

```
@inproceedings{LiPG26,  
  author    = {Zhuojin Li and Marco Paolieri and Leana Golubchik},  
  title     = {Accelerating Mobile Inference through Fine-Grained {CPU}-{GPU} Co-Execution},  
  booktitle = {Selected Papers of {EPEW} 2025},  
  series    = {Lecture Notes in Computer Science}, volume = {To appear}, pages = {--},  
  publisher = {Springer}, year = {2026}, doi = {}  
}
```

Accelerating Mobile Inference through Fine-Grained CPU-GPU Co-Execution

Zhuojin Li , Marco Paolieri , and Leana Golubchik 

University of Southern California, Los Angeles, California, USA
{zhuojinl,paolieri,leana}@usc.edu

Abstract. Deploying deep neural networks on mobile devices is increasingly important but remains challenging due to limited computing resources. On the other hand, their unified memory architecture and narrower gap between CPU and GPU performance provide an opportunity to reduce inference latency by assigning tasks to both CPU and GPU. The main obstacles for such collaborative execution are the significant synchronization overhead required to combine partial results, and the difficulty of predicting execution times of tasks assigned to CPU and GPU (due to the dynamic selection of implementations and parallelism level). To overcome these obstacles, we propose both a lightweight synchronization mechanism based on OpenCL fine-grained shared virtual memory (SVM) and machine learning models to accurately predict execution times. Notably, these models capture the performance characteristics of GPU kernels and account for their dispatch times. A comprehensive evaluation on four mobile platforms shows that our approach can quickly select CPU-GPU co-execution strategies achieving up to 1.89x speedup for linear layers and 1.75x speedup for convolutional layers (close to the achievable maximum values of 2.01x and 1.87x, respectively, found by exhaustive grid search on a Pixel 5 smartphone).

Keywords: Mobile Inference · Co-Execution · Latency · Prediction

1 Introduction

Machine learning (ML) techniques have achieved rapid growth in recent years, driven by the breakthroughs in large-scale architectures such as Large Language Models (LLMs) and Large Vision Models (LVMs). These state-of-the-art models exhibit impressive capabilities in a wide range of applications, including question answering, image recognition, and video analysis. While these models are typically trained on powerful cloud servers, there is a growing demand for deployment on mobile platforms to serve inference tasks, since on-device deployment not only protects user privacy, but also provides offline availability and reduces latency for real-time applications.

However, deploying large-scale models on mobile platforms remains challenging, due to limited computational resources and energy constraints. To improve inference latency on mobile platforms, substantial research and industrial efforts

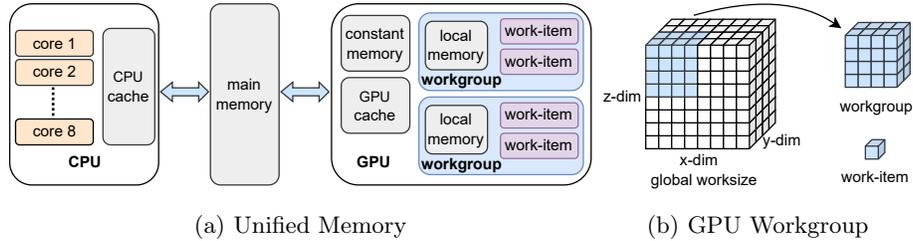


Fig. 1: Introduction to Mobile Platforms

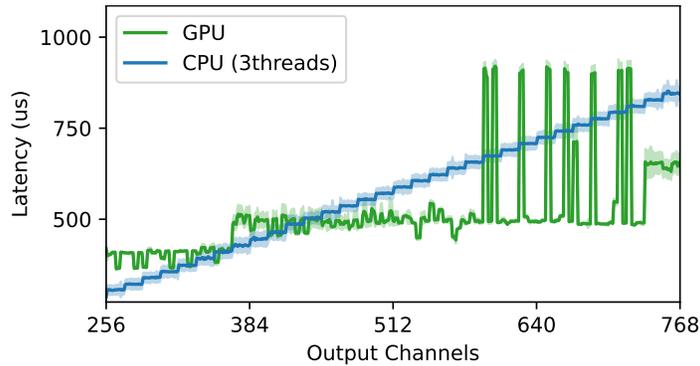


Fig. 2: Comparison of CPU and GPU Latencies for Linear Operations with Input Shape (50, 3072), with 95% Confidence Intervals (OnePlus 11)

have focused on developing efficient neural network architectures [16] and specialized hardware [7]. On top of these efforts, our work explores an additional dimension: distributing fine-grained operations (e.g., forward propagation of a neural network layer) across heterogeneous computing devices (specifically, CPU and GPU) available on modern mobile platforms. In particular, we pursue this approach by leveraging the following opportunities.

The unified memory architecture on mobile platforms enables both the main processor (CPU) and specialized accelerators (e.g., GPU and NPU) to directly access shared areas of main memory, as illustrated in Fig. 1a. In contrast, GPUs of cloud servers maintain separate on-chip memories and require data transfers to share data with the CPU through the main memory. By eliminating the need for memory transfers to main memory, unified memory on mobile platforms facilitates collaborative execution of inference tasks across compute devices.

