Work-in-Progress: Modeling and Analysis of Inference Latency on USB Edge TPUs

Haopeng Gao*, Hyunjong Choi† and Yidi Wang*
*Santa Clara University, USA
hgao2@scu.edu, yidi.wang@scu.edu
†San Diego State University, USA
hyunjong.choi@sdsu.edu

***Abstract*—The USB-connected Edge TPU is widely adopted for edge inference, yet its performance characteristics remain poorly understood in real-world settings. Its inference latency is shaped not only by compute but also by parameter streaming over a constrained and potentially contended I/O channel. Prior work has largely relied on profiling with modified SRAM capacities, which provides only isolated insights and fails to capture runtime behavior when weights exceed on-chip memory or when multiple devices contend for USB bandwidth. In this paper, we present a streaming-aware latency model for the USB Edge TPU. In evaluation, we validate the model using several practical DNNs. Experimental results show that the predicted upper and lower bounds can effectively bound the measured latencies. Our work offers a foundation for future work on a predictive framework that generalizes to multi-device environments that require real-time guarantees, for which USB contention introduces nondeterminism.**

I. Introduction

Edge TPU has emerged as a popular ASIC-based hardware accelerator for deep learning inference at the edge due to its low power consumption and compact form factor. However, they are not designed to be fully preemptible: once a network is offloaded, execution runs to completion without interruption. To reduce the task blocking time introduced by such execution behavior, many prior works have proposed partitioning the network into multiple segments, and preemption can occur at the segment boundaries.

Existing performance studies of Edge TPU devices have predominantly relied on profiling approaches. For example, SPET [3] partitions models and jointly decides each task's SRAM share and the number of segments via a MIP formulation to meet real-time constraints. Sun et al. build an iterative DNN-based partitioner that searches for better cut points and selectively recompiles candidates until latencies are balanced [4]. Also, Zou et al. use a Transformer-based predictor to estimate segment latencies for Edge TPU pipelines [8]. While these profiling-based methods provide useful insights into compute behaviors, they do not sufficiently capture the role of I/O bottlenecks, especially for the widely deployed USB Edge TPU. In practice, when model parameters do not fit into SRAM, the TPU must stream weights at runtime from the host to device over USB communication, and it partially overlaps with computation on TPU but is still constrained by host-to-device bandwidth. As a result, if I/O is the bottleneck, contention naturally arises in multi-TPU setups or when the TPU co-exists with other USB devices. Profiling may still provide approximate insights when the TPU is the only device on the system, but in practice, such contention introduces nondeterminism that undermines the timing guarantees required by real-time systems.

Contribution. In this WiP paper, we propose a modeling-based approach to characterize the latency of USB-connected Edge TPUs. Our model explicitly decomposes execution into input transfer, output transfer, compute time, and weight transfer, while carefully accounting for the overlap between compute and streaming. By grounding our analysis in a system model rather than static profiles, we aim to build a foundation for reasoning about Edge TPU latency under realistic and shared I/O environments. It targets latency-sensitive edge inference scenarios, such as perception and control pipelines in embedded and robotic systems where predictable timing behavior is essential. This can be further extended to support timing guarantees in real-time systems.

The rest of the paper is organized as follows: Sec. II gives a brief summary of our experimental setup; Sec. III and IV show our proposed model for Edge TPU under compute-I/O interaction and its performance; and Sec. V concludes the paper.

II. Experiment Setup

This experiment utilizes Google Coral USB Edge TPU [1] and Raspberry Pi 5 featuring an SoC with four ARM Cortex-A76 CPU cores clocked at 2.4GHz, with 8 GB RAM as the experimental hardware, running on native Raspberry Pi OS, with `libedgetpu1-std` [2] compatible with USB 3.0. All models are first packaged as TensorFlow Lite (TFLite) models, then quantized and compiled using the TPU compiler to ensure compatibility and execution on the EdgeTPU. USB I/O traces are collected with Linux `usbmon`, which records submission and completion events with byte counts and timestamps.

III. Inference Latency Analysis

A. Example

Before diving into the segment-level analysis, we first illustrate the difference between *fully on-chip* and *partially on-chip* execution using a quantized, un-partitioned MobileNetV2-224

TABLE I: Partitions of MobileNetV2 in Sec. III-A.

