IEICE TRANSACTIONS on Information and Systems

VOL. E105-D NO. 2
FEBRUARY 2022

A PUBLICATION OF THE INFORMATION AND SYSTEMS SOCIETY
The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

LETTER

Layerweaver+: A QoS-Aware Layer-Wise DNN Scheduler for Multi-Tenant Neural Processing Units*

Young H. OH†, *Student Member*, Yunho JIN††, Tae Jun HAM††, *Nonmembers*, *and* Jae W. LEE††a), *Member*

SUMMARY Many cloud service providers employ specialized hardware accelerators, called *neural processing units* (NPUs), to accelerate deep neural networks (DNNs). An NPU scheduler is responsible for scheduling incoming user requests and required to satisfy the two, often conflicting, optimization goals: maximizing system throughput and satisfying quality-of-service (QoS) constraints (e.g., deadlines) of individual requests. We propose *Layerweaver+*, a low-cost layer-wise DNN scheduler for NPUs, which provides both high system throughput and minimal QoS violations. For a serving scenario based on the industry-standard MLPerf inference benchmark, *Layerweaver+* significantly improves the system throughput by up to 266.7% over the baseline scheduler serving one DNN at a time.

key words: *inference serving system, neural networks, multi-tasking*

1. Introduction

With the widespread adoption of deep learning-based applications and services, computational demands for efficient deep neural network (DNN) processing have recently surged in datacenters [1]. In addition to well-established domains such as advertisement, social networks, and personalized assistants, emerging services in the domains of automotives and Internet-of-Things (IoT) further explodes those computational demands for DNNs.

Many cloud service providers, including Google, Microsoft, and Amazon, to name a few, often exploit specialized DNN accelerators on their cloud, called *neural processing units (NPUs)*. The characteristics of each NPU (e.g. compute-to-memory bandwidth ratio) can vary widely as their target applications and design objectives are different. For example, Google TPUv3 [2], which targets both DNN training and inference, features 128 TOP/s of computation and 900 GB/s off-chip memory bandwidth. In contrast, TPUv4i [3] targets DNN inference only, and its compute-to-memory bandwidth ratio is substantially higher with 123 TOP/s of compute and 614 GB/s of memory bandwidth. While the microarchitectures of these two designs are quite similar, the variability of NPUs in compute-to-memory bandwidth ratio tends to get wider across vendors with different targets, microarchitectures, and technologies.

On the other hand, DNN models in service have very different arithmetic intensities depending on their layer structures, operators, etc. Thus, there is no one-size-fits-all accelerator that works well for all of those DNN models. For example, convolutional neural networks (CNNs) are conventionally known to be most compute-intensive, hence featuring the highest arithmetic intensities. In contrast, natural language processing (NLP) models often consist of fully-connected (FC) layers with little weight reuse to be more memory-intensive. Forming a larger batch also increases the arithmetic intensity by increasing the weight reuse.

As a datacenter NPU is required to run a variety of DNN models, a mismatch between its compute-to-memory bandwidth ratio and the arithmetic intensity of the model it runs can cause a serious imbalance between compute and memory access time. This yields a low system throughput due to the under-utilization of either processing elements (PEs) or off-chip DRAM bandwidth. This waste leads to an increase in the total cost of ownership (TCO) in datacenters. Once either resource gets saturated (while the other resource is still available), it is difficult to further increase throughput without scaling the bottlenecked resource.

There exist some proposals that target to maximize throughput with latency-hiding techniques [4] or layer-wise time-multiplexed scheduling [5]. However, merely optimizing throughput is not sufficient for DNN inference as datacenters typically impose service-level agreement (SLA) goals [6] to set a bound for the maximum service latency. Unfortunately, maximizing system throughput is often at odds with providing QoS for individual requests; thus, the NPU scheduler plays a key role in balancing latency and throughput. In the context of GPUs, multi-tasking deadline-aware schedulers have recently been proposed [7]–[9]. However, NPUs feature a much simpler execution model than GPUs, thus leading to a more deterministic and predictable execution time. This enables us to derive a simpler but more effective solution that leverages DNN-specific features (e.g., characteristics of each layer). Several QoS-aware GPU schedulers in a datacenter environment [10], [11] perform task prioritization based on estimated QoS slack time. However, their estimation is less precise in an NPU setting for not utilizing data fetch and

Manuscript received September 13, 2021.
Manuscript revised October 24, 2021.
Manuscript publicized November 11, 2021.