Moreover, in contrast with existing works [8, 19] which report poor performance on mobile CPUs (due to inefficient CPU implementations in some ML libraries such as OpenVINO [6]), our empirical analysis shows that the performance gap between mobile CPUs and GPUs can be narrow for important operations when using the XNNPACK [4] library in TensorFlow Lite (TFLite), which provides high-performance implementations based on advanced SIMD in-

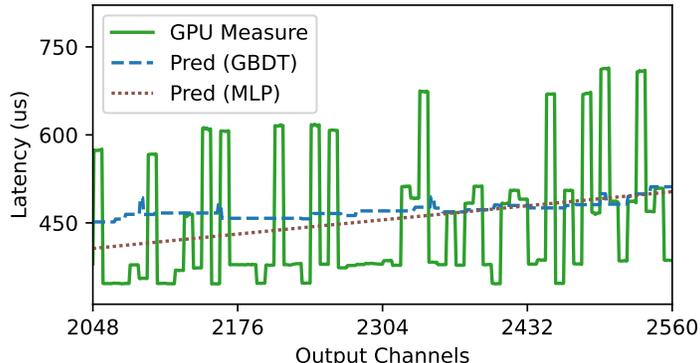


Fig. 3: Inadequate Modeling of GPU Latency Spikes by Existing Methods for Linear Operations with Input Shape (50, 768) (OnePlus 11)

structions for ARM CPUs. For instance, as illustrated in Fig. 2, for matrix multiplications (i.e., a linear layer in a neural network) of sizes 50×3072 and $3072 \times C_{out}$, a CPU implementation with 3 threads achieves even lower latency than a GPU kernel when $C_{out} < 425$. The significant performance of mobile CPUs motivates us to *assign compute tasks to both CPU and GPU to achieve lower inference latency through parallel execution*.

However, designing an effective co-execution strategy across multiple accelerators remains challenging. In our empirical evaluation, existing approaches achieve limited performance gains mainly due to the following two reasons.

First, optimal workload partitioning across CPUs and GPUs is difficult, primarily because GPU kernels exhibit complex non-linear performance characteristics. Our experiments, reported in Fig. 3, show significant latency spikes (green line) for linear operations as the output dimension C_{out} increases, due to heuristic choices in GPU kernel implementations and in the assignment of tasks to *work-groups* (i.e., to groups of threads executing the same code on different subsets of the input data, as illustrated in Fig. 1b). Consequently, co-execution frameworks relying on linear models for GPU latency prediction (e.g., [2]) can make poor partitioning decisions. Although non-linear ML models based on operation configurations (e.g., input/output channels) have been proposed for more accurate latency predictions [9, 13, 15, 22], our evaluations in Fig. 3 indicate that these methods¹ still fail to capture sudden latency spikes for specific configurations.

Second, synchronization overhead between compute devices is often non-negligible. Existing work [9] reports overhead of up to 1 ms for the GPU to notify the CPU that data mapping of a shared memory region has completed; as a comparison, our measurements indicate that, using the GPU on the OnePlus 11 smartphone, the longest linear operation of the ViT-Base-32 neural network [3]

¹ GBDT hyperparameters are detailed in Section 5.2. The MLP predictor was selected by varying the number of layers (1 to 4), neurons at each layer (32, 64 or 128), dropout rate (0 to 0.5), learning rate (10^{-5} to 0.1) and weight decay (10^{-6} to 10^{-3}).

takes only 660 μs . Such overhead can completely wipe out the benefits of co-execution; thus, minimizing synchronization overhead is crucial.

Our main contributions to tackling these problems are as follows:

- We develop accurate latency predictors that use kernel implementation details and kernel dispatch behaviors from TFLite (Section 3). Using detailed information, our predictors accurately capture complex latency characteristics, including the discontinuity due to heuristic workgroup choices and kernel selection, enabling more effective workload partitioning decisions.
- To address significant synchronization overhead, we use fine-grained shared virtual memory in OpenCL to implement an efficient CPU-GPU synchronization mechanism on mobile platforms (Section 4). Our design reduces expensive data mapping operations required for cache coherence and avoids notification delay through active querying. As a result, synchronization overhead is substantially reduced, e.g., from 162 μs to 7 μs for linear operations on a Motorola Edge Plus 2022 smartphone.
- We create a comprehensive dataset consisting of latency measurements of 2,039 linear and 2,051 convolution operations executed on four mobile devices using co-execution strategies with 1 to 3 CPU threads and the GPU. The evaluation shows that our predictors can quickly identify co-execution strategies achieving speedups up to 1.89x for linear operations and 1.75x for convolution operations (on Pixel 5 smartphones); we show that speedups are comparable to those identified through brute-force exploration of co-execution strategies (Section 5).