Seg. No.	Position	Param. Size	In. Size	Out. Size
1	block_12_add	1.01 MiB	147.00 KiB	19.92 KiB
2	block_15_add	1.13 MiB	19.92 KiB	8.42 KiB
3	global_average_pooling2d	1.17 MiB	8.42 KiB	1.38 KiB
4	output	1.42 MiB	1.38 KiB	0.98 KiB
Overall	—	4.76 MiB	176.63 KiB	30.70 KiB

model. Fig. 1 compares the execution under two settings: *partially on-chip* and *fully on-chip*, measured in a "*cold-start*" condition, where the TPU's on-chip SRAM is initially empty and no weight parameters are resident at invocation. In the *fully on-chip* scenario (Fig. 1a), the entire model can be fitted into the SRAM and all weights must be transferred before the execution can start. In the *partially on-chip* scenario (Fig. 1b), only part of on-chip SRAM is allocated to the model's weight parameters. The TPU begins execution once the minimum warm-up weights are loaded, and the remaining parameters are streamed from the host over USB during execution. However, the overlaps between runtime weight streaming and TPU execution, as shown in Fig. 1b, shorten the total makespan, although the setting in Fig. 1a can eliminate runtime weight streaming. We make the following observations:

- Interestingly, at "cold-start", larger SRAM capacity and pre-loading all weights do not necessarily yield faster inference.
- When the weight parameters exceed SRAM capacity, the latency is no longer a simple sum of compute and I/O, but rather depends on how effectively the transfers between host and device can be hidden within the compute interval.

Next, we turn to a segment-level view. To better illustrate the decomposition of the DNN inference latency, the MobileNetV2-224 model is partitioned at the quartiles of cumulative parameter size. The partitions are shown in Tab. I.

Fig. 2 presents the latency decomposition of the four MobileNet segments under "cold-start" execution. Sufficient SRAM is allocated, and all weights can be fully loaded onto the TPU before execution. The trend shows that weight streaming dominates latency across all segments, while compute contributes relatively little. Building on these empirical insights, in the next section, we will formalize the latency model and analysis, which can explicitly capture the interaction between execution and parameter streaming.

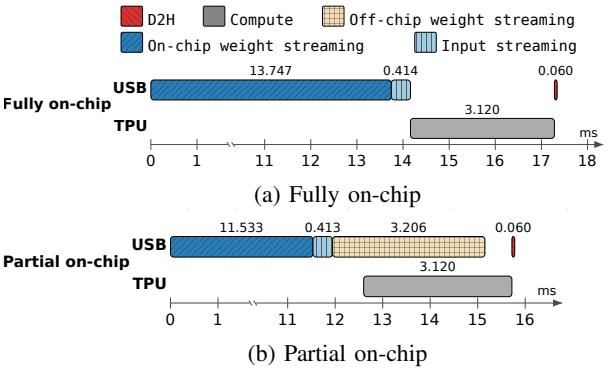
B. Analysis Under Compute-I/O Interaction

We consider a single Edge TPU-compatible model M that is partitioned into an ordered sequence of n segments $\{S_1, S_2, \ldots, S_n\}$ and executed continuously as a chain. An end-to-end inference of model M therefore consists of invoking $S_1, S_2, \ldots, S_n$ in order on the device. Let d_i^{in} and d_i^{out} denote the input and output tensors of segment S_i respectively, we have:

$$d_{i+1}^{in} = d_i^{out}, \quad 1 \leqslant i < n$$

At the chain boundaries, d_1^{in} is produced by the host from the original model input, and d_n^{out} is returned to the host as the final model output. We explicitly model the input and output tensor transmission of each segment S_i. We denote:

- C_i^{in}: time to transfer d_i^{in} from host to device (USB, h2d) before S_i executes.



(a) Fully on-chip

(b) Partial on-chip

Fig. 1: Execution with fully on-chip and partial on-chip

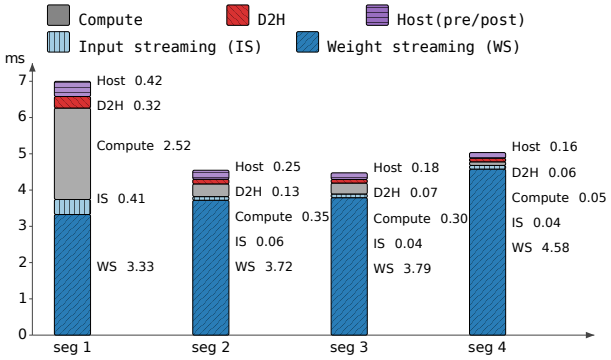


Fig. 2: Mobilenet segments inference

- C_i^{out}: time to transfer d_i^{out} from device to host (USB, d2h) after S_i completes.

