Layerweaver: Maximizing Resource Utilization of Neural Processing Units via Layer-Wise Scheduling

Young H. Oh* Seonghak Kim† Yunho Jin† Sam Son† Jonghyun Bae†
Jongsung Lee† Yeonhong Park† Dong Uk Kim† Tae Jun Ham† Jae W. Lee†

*Department of Electrical and Computer Engineering
Sungkyunkwan University, Suwon, Korea

†Department of Computer Science and Engineering
Neural Processing Research Center (NPRC)
Seoul National University, Seoul, Korea

*younghwan@skku.edu †{ksh1102,yhjin0509,sosson97,jonghbae,leitia,ilil96,dongukim12,taejunham,jaewlee}@snu.ac.kr

***Abstract*—To meet surging demands for deep learning inference services, many cloud computing vendors employ high-performance specialized accelerators, called *neural processing units* (NPUs). One important challenge for effective use of NPUs is to achieve high resource utilization over a wide spectrum of deep neural network (DNN) models with diverse arithmetic intensities. There is often an intrinsic mismatch between the compute-to-memory bandwidth ratio of an NPU and the arithmetic intensity of the model it executes, leading to under-utilization of either compute resources or memory bandwidth. Ideally, we want to saturate both compute TOP/s and DRAM bandwidth to achieve high system throughput. Thus, we propose *Layerweaver*, an inference serving system with a novel multi-model time-multiplexing scheduler for NPUs. *Layerweaver* reduces the temporal waste of computation resources by interweaving layer execution of multiple different models with opposing characteristics: compute-intensive and memory-intensive. *Layerweaver* hides the memory time of a memory-intensive model by overlapping it with the relatively long computation time of a compute-intensive model, thereby minimizing the idle time of the computation units waiting for off-chip data transfers. For a two-model serving scenario of batch 1 with 16 different pairs of compute- and memory-intensive models, *Layerweaver* improves the temporal utilization of computation units and memory channels by 44.0% and 28.7%, respectively, to increase the system throughput by 60.1% on average, over the baseline executing one model at a time.**

***Keywords*-Layer-wise Scheduling, Systems for Machine Learning, Inference Serving System, Neural Networks, Accelerator Systems, Multi-tasking**

I. Introduction

With widespread adoption of deep learning-based applications and services, computational demands for efficient deep neural network (DNN) processing have surged in datacenters [1]. In addition to already popular services such as advertisement [2], [3], social networks [4]–[7], and personal assistants [8], [9], emerging services for automobiles and IoT devices [10]–[12] are also attracting great attention.

To accelerate key deep-learning applications, datacenter providers widely adopt high-performance serving systems [13]–[16] based on specialized DNN accelerators, or *neural processing units (NPUs)* [17]–[19]. Such accelerators have a massive amount of computation units delivering up to several hundreds of tera-operations per second (TOP/s) as well as hundreds of giga-bytes per second (GB/s) DRAM bandwidth. NPUs targeting only inference tasks feature a relatively high compute-to-memory bandwidth ratio (TOP/GB) [18], [20], whereas those targeting both inference and training a much lower ratio due to their requirement for supporting floating-point arithmetic, which incurs larger area and bandwidth overhead.

Due to a wide spectrum of arithmetic intensities of DNN models, there is no one-size-fit-all accelerator that works well for all of them. For example, convolutional neural networks (CNNs) [21]–[23] are traditionally known to be most compute-intensive. In contrast, machine translation and natural language processing (NLP) models [4], [24]–[26] are often composed of fully-connected (FC) layers with little weight reuse to be more memory-intensive. To first order, the computational structure of a DNN model determines its arithmetic intensity, and hence its suitability to a particular NPU.