2 Inference Workload Partitioning

ML Operation Partitioning. Linear and convolutional layers are fundamental building blocks in deep neural networks. A linear layer multiplies an input matrix $\mathbf{X} \in \mathbb{R}^{L \times C_{in}}$ (activations from the previous layer) by the *weights* $\mathbf{W} \in \mathbb{R}^{C_{in} \times C_{out}}$ (the trainable parameters of the layer); each column of the output $\mathbf{Y} = \mathbf{X}\mathbf{W} \in \mathbb{R}^{L \times C_{out}}$ represents a different output feature. A convolutional layer is a generalization of a linear layer that is used to process data with grid-like topology (e.g., images): each output value $(\mathbf{Y})_{ijk}$ is obtained through the dot product of a *kernel* (or *filter*) $\mathbf{W}_k \in \mathbb{R}^{K \times K \times C_{in}}$ with a $K \times K$ patch of the input *feature map* $\mathbf{X} \in \mathbb{R}^{H_{in} \times W_{in} \times C_{in}}$ (2D activations from a previous layer), i.e., $(\mathbf{Y})_{ijk} = \mathbf{X}(i, j)\mathbf{W}_k$ where $\mathbf{X}(i, j)$ restricts the first two dimensions of \mathbf{X} to $[i - \lfloor \frac{K}{2} \rfloor, i + \lfloor \frac{K}{2} \rfloor]$ and $[j - \lfloor \frac{K}{2} \rfloor, j + \lfloor \frac{K}{2} \rfloor]$, respectively. In the output $\mathbf{Y} \in \mathbb{R}^{H_{out} \times W_{out} \times C_{out}}$, input height and width can be reduced by using a *stride* $S > 1$ to increment i and j , i.e., $H_{out} = \lfloor H_{in}/S \rfloor$ and $W_{out} = \lfloor W_{in}/S \rfloor$; the number of *output channels* C_{out} is equal to the number of different kernels \mathbf{W}_k .

In this work, we focus on partitioning the computation of linear or convolutional layers along output channels. Since each output channel corresponds to a distinct column of a linear weight matrix \mathbf{W} or to a distinct convolution kernel \mathbf{W}_k , each compute unit can store and manage its own subset of weights.

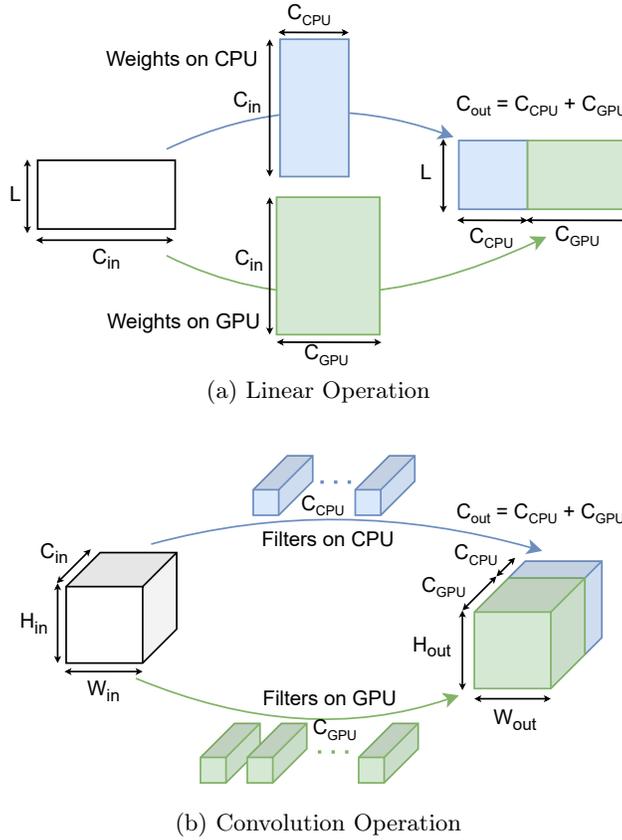


Fig. 4: Illustration of Computation Partitioning over Output Channels

Fig. 4 illustrates this strategy: the total number of output channels C_{out} is partitioned as $C_{CPU} + C_{GPU} = C_{out}$. Accordingly, the original weights are split by assigning the first C_{CPU} columns of \mathbf{W} or the first C_{CPU} kernels \mathbf{W}_k to the CPU and the rest to the GPU. During execution, CPU and GPU calculate their assigned portion of output independently using the shared input \mathbf{X} .

Problem Formulation. Formally, given a total of C_{out} output channels for a linear or convolution operation, our goal is to determine the optimal partitioning $c_1 + c_2 = C_{out}$ that minimizes the parallel execution time, i.e.,

$$\min_{c_1+c_2=C_{out}} T_{overhead}(c_1, c_2) + \max(T_{CPU}(c_1), T_{GPU}(c_2)).$$

Here, $T_{CPU}(c_1)$ and $T_{GPU}(c_2)$ represent the computation latency on CPU and GPU, respectively. $T_{overhead}(c_1, c_2)$ accounts for additional latency from synchronization overhead; in particular, we observe that the overhead remains constant in our measurements when a co-execution strategy is used (Section 4), while

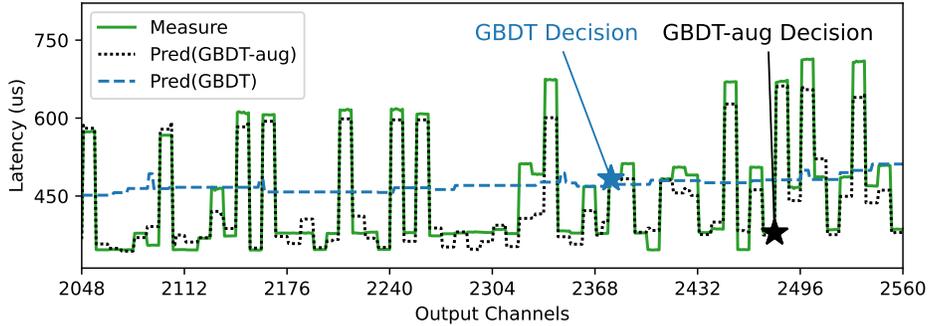


Fig. 5: Latency Prediction Improvement using Additional Features (OnePlus 11)

$T_{overhead}(c_1, c_2) = 0$ when $c_1 = C_{out}$ or $c_2 = C_{out}$ (i.e., exclusive execution on CPU or GPU, respectively). The formulation aims at balancing computational loads across CPU and GPU, ensuring high resource utilization.

