PARALLAX: RUNTIME PARALLELIZATION FOR OPERATOR FALLBACKS IN HETEROGENEOUS EDGE SYSTEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

The growing demand for real-time DNN applications on edge devices necessitates faster inference of increasingly complex models. Although many devices include specialized accelerators (e.g., mobile GPUs), dynamic control-flow operators and unsupported kernels often fall back to CPU execution. Existing frameworks handle these fallbacks poorly, leaving CPU cores idle and causing high latency and memory spikes. We introduce Parallax, a framework that accelerates mobile DNN inference without model refactoring or custom operator implementations. Parallax first partitions the computation DAG to expose parallelism, then employs branch-aware memory management with dedicated arenas and buffer reuse to reduce runtime footprint. An adaptive scheduler executes branches according to device memory constraints, meanwhile, fine-grained subgraph control enables heterogeneous inference of dynamic models. By evaluating on representative DNNs across different mobile devices, Parallax achieves up to 46% latency reduction, maintains controlled memory overhead (average 26.5%), and delivers up to 30% energy savings compared with state-of-the-art frameworks, offering improvements aligned with the responsiveness demands of real-time mobile inference.

1 INTRODUCTION

Efficient on-device inference of deep neural networks (DNNs) has become an integral part of modern mobile computing (Xu et al., 2025; Cai et al., 2019). As mobile applications shift from conventional recognition tasks to more sophisticated workloads, they increasingly contend with dynamic tensor shapes and control flow (Shen et al., 2021). For example, object detectors predict a variable number of bounding boxes, and automatic speech recognition (ASR) employs beam search to dynamically adjust outputs. Meanwhile, continued innovation in DNN architectures introduces new operators that may lack universal support across hardware backends (Ghodrati et al., 2024).

State-of-the-art (SOTA) mobile inference frameworks, such as ONNXRuntime (ORT) (Bai et al., 2019), ExecuTorch (Foundation, 2024), and TensorFlow Lite (TFLite) (David et al., 2021), provide hardware delegation (GPUs, TPUs, NPUs) and compiler optimizations like operator fusion. However, they assume static graphs and predefined operators. When faced with dynamic or unsupported operations, frameworks fall back to CPU execution. TFLite often offloads the entire graph (Niu et al., 2024), while ORT and ExecuTorch may reject or partially offload graphs with control flow (Shen et al., 2021), leading to extra data transfers, memory copies and synchronization stalls (Zhang et al., 2022). Recent solutions handle inference overhead from various perspective. ASPEN (Park et al., 2023) improves CPU-only operator-level parallelism by converting models into fine-grained tile graphs for dynamic scheduling. NN-Stretch (Wei et al., 2023) and CoDL (Jia et al., 2022) accelerate static graphs by scheduling different model segments to run concurrently on heterogeneous processors. SoD^2 (Niu et al., 2024) addresses dynamic shapes by generating fixed-shape kernel variants offline based on operator dependencies and selecting the appropriate version at runtime. MikPoly (Yu et al., 2024) follows a similar approach by compiling shape-specialized micro-kernels and linking only those matching observed tensor sizes. On the hardware side, Tandem Processor (Ghodrati et al., 2024) integrates a small programmable core alongside the main accelerator, allowing unsupported or irregular operators to run on-chip and avoid costly fallbacks to the host CPU.

Table 1: Parallax uniquely handles dynamic operators, avoids model/kernel modifications, supports heterogeneous inference and accelerates parallel CPU fallback execution.

Method	Handles Dynamic Ops	No Model Refactoring / Kernel Creation	Heterogeneous Inference	Parallelization Acceleration
SoD^2 (Niu et al., 2024)	✓	✗	✓	Not Discussed
MikPoly (Yu et al., 2024)	✓	✗	✓	Not Discussed
NN-Stretch (Wei et al., 2023)	✗	✗	✓	✓
CoDL (Jia et al., 2022)	✗	✗	✓	✓
ASPEN (Park et al., 2023)	✗	✗	✗	✓
Parallax (Ours)	✓	✓	✓	✓

While these methods offer promising improvements, they have practical limitations. Static code generation and micro-kernel libraries require continuous maintenance, as evolving models and operators can invalidate precompiled kernels (Niu et al., 2024; Yu et al., 2024). Hardware-centric solutions depend on new silicon and remain incompatible with most deployed mobile devices (Ghodrati et al., 2024). Many other approaches like NN-Stretch require model refactoring, custom operator maintenance, or constrain model flexibility to preserve accelerator compatibility (Wei et al., 2023; Jia et al., 2022). Thus, a clear need persists for an approach that utilizes readily available on-device resources to offset performance penalties from dynamic operations and unsupported operators.

To meet this need, we present Parallax, a scheduling framework that accelerates mobile DNN inference without model changes. Parallax non-invasively traverses the computation Directed Acyclic Graph (DAG), partitions it to expose parallelism, and assigns dedicated memory arenas per branch using efficient intra- and inter-branch buffer sharing. It then performs static shape inference and linear liveness scanning to estimate each branch's peak memory. Branches are scheduled in parallel if within available system RAM (with safety margin), otherwise sequentially to prevent out-of-memory (OOM) issues. We evaluate Parallax on five DNNs across three mobile devices. Compared to existing solutions (Table 1), Parallax shows unique advantages. Furthermore, results demonstrate that Parallax reduces inference latency by up to 46%, maintains controlled memory overhead (26.5% average increase), and delivers up to 30% energy savings, all without requiring model refactoring or custom operator implementations. Our key contributions are:

- **Graph Analysis and Parallel Scheduling**: A non-invasive DAG analysis that identifies and schedules heterogeneous subgraph execution without modifying the model.
- **Branch-Aware Memory Management**: Dedicated memory arenas per branch with region-based allocation and buffer reuse to eliminate contention and minimize footprint.
- **Device-Adaptive Scheduling**: A resource-constrained algorithm that enforces an adaptive memory budget to maximize safe parallel CPU utilization.

2 Related Work

Offline Model Compression. A common approach to optimize DNN inference on resource-constrained devices is offline model compression. Techniques such as quantization reduce numerical precision (e.g., to INT8) to shrink model size and accelerate computation (Kim et al., 2022; Yao et al., 2020), while pruning removes less important weights or channels to lower FLOPs (Jiang et al., 2022). Operator fusion merges adjacent kernels to reduce memory access and boost throughput (Niu et al., 2021). Architecture-level designs like MobileNets (Howard, 2017) and MobileBERT (Sun et al., 2020) offer efficiency through depthwise convolutions and compact Transformer blocks. Recent innovations, such as LookupFFN (Zeng et al., 2023), replace matrix multiplications with table lookups. Although these methods reduce latency and memory, they often require retraining or calibration data, may degrade accuracy, and often involve model tuning. Critically, they do not address runtime penalties from dynamic behavior or unsupported operators. Parallax complements these approaches by directly handling fallback scenarios at runtime without further model changes.