Even if the arithmetic intensity of a DNN model is perfectly balanced with the compute-to-memory ratio of the NPU, it is still challenging to fully saturate the hardware resources. One such difficulty comes from varying batch size at runtime. For example, the dynamic fluctuation of user requests towards a DNN serving system [13], [15], [16] or application-specified batch size from cameras and sensors in an autonomous driving system [10], [27] mandates the system to run on a sub-optimal batch size. Thus, the batch size is often highly workload-dependent and not freely controllable to make it nearly intractable to build a single NPU maintaining high resource utilization for all such workloads.

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

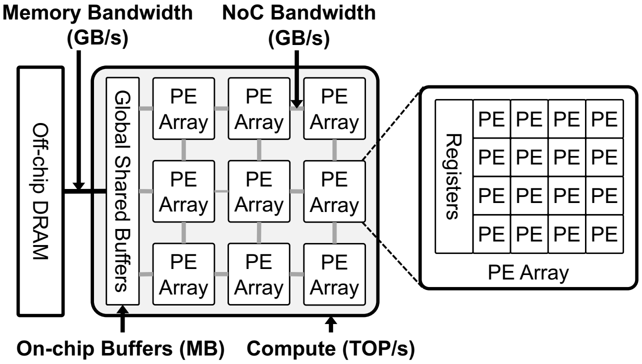


Fig. 1. An abstract single-chip NPU architecture.

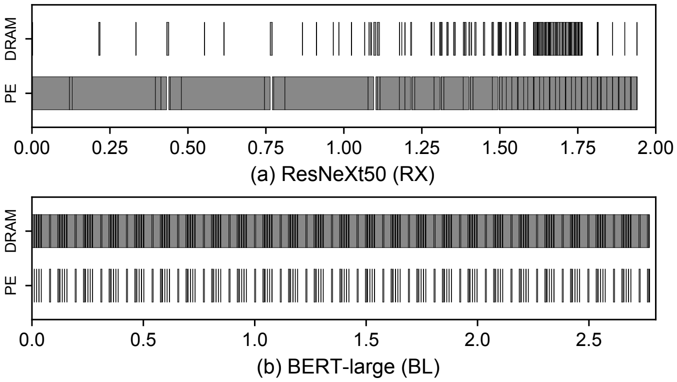


Fig. 2. Layer-wise execution timeline on a unit-batch (batch 1) input in milliseconds with (a) compute-intensive and (b) memory-intensive workload.

To improve resource utilization of the NPU, latency-hiding techniques such as pipelining [28]–[30], decoupled memory access and execution [31] have been proposed. These techniques overlaps compute time with memory access time by fetching dependent input activations and weights from the off-chip memory while performing computation. While mitigating the problem to a certain extent, the imbalance between compute and memory access time eventually limits their effectiveness. To eliminate the memory bandwidth bottleneck, one popular approach is to employ expensive high-bandwidth memory technologies such as HBM DRAM. However, this abundant bandwidth is wasted when running compute-intensive models (like CNNs). On the other hand, small-sized NPUs traditionally count on double buffering [32], [33] and dataflow optimization to minimize off-chip DRAM accesses [34]–[36].

Instead, we take a software-centric approach that exploits concurrent execution of multiple DNN models with opposite characteristics to balance NPU resource utilization. To this end, we propose *Layerweaver*, an inference serving system with a novel time-multiplexing layer-wise scheduler. The low-cost scheduling algorithm searches for an optimal time-multiplexed layer-wise execution order from multiple heterogeneous DNN serving requests. By interweaving the execution of both compute- and memory-intensive layers, *Layerweaver* effectively balances the temporal usage of both compute and memory bandwidth resources. Our evaluation of *Layerweaver* on 16 pairs of compute- and memory-intensive DNN models demonstrates an average of 60.1% and 53.9% improvement in system throughput for single- and multi-batch streams, respectively, over the baseline executing one model at a time. This is attributed to an increase in temporal utilization of PEs and DRAM channels by 44.0% (22.8%) and 28.7% (40.7%), respectively, for single-batch (multi-batch) streams.