When the model is expressed in terms of tensor sizes and link bandwidths, we write:

$$C_i^{in} = \frac{d_i^{in}}{B^{h2d}}, \quad C_i^{out} = \frac{d_i^{out}}{B^{d2h}}$$

where B^{h2d} and B^{d2h} are the *effective* host-to-device and device-to-host bandwidths at runtime. Therefore, formally, a segment S_i can be characterized as follows:

$$S_i := (C_i^{in}, C_i^{out}, C_i^{e}, w_i^{tot}, C_i^{w})$$

- C_i^{e}: the intrinsic cumulative TPU computation requirements, assuming the TPU is fully fed without stalls (explained later).
- w_i^{tot}: the total weight of bytes required to complete S_i.
- C_i^{w}: the *effective* weight transfer cost, defined as the portion of parameter transfer that cannot be hidden by the overlap with compute.

Here C_i^{w} exists only when the model parameters cannot fully reside in the on-chip SRAM; therefore, runtime parameter transfers are required to enable inference [3]. Bringing these elements together, the makespan for one pass through segment S_i is expressed as:

$$W_i = C_i^{in} + C_i^{out} + C_i^{e} + C_i^{w} + \epsilon \tag{1}$$

where ϵ is a small fixed per-segment control overhead including queuing, fencing, synchronization, etc. To understand C_i^{w}, we further break it down using the total weight bytes that must be delivered during segment execution:

$$w_i^{tot} = w_i^{warm} + w_i^{rem} \tag{2}$$

where

- w_i^{warm}: required warm-up weight bytes on device before compute can legally start. This is only required in "cold-start" situations.
- w_i^{rem}: remaining weight bytes in S_i that must be streamed.

Accordingly, the parameter transfer time C_i^w is the sum of the warm-up time and the residual streaming time:

$$C_i^w := t_i^{warm} + t_i^{rem} \tag{3}$$

The warm-up time t_i^{warm} can be converted from weight bytes w_i^{warm}:

$$t_i^{warm} = \begin{cases} \frac{w_i^{warm}}{B^{h2d}} & \text{, if } w_i^{warm} \text{ is not in SRAM,} \\ 0 & \text{, if } w_i^{warm} \text{ is in SRAM} \end{cases} \tag{4}$$

where B^{h2d} is the *effective* host-to-device bandwidth for weights. Once compute begins, the TPU execution and USB streaming proceed in parallel. Technically, the TPU core fetches operands from its on-chip SRAM and continues its computation, while in the background, the USB controller and host driver move the remaining model parameters into available SRAM banks. This design avoids USB TPU stalling. In this paper, this parallelism is captured by treating the TPU's compute window C_i^e as an overlap window, during which up to $B^{h2d} \cdot C_i^e$ bytes can be streamed without extending the makespan. Any excess bytes beyond this capacity become a residual tail. Such an approach is used in some prior research works [5, 7]; therefore, we derive the lower bound of the time to stream the remaining weights:

$$\begin{aligned} \breve{t}_i^{rem} &= \max\left(\frac{w_i^{rem} - B^{h2d} \cdot C_i^e}{B^{h2d}}, 0\right) \\ &= \max\left(\frac{w_i^{tot} - w_i^{warm}}{B^{h2d}} - C_i^e, 0\right) \end{aligned} \tag{5}$$

In contrast, when no overlap is assumed, the streaming of the remaining weights will be fully serialized; thus the upper bound of t_i^{rem} is:

$$\hat{t}_i^{rem} = \frac{w_i^{rem}}{B^{h2d}} \tag{6}$$

Based on the above decomposition, we can obtain the lower and upper bounds respectively for the makespan W_i in Eq. 1.

IV. Evaluation

In this section, we first present the parameter estimation, and then use realistic DNN models to evaluate the effectiveness of the proposed analytical model using the experiment setup mentioned in Sec. II.

Benchmark Selection. We use a set of well-established DNN models, including DenseNet-201, InceptionV3, ResNet-101, ResNet-50, and Xception. Each benchmark model is partitioned into eight segments. These cut points are determined at natural structural boundaries of the networks, where the output tensors are of moderate size and can be compiled for execution on the Edge TPU. For example, in ResNet-50, the "add" marks the end of the block where the skip connection is fused. Tab. II provides a summary of the models used in this study and reports the partition boundaries and the size of each segment.