Hardware Acceleration and Heterogeneous Execution. Standard mobile frameworks (e.g., TFLite, ORT, ExecuTorch) use hardware delegation and compiler optimizations to accelerate DNN inference, but rely on static graphs. To improve acceleration, prior work explored co-execution and graph partitioning. ASPEN (Park et al., 2023) dynamically schedules fine-grained tiles for CPU-only parallelism. μLayer (Kim et al., 2019) distributes layers across CPU and GPU based

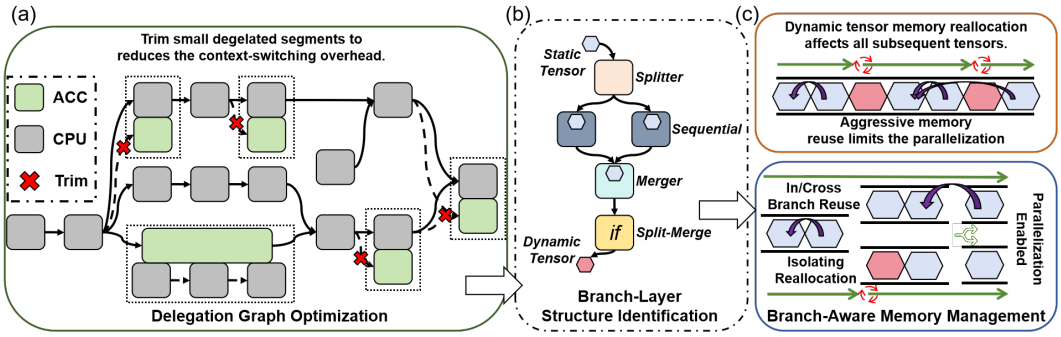

Figure 1: Parallax overview. (a) Delegation graph optimization trims small delegated segments to reduce synchronization overhead. (b) Branch-layer structure identification classifies nodes to expose parallel execution paths. (c) Branch-aware memory management isolates branch arenas to avoid contention and safely reuse arenas, enabling efficient parallel execution under dynamic tensor shapes.

on workload and datatype performance; CoDL (Jia et al., 2022) enables intra-operator parallelism across processors. NN-Stretch (Wei et al., 2023) restructures sequential DNNs into parallel branches for heterogeneous cores, and BAND (Jeong et al., 2022) schedules multiple DNNs across systems. OPTiC (Wang et al., 2018) optimizes CPU-GPU partitioning and core frequencies under thermal constraints. While effective for static models, these methods often depend on heuristic-based partitioning and require careful coordination across separate memory spaces, which can introduce complexities and overhead.

Dynamic Operations and Memory Management. To address static graph limitations, SoD^2 (Niu et al., 2024) uses symbolic analysis to infer tensor shapes and generate optimized code paths, while MikPoly (Yu et al., 2024) pre-compiles shape-specialized micro-kernels for runtime linking. Although these methods offer flexible solutions, they rely on offline analysis and dispatch generation that require regeneration as models or operators evolve. Hardware-centric solutions like Tandem Processor (Ghodrati et al., 2024) integrate programmable cores for irregular operations, but such designs require new silicon and are incompatible with the existing edge device ecosystem. SOTA framework memory allocators (David et al., 2021; Bai et al., 2019) minimize memory via aggressive buffer reuse but create data dependencies that block branch-level parallelism. Parallax overcomes this by assigning dedicated memory arenas per branch with efficient reuse within and across branches, minimizing contention and enabling reliable parallelism. Combined with lightweight memory estimation and adaptive scheduling, Parallax maximizes concurrency within tight memory budgets and remains adaptable across diverse models and edge platforms.

3 PARALLAX

Problem Formulation. Mobile DNN inference increasingly encounters dynamic graphs, creating two major challenges: **(i) Parallel execution disruption.** Mixed graphs with CPU fallbacks and delegateable segments are executed sequentially, causing repeated host-device transfers, synchronization delays and idle CPU cores. **(ii) Inefficient memory handling.** Static memory planners allocate global heaps based on fixed shapes. When shapes are resolved only at runtime (e.g., conditional branches, dynamic decoding), allocators must invalidate and reallocate large regions, adding overhead and limiting parallelism. Existing methods either overlook these problems or rely on offline preprocessing and static kernel generation, which struggle to generalize across evolving models.

Parallax addresses these challenges through a compiler-level framework. It applies graph analysis, optimized delegation and fallback handling to expose parallelizable branches and reduce synchronization overhead. The branch-aware memory allocation further minimizes unnecessary reallocations, supporting efficient execution under dynamic shapes. The design comprises three coordinated stages (Fig. 1), detailed in the following subsections.

3.1 GRAPH ANALYSIS AND PARTITIONING

As the foundation step, Parallax addresses an underexplored challenge: restructuring complex, heterogeneous DNN computation graphs to expose independent, parallelizable execution paths. Without this, fallback regions remain fragmented and tightly coupled to delegate segments, limiting parallelism and efficient memory planning. We preprocess the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ are operations and $\mathcal{E}$ are tensor dependencies, to (i) identify accelerator-worthy regions, (ii) expose branch-level parallelism in mixed CPU-delegate graphs, and (iii) generate per-branch workload metadata for later stages.

Optimized Delegate Partitioning. Identifying accelerator-worthy regions within dynamic, heterogeneous DNN graphs is both challenging and essential: naively offloading small or low-compute subgraphs causes kernel-launch and data-transfer overheads that negate any acceleration gains. Parallax applies an analytical cost model, grounded in representative mobile SoC parameters, to prune inefficient delegate candidates (Fig. 1a).