Our contributions are summarized as follows.

- We observe the existence of a significant imbalance between the compute-to-memory bandwidth ratio of an NPU and the arithmetic intensity of DNN models to identify opportunities for balancing the resource usage via layer-wise time-multiplexing of multiple DNN models.
- We devise a novel time-multiplexing scheduler to balance the NPU resource usage, which is applicable to a wide range of NPUs and DNN models. The proposed algorithm computes the resource idle time and selects the best schedules within a candidate group of layers. For this reason, *Layerweaver* finds a near-optimal schedule that achieves better performance than heuristic approaches.
- We provide a detailed evaluation of *Layerweaver* using 16 pairs of state-of-the-art DNN models with two realistic inference scenarios. We demonstrate the effectiveness of *Layerweaver* for increasing system throughput by eliminating idle cycles of compute and memory bandwidth resources.

II. Background and Motivation

A. Neural Processing Units

While GPUs and FPGAs are popular for accelerating DNN workloads, a higher efficiency can be expected by using specialized ASICs [17], [18], [29], [37], or neural processing units (NPUs). Figure 1 depicts a canonical NPU architecture. A 2D array of processing elements (PE) are placed and interconnected by a network-on-a-chip (NoC), where each PE array has local SRAM registers. There are also globally shared buffers to reduce off-chip DRAM accesses.

Recently, with ever growing demands for energy-efficient DNN processing, specialized accelerator platforms have been actively investigated for both datacenters [8], [20], [38], [39] and mobile/IoT devices [12], [40]–[43]. Depending on the application each request may be a single input instance or mini-batch with multiple instances. When a request arrives at the host device to which an NPU is attached, it transfers model parameters and input data to the device-side memory via the PCIe interface (or on-chip bus in an SoC). Then the NPU fetches the data and commences inference computation.

B. DNN Model Characteristics

The characteristics of a DNN is determined by the characteristics of the layers it comprises. For convolution neural networks (CNNs), convolution layers typically take up the most of the inference time with the remaining time spent on batch normalization (BN), activation, fully-connected (FC), and pooling layers. In a convolution layer most of the data (activations and filter weights) are reused with a sliding window, so it has a very high compute-to-memory bandwidth ratio (i.e., highly compute-intensive). Figure 2(a) shows an example execution timeline of ResNeXt50 [21], where PEs

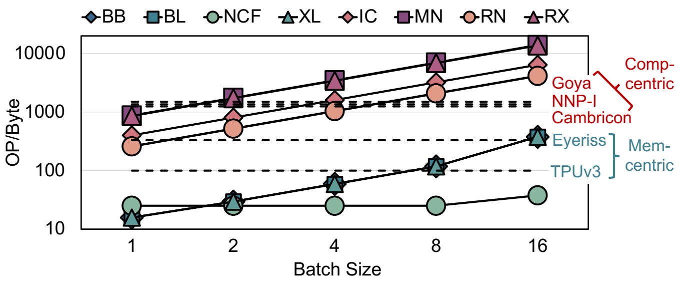


Fig. 3. Compute-to-memory bandwidth ratios across varying batch sizes. Arithmetic intensity (Number of operations divided by off-chip access bytes) is measured for DNN models. Peak TOPS is divided by peak off-chip DRAM bandwidth for NPUs.

are very heavily used while DRAM bandwidth is under-utilized. In contrast, most neural language processing (NLP) and recommendation models are dominated by fully-connected (FC) layers which have little reuse of weight data. Thus, the FC layer has memory-intensive characteristics. As shown in Figure 2(b), BERT-large [24] has high utilization of off-chip DRAM bandwidth but much lower utilization of PEs.

C. NPU Resource Under-utilization Problem

Compute-centric vs. Memory-centric NPUs. In a datacenter environment NPUs are generally capable of servicing multiple requests at once as service throughput is an important measure. In contrast, in a mobile environment, NPUs are often required to provide a low latency for a single request to not degrade user experience. As such, depending on the requirements from the target environment, NPUs have varying compute-to-memory bandwidth ratios.