Obtaining Parameters. The host-to-device transfer bandwidth B^{h2d} is obtained by transferring large contiguous tensors from host memory to the TPU. Since the official USB 3.0 specification allows for a much higher throughput than that observed in our measurements, we directly measured this bandwidth and used the average value of about $340\,\mathrm{MiB\,s^{-1}}$ by running the benchmark models 100 times each.

The device-to-host transfer bandwidth B^{d2h} is measured in a similar way by repeatedly transferring results from TPU back to the host. However, in contrast to B^{h2d}, the measurements are both slower and more variable, i.e., ranging from $35\,\mathrm{MiB\,s^{-1}}$ to $87\,\mathrm{MiB\,s^{-1}}$, which we attribute to firmware buffering and host controller scheduling arbitrations. When computing the upper and lower bounds of inference latency, we use the observed minimum and maximum B^{d2h} respectively. Nevertheless, since output tensors are generally much smaller than input and weight transfers, this variability has a limited impact on overall inference latency.

We then obtain the intrinsic TPU execution time C_i^e by measuring the latency of each segment when $C_i^w = 0$, and then using these profiles in all the following experimental settings. Since the execution on TPU is based on fixed functions on hardware without scheduling variability, the measurements are highly stable and we take the average. Such execution appears as the idle gap between the completion of the host-to-device burst and the arrival of the first device-to-host packet [1]

To estimate the per-segment control overhead ϵ, we measure the end-to-end latency of each segment and subtract all active contributions in Eq. 1. It exhibits a range from 0.3 to 2.0 ms, and we take the average value of approximately 1.0 ms.

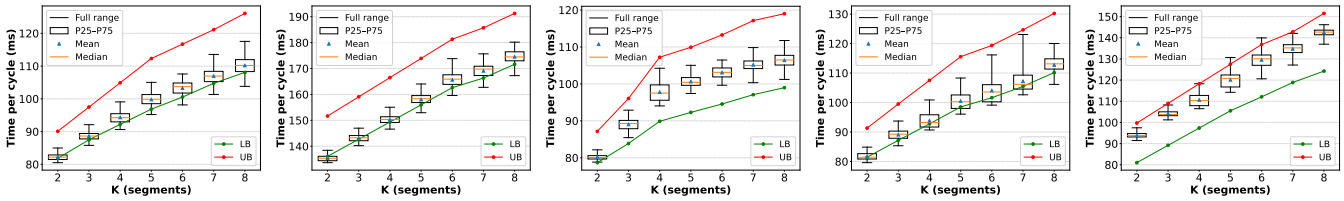
System Evaluation. To evaluate the accuracy of the proposed latency model, we consider several real DNN models listed in Tab. II, each manually partitioned into eight candidate segments of varying sizes. To systematically explore different partitioning granularities, we form multiple segmentation configurations by progressively increasing the number of partitioning points. Specifically, let k denote the number of partitions, where $k \in \{2, \ldots, 8\}$. For example, for $k = 2$, the model is divided into two contiguous groups: the first includes all the layers from the input up to the first cut point, and the second includes the remaining layers from that cut point to the output. For $k = 3$, we insert one additional cut at the second cut point, forming three contiguous groups. This process continues until $k = 8$, where each candidate cut point defines a segment boundary. In this way, the same network yields multiple segmentation granularities, ranging from coarse (with 2 segments) to fine (with 8 segments) partitioning.

For each segmentation configuration and each benchmark model, we execute the inference 100 times and record the distribution of the measured end-to-end latency. We then compare these measured latencies against the analytical predictions of

[1] Although we measure this parameter via USB transaction trace, C_i^e is not necessarily to be profile-dependent. It represents the intrinsic execution time of the TPU, which can be estimated analytically from the model's multiply-accumulate (MAC) operations and the theoretical throughput of the TPU [6].