We characterize each candidate region S by (1) its operation count $N = |V(S)|$, (2) total compute $F = \sum_{v \in S} \mathrm{FLOPs}(v)$ MACs (Multiply–Accumulate operations) estimated using operator-level FLOPs (Appendix A) and (3) boundary transfer size $B = \sum_{T \in \partial S} \mathrm{numel}(T) \times \mathrm{sizeof}(\mathrm{dtype})$, where ∂S is the set of boundary tensors between S and the rest of the graph, $\mathrm{numel}(T)$ is the number of elements in tensor T, and $\mathrm{sizeof}(\mathrm{dtype})$ is the byte width of the tensor's data type. A region is offloaded only if:

$$N \geq 3, \quad F \geq 1 \times 10^9, \quad \frac{B}{F} \leq 0.1.$$

The thresholds arise from requiring that total offload time

$$T_{\text{offload}} = L + \frac{F}{R_{\text{acc}}} + \frac{B}{B_{\text{bw}}}$$

be less than CPU execution time $\frac{F}{R_{\text{cpu}}}$, where L is the accelerator dispatch latency, R_{acc} is the accelerator's peak throughput (in MACs/s), and B_{bw} is the peak memory bandwidth (in bytes/s). This simplifies (Appendix B) to:

$$F > L R_{\text{acc}}, \quad \frac{B}{F} < \frac{B_{\text{bw}}}{R_{\text{acc}}}.$$

We adopt representative values from published hardware reports: (i)median mobile GPU dispatch latency $L = 0.2\,\mathrm{ms}$, typical for NNAPI burst mode across devices (Kim et al., 2021); (ii) peak accelerator throughput $R_{\text{acc}} = 2.6 \times 10^{13}\,\mathrm{MAC/s}$, matching Qualcomm Snapdragon 8 Gen 1 specifications (Qualcomm Technologies, Inc., 2021); (iii) memory bandwidth $B_{\text{bw}} = 51.2\,\mathrm{GB/s}$, reflecting LPDDR5 in 2021–2022 flagship SoCs (Qualcomm Technologies, Inc., 2020). Substituting yields $F > 5.2 \times 10^9$ MACs and $\frac{B}{F} < 0.002$ bytes/MAC. We relax these to $F \geq 1 \times 10^9$ and $\frac{B}{F} \leq 0.1$ to account for device variability, kernel inefficiencies and runtime scheduling fluctuations.

Branch and Layer Extraction. After delegate partitioning, the graph intermixes accelerator-offload and CPU-fallback nodes whose dependencies limit independent scheduling. Parallax decomposes the graph into a Branch-Layer structure that proceeds in two stages. First, Parallax classifies each node by connectivity: *Sequential* (`in`=1, `out`=1), *Splitter* (`in`=1, `out`>1), *Merger* (`in`>1, `out`=1), or *Split-Merge* (`in`>1, `out`>1). Delegate regions are treated as indivisible units, while control-flow operators (e.g., `If`, `While`) are marked Split-Merge to ensure sequential correctness. Maximal branches are then extracted via traversal as shown in Algorithm 1. Finally, branches are grouped into layers via a topological sort over the branch dependency map 2. This entire process runs in $O(|\mathcal{V}| + |\mathcal{E}|)$ time and outputs $\mathcal{B}$ and $\mathcal{L}$, which enable efficient parallel scheduling and memory management in subsequent stages (Fig. 1b). (Expanded algorithm listings are provided in Appendix C.)

Further Refinement. Parallax further refines its execution plan by enforcing minimal workloads for parallelizable layers and balancing workloads across branches within each layer. To quantify these workloads, we reuse the metrics N and F. For a layer's branches to execute in parallel, each branch must satisfy:

$$N > 2, \quad \text{and} \quad \frac{F_{\max}}{F_{\min}} \leq \beta$$

Algorithm 1 Node Classification and Branch Identification

Require: Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Ensure: Branch set $\mathcal{B}$

```
for all v ∈ V do
    compute (d_in(v), d_out(v))
    label v as Sequential / Splitter / Merger / Split-Merge
end for
B ← ∅, V_vis ← ∅
for all v ∈ V if v ∉ V_vis and not Merger/Split-Merge do
    b ← []
    while v Sequential and v ∉ V_vis do
        append v to b; mark v visited; v ← successor
    end while
    add b to B
end for
```

Algorithm 2 Layer Construction via Topological Sort

Require: Branches $\mathcal{B}$, dependencies $\mathcal{E}$
Ensure: Layers $\mathcal{L}$

```
compute d[b] ← in-degree for b ∈ B
initialize empty queue Q ← {b | d[b] = 0}
initialize empty list L ← []
while Q ≠ ∅ do
    layer ← Q; Q ← ∅
    for all b ∈ layer do
        process branch b; remove from Q
        for all b′ dependent on b do
            decrement d[b′]; add b′ to Q if d[b′] = 0
        end for
    end for
    append layer to L
end while
```

where $F_{\max}$ and $F_{\min}$ denote the FLOPs of the heaviest and lightest branches, respectively. The threshold β was empirically determined (1.5 in experiments) to balance workload effectively without incurring synchronization overhead.

3.2 Branch-Aware Memory Management

Parallel execution of branches requires careful memory isolation to prevent contention and unsafe reuse. Parallax assigns each branch b_i a dedicated memory arena $\mathcal{A}_i$, ensuring all tensor allocations remain within $\mathcal{A}_i$ to avoid cross-branch conflicts (Fig. 1c).

In-Branch Memory Reuse. Within each arena, Parallax uses a bump-pointer allocator with liveness analysis. When a tensor's last use completes, its buffer is reclaimed into a free list for reuse by subsequent tensors. Buffer reuse is safe if tensor lifetimes do not overlap:

$$\text{reuse}(T_j, T_k) \iff \text{lifetime}(T_j) \cap \text{lifetime}(T_k) = \varnothing. \tag{1}$$

This approach minimizes fragmentation and reduces peak memory usage, which is critical for resource-constrained mobile inference.

Cross-Arena Buffer Sharing. Parallax also enables safe cross-arena buffer reuse. When branches b_i and b_j reside in different, non-concurrent layers, freed buffers from $\mathcal{A}_i$ can be transferred to $\mathcal{A}_j$. This further reduces peak memory without incurring synchronization overhead.

Handling Dynamic Tensor Shapes. Dynamic operations generate tensors with unknown shapes at runtime, requiring on-the-fly memory (re)allocations. Naive handling risks synchronization delays or overwriting buffers of concurrent branches. Parallax avoids this by confining all dynamic tensor allocations and resizes to the arena $\mathcal{A}_i$ of the originating branch. This isolation eliminates cross-branch conflicts and enables safe parallel execution under constrained memory.

3.3 Resource-Constrained Parallel Scheduling

Even with balanced and memory-isolated branches, running parallel branches simultaneously could exceed the device's memory capacity. To prevent OOM issues while maximizing available concurrency, Parallax finally employs a resource-constrained scheduling strategy.

Branch Peak Memory Estimation. Parallax estimates the peak memory M_i of each branch b_i in three steps. First, *shape inference* computes tensor sizes from operator metadata (dimensions, data types). Second, *liveness analysis* tracks each tensor's lifetime within the branch; tensors needed downstream remain active. Finally, a linear scan over interval endpoints maintains a running total of active memory, recording the peak as M_i. This sweep runs in $O(|V|)$ time with negligible overhead, as it is fused with branch structure identification (§3.1).

