | Lookahead-HEFT    | 1.29       | 1.42  | 1.31          | 1.47  | 1.47    | 1.78  | 1.52    | 1.84  |
|                                  | PEFT              | 1.29       | 1.45  | 1.33          | 1.56  | 1.49    | 1.82  | 1.53    | 1.92  |
|                                  | LDCP              | 1.28       | 1.44  | 1.30          | 1.47  | 1.47    | 1.78  | 1.52    | 1.84  |
|                                  | CEFT              | 1.49       | 1.61  | 1.50          | 1.68  | 1.66    | 2.08  | 1.69    | 2.07  |
|                                  | PSLS              | 1.28       | 1.49  | 1.31          | 1.57  | 1.49    | 1.80  | 1.56    | 1.85  |
|                                  | Gurobi            | 1.22       | 1.42  | 1.24          | 1.45  | 1.44    | 1.77  | 1.49    | 1.84  |
| Expanded-DAG (Min-local-latency) | HEFT              | 1.15       | 1.26  | 1.15          | 1.32  | 1.11    | 1.26  | 1.09    | 1.20  |
|                                  | Lookahead-HEFT    | 1.15       | 1.27  | 1.15          | 1.32  | 1.11    | 1.25  | 1.09    | 1.20  |
|                                  | PEFT              | 1.15       | 1.26  | 1.15          | 1.33  | 1.24    | 1.35  | 1.08    | 1.22  |
|                                  | LDCP              | 1.16       | 1.27  | 1.16          | 1.36  | 1.11    | 1.25  | 1.09    | 1.22  |
|                                  | CEFT              | 1.35       | 1.56  | 1.35          | 1.54  | 1.29    | 1.45  | 1.27    | 1.44  |
|                                  | PSLS              | 1.17       | 1.33  | 1.17          | 1.36  | 1.11    | 1.23  | 1.09    | 1.21  |
|                                  | Gurobi            | 1.09       | 1.15  | 1.08          | 1.22  | 1.05    | 1.15  | 1.04    | 1.13  |
| Expanded-DAG (Equal-size)        | HEFT              | 1.34       | 1.52  | 1.34          | 1.60  | 1.25    | 1.42  | 1.18    | 1.36  |
|                                  | Lookahead-HEFT    | 1.34       | 1.52  | 1.34          | 1.60  | 1.25    | 1.42  | 1.18    | 1.36  |
|                                  | PEFT              | 1.40       | 1.55  | 1.38          | 1.59  | 1.30    | 1.44  | 1.22    | 1.39  |
|                                  | LDCP              | 1.35       | 1.53  | 1.35          | 1.61  | 1.27    | 1.45  | 1.19    | 1.38  |
|                                  | CEFT              | 1.50       | 1.70  | 1.48          | 1.80  | 1.35    | 1.55  | 1.28    | 1.50  |
|                                  | PSLS              | 1.37       | 1.54  | 1.37          | 1.58  | 1.29    | 1.42  | 1.21    | 1.36  |
|                                  | Gurobi            | 1.30       | 1.45  | 1.28          | 1.51  | 1.20    | 1.36  | 1.14    | 1.32  |
| Partition-aware heuristic        | HEFT              | 1.12       | 1.21  | 1.12          | 1.29  | 1.08    | 1.19  | 1.07    | 1.18  |
| Iterative search                 | HEFT              | 1.04       | 1.08  | 1.03          | 1.11  | 1.01    | 1.06  | 1.02    | 1.05  |
|                                  | PEFT              | 1.05       | 1.12  | 1.04          | 1.12  | 1.03    | 1.08  | 1.03    | 1.07  |
|                                  | PSLS              | 1.05       | 1.13  | 1.04          | 1.14  | 1.03    | 1.08  | 1.03    | 1.06  |
|                                  | Lookahead-HEFT    | 1.04       | 1.08  | 1.03          | 1.12  | 1.01    | 1.06  | 1.02    | 1.05  |