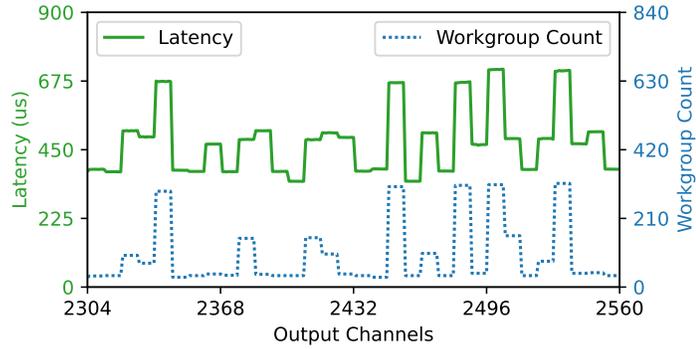
Since the number of output channels can be thousands in practice (e.g., 3,072 in the vision transformer ViT-Base-32 [3]) and the optimal partitioning varies across hardware platforms and other operation configurations (e.g., convolution kernel size), exhaustively measuring latency for every possible partitioning is costly. Thus, existing works [9, 11, 20] typically use ML-based latency predictors that rely on the operation parameters (e.g., input/output dimensions). However, as detailed in Section 3.1, these approaches can hardly capture the complex characteristics of GPU performance, resulting in suboptimal partitioning decisions. Additionally, as presented in Section 4, synchronization overhead $T_{overhead}$ must be accounted for when evaluating the achieved speedup, since significant overhead can diminish performance gains achieved through co-execution.

3 Accurate Latency Prediction

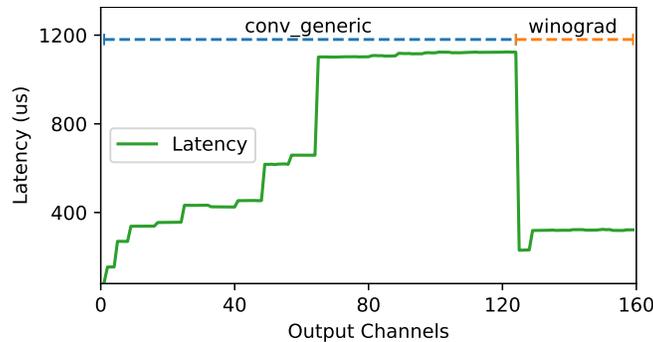
In this section, we present our approach to improve latency predictors used for partitioning decisions.

3.1 GPU Kernel Characterization

As motivated in Section 2, accurate latency prediction is crucial for effective partitioning decisions. To illustrate this, we conduct an experiment on a linear operation $\mathbf{W} \in \mathbb{R}^{768 \times 3072}$ from the vision transformer ViT-Base-32 [3], where it is used to transform an input feature map $\mathbf{X} \in \mathbb{R}^{50 \times 768}$ into $\mathbf{Y} \in \mathbb{R}^{50 \times 3072}$. To partition this operation, we measure the GPU latency of linear operations with size $768 \times C_{out}$ for $C_{out} \in [2048, 2560]$. As depicted in Fig. 5, GPU latency exhibits significant spikes, e.g., the linear operation with $C_{out} = 2500$ is counterintuitively 1.85 times slower than the one with $C_{out} = 2520$. Notably, a gradient-boosted decision tree (GBDT, an ML model broadly used in prior



(a) Heuristic Workgroup for Linear Operations with Input Shape (50, 768)



(b) Kernel Switch for 3x3 Convolution Operations with Input Shape (64, 64, 128)

Fig. 6: Reasons for Discontinuity in Latency Curve on Mobile GPUs (OnePlus 11)

work [9, 13, 15, 22]) using matrix sizes as input features only captures the overall increasing trend of latency (blue curve) rather than its sudden changes; using GBDT latency predictions, 2,378 output channels are assigned to the GPU (measured latency of $483 \mu\text{s}$) and 694 channels are assigned to the CPU (measured latency of $424 \mu\text{s}$), achieving only 1.02x speedup with respect to using only the GPU (as a reference, our approach in Section 3.2 achieves 1.29x speedup by assigning 2,480 channels to the GPU, based on more accurate predictions). To understand the discontinuity in GPU performance, we analyzed the source code of the ML framework TFLite [12] and identified two primary factors accounting for sudden latency changes:

1. *Heuristic Workgroup Choices.* In GPU computing, a workgroup is a collection of threads (work items) concurrently executing the same code and sharing resources such as local memory; the workgroup size determines how GPU threads are grouped for scheduling and is crucial for the efficiency of a GPU kernel. ML frameworks typically decide the workgroup size heuristically based on the kernel implementation and on the underlying hardware architecture. Fig. 6a presents the latencies and workgroup counts for linear operations with input size 50×768

and varying number of output channels C_{out} ; here, we observe a strong correlation between the number of workgroups and kernel latency.

2. *Kernel Selection.* In order to enhance performance, TFLite provides multiple GPU kernel implementations of convolution operations for different parameters (kernel size/stride/channels, input/output size). For example, Fig. 6b shows that, for a convolutional layer with 3×3 filter and input size of $64 \times 64 \times 128$, when the number of output channels exceeds 128, the kernel implementation will switch to the Winograd algorithm, which offers higher efficiency for operations with more output channels. This change of kernel implementations results in substantial latency anomalies due to their distinct performance characteristics.