Greedy Layer Scheduling. During runtime, Parallax continuously queries the operating system for available free memory. Then it set a safety margin of 30-50% to be the working memory budget M_{budget}. Within each layer, Parallax then selects the largest possible subset of branches whose combined estimated peak memory does not exceed this budget:

$$\sum_{b_i \in \text{chosen}} M_i \;\leq\; M_{\text{budget}}.$$

Branches not selected for parallel execution are run sequentially. This simple yet effective strategy avoids OOM failures while maximizing safe concurrency within the device's memory limits.

4 EXPERIMENT

Parallax is integrated into TensorFlow 2.18.1 via targeted modifications to `Invoke`, `InvokeImpl` and `PartitionGraphIntoIndependentNodeSubsets`, focusing on `Subgraph`, `Interpreter`, and `GraphInfo` to enable graph partitioning and branch parallelization. A custom memory planner extends `SimpleMemoryArena` for branch-aware memory management and isolated parallel execution. We also develop an evaluation app to benchmark Parallax on Android devices with Kirin, Dimensity and Google Tensor processors, measuring latency, peak memory and energy under a unified protocol.

4.1 EXPERIMENTAL SETUP

Table 2: Summary of DNN models. Symbolic input dimensions represent variable input size.

Model	Task	Input Shape	Precision	Params
YOLOv8n (Jocher et al., 2023)	Object detection	[1, 3, 640, 640]	FP32	3.19M
Whisper-Tiny (Radford et al., 2023)	Speech recognition	[1, 3000]	INT8/FP32	46.51M
SwinV2-Tiny (Liu et al., 2022)	Image classification	[1, 3, 224, 224]	FP16	28.60M
CLIP Text Encoder (Radford et al., 2021)	Text embedding	[batch, sequence_len]	FP32	63.17M
DistilBERT (Sanh et al., 2019)	Sentiment Classification	[batch, sequence_len]	FP32	66.96M

Baselines & Benchmarks. We evaluate Parallax across 5 representative DNN models, summarized in Table 2. Comparisons are made against three SOTA mobile inference frameworks: ORT, ExecuTorch and TFLite. For consistency, all models were initially implemented in PyTorch and subsequently converted to the target formats of each framework. An exception was Whisper-Tiny, where publicly available, framework-specific open-source implementations were used due to conversion difficulties. All baselines were evaluated on identical hardware and software setups.

Testing Platforms. Experiments used three smartphones: Google Pixel 6 (Google Tensor SoC, TPU, 8-core CPU, 2.80GHz), Huawei P30 Pro (Kirin 980 SoC, GPU, 8-core CPU, 2.60GHz), and Redmi K50 (Dimensity 8100 SoC, MDLA/DSP/GPU, 8-core CPU, 2.85GHz). Hardware offloading was enabled via Android's Neural Networks API (NNAPI) for heterogeneous inference. Although Kirin 980 includes a Mali-G76 GPU and dual-core NPU, neither is NNAPI-accessible. In this case, we leverage the OpenCL backend in TFLite to enable heterogeneous inference.

Performance Metrics. We report end-to-end inference latency (averaged over 20 runs after 5 warm-up runs), peak runtime memory usage and estimated energy consumption measured with Android Studio Energy Profiler. Benchmark inputs include random images from the COCO validation set (YOLOv8n), audio samples from LibriSpeech test-clean (Whisper-Tiny), images from the ImageNet validation set (MobileNetV2, SwinV2-Tiny), and sentences randomly selected from the SST-2 dataset (CLIP Text Encoder, DistilBERT), 30 for each.

4.2 RESULTS & DISCUSSION

Parallax leaves model weights and structure unchanged, ensuring identical outputs and accuracy to the original pretrained models. Therefore, while improving inference latency and energy, it does not affect functional performance across evaluations.

Latency Performance. Table 3 reports Parallax latency versus baselines. In CPU-only execution, Parallax consistently outperforms others, with 15–31% reductions on large-input models such as YOLOv8n ($3\times640\times640$) and Whisper-Tiny (30-second audio). For text encoders with 16–77 tokens

Table 3: End-to-end inference latency (ms) on three devices. Each entry reports the minimum / maximum latency. "Het" denotes heterogeneous inference. "-" indicates not supported due to operator-set mismatch, lack of backend support or inability to handle dynamic input tensors without manual shape fixing (in Parallax). The best results are in **bold**.

Model	ORT		ExecuTorch		TFLite		Parallax	
	CPU	Het	CPU	Het	CPU	Het	CPU	Het
Google Pixel 6								
YOLOv8n	83 / 974	-	77 / 933	-	85 / 1355	-	63 / 794	**54 / 743**
Whisper-Tiny	397 / 2114	486 / 3142	421 / 2376	-	543 / 2076	-	350 / 1780	**323 / 1706**
SwinV2-Tiny	82 / 87	1323 / 1726	84 / 93	-	96 / 108	1107 / 1994	**64** / 83	69 / **79**
CLIP Text Encoder	16 / 43	15 / 44	17 / 56	-	13 / 37	-	**11 / 29**	-
DistilBERT	**14** / 54	19 / **48**	15 / 51	-	27 / 57	-	17 / 49	-
Huawei P30 Pro								
YOLOv8n	136 / 1327	-	138 / 1475	-	127 / 1463	-	101 / 1016	**91 / 957**
Whisper-Tiny	433 / 3298	-	487 / 2954	-	412 / 2976	-	**340 / 2239**	-
SwinV2-Tiny	128 / 134	-	131 / 143	-	136 / 148	987 / 1753	109 / 126	**107 / 122**
CLIP Text Encoder	18 / 68	-	19 / 72	-	17 / 72	-	**15 / 57**	-
DistilBERT	19 / 73	-	17 / 83	-	25 / 63	-	**14 / 61**	-
Redmi K50								
YOLOv8n	98 / 1032	-	117 / 1215	-	124 / 1273	-	91 / 912	**84 / 881**
Whisper-Tiny	501 / 3326	634 / 3793	496 / 3182	-	499 / 2980	-	**317 / 2010**	-
SwinV2-Tiny	86 / 94	-	91 / 98	-	83 / 89	1046 / 1935	54 / 62	**53 / 59**
CLIP Text Encoder	**16** / 46	14 / 51	17 / 51	-	21 / 49	-	**16 / 43**	-
DistilBERT	**15** / 55	38 / 77	16 / 61	-	21 / 66	-	17 / **47**	-

Table 4: Peak runtime memory usage (MB), including model static memory and dynamic temporary activations or tensor allocation. Lowest peak memory results are in **bold** across devices.