†The author is with Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon 16419, Korea.
††The authors are with Department of Computer Science and Engineering, Seoul National University, Seoul 08826, Korea.

*This work is supported by Samsung Advanced Institute of Technology, Samsung Electronics Co., Ltd., IC Design Education Center (IDEC), Korea (for EDA tools), and Institute of Information & Communications Technology Planning & Evaluation (IITP) grant (2021-0-00853, Developing Software Platform for PIM Programming) funded by the Korea Government (MSIT).

a) E-mail: jaewlee@snu.ac.kr

DOI: 10.1587/transinf.2021EDL8084

compute time, which can be readily estimated on NPUs with high precision. Thus, it is highly desirable to design a lightweight QoS-aware scheduler that leverages unique characteristics of NPUs to maximize system throughput while satisfying QoS constraints of individual requests.

To address this, we take a software-centric approach that exploits concurrent execution of multiple DNN models of opposing characteristics to effectively balance NPU resource utilization, while minimizing QoS violations for individual requests. To this end, we propose *Layerweaver+*, an inference serving system with a novel QoS-aware, layer-wise time-multiplexing scheduler. The low-cost scheduling algorithm of *Layerweaver+* searches for an efficient layer-wise schedule from multiple heterogeneous DNN serving requests to maximize throughput. To prevent QoS violation, the *Layerweaver+* scheduler prioritizes a request whose deadline is imminent. Our evaluation of *Layerweaver+* on various mixes of compute- and memory-intensive DNN models demonstrates an average of 49.2% improvement in system throughput for a multi-stream server scenario over the baseline scheduler executing one model at a time.

Our contributions are summarized as follows.

- We identify the limitations of a state-of-the-art multi-tenant DNN scheduler (*Layerweaver* [5]), which does not take into account the QoS constraints of individual requests.
- To address this, we propose *Layerweaver+*, which finds a more balanced schedule for both throughput and latency by switching between two scheduling modes: QoS-aware request prioritization and throughput-oriented scheduling.
- We provide a detailed analysis of *Layerweaver+* using a realistic inference scenario with QoS constraints based on a server scenario in the MLPerf inference benchmark.

2. QoS-Aware Multi-DNN Scheduler

2.1 Limitations of the Prior Art

Due to the wide spectrum of NPUs and DNN models, it is nearly impossible to balance the usage of NPU resources for *all* DNNs. Recently, several proposals have addressed this problem by time-multiplexing layer-wise execution of multiple DNN models with opposing characteristics (e.g., memory-intensive and compute-intensive) to saturate both compute and memory bandwidth [5], [12]. Figure 1 illustrates how the state-of-the-art scheduler interweaves two heterogeneous DNN models to balance resource usage. The naïve schedule in Fig. 1 (a) does not allow the interleaving of DNN layers across different models. This results in a significant amount of PE idle time during the execution of M1, and DRAM idle time during the execution of M2. On the contrary, by interleaved execution of two models can achieve higher throughput and resource utilization as shown in Fig. 1 (b).

Challenge in Multi-DNN Scheduling. However, multi-DNN scheduling incurs an inevitable slowdown of individual requests. The vertical line in Fig. 1 marks the start and end times of each individual request. For the baseline schedule (a), the execution of the second request (M2) is postponed until the end of the first request (M1) to increase the tail latency. While interleaving two requests as in (b) slightly mitigates this problem, it is not sufficient. Figure 2 shows the latency distribution of two randomly-arriving request streams with different QoS constraints (taken from MLPerf-inference [6]): MobileNetV2 (MN) and BERT-large (BL). A majority of the requests with both the baseline and *Layerweaver* violate the QoS constraints. It is because their scheduling logic is *QoS-oblivious*. In the case of *Layerweaver*, while most of the BERT-large (NLP) requests satisfy the QoS constraints, over 90% of the MobileNetV2 (vision) requests violate them. Due to the growing importance of support for multi-tenancy on NPUs, the lack of QoS is a serious drawback in datacenters [3].