3.2 Feature Augmentation

To accurately predict the GPU latency, we propose a white-box approach to capture the two aforementioned factors. Specifically, we analyzed the algorithms of TFLite to determine kernel implementation and workgroup size for a given operation configuration. Below, we summarize our analysis.

First, TFLite mainly uses three kernel implementations for convolution operations: (1) `conv_constant`, which leverages the faster on-chip constant memory when sufficient registers (estimated based on output channels) are available and convolution filters can fit within the constant memory, (2) `winograd`, which reduces the number of multiplications when the filter size is small and input sizes are large enough to make the transformation overhead negligible compared to the savings in multiplications, and (3) `conv_generic`, which is the default implementation for general use cases. Second, once a kernel implementation is selected, ML frameworks heuristically select workgroup configurations by considering hardware-specific factors, such as number of registers, compute unit occupancy, and memory access patterns.

To leverage this additional information about GPU kernels, we (1) construct separate latency predictors for each kernel implementation, and (2) augment the predictor input features to include kernel dispatch information, such as size and number of workgroups; these dispatch-related features can be calculated based on the hardware specification and on the parameters of the operation being executed. Fig. 5 illustrates that our approach (black curve) accurately captures the spikes of inference latency through feature augmentation. As a result, our enhanced latency prediction enables more effective workload partitioning for co-execution: using 2,480 output channels on GPU (with measured latency of $379 \mu s$ and prediction improved from $481 \mu s$ to $375 \mu s$) and 592 output channels on CPU (with measured latency of $354 \mu s$), the speedup improves from 1.02x to 1.29x (i.e., from $495 \mu s$ to $393 \mu s$).

4 Reduction of CPU-GPU Synchronization Overhead

As discussed in Section 2, significant synchronization overhead can diminish the performance gains of co-execution. We identify two main sources of overhead.

1. *Data Mapping for Cache Coherence.* When CPU and GPU collaboratively compute the output of a neural network layer, data transfers between their memory hierarchies are necessary. In particular, since mobile CPU and GPU have distinct caches, explicit data mapping operations are required to maintain cache coherence. Previous works [9] report up to 1 ms data mapping overhead.

2. *Inter-Processor Notifications.* Since CPU and GPU computations proceed in parallel, synchronization points are required to manage data dependencies and enforce execution order. For example, in OpenCL, synchronization between the host (CPU) and device (GPU) can be implemented by making the GPU kernel dependent on a user event; once the CPU finishes its computation, it marks this event as completed. However, there is a delay before the GPU recognizes the updated event state. Similarly to earlier work [9], our measurements indicate that the delay is on average 162 μ s on Motorola Edge Plus 2022 across 2,039 linear operations, which accounts for 39.9% of the total co-execution latency.

To reduce synchronization overhead, we use two techniques:

1. During inference, we store layer outputs in OpenCL *fine-grained shared virtual memory* (SVM), allowing both the OpenCL host (CPU) and device (GPU) to read and write directly to the same region in main memory. For instance, when a previous layer is evaluated collaboratively by CPU and GPU, the output results can be saved to this shared memory and read by subsequent CPU and GPU operations without additional copies between CPU and GPU buffers in main memory. In addition, unlike *coarse-grained SVM*, *fine-grained SVM* does not require explicit data mapping and unmapping operations since the hardware guarantees cache coherence for memory shared between CPU and GPU. Hence, this technique not only avoids copying data between CPU and GPU, but also eliminates the cost of data mapping operations.

2. To reduce the notification delay, we dispatch an active polling OpenCL kernel after each GPU computation. This kernel updates two synchronization variables (`cpu_flag` and `gpu_flag`) stored in fine-grained SVM. The kernel first updates `gpu_flag` to indicate GPU completion, then repeatedly checks `cpu_flag` to wait for CPU completion. Meanwhile, the CPU updates `cpu_flag` once finishing the computation and keeps polling for `gpu_flag` to be updated by the GPU. Notably, in order to minimize synchronization overhead, our polling-based implementation requires *busy waiting* on both CPU and GPU, which leads to additional power consumption in the case of unbalanced workload partitioning. However, our accurate latency predictions help balance CPU and GPU computation times and mitigate this issue.

Overall, by eliminating data mapping operations and reducing notification delay, we reduce the mean synchronization overhead to 7 μ s on Motorola Edge Plus 2022 across 2,039 linear operations, significantly improving the efficiency of CPU-GPU co-execution.

5 Experimental Results

In this section, we conduct comprehensive evaluations of our co-execution strategy using 2,039 linear and 2,051 convolution operations on four mobile platforms: Pixel 4, Pixel 5, Motorola Edge Plus 2022 (Moto 2022), and OnePlus 11.

5.1 Experimental Setup

Building on TFLite, we developed a C++ benchmarking tool that co-executes OpenCL kernels from the TFLite GPU Delegate [12] and CPU kernels from the XNNPACK library [4]; CPU-GPU synchronization kernels were implemented in OpenCL and dispatched to the same GPU queue. The compiled binary was uploaded to each smartphone to benchmark the latency of co-execution.