Model	Pixel 6				P30 Pro				Redmi K50			
	ORT	ET	TFLite	Parallax	ORT	ET	TFLite	Parallax	ORT	ET	TFLite	Parallax
YOLOv8n	24.9	25.1	**24.5**	26.6	29.6	28.3	29.7	32.5	34.2	-	26.7	33.0
Whisper-Tiny	50.1	48.2	47.8	63.6	52.4	49.6	**45.3**	72.9	51.8	-	49.6	71.2
SwinV2-Tiny	86.7	84.9	**82.3**	107.6	102.3	95.7	96.4	132.0	97.3	99.1	102.4	124.8
CLIP Text Encoder	137.2	140.7	134.6	184.4	143.5	139.2	142.0	191.2	135.9	139.0	**133.7**	174.3
DistilBERT	206.7	203.2	**197.6**	233.5	215.9	211.1	207.4	234.7	230.4	227.3	229.0	265.2

(CLIP, DistilBERT), Parallax shows comparable latency, with minor overheads (e.g., DistilBERT on Pixel 6) from branch scheduling.

In heterogeneous execution, ORT remains the strongest baseline for dynamic inputs (e.g., CLIP, DistilBERT), as ExecuTorch lacks NNAPI support and TFLite reverts to CPU for dynamic operators. Parallax's fine-grained subgraph control and branch-isolation memory enable partial offloading to accelerators, yielding 9–46% latency gains over ORT and 20–45% over TFLite. Gains are most notable on complex models like Whisper-Tiny and SwinV2-Tiny, where conventional frameworks suffer from context-switching overhead due to fragmented delegation. These results demonstrate that Parallax effectively leverages parallel branch execution and selective offloading to mitigate performance penalties for dynamic and fragmented models in heterogeneous edge environments.

Runtime Memory Analysis. Table 4 shows Parallax peak memory versus baselines across devices. Concurrent branch execution moderately increases memory but stays within device limits due to resource-constrained scheduling (§3.3). The largest rise is on Whisper-Tiny (72.9 MB vs. 45.3 MB, +60.9% on P30 Pro), with smaller increases for YOLOv8n (33.0 MB vs. 26.7 MB, +23.6% on Redmi K50) and SwinV2-Tiny (124.8 MB vs. 102.4 MB, +21.9%). On average, memory rises 26.5%, more pronounced for large fragmented models (Whisper, SwinV2), but negligible for smaller workloads (CLIP, DistilBERT).

Table 5 further quantifies Parallax's branch-aware allocator overhead (§3.2). Its arena footprint averages 46.3% larger than TFLite and 37.7% larger than ORT, reflecting our choice of branch isolation over aggressive reuse. Compared to a naïve planner (one buffer per tensor), Parallax reduces arena size by 43.2% (like YOLOv8n: 85.8 MB vs. 203.6 MB, –57.8%). This demonstrates that Parallax achieves effective intra-branch memory reuse while enabling parallel execution across branches. We argue this memory overhead represents a worthwhile trade-off, as it supports the substantial latency improvements required for real-time edge inference and ensures memory remains predictable and

Table 5: Peak memory footprint (MB) of tensor arena allocations across frameworks. "Naive" denotes no liveness reuse, and every tensor gets separate memory.

Model	ORT	ExecuTorch	TFLite	TFLite (Naive)	Parallax
YOLOv8n	69.98	72.45	74.28	203.62	85.82
Whisper-Tiny	28.94	35.12	31.67	90.15	43.20
SwinV2-Tiny	17.22	12.87	31.67	38.40	46.75
CLIP Text Encoder	3.21	4.09	3.42	7.69	4.97
DistilBERT	4.67	5.18	4.02	9.41	7.42

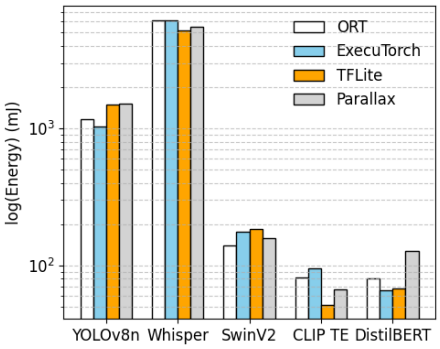

Figure 2: Energy performance on Google Pixel 6. Results are for CPU-only inference, which supports the majority of models.

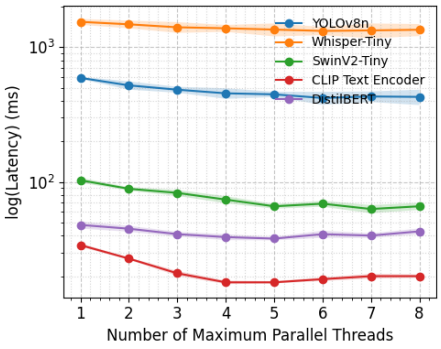

Figure 3: Maximum parallel threads vs. latency for Parallax (Pixel 6 CPU-only inference).

bounded under the parallel execution model. This also preserves deployment flexibility by eliminating the need for model refactoring or custom kernel modifications.

Energy Consumption. Figure 2 shows energy costs across frameworks on Google Pixel 6. Parallax achieves energy savings through reduced inference time. On Whisper-Tiny, energy decreases by 10.0% compared to ORT and 10.7% compared to ExecuTorch. Similarly, for the CLIP Text Encoder, Parallax lowers energy by 18.3% and 30.0% compared to ORT and ExecuTorch, respectively. For SwinV2-Tiny, Parallax energy rises 13.3% over ORT but improves 14.1% over TFLite. For YOLOv8n and DistilBERT, parallel execution overhead increases energy up to 47.0% and 92.2% vs. ExecuTorch, but still remains within practical mobile inference. Note that during all tests, device battery temperature fluctuation remained modest, ranging from 27.9 °C to 33.4 °C.

Although TFLite generally achieves the lowest peak memory and better energy in some models, it runs 1.2–1.8× slower than Parallax. ORT provides a marginal improvement than Parallax for latency in dynamic models, but Parallax improves efficiency via branch-level parallel scheduling, offering a compelling alternative for energy-constrained real-time inference.

4.3 Ablation Study

Layer-Level Latency Analysis. We profiled Whisper (CPU) and SwinV2-Tiny (CPU+TPU) on Pixel 6 using our graph analysis method (§3.1). Table 6 shows selected layers with TFLite vs. Parallax latencies and branch counts (BR). On multi-branch layers, Parallax achieves up to 32.8% improvement on Whisper layer 7 (6 branches) and 43.5% on SwinV2 layer 53 (1 TPU + 3 CPU branches), with averages of 23.7% and 35.5%. Single-branch layers, including delegated ones, show minor overheads ($\leq 4.4\%$) from thread setup and coordination. Overall, Parallax accelerates parallelizable segments while imposing negligible penalties on sequential layers.