Figure 3 overlays the arithmetic intensity of eight popular DNN models with the compute-to-memory bandwidth ratio of five NPUs while varying the batch size of the input. The following four of the eight models are known to be *compute-intensive*: InceptionV3 (IC) [44], MobileNetV2 (MN) [45], ResNet50 (RN) [22] and ResNeXt50 (RX) [21]. The remaining four models are *memory-intensive*: BERT-base (BB), BERT-large (BL) [24], NCF (NCF) [26] and XLNet (XL) [25]. Furthermore, we also classify five NPUs into *compute-centric* designs [18], [19], [46], which has a relatively high compute-to-memory bandwidth ratio, and *memory-centric* designs [17], [34], which is characterized by a low ratio. If a compute-intensive model (e.g., ResNeXt50) is executed on a memory-centric NPU (e.g., TPUv3), the memory bandwidth resource is likely to be under-utilized (i.e., bottlenecked by PEs) to yield sub-optimal system throughput.

Imbalance between DNN Model and NPU. There is a wide spectrum of NPUs and DNN models to make it nearly impossible to balance the NPU resources with the requirements from *all* DNN models to execute. In Figure 3, for a given NPU, the farther the distance from the horizontal line of the NPU to the arithmetic intensity of the workload, the greater degree of imbalance exists to yield poor resource utilization of either PEs or DRAM bandwidth. If the arithmetic intensity of the DNN model in question is above the horizontal line of the NPU, DRAM bandwidth is under-utilized, or vice versa.

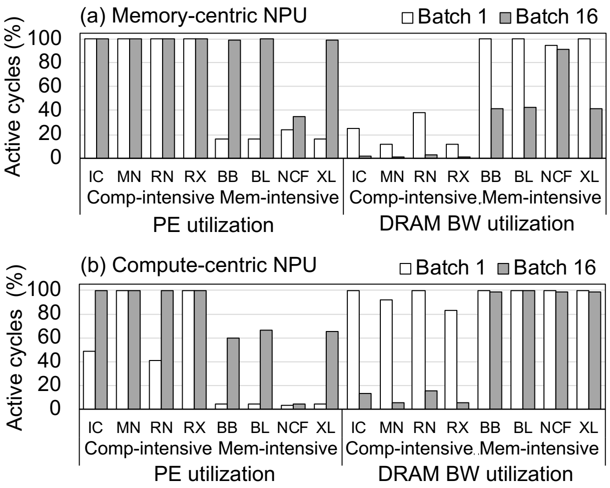


Fig. 4. Normalized active cycles of compute-centric and memory-centric NPUs on various DNN models.

Figure 4 characterizes the resource utilization in greater details. The two bars for each DNN model shows normalized active cycles (i.e., temporal utilization) of both resources for unit-batch (batch 1) and multi-batch (batch 16) inputs, respectively. We use both compute-centric (NNP-I) and memory-centric NPUs (TPUv3) in Figure 4(a) and (b). The details of this experimental setup is available in Section IV-A.

In Figure 4(a), the memory-centric NPU (TPUv3-like) under-utilizes DRAM bandwidth for compute-intensive models and PE resources of memory-intensive models at the unit batch (batch 1, white bars). This under-utilization is explained by the large gap between the horizontal line of the NPU and the DNN arithmetic intensity. However, at batch 16 (gray bars), memory-intensive DNN models show high PE utilization and lower DRAM bandwidth utilization compared to unit batch size except for NCF, which is extremely memory-intensive, due to an increase in the arithmetic intensity. Figure 3 shows that the points of the memory-intensive workloads are located above the horizontal line except NCF. Still, the compute-intensive models show very low DRAM bandwidth utilization.