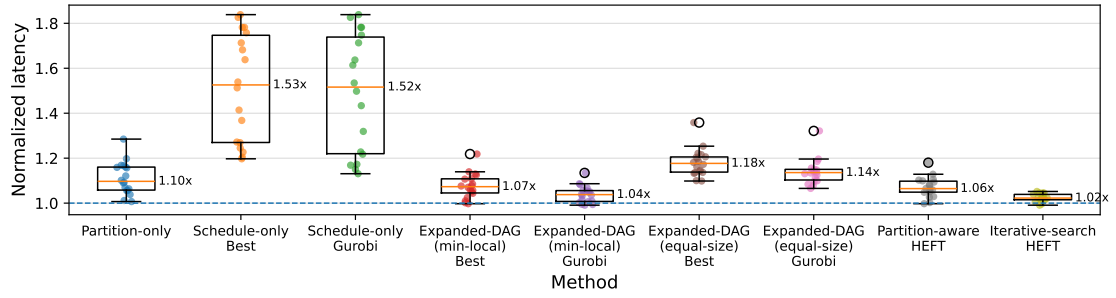
Table C.9: Simulation results across four phones. Latency is normalized to JOINT-GUROBI; lower is better.



(a) Motorola 2022



(b) Pixel 4



(c) Pixel 5

Figure C.7: Distribution of simulation latency across 18 models, normalized to JOINT-GUROBI (annotated values report the median)

| Approach                         | Variant           | OnePlus 11 |       | Motorola 2022 |       | Pixel 4 |       | Pixel 5 |       |
|----------------------------------|-------------------|------------|-------|---------------|-------|---------|-------|---------|-------|
|                                  |                   | Avg        | Worst | Avg           | Worst | Avg     | Worst | Avg     | Worst |
| Joint-search                     | Gurobi            | 1.00       | 1.00  | 1.00          | 1.00  | 1.00    | 1.00  | 1.00    | 1.00  |
| Single-device                    | GPU               | 1.62       | 1.95  | 1.60          | 2.01  | 1.94    | 2.46  | 1.93    | 2.10  |
|                                  | CPU (L)           | 7.68       | 11.72 | 7.45          | 10.29 | 6.60    | 9.11  | 4.45    | 5.54  |
|                                  | CPU (M)           | 5.49       | 8.64  | 5.21          | 7.32  | 3.21    | 3.75  | 4.71    | 5.76  |
| Partition-only                   | Min-local-latency | 1.37       | 2.92  | 1.25          | 1.61  | 1.20    | 1.93  | 1.12    | 1.52  |
| Schedule-only                    | HEFT              | 1.16       | 1.73  | 1.08          | 1.30  | 1.26    | 1.45  | 1.33    | 1.66  |
|                                  | Lookahead-HEFT    | 1.25       | 2.67  | 1.09          | 1.30  | 1.26    | 1.44  | 1.33    | 1.66  |
|                                  | PEFT              | 1.25       | 1.73  | 1.12          | 1.31  | 1.25    | 1.52  | 1.36    | 1.76  |
|                                  | LDCP              | 1.15       | 1.33  | 1.08          | 1.26  | 1.25    | 1.45  | 1.34    | 1.66  |
|                                  | CEFT              | 1.33       | 1.50  | 1.27          | 1.52  | 1.39    | 1.67  | 1.49    | 2.00  |
|                                  | PSLS              | 1.16       | 1.36  | 1.10          | 1.31  | 1.30    | 1.45  | 1.37    | 1.66  |
|                                  | Gurobi            | 1.08       | 1.22  | 1.07          | 1.25  | 1.24    | 1.48  | 1.31    | 1.68  |
| Expanded-DAG (Min-local-latency) | HEFT              | 1.38       | 2.56  | 1.17          | 1.40  | 1.13    | 1.83  | 1.10    | 1.49  |
|                                  | Lookahead-HEFT    | 1.39       | 2.60  | 1.17          | 1.40  | 1.13    | 1.86  | 1.10    | 1.50  |
|                                  | PEFT              | 1.42       | 2.86  | 1.19          | 1.32  | 1.22    | 1.75  | 1.08    | 1.46  |
|                                  | LDCP              | 1.34       | 1.61  | 1.16          | 1.31  | 1.13    | 1.86  | 1.11    | 1.69  |
|                                  | CEFT              | 1.77       | 3.05  | 1.41          | 1.69  | 1.26    | 2.06  | 1.23    | 1.59  |
|                                  | PSLS              | 1.37       | 1.95  | 1.21          | 1.45  | 1.13    | 1.73  | 1.11    | 1.44  |
|                                  | Gurobi            | 1.19       | 1.33  | 1.13          | 1.36  | 1.09    | 1.69  | 1.07    | 1.53  |
| Expanded-DAG (Equal-size)        | HEFT              | 1.73       | 2.50  | 1.44          | 1.81  | 1.22    | 1.90  | 1.20    | 1.82  |
|                                  | Lookahead-HEFT    | 1.68       | 2.08  | 1.43          | 1.80  | 1.22    | 1.89  | 1.21    | 1.84  |
|                                  | PEFT              | 1.85       | 2.24  | 1.50          | 1.82  | 1.26    | 1.94  | 1.22    | 1.84  |
|                                  | LDCP              | 1.75       | 2.34  | 1.45          | 1.72  | 1.22    | 1.91  | 1.21    | 1.89  |
|                                  | CEFT              | 1.86       | 2.29  | 1.57          | 2.01  | 1.28    | 1.83  | 1.26    | 1.91  |
|                                  | PSLS              | 1.76       | 2.14  | 1.49          | 1.81  | 1.29    | 1.85  | 1.23    | 1.90  |
|                                  | Gurobi            | 1.56       | 1.83  | 1.63          | 2.16  | 1.20    | 1.88  | 1.18    | 1.88  |
| Partition-aware heuristic        | HEFT              | 1.36       | 1.69  | 1.18          | 1.39  | 1.11    | 1.78  | 1.10    | 1.60  |
| Iterative search                 | HEFT              | 1.02       | 1.11  | 1.01          | 1.10  | 1.00    | 1.29  | 1.04    | 1.15  |
|                                  | PEFT              | 1.12       | 1.44  | 1.07          | 1.23  | 1.00    | 1.57  | 1.08    | 1.42  |
|                                  | PSLS              | 1.05       | 1.12  | 1.03          | 1.20  | 1.02    | 1.30  | 1.06    | 1.19  |
|                                  | Lookahead-HEFT    | 1.06       | 1.54  | 1.02          | 1.13  | 1.01    | 1.21  | 1.04    | 1.13  |