To ensure stable performance measurements, we prepared the smartphones using the same configurations as [14]. Specifically, we enabled the performance mode on the GPU and all CPU cores to encourage near-maximum clock frequencies, reducing performance fluctuations due to dynamic frequency scaling on Android platforms. The CPU threads were scheduled to high-performance cores by specifying CPU affinity. In addition, we attached an external cooling fan to the back of the smartphones and allowed for a cool-down time (1 second) after profiling each operation configuration, in order to mitigate thermal throttling effects, which can significantly hinder sustained performance.

5.2 Training Dataset Generation and Latency Prediction Accuracy

We construct a training dataset for ML predictors by sampling a broad range of operation parameters. For linear layers, operation dimensions (input length L , input channels C_{in} , and output channels C_{out}) are selected using a structured random sampling approach: first, we randomly pick an interval from $\{[2^k, 2^{k+1}] \mid 2 \leq k \leq 9\}$, and then we sample the dimensions uniformly from the selected interval. For convolutional layers, we sample input height H_{in} , input width W_{in} , input channels C_{in} , and output channels C_{out} using the same approach, and we sample the kernel (filter) shape K from $\{1, 3, 5, 7\}$ and its stride S from $\{1, 2\}$. In total, we collect latency measurements for 12,500 distinct configurations for each type of layer (linear or convolutional), using 20% for testing.

To train latency predictors, we use gradient-boosted decision trees (GBDTs) from LightGBM [10] and Optuna [1] to tune hyperparameters, which include learning rate (0.01 to 0.2), number of estimators (100 to 1000), depth (5 to 20), number of leaves (16 to 512), L1/L2 regularization terms (10^{-8} to 1), and subsample ratios (0.5 to 1). As discussed in Section 3.2, we train separate predictors for each kernel implementation, based on features including operation configurations and workgroup information. Fig. 7 presents the gain importance (i.e., the total loss improvement for all splits of a feature) for the top eight features of convolutional layers. As shown, workgroup size and total workgroup count are important factors affecting latency, which motivates their inclusion as input features. The GBDT predictors typically take 3-4 ms to determine the optimal

Device	Operations	MAPEs			
		GPU	1 CPU	2 CPUs	3 CPUs
Pixel 4	Linear	4.4%	11.5%	7.1%	5.8%
	Convolutional	8.5%	11.4%	8.8%	7.2%
Pixel 5	Linear	3.7%	6.2%	7.8%	7.2%
	Convolutional	7.7%	6.9%	8.1%	7.1%
Moto 2022	Linear	4.0%	2.5%	2.6%	2.4%
	Convolutional	9.0%	4.0%	3.6%	3.5%
OnePlus 11	Linear	3.7%	3.1%	2.9%	3.1%
	Convolutional	7.4%	4.8%	4.2%	4.4%

Table 1: MAPEs of GBDT Predictors

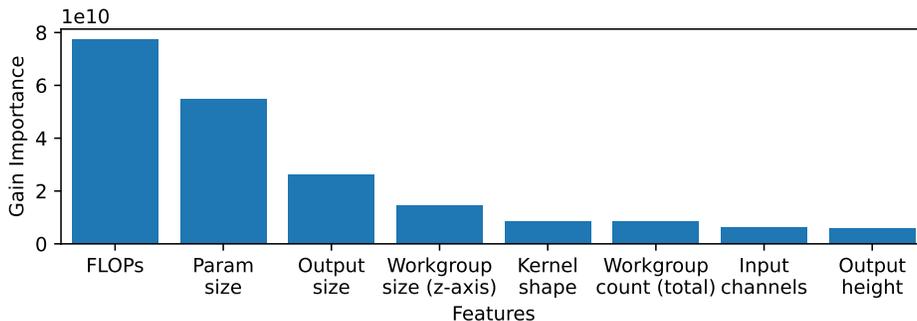


Fig. 7: Gain Improvement from GBDT Input Features (Convolution, Moto 2022)

partitioning for each operation; these partitioning decisions can be made offline before deployment to a device, as part of the compilation process.

Table 1 presents the prediction Mean Average Percentage Error (MAPE); errors are generally higher for convolutions due to the greater number of parameters (e.g., filter shape, stride) and multiple kernel implementations.

5.3 Co-Execution Speedup of Individual Layers

Table 2 reports the speedups of our co-execution strategy, together with the best speedups found by a grid search over $[0, C_{out}]$ with step size equal to 8. Given the long times required for repeated measurements, grid search is evaluated only on a random subset including 10% of the test cases. Note that grid search is only a baseline that is not applicable to real-world systems due to the long measurement times required for each new set of parameters of the linear and convolution operations. Test cases are selected as follows.

Device	Method	Speedup of Linear			Speedup of Convolutional		
		1 thread	2 threads	3 threads	1 thread	2 threads	3 threads
Pixel 4	GBDT	1.21x	1.52x	1.84x	1.22x	1.46x	1.69x
	Search	1.29x	1.59x	1.92x	1.31x	1.56x	1.79x
Pixel 5	GBDT	1.51x	1.78x	1.89x	1.45x	1.69x	1.75x
	Search	1.63x	1.92x	2.01x	1.49x	1.80x	1.87x
Moto 2022	GBDT	1.20x	1.32x	1.44x	1.16x	1.27x	1.39x
	Search	1.23x	1.36x	1.49x	1.22x	1.34x	1.46x
OnePlus 11	GBDT	1.06x	1.17x	1.26x	1.07x	1.22x	1.35x
	Search	1.13x	1.25x	1.35x	1.12x	1.27x	1.40x

Table 2: Average Speedups Obtained through CPU-GPU Co-Execution

– *Linear Layers*: Dimensions are selected from $\{i \cdot 2^j \mid 4 \leq i \leq 6, 2 \leq j \leq 9\}$. Then, we keep operations with FLOPs in $[4 \cdot 10^6, 10^9]$, resulting in a total of 2,039 linear operations. Our predictor leads to up to 1.89x average speedup (on Pixel 5), close to the best average speedup of 2.01x obtained through grid search.