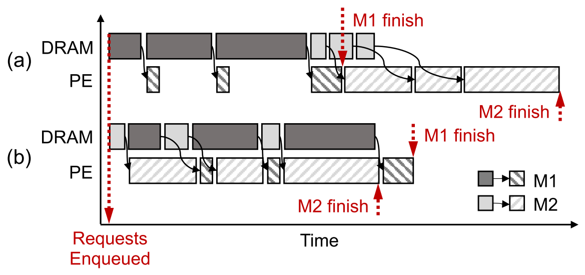

Fig. 1 Execution timeline with two DNNs: memory-intensive (M1) and compute-intensive (M2) models. (a) illustrates a schedule *without* layer-wise interleaving across models and (b) *with* interleaving across models.

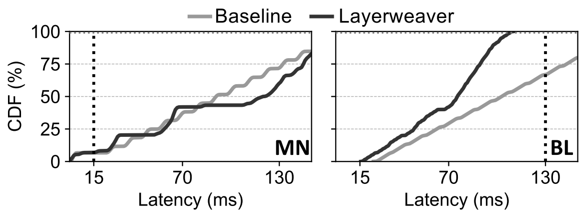

Fig. 2 Latency distribution of MobileNetV2 (MN) and BERT-large (BL) with their QoS constraints (15ms and 130ms) in vertical lines [6].

2.2 Design and Implementation

Layerweaver+ replaces the greedy scheduler of *Layerweaver* [5] to achieve the two design goals: maximizing the inference throughput and bounding each request's latency to a given deadline. To this end, we introduce two operation modes in *Layerweaver+*: 1) *Select the layer that causes the minimum resource idle time first*, and 2) *Select the layer with the minimum QoS slack time*. In the first mode, the QoS slack time is used for a tie-breaker (i.e., prioritizing the layer with the smallest slack time). *Layerweaver+* usually operates in Mode 1 to maximize system throughput. However, when QoS violation is imminent for a request, the scheduler switches to Mode 2 to schedule the next layer from that request.

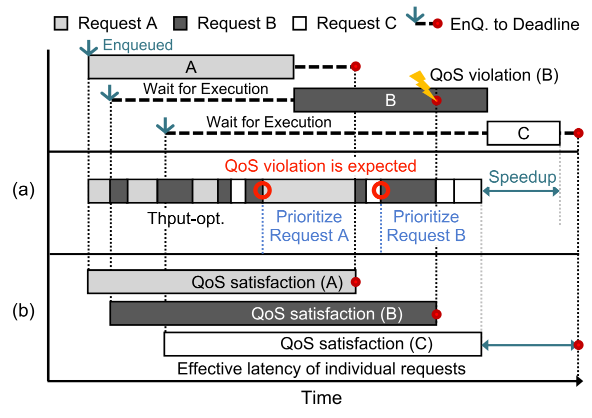

Fig. 3 Illustration of *Layerweaver+* scheduling.

Scheduling Algorithm. Figure 3 compares the scheduling behaviors of *Layerweaver* with the baseline scheduler using a simple example. In the baseline scheduler executing one request at a time (Top), each request will be executed in the order of arrival. In this case, a QoS violation happens for Request B. Although omitted for brevity, *Layerweaver* with no consideration for QoS can also cause potential QoS violations for both Request A and B as merely optimizing throughput can cause an unbounded slowdown to starved requests. In contrast, Fig. 3 (a) and (b) show the scheduling behaviors of *Layerweaver+*, which prioritizes a request whose deadline is imminent. With this mechanism, *Layerweaver+* generates a schedule that meets the deadlines of all three requests at the expense of slight degradation in system throughput due to prioritization without interleaving multiple models.