Table C.10: Full measurement results across four phones. Latency is normalized to JOINT-GUROBI; lower is better

| Model              | Prediction overhead (ms) | Scheduling overhead (ms) | Initialization time (ms) |
|--------------------|--------------------------|--------------------------|--------------------------|
| inceptionv3        | 135                      | 303                      | 1048                     |
| inceptionv4        | 130                      | 420                      | 1534                     |
| inceptionresnetv1  | 88                       | 249                      | 1100                     |
| inceptionresnetv2  | 123                      | 627                      | 1783                     |
| squeezenet_v1_0    | 62                       | 25                       | 479                      |
| squeezenet_v1_1    | 51                       | 24                       | 482                      |
| squeezeresnet_v1_0 | 60                       | 24                       | 504                      |
| squeezeresnet_v1_1 | 51                       | 24                       | 504                      |
| peleenet           | 81                       | 265                      | 778                      |
| hrnet_w18_small_v1 | 122                      | 311                      | 1424                     |
| hrnet_w18_small_v2 | 134                      | 446                      | 1625                     |
| hrnetv2_w18        | 134                      | 855                      | 1817                     |
| hrnetv2_w30        | 163                      | 1046                     | 2204                     |
| hrnetv2_w32        | 124                      | 1084                     | 2272                     |
| hrnetv2_w40        | 156                      | 1140                     | 2574                     |
| hrnetv2_w44        | 185                      | 1127                     | 2692                     |
| hrnetv2_w48        | 161                      | 1339                     | 2955                     |
| hrnetv2_w64        | 148                      | 1116                     | 3675                     |