TABLE II: Segments information of the benchmark models. Parameter sizes are in MiB

	ResNet-50		ResNet-101		DenseNet-201		InceptionV3		Xception	
Cut	Position	Size	Position	Size	Position	Size	Position	Size	Position	Size
1	conv4_block1_add	2.98	conv4_block3_add	5.16	conv4_block1_add	2.98	mixed4	3.74	add_3	3.24
2	conv4_block4_add	3.31	conv4_block8_add	5.49	conv4_block4_add	3.31	mixed5	1.88	add_5	3.51
3	conv4_block5_out	1.13	conv4_block13_add	5.49	conv4_block5_out	1.13	mixed6	1.88	add_7	3.51
4	conv4_block6_add	1.13	conv4_block18_add	5.49	conv4_block6_add	1.13	mixed7	2.13	add_9	3.51
5	conv5_block1_add	5.86	conv4_block23_add	5.49	conv5_block1_add	5.86	mixed8	1.68	add_10	1.77
6	conv5_block2_add	4.33	conv5_block1_add	5.86	conv5_block2_add	4.33	mixed9	4.89	block14_sepconv1_act	3.67
7	conv5_block3_add	4.33	conv5_block2_out	4.33	conv5_block3_add	4.33	mixed10	5.88	block14_sepconv2_act	3.13
8	output	2.10	output	6.40	output	2.10	output	2.10	output	2.10



(a) ResNet-50 (b) ResNet-101 (c) DenseNet-201 (d) InceptionV3 (e) Xception

Fig. 3: Latency vs. segment count k for five models

our model, specifically the lower and upper bounds of W_i derived from Eq. 1. Fig. 3 shows the results. As the number of segments increases, the total end-to-end inference time of each model increases, due to the costs introduced by the extra I/O activities at the boundaries of the segments. The lower bound (LB) curve represents the optimistic case where runtime weight streaming fully overlaps with compute. The upper bound (UB) represents the pessimistic case where streaming is serialized with compute. For all models and segmentations, the means and medians of the measured latency fall between LB and UB, with an overall coverage of 83.6% and a numeric violation severity of 0.85%. However, fluctuations are observed, mainly due to three factors: (i) the variance of B^{d2h} and B^{h2d}, (ii) the inconsistency of the overlap between TPU execution and weight streaming across the runs, and (iii) the small fluctuation of the host-side processing overhead ϵ. Xception (Fig. 3e) exhibits a 1.35% violation over the upper bound, primarily because its early depthwise-separable layers generate numerous small, fragmented host-to-device transfers, which lower the effective B^{h2d} compared to the averaged bandwidth used in our analytical model.

By validating our model across different segmentation granularities, we demonstrate its ability to support design trade-offs between predictability and efficiency, which are important for real-time inference deployment.

V. Conclusion and Future Work

This Work-in-Progress paper introduces a modeling-based approach to predict the latency of USB-connected Edge TPUs, explicitly decomposing end-to-end inference into input/output transfers, intrinsic compute, and residual weight streaming while capturing the overlap between compute and communication. The evaluation shows that our model yields practical lower and upper latency bounds across five standard DNNs with variant partitioning granularities. The study also quantifies the non-trivial boundary costs introduced by segmentation and clarifies when off-chip weight streaming can or cannot be hidden by compute.

For future work, we plan to extend the proposed model beyond single inference scenarios to include those with multiple concurrent workloads and shared resources. This includes accounting for contention on the USB bus when multiple other I/O devices are active. We also aim to integrate these extensions with real-time schedulability analysis, so that our framework can guide system-level decision-making such as model segmentation, batching, and device allocation.

References

[1] Coral USB Accelerator. URL: https://coral.ai/products/accelerator.

[2] Edge TPU runtime library (libedgetpu). URL: https://github.com/google-coral/libedgetpu.

[3] Changhun Han, Hoon Sung Chwa, Kilho Lee, and Sangeun Oh. Spet: Transparent SRAM allocation and model partitioning for real-time DNN tasks on Edge TPU. In *Proceedings of the 60th Annual ACM/IEEE Design Automation Conference*, pages 1–6, 2023.

[4] Binqi Sun, Bohua Zou, Yigong Hu, Tomasz Kloda, Ling Wang, Tarek Abdelzaher, and Marco Caccamo. Sapar: A surrogate-assisted DNN partitioner for efficient inferences on Edge TPU pipelines. *ACM Transactions on Embedded Computing Systems*, 24(5s), September 2025.

[5] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, et al. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 1, pages 93–106, 2022.

[6] Yu Wang, Gu-Yeon Wei, and David Brooks. A systematic methodology for analysis of deep learning hardware and software platforms. In *Proceedings of Machine Learning and Systems*, volume 2, pages 30–43, 2020.

[7] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, April 2009.

[8] Bohua Zou, Binqi Sun, Yigong Hu, Tomasz Kloda, Marco Caccamo, and Tarek Abdelzaher. A performance prediction-based DNN partitioner for Edge TPU pipelining. In *IEEE Military Communications Conference*, pages 1–6, 2024.