Algorithm 1 shows a pseudocode of the *Layerweaver+*'s scheduling function invoked to select the next layer. It takes the model information (M_i), which includes its layer-wise cycle execution time and progress, the QoS constraint (QoS_i) and the request arrival time ($EnqTime_i$) as input. The algorithm then finds the layer that incurs the minimum idle time to optimize throughput (Line 5-9) and temporarily updates the schedule state (Line 10). The Mode 1 scheduling is similar to *Layerweaver* [5] except for the case where multiple candidate layers of the same minimum idle time compete for selection. If there are such layers, the algorithm selects a layer from the request with the least QoS slack time as a tie-breaker. Then, the algorithm now checks if QoS violation could happen when applying the previous throughput-first decision (Line 12-15). Finally, if the QoS slack time is long enough so that no QoS violation is expected, throughput optimization (Line 16-18) is chosen or vice-versa (Line 20-22) with the finalization process for the next scheduler call.

For the estimation of the remaining standalone execution time for a request, we utilize an analytical model with the following equation: $\sum_{L=progress,...,end} \max(m_L, c_L)$ where m_L and c_L denote memory access time and computation time of layer L, respectively. This equation approximates the remaining standalone execution time of a DNN by approximating each layer's execution time as the larger of its memory access time and compute time. This simple equation allows the scheduler to calculate the slack time quickly and effectively prevents QoS violation according to our evaluation in Sect. 3.2 (*Layerweaver+ (Approx.)*).

Algorithm 1 QoS-aware multi-DNN scheduling func.

```
Input: requestQ containing a list of
    <Model info $M_i$, QoS target $QoS_i$, Request arrival time $EnqTime_i$>
Output: A next layer to be scheduled
1:  function SCHEDULE(requestQ)
2:      $CurSched \leftarrow$ Current schedule state of NPU
3:      $T_{PE} \leftarrow$ Latest timestamp of PE util.
4:      // Mode 1: Maximize throughput
5:      for $M_i, QoS_i, EnqTime_i \in$ requestQ.all() do
6:          $SlackTime_i \leftarrow EnqTime_i + QoS_i - T_{PE}$
7:          $IdleTime_i \leftarrow$ GetIdleTime($M_i, CurSched$)
8:      end for
9:      $thput \leftarrow \mathbf{argmin}_i\ IdleTime_i$, where tie-break w/ $SlackTime_i$
10:     $TT_{PE}, TmpSched \leftarrow$ UpdateSchedule($M_{thput}$)
11:     // Mode 2: If QoS violation is predicted, prioritize it
12:     $urgent \leftarrow \mathbf{argmin}_i\ SlackTime_i$
13:     $StnTime_{urgent} \leftarrow$ GetStandaloneExecTime($M_{urgent}$)
14:     $SlackTime_{urgent} \leftarrow EnqTime_{urgent} + QoS_{urgent} - TT_{PE}$
15:     if $SlackTime_{urgent} > StnTime_{urgent}$ then
16:         $CurSched \leftarrow TmpSched$
17:         CheckEndThenDequeue($M_{thput}$, requestQ)
18:         return $M_{thput}$.GetLayerInProgress()
19:     else
20:         $t_{PE}, CurSched \leftarrow$ UpdateSchedule($M_{urgent}$)
21:         CheckEndThenDequeue($M_{urgent}$, requestQ)
22:         return $M_{urgent}$.GetLayerInProgress()
23:     end if
24: end function
```

3. Evaluation

3.1 Methodology

Simulation Setup. To model cycle-level behaviors of an NPU, we have extended MAESTRO [13] to estimate the computation and data transfer time while considering the effect of data prefetch and random query arrivals. We used an NPU accelerator featuring 128 TOP/s of computation, 100 GB/s off-chip memory bandwidth and 50MB on-chip buffer size supposing inference-only NPU. We select three compute-intensive models: InceptionV3 (IC), MobileNetV2 (MN) and three ResNet50 (RN); and memory-intensive models: BERT-base (BB), BERT-large (BL), and XLNet (XL).