– *Convolutional Layers*: Mobile vision models adopt hierarchical stages [16], with earlier stages capturing the local details (e.g., texture) and later stages capturing global features (e.g., shape). Accordingly, we define a hierarchy of 4 stages: convolutions of the first stage have input resolution $H_{in}, W_{in} \in \{64, 56, 48, 40\}$, kernel (filter) shape $K \in \{1, 3, 5, 7\}$, stride $S \in \{1, 2\}$, and input/output channels $C_{in}, C_{out} \in \{256/i, 320/i, 384/i, 448/i, 512/i\}$ where $i = 1, 1, 4, 8$ for $K = 1, 3, 5, 7$, respectively (to have similar computation loads). In stages 2, 3, 4, we halve the resolution and double the number of channels. We keep the generated layers with FLOPs within range $[4 \cdot 10^6, 10^9]$, resulting in a total of 2,051 convolutional layers. As shown in Table 2, our predictor achieves up to 1.75x speedup on Pixel 5, close to the best measured speedup of 1.87x. Notably, speedups are higher on devices with smaller CPU/GPU performance gap (e.g., Pixel 4 and Pixel 5), as a larger fraction of the task can be offloaded to the CPU.

5.4 Co-Execution Speedup of End-to-End Models

In this section, we present experiments demonstrating the speedups in end-to-end latency. We focus on four common neural networks: VGG16 [17], ResNet-18 [5], ResNet-34, and Inception-v3 [18]. In Table 3, we report the baseline latency for running each model on GPU. As comparison, we evaluate the performance of individual convolution and linear operations through co-execution on GPU and 3 CPU threads; we also conduct end-to-end experiments that incorporate the offline partitioning decisions for each operation and correspondingly schedule CPU and GPU kernels for the entire model; notably, pooling operations are always scheduled on the GPU, since their latency is negligible and this can avoid

Device	Neural Network	Baseline (ms)	Individual Ops		End-to-End	
			Latency (ms)	Speedup	Latency (ms)	Speedup
Pixel 4	VGG16	83.3	70.8	1.18x	73.0	1.14x
	ResNet-18	17.5	11.0	1.59x	11.4	1.54x
	ResNet-34	37.5	22.2	1.69x	22.5	1.67x
	Inception-v3	99.5	59.1	1.68x	61.5	1.62x
Pixel 5	VGG16	194.8	119.8	1.63x	125.1	1.56x
	ResNet-18	33.2	18.2	1.82x	18.6	1.78x
	ResNet-34	65.2	36.9	1.77x	37.1	1.76x
	Inception-v3	184.3	99.9	1.85x	102.9	1.79x
Moto 2022	VGG16	32.0	28.7	1.11x	29.7	1.08x
	ResNet-18	7.5	6.3	1.18x	6.7	1.11x
	ResNet-34	14.7	12.2	1.20x	12.9	1.14x
	Inception-v3	52.0	38.8	1.34x	41.1	1.27x
OnePlus 11	VGG16	27.4	25.3	1.09x	26.2	1.05x
	ResNet-18	8.5	6.6	1.29x	6.8	1.25x
	ResNet-34	17.6	13.5	1.31x	13.8	1.27x
	Inception-v3	44.2	36.2	1.22x	37.8	1.17x

Table 3: End-to-End Speedups from GPU and 3 CPU Threads Co-Execution

the synchronization overhead. The end-to-end improvement is slightly lower than that of individual operations, potentially due to memory access overhead between layers. The results show that our proposed method can achieve up to 1.67x, 1.79x, 1.27x, and 1.27x average speedups on Pixel 4, Pixel 5, Moto 2022, and OnePlus 11, respectively; these results validate the effectiveness of approach to real-world neural networks.

We observe that related work [9] also evaluated co-execution of VGG16, reducing its inference latency on Pixel 4 from a baseline of 200 ms (using only the GPU) to around 150 ms (using CPU and GPU). In contrast, our evaluation of VGG16 on Pixel 4 started from a baseline of 83.3 ms, which was reduced to 73.0 ms through co-execution. The difference in baseline inference latency is due to the different performance of the MACE ML framework [21] (used in [9]) and TFLite (used in our work). In particular, TFLite (1) already leverages image storage types to take advantage of L1 texture cache (which was used in [9] to improve the MACE baseline) and (2) implements efficient Winograd kernels to accelerate convolutional layers in VGG16 (Section 3.2).