Graph Optimization Effect. Table 7 summarizes how Parallax transforms each model's graph (§3.1). After initial operator delegation ("Post"), node counts drop sharply (e.g., Whisper-Tiny: 627 to 202), but graphs fragment into fine-grained layers (Whisper-Tiny: 75 to 184; SwinV2-Tiny: 151 to 270). Parallax applies graph partitioning to fallback small delegate regions, yielding more compact structures. It also exposes greater parallelism: parallelizable layers increase from 10 to

Table 6: Layer-wise latencies and branch counts for Whisper (CPU) vs. SwinV2-Tiny (CPU+TPU).

Whisper (CPU)				SwinV2-Tiny (CPU+TPU)			
Layer ID	**TFLite (ms)**	**Parallax (ms)**	**BR.**	**Layer ID**	**TFLite (ms)**	**Parallax (ms)**	**BR.**
3	37.26	27.59	3	2	4.32	2.97	4
7	6.37	4.28	6	10	1.59	1.66	1
8	12.40	12.64	1	37	7.45	5.09	6
24	45.37	39.74	8	53	13.46	7.61	4 (1D+3)
30	2.35	2.27	1	103	2.33	2.41	1 (D)

Table 7: Graph Structure and parallelism performance on Pixel 6. "Pre" represents pre-delegation graphs i.e. original graphs; "Post" means delegated graphs; "Parallax" means graphs optimized by Parallax.

Model	**Nodes**			**Layers**			**Par-Layers**			**Max-Branches**		
	Pre	Post	Parallax	Pre	Post	Parallax	Pre	Post	Parallax	Pre	Post	Parallax
YOLOv8n	480	5	4	120	6	4	69	2	1	6	2	2
Whisper-Tiny	627	202	367	75	184	57	19	10	24	8	5	8
SwinV2-Tiny	1108	356	674	151	270	143	92	28	52	8	8	8
CLIP Text Encoder	635	216	446	90	173	74	53	19	49	4	3	4
DistilBERT	353	123	211	53	115	49	28	23	17	4	2	4

24 (Whisper-Tiny) and from 28 to 52 (SwinV2-Tiny), while maximum concurrent branches recover from 5 to 8 in both. These results show Parallax simplifies fragmented graphs while improving scheduling flexibility and execution efficiency.

Thread Parallelism Sensitivity. Figure 3 shows the effect of varying Parallax's maximum parallel threads on Google Pixel 6. Latency decreases as threads increase, especially between 1 and 4, where parallelism is best utilized. Beyond 4 threads, improvements fluctuate based on model branch count and workload. For smaller models (CLIP, DistilBERT), excessive threads introduce management overhead, offsetting gains.

We set Parallax's maximum thread count to 6 in experiments. For fairness, other frameworks were also set to 6 where applicable. We observed similar trends when increasing their number of threads: using all available cores sometimes degraded performance significantly due to scheduling overhead (e.g., in TFLite multi-threaded inference). This safe setting provided stable and comparable results across all platforms.

5 Conclusion and Future Work

We introduced Parallax, a framework for accelerating heterogeneous mobile inference through non-invasive graph analysis, fine-grained subgraph control, branch-aware memory management, and resource-aware parallel scheduling. Across five DNNs on three smartphones, Parallax achieves 15–31% CPU speedups, 9–46% heterogeneous latency reductions, and up to 30% energy savings. With a moderate 26.5% memory overhead, it balances real-time performance and deployment simplicity, requiring no model refactoring or custom kernels. Ablation studies highlight the impact of graph partitioning, branch isolation, and thread scaling.

Future work will address three key limitations: **(i) Partial Dynamic-Flow Support.** Current support for dynamic tensors and control-flow is limited, with unsupported operators defaulting to the CPU. Extending accelerator support for arbitrary dynamic shapes and flows remains an open challenge. **(ii) Energy Overhead.** While our evaluation emphasizes latency and memory, some models incur higher net energy use due to branch scheduling. Incorporating energy-aware scheduling and graph optimization is a priority. **(iii) Model Conversion and Compatibility.** Real-world deployment faces issues such as precision mismatches and backend-specific constraints. We are extending our framework with automated conversion and delegation tools to broaden model support with minimal manual effort.

REPRODUCIBILITY STATEMENT

All models used in this work (YOLOv8n, Whisper-Tiny, SwinV2-Tiny, CLIP Text Encoder, DistilBERT) are publicly available, and we provide detailed descriptions of models, data and evaluation protocols in Section 4 and Appendix A. Hardware and software environments (three smartphones, TensorFlow 2.18.1 integration, NNAPI/OpenCL backends, thread settings) are fully specified in Section 4. Measurement procedures for latency, memory, and energy (including run counts, warm-ups, and profiling tools) are described in Section 4. Algorithmic details, such as graph partitioning, pruning thresholds, and memory budgeting, are given in Section 3 and Appendix C. To ensure reproducibility, we will release the full source code, runtime modifications, and experiment scripts upon acceptance of the paper.

ETHICS STATEMENT

This work focuses on improving the efficiency of mobile deep learning inference. It does not involve the collection or use of sensitive personal data, human subjects, or potentially harmful content. All models and datasets used are publicly available under their respective licenses. The main ethical considerations relate to responsible reporting of performance and energy measurements, which we have detailed in the paper. We do not foresee any direct negative societal impacts from this research.

REFERENCES

Daliang Xu, Hao Zhang, Liming Yang, Ruiqi Liu, Gang Huang, Mengwei Xu, and Xuanzhe Liu. Fast on-device llm inference with npus. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 445–462, 2025.

Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. *arXiv preprint arXiv:1908.09791*, 2019.

Haichen Shen, Jared Roesch, Zhi Chen, Wei Chen, Yong Wu, Mu Li, Vin Sharma, Zachary Tatlock, and Yida Wang. Nimble: Efficiently compiling dynamic neural networks for model inference. *Proceedings of Machine Learning and Systems*, 3:208–222, 2021.

Soroush Ghodrati, Sean Kinzer, Hanyang Xu, Rohan Mahapatra, Yoonsung Kim, Byung Hoon Ahn, Dong Kai Wang, Lavanya Karthikeyan, Amir Yazdanbakhsh, Jongse Park, et al. Tandem processor: Grappling with emerging operators in neural networks. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1165–1182, 2024.

Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. `https://github.com/onnx/onnx`, 2019.

PyTorch Foundation. ExecuTorch. GitHub repository, 2024. URL `https://github.com/pytorch/executorch`. Accessed: 2025-05-08.

Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, et al. Tensorflow lite micro: Embedded machine learning for tinyml systems. *Proceedings of Machine Learning and Systems*, 3:800–811, 2021.