Service Scenario. We extend the server scenario of MLPerf inference [6] benchmark to support multiple different kinds of inference requests. It models a case where a single NPU serves randomly arrived requests for different DNN services. Each request has a specific QoS constraint which represents its hard deadline. We set the QoS constraints to be <15ms, <130ms for computer vision tasks (RN, RX, and MN) and NLP tasks (BB, BL, and XL), respectively. We compare the maximum STP (System Throughput) that the underlying system can sustain with less than 1% QoS violation [6]. We evaluate three schedulers explained in Sect. 2.2:

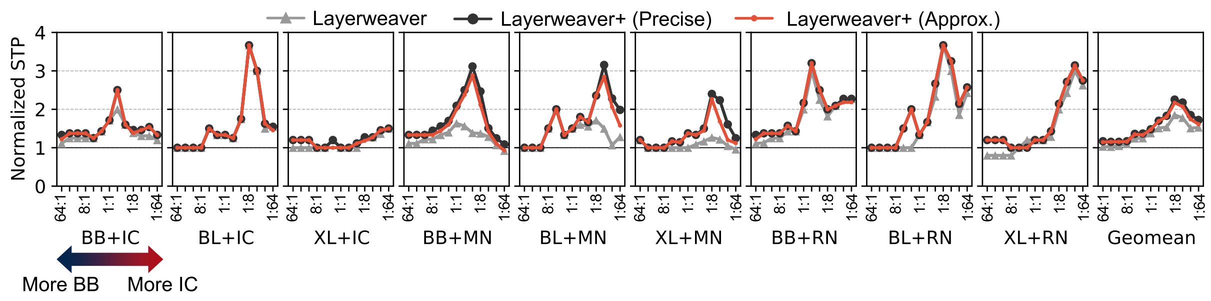

Fig. 4 Maximum sustainable system throughput (STP) normalized to *Baseline* for multi-stream server scenario with two streams. The X-axis represents the varying request arrival ratio between the two streams (with 2× per step). The leftmost tick corresponds to a 64:1 ratio where the memory-intensive model (e.g., BB) has a 64× higher arrival rate than the compute-intensive model (e.g., IC).

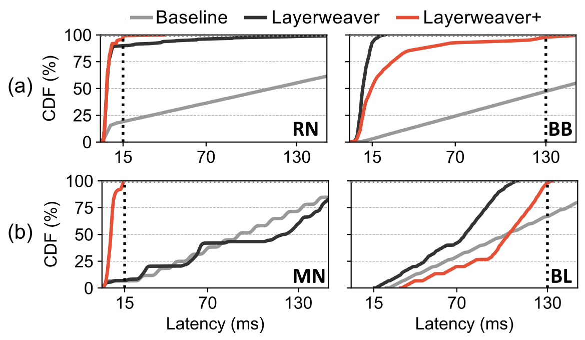

Fig. 5 Tail latency of (a) ResNet50 (RN) and BERT-base (BB); and (b) MobileNetV2 (MN) and BERT-large (BL). The vertical lines represent QoS constraints for vision (15ms) and NLP (130ms) tasks [6].

Baseline [14], *Layerweaver* [5] and *Layerweaver+*.

3.2 Results

Throughput. Figure 4 shows the maximum sustainable system throughput (STP) normalized to the baseline. Overall, *Layerweaver* even without considering QoS achieves notably better throughput than the baseline with request batching. This is because *Layerweaver* can interweave two requests, achieving much higher resource utilization than the baseline which serves one batch of the same model at a time. *Layerweaver+* (Sect. 2.2) further improves the throughput as it substantially reduces the number of requests violating QoS constraints. Moreover, it also shows that the approximate scheduler utilizing the simple equation for estimating standalone execution time (Layerweaver+ *(Approx.)*) delivers comparable throughput to a hypothetical precise scheduler utilizing the precise standalone execution time (Layerweaver+ *(Precise)*). Thus, the approximate scheduler works well with negligible performance degradation while reducing the computation cost of the scheduler significantly.
Tail Latency. Figure 5 compares the tail latency distribution of the three schedulers in question. Specifically, Fig. 5 (a) shows the tail latency of two interleaved models (RN and BB) when requests are injected at the rate of $QPS_{RN} = 800$ and $QPS_{BB} = 200$. For the baseline, about 80% (50%) of the RN (BB) requests fail to meet the 15 ms (130 ms) deadline, mainly because the overall system throughput is too low without layer-wise interleaving. *Layerweaver* substantially improves the system throughput. However, 10% of the RN requests still violate the 15 ms deadline because *Layerweaver* schedules BB more frequently to maximize the overall throughput, even though BB has a relatively loose deadline and thus does not have to be scheduled that frequently. In contrast, *Layerweaver+* prioritizes RN requests having a tighter deadline when their QoS slack becomes small. With this mechanism, *Layerweaver+* can serve 99%+ requests within the specified deadlines. A similar behavior is observed in Fig. 5 (b) at the injection rate of $QPS_{MN} = 7530$, $QPS_{BL} = 470$ as well.