5.5 Ablation Study

We conduct an ablation study on Moto 2022 to evaluate the individual impact of our proposed techniques. First, we observe that our augmentation technique

Method	Speedup of Linear			Speedup of Convolutional		
	1 thread	2 threads	3 threads	1 thread	2 threads	3 threads
Ours	1.20x	1.32x	1.44x	1.16x	1.27x	1.39x
w/o Augmentation	1.12x	1.24x	1.37x	1.08x	1.19x	1.31x
Original Overhead	0.76x	0.81x	0.88x	0.98x	1.07x	1.17x

Table 4: Ablation Study: Co-execution Speedup (Moto 2022)

reduces latency prediction MAPE of linear layers from 9.3% to 4.4% and of convolutional layers from 14.1% to 9.3%. These improvements lead to better partitioning strategies, e.g., for convolutional layers using the GPU and 1 CPU thread, the latency reduction improves from 1.08x to 1.16x (Table 4).

Next, to evaluate the contribution of our overhead reduction technique, we compare our active polling implementation with a baseline where the CPU *passively* waits for GPU kernel completions using the OpenCL `clWaitForEvents` API. The baseline incurs average overhead of 162 μ s across 2,039 linear layers and 141 μ s across 2,051 convolutional layers, accounting for 39.9% and 15.8% of the total co-execution latency, respectively, in the case of GPU co-execution with 1 CPU thread. Such high overhead negates the benefits of co-execution; in contrast, our active polling approach makes the synchronization overhead negligible (average of 7.0 μ s for linear layers and 5.4 μ s for convolutional layers).

6 Conclusions

We explored inference latency optimization for deep neural networks on mobile platforms by partitioning individual linear and convolutional layers across CPU and GPU. To address challenges in accurately predicting complex GPU latency behaviors and reducing CPU-GPU synchronization overhead, we developed enhanced latency predictors incorporating kernel implementation and dispatching information, and we proposed a lightweight synchronization method using OpenCL shared virtual memory. Comprehensive experimental evaluations showed that our approach achieves significant speedups that are close to the achievable best. In future work, we plan to investigate parallel execution on CPU, GPU, and NPU, and the effects of model quantization.

Acknowledgments

This work was supported in part by the NSF CNS-1816887, CCF-1763747, and IIS-1833137 awards. The authors would like to thank the anonymous EPEW reviewers for their insightful comments that helped improve this paper.

References

1. Akiba et al.: Optuna: A next-generation hyperparameter optimization framework. In: Proceedings of KDD. pp. 2623–2631 (2019)
2. Chen et al.: HeteroLLM: Accelerating large language model inference on mobile SoCs platform with heterogeneous AI accelerators. arXiv:2501.14794 (2025)
3. Dosovitskiy et al.: An image is worth 16x16 words: Transformers for image recognition at scale. arXiv:2010.11929 (2020)
4. Google: XNNPACK: High-efficiency floating-point neural network inference operators for mobile, server, and web. <https://github.com/google/XNNPACK> (2023)
5. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of CVPR. pp. 770–778. IEEE (2016)
6. Intel: OpenVINO Toolkit. <https://github.com/openvinotoolkit> (2025)
7. Jang et al.: Sparsity-aware and re-configurable NPU architecture for Samsung flagship mobile SoC. In: Proceedings of ISCA. pp. 15–28. IEEE (2021)
8. Jayanth, R., Gupta, N., Prasanna, V.: Benchmarking edge AI platforms for high-performance ML inference. arXiv:2409.14803 (2024)
9. Jia et al.: CoDL: Efficient CPU-GPU co-execution for deep learning inference on mobile devices. In: Proceedings of MobiSys. vol. 22, pp. 209–221 (2022)
10. Ke et al.: LightGBM: A highly efficient gradient boosting decision tree. Proceedings of NeurIPS **30** (2017)
11. Kim, Y., Kim, J., Chae, D., Kim, D., Kim, J.: μ layer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization. In: Proceedings of EuroSys. pp. 1–15 (2019)
12. Lee et al.: On-device neural net inference with mobile GPUs. arXiv:1907.01989 (2019)
13. Li, Z., Paolieri, M., Golubchik, L.: Predicting inference latency of neural architectures on mobile devices. In: Proceedings of ICPE. pp. 99–112. ACM (2023)
14. Li, Z., Paolieri, M., Golubchik, L.: A benchmark for ML inference latency on mobile devices. In: Proceedings of EdgeSys. pp. 31–36. ACM (2024)
15. Li, Z., Paolieri, M., Golubchik, L.: Inference latency prediction for CNNs on heterogeneous mobile devices and ML frameworks. Perform. Eval. **165**, 102429 (2024)
16. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., Chen, L.C.: MobileNetV2: Inverted residuals and linear bottlenecks. In: Proceedings of CVPR. pp. 4510–4520 (2018)
17. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)
18. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., Wojna, Z.: Rethinking the inception architecture for computer vision. In: Proceedings of CVPR. pp. 2818–2826. IEEE (2016)
19. Tang et al.: To bridge neural network design and real-world performance: A behaviour study for neural networks. Proceedings of MLSys **3**, 21–37 (2021)
20. Wei, J., Cao, T., Cao, S., Jiang, S., Fu, S., Yang, M., Zhang, Y., Liu, Y.: NN-Stretch: Automatic neural network branching for parallel inference on heterogeneous multi-processors. In: Proceedings of MobiSys. pp. 70–83 (2023)
21. XiaoMi: Mace: A deep learning inference framework for mobile heterogeneous computing platforms (2025), <https://github.com/XiaoMi/mace>, accessed: 2025-08-27
22. Zhang et al.: NN-Meter: Towards accurate latency prediction of deep-learning model inference on diverse edge devices. In: Proceedings of MobiSys. pp. 81–93 (2021)