Wei Niu, Gagan Agrawal, and Bin Ren. Sod2: Statically optimizing dynamic deep neural network execution. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 386–400, 2024.

Qiyang Zhang, Xiang Li, Xiangying Che, Xiao Ma, Ao Zhou, Mengwei Xu, Shangguang Wang, Yun Ma, and Xuanzhe Liu. A comprehensive benchmark of deep learning libraries on mobile devices. In *Proceedings of the ACM Web Conference 2022*, pages 3298–3307, 2022.

Jongseok Park, Kyungmin Bin, Gibum Park, Sangtae Ha, and Kyunghan Lee. Aspen: Breaking operator barriers for efficient parallelization of deep neural networks. *Advances in Neural Information Processing Systems*, 36:68625–68638, 2023.

Jianyu Wei, Ting Cao, Shijie Cao, Shiqi Jiang, Shaowei Fu, Mao Yang, Yanyong Zhang, and Yunxin Liu. Nn-stretch: Automatic neural network branching for parallel inference on heterogeneous multi-processors. In *Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services*, pages 70–83, 2023.

Fucheng Jia, Deyu Zhang, Ting Cao, Shiqi Jiang, Yunxin Liu, Ju Ren, and Yaoxue Zhang. Codl: efficient cpu-gpu co-execution for deep learning inference on mobile devices. In *MobiSys*, volume 22, pages 209–221, 2022.

Feng Yu, Guangli Li, Jiacheng Zhao, Huimin Cui, Xiaobing Feng, and Jingling Xue. Optimizing dynamic-shape neural networks on accelerators via on-the-fly micro-kernel polymerization. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 797–812, 2024.

Sehoon Kim, Amir Gholami, Zhewei Yao, Nicholas Lee, Patrick Wang, Aniruddha Nrusimha, Bohan Zhai, Tianren Gao, Michael W Mahoney, and Kurt Keutzer. Integer-only zero-shot quantization for efficient speech recognition. In *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4288–4292. IEEE, 2022.

Yiwu Yao, Yuchao Li, Chengyu Wang, Tianhang Yu, Houjiang Chen, Xiaotang Jiang, Jun Yang, Jun Huang, Wei Lin, Hui Shu, et al. Int8 winograd acceleration for conv1d equipped asr models deployed on mobile devices. *arXiv preprint arXiv:2010.14841*, 2020.

Yuang Jiang, Shiqiang Wang, Victor Valls, Bong Jun Ko, Wei-Han Lee, Kin K Leung, and Leandros Tassiulas. Model pruning enables efficient federated learning on edge devices. *IEEE Transactions on Neural Networks and Learning Systems*, 34(12):10374–10386, 2022.

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

Andrew G Howard. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. Mobilebert: a compact task-agnostic bert for resource-limited devices. *arXiv preprint arXiv:2004.02984*, 2020.

Zhanpeng Zeng, Michael Davies, Pranav Pulijala, Karthikeyan Sankaralingam, and Vikas Singh. Lookupffn: making transformers compute-lite for cpu inference. In *International Conference on Machine Learning*, pages 40707–40718. PMLR, 2023.

Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. μlayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15, 2019.

Joo Seong Jeong, Jingyu Lee, Donghyun Kim, Changmin Jeon, Changjin Jeong, Youngki Lee, and Byung-Gon Chun. Band: coordinated multi-dnn inference on heterogeneous mobile processors. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, pages 235–247, 2022.

Siqi Wang, Gayathri Ananthanarayanan, and Tulika 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):393–406, 2018.

Sumin Kim, Seunghwan Oh, and Youngmin Yi. Minimizing gpu kernel launch overhead in deep learning inference on mobile gpus. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, pages 57–63, 2021.

Qualcomm Technologies, Inc. Snapdragon 8 Gen 1 Mobile Platform Product Specifications. Technical specifications, 2021. Available at: `https://www.qualcomm.com/products/mobile/snapdragon-8-gen-1-mobile-platform`.

Qualcomm Technologies, Inc. 6th Generation Qualcomm AI Engine with Snapdragon 888 Re-Engineered Hexagon 780 Processor. Whitepaper, 2020. Available at: https://www.qualcomm.com/news/releases/2020/12/qualcomm-announces-snapdragon-888-5g-mobile-platform.

Glenn Jocher et al. Yolov8: The next evolution of yolo models, 2023. URL https://github.com/ultralytics/ultralytics. Ultralytics technical report and documentation.

Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. In *International conference on machine learning*, pages 28492–28518. PMLR, 2023.

Ze Liu, Han Hu, Yutong Lin, Zhenda Yao, Zhuliang Xie, Yixuan Wei, Jia Ning, Xinyang Cao, Zheng Zhang, Li Dong, Lu Yuan, and Baining Guo. Swin transformer v2: Scaling up capacity and resolution. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 12009–12019, 2022.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. In *Proceedings of the 38th International Conference on Machine Learning (ICML)*, pages 8748–8763, 2021.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. In *NeurIPS 2019 Workshop on Energy Efficient Machine Learning and Cognitive Computing*, 2019.

USE OF LARGE LANGUAGE MODELS

We used large language models (LLMs) solely to assist with grammar checking, wording suggestions, and polishing of the manuscript text. No LLMs were used for idea generation, experimental design, analysis, or results.

A FLOP ESTIMATION AND DELEGATION THRESHOLD JUSTIFICATION

A.1 OPERATOR-LEVEL FLOP ESTIMATION

To support Parallax's delegation pruning (§3.1), we estimate the compute intensity F of candidate regions using operator-level FLOP counts. We categorize TFLite operators into coarse-grained groups, each with a simplified FLOP estimator:

Table 8: Operator Classes and FLOP Estimators

Op Class	**Examples**	**FLOPs per Node**
Conv2D / Depthwise	Conv2D, DepthwiseConv2D	$2 \cdot C_{\text{in}} \cdot H_{\text{out}} \cdot W_{\text{out}} \cdot K_h \cdot K_w \cdot C_{\text{out}}$
MatMul / Dense	FullyConnected, MatMul	$2 \cdot M \cdot N \cdot K$
Elementwise	Add, Mul, ReLU, Sub	output_size
Pooling / Reduce	AvgPool, MaxPool, Mean, Sum	$H_{\text{out}} \cdot W_{\text{out}} \cdot K_h \cdot K_w$
Misc. / Other	Reshape, Slice, Transpose	0 (or $0.5 \cdot$ output_size optionally)

This coverage accounts for the majority of compute in modern CNNs and Transformers. Unrecognized or non-compute-heavy ops are either treated as 0-FLOP or assigned a small constant workload.