4. Conclusion

This paper presents *Layerweaver+*, a DNN inference serving system with a novel QoS-aware multi-model scheduler. The proposed algorithm effectively prevents QoS violation while maintaining much higher system throughput (STP) than the baseline decoupled execution with no overlap between models. *Layerweaver+* demonstrates an average of 49.2% improvement (by up to 266.7%) in system throughput compared to the baseline for a realistic QoS-constrained scenario.

References

[1] L.A. Barroso, J. Clidaras, and U. Hölzle, The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition, Morgan & Claypool Publishers, 2013.
[2] Google, "Cloud TPU at Google Cloud," https://cloud.google.com/tpu/docs/system-architecture, 2018.
[3] N.P. Jouppi, D.H. Yoon, M. Ashcraft, M. Gottscho, T.B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma, T. Norrie, N. Patil, S. Prasad, C. Young, Z. Zhou, and D. Patterson, "Ten lessons from three generations shaped Google's TPUv4i," The 48th Annual International Symposium on Computer Architecture (ISCA), 2021.
[4] M. Pellauer, Y.S. Shao, J. Clemons, N. Crago, K. Hegde, R. Venkatesan, S.W. Keckler, C.W. Fletcher, and J. Emer, "Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration," The 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2019.
[5] Y.H. Oh, S. Kim, Y. Jin, S. Son, J. Bae, J. Lee, Y. Park, D.U. Kim,

T.J. Ham, and J.W. Lee, "Layerweaver: Maximizing resource utilization of neural processing units via layer-wise scheduling," The 27th IEEE International Symposium on High Performance Computer Architecture (HPCA), 2021.

[6] MLCommons, "MLPerf Inference Benchmark Suite," https://github.com/mlcommons/inference, 2021.

[7] H. Zhou, S. Bateni, and C. Liu, "S^3DNN: Supervised streaming and scheduling for gpu-accelerated real-time dnn workloads," IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2018.

[8] N. Capodieci, R. Cavicchioli, M. Bertogna, and A. Paramakuru, "Deadline-based scheduling for GPU with preemption support," IEEE Real-Time Systems Symposium (RTSS), 2018.

[9] Y. Xiang and H. Kim, "Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference," IEEE Real-Time Systems Symposium (RTSS), 2019.

[10] Q. Chen, H. Yang, M. Guo, R.S. Kannan, J. Mars, and L. Tang, "Prophet: Precise QoS prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers," The 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2017.

[11] T.T. Yeh, M.D. Sinclair, B.M. Beckmann, and T.G. Rogers, "Deadline-aware offloading for high-throughput accelerators," The 27th IEEE International Symposium on High Performance Computer Architecture (HPCA), 2021.

[12] E. Baek, D. Kwon, and J. Kim, "A multi-neural network acceleration architecture," The 47th Annual International Symposium on Computer Architecture (ISCA), 2020.

[13] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Understanding reuse, performance, and hardware cost of DNN dataflow: A data-centric approach," The 52nd Annual International Symposium on Microarchitecture (MICRO), 2019.

[14] NVIDIA, "NVIDIA Triton Inference Server." https://developer.nvidia.com/nvidia-triton-inference-server, 2020.