Table C.11: Overhead compared with TFLite model initialization time

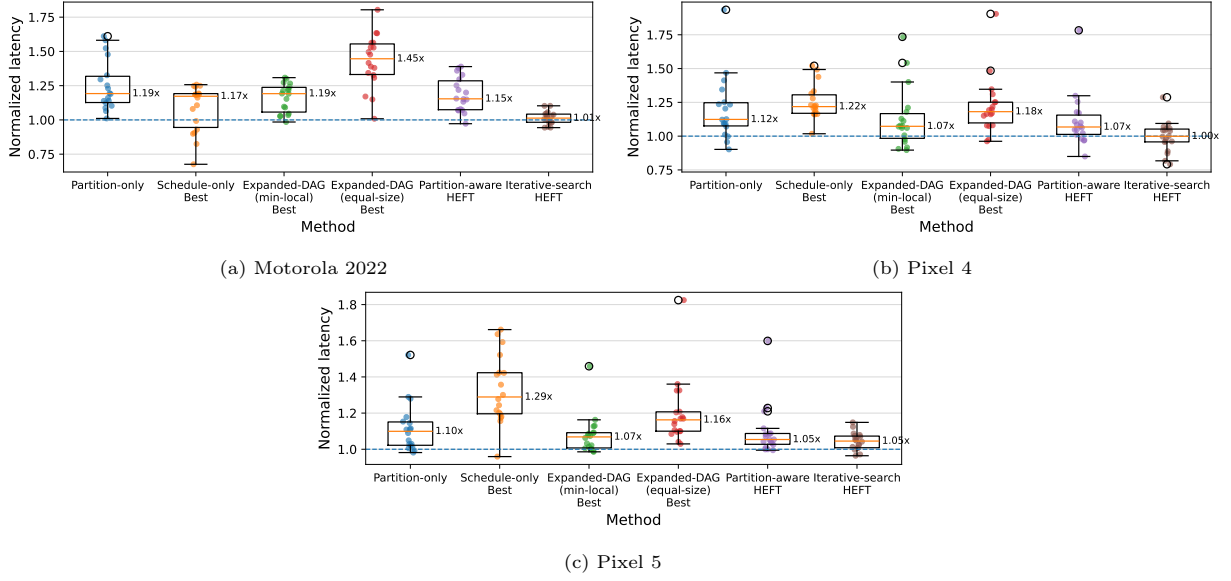


Figure C.8: Distribution of measured end-to-end latency across 18 models, normalized to JOINT-GUROBI (annotated values report the median)

|                  | Configuration                 | Normalized latency | Search cost (s) |
|------------------|-------------------------------|--------------------|-----------------|
| <b>Default</b>   | <b>HEFT, split=3, staging</b> | <b>1.03</b>        | <b>10.8</b>     |
| Heuristics       | PEFT                          | 1.04               | 12.3            |
|                  | PSLS                          | 1.04               | 16.7            |
|                  | Lookahead-HEFT                | 1.03               | 28.3            |
|                  | Min-local-latency Init        | 1.05               | 17.7            |
| Staging          | no staging                    | 1.14               | 24.7            |
| Number of splits | split=2                       | 1.06               | 9.0             |
|                  | split=3                       | 1.03               | 10.8            |
|                  | split=4                       | 1.05               | 12.0            |
|                  | split=5                       | 1.06               | 15.2            |
|                  | split=6                       | 1.07               | 22.5            |
|                  | split=7                       | 1.08               | 43.5            |
|                  | split=8                       | 1.09               | 103.9           |

Table C.12: Ablation and split-sensitivity results for iterative search. Lower normalized latency (on OnePlus 11) and lower search cost are preferred.

| Timeout (s) | Normalized latency | Search cost (s) |
|-------------|--------------------|-----------------|
| Unlimited   | 1.03               | 10.78           |
| 0.75        | 1.03               | 9.30            |
| 0.5         | 1.04               | 7.96            |
| 0.4         | 1.04               | 7.12            |
| 0.3         | 1.04               | 6.08            |
| 0.2         | 1.05               | 4.54            |
| 0.1         | 1.07               | 2.61            |
| 0.05        | 1.09               | 1.64            |
| 0.04        | 1.09               | 1.50            |
| 0.03        | 1.12               | 0.89            |
| 0.02        | 1.13               | 0.75            |
| 0.01        | 1.14               | 0.64            |

Table C.13: Timeout-sensitivity results for iterative search. Lower timeout values reduce search cost but can degrade normalized latency (measured on OnePlus 11).