B DELEGATION COST MODEL DERIVATION

This section provides the full derivation and numerical justification for the delegate–pruning thresholds used in §3.1.

B.1 OFFLOAD VS. CPU EXECUTION TIME

Consider a candidate region S with

$$F = \sum_{v \in S} \text{FLOPs}(v) \quad \text{and} \quad B = \sum_{T \in \partial S} \text{numel}(T) \times \text{sizeof(dtype)}.$$

Then:

$$T_{\text{offload}} = L \; + \; \frac{F}{R_{\text{acc}}} \; + \; \frac{B}{B_{\text{bw}}}$$

is the total time to dispatch, compute on the accelerator and transfer boundary tensors back. Meanwhile, running the same region on the CPU takes

$$T_{\text{CPU}} = \frac{F}{R_{\text{cpu}}}$$

where R_{cpu} is the CPU's MAC/s rate.

We choose to offload only when

$$T_{\text{offload}} < T_{\text{CPU}} \quad \Longleftrightarrow \quad L + \frac{F}{R_{\text{acc}}} + \frac{B}{B_{\text{bw}}} < \frac{F}{R_{\text{cpu}}}.$$

B.2 SIMPLIFIED BOUNDS

Since $R_{\text{cpu}} \ll R_{\text{acc}}$, we can decompose the above condition into two practical sub-conditions:

Compute-bound condition. Neglecting memory transfer ($B/B_{\text{bw}} \approx 0$) and using $R_{\text{cpu}} \ll R_{\text{acc}}$, the inequality reduces to

$$L + \frac{F}{R_{\text{acc}}} \;<\; \frac{F}{R_{\text{cpu}}} \;\approx\; \frac{F}{R_{\text{cpu}}} \;\Longrightarrow\; F \;>\; L\,R_{\text{cpu}}.$$

Thus, the region's total MACs F must exceed the number of MACs a CPU can perform during the dispatch latency L.

Algorithm 3 Node Classification and Branch Identification

Input: Computation graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Output: Set of branches $\mathcal{B}$
for each node $v \in \mathcal{V}$ **do**
  Compute in-degree $d_{in}(v)$ and out-degree $d_{out}(v)$
  if $d_{in}(v) = 1$ and $d_{out}(v) = 1$ **then**
    Label v as `Sequential`
  else if $d_{in}(v) = 1$ and $d_{out}(v) > 1$ **then**
    Label v as `Splitter`
  else if $d_{in}(v) > 1$ and $d_{out}(v) = 1$ **then**
    Label v as `Merger`
  else if $d_{in}(v) > 1$ and $d_{out}(v) > 1$ **then**
    Label v as `Split-Merge`
  end if
end for
Initialize empty set of branches $\mathcal{B}$
Initialize visited set $\mathcal{V}_{visited} \leftarrow \emptyset$
for each node $v \in \mathcal{V}$ **do**
  if $v \notin \mathcal{V}_{visited}$ and v is not labeled as `Merger` or `Split-Merge` **then**
    Initialize new branch $b \leftarrow [\,]$
    while v is labeled as `Sequential` and $v \notin \mathcal{V}_{visited}$ **do**
      Append v to b
      Add v to $\mathcal{V}_{visited}$
      $v \leftarrow$ successor of v
    end while
    Add branch b to $\mathcal{B}$
  end if
end for

Algorithm 4 Layer Construction via Topological Sorting

Input: Set of branches $\mathcal{B}$, dependencies $\mathcal{E}$
Output: Ordered list of layers $\mathcal{L}$
Initialize in-degree map $d : \mathcal{B} \to \mathbb{N}$
for each branch $b \in \mathcal{B}$ **do**
  $d[b] \leftarrow$ number of incoming dependencies to b
end for
Initialize queue $Q \leftarrow$ branches with $d[b] = 0$
Initialize empty list of layers $\mathcal{L} \leftarrow [\,]$
while Q is not empty **do**
  Initialize empty list $layer \leftarrow [\,]$
  for each branch $b \in Q$ **do**
    Append b to $layer$
    for each branch b' dependent on b **do**
      Decrement $d[b']$ by 1
      if $d[b'] = 0$ **then**
        Add b' to Q
      end if
    end for
  end for
  Append $layer$ to $\mathcal{L}$
end while

Memory-bound condition. Neglecting compute time ($F/R_{\text{acc}} \approx 0$), offloading helps only if

$$L + \frac{B}{B_{\text{bw}}} \;<\; \frac{F}{R_{\text{cpu}}}.$$

However, more commonly one asks whether the accelerator itself would be memory-bound: comparing compute to transfer on the accelerator gives

$$\frac{B}{B_{\text{bw}}} \;<\; \frac{F}{R_{\text{acc}}} \quad\Longleftrightarrow\quad \frac{B}{F} \;<\; \frac{B_{\text{bw}}}{R_{\text{acc}}}.$$

This ensures the accelerator spends more time computing than waiting on memory.

B.3 Numerical Substitution

Using the representative SoC parameters from §3.1:

$L = 0.2\,\text{ms}, \quad R_{\text{cpu}} \approx 1 \times 10^9\ \text{MAC/s}, \quad R_{\text{acc}} = 2.6 \times 10^{13}\ \text{MAC/s}, \quad B_{\text{bw}} = 51.2 \times 10^9\ \text{B/s},$

we obtain:

$$L\,R_{\text{cpu}} = 0.2 \times 10^{-3} \times 10^{9} = 2 \times 10^{5} \text{ MACs}, \quad \frac{B_{\text{bw}}}{R_{\text{acc}}} = \frac{51.2 \times 10^{9}}{2.6 \times 10^{13}} \approx 0.002 \text{ bytes/MAC}.$$

To provide a safety margin for driver overhead, runtime jitter and variability across devices, we conservatively enforce:

$$F \;\geq\; 1 \times 10^{9} \quad \text{and} \quad \frac{B}{F} \;\leq\; 0.1 \text{ bytes/MAC},$$

and since each operator typically costs on the order of 10^{8}–10^{9} MACs, we also require $N = |V(S)| \geq 3$ to ensure meaningful per-region compute.

C Graph Partitioning and Scheduling Algorithms

To support the layer-branch abstraction (§3.1), Parallax employs two algorithms:

- **Node Classification and Branch Identification:** This algorithm classifies each node in the computation graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ based on its in-degree and out-degree, and identifies branches as maximal linear sequences of nodes without internal splits or merges, shown in Algorithm 3.
- **Layer Construction via Topological Sorting:** This algorithm constructs layers by performing a topological sort on the identified branches, ensuring that all dependencies are respected and that branches within the same layer can be executed in parallel, shown in Algorithm 4.