In Figure 4(b), the compute-centric NPU (NNP-I-like) shows opposite results. Some compute-intensive workloads demonstrate relatively low PE utilization because of insufficient computations at the unit batch. However, with batch 16, now DRAM bandwidth is under-utilized while PE resources are fully saturated. Conversely, in the case of memory-intensive workloads, PE resources are still not fully saturated even at batch 16 as DRAM bandwidth gets saturated first.

III. LayerWeaver

A. Overview

The analysis in Section II-C implies that (i) either PE or DRAM bandwidth resources (or both) may be under-utilized due to a mismatch between the arithmetic intensity of a DNN model and the compute-to-memory bandwidth ratio of the NPU; (ii) the imbalance of resource usage cannot be eliminated by simply adjusting the batch size of a single DNN model. Thus, we propose to execute two different DNN models with opposite

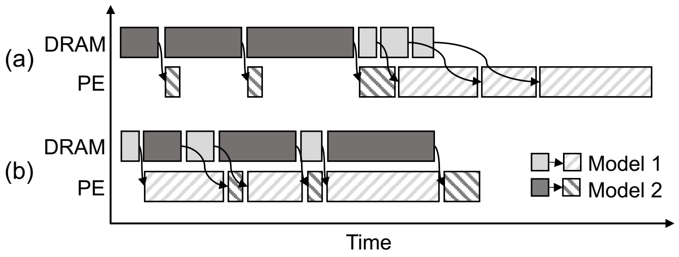


Fig. 5. A timeline with different scheduling. Based on decoupled memory system, (a) illustrates the schedule without reordering and (b) with reordering.

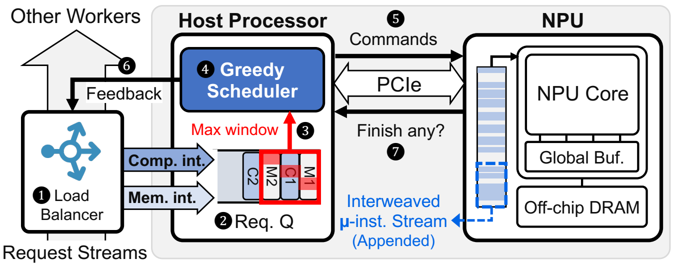


Fig. 6. A baseline NPU-incorporated serving system.

characteristics via layer-wise time-multiplexing to balance the resource usage for a given NPU.

Figure 5 illustrates how interweaving the layers of two heterogeneous DNN models can lead to balanced resource usage. In this setup we assume that the DNN serving system needs to execute two hypothetical 3-layer models, a compute-intensive one (Model 1) and a memory-intensive one (Model 2). The baseline schedule in Figure 5(a) does not allow reordering between layers. During the execution of Model 2, the PE time is under-utilized, whereas during the execution of Model 1, the DRAM time is under-utilized. However, by reordering layers across the two models appropriately as in Figure 5(b), the two models as a whole utilize both PE and DRAM bandwidth resources in a much more balanced manner.

Challenges for Efficient Layer-wise Scheduling. Determining an efficient schedule to fully utilize both compute and memory resources is not a simple task. It would have been very easy if the on-chip buffer had an infinite size. In such a case, simply prioritizing layers having longer compute time than memory time would be sufficient to maximize compute utilization as the whole memory time will be completely hidden by computation. However, unfortunately, the on-chip buffer size is very limited, and thus one needs to carefully consider the prefetched data size as well as the remaining on-chip buffer size. This is because prefetching too early incurs memory idle time as described in the “Memory Idle Time” paragraph in Section III-D.

Layerweaver. *Layerweaver* is an inference serving system with a layer-wise weaving scheduler for NPU. The core idea of *Layerweaver* is to interweave multiple DNN models of opposite characteristics, thereby abating processing elements (PEs) and DRAM bandwidth idle time. Figure 6 represents the overall architecture of *Layerweaver*. The design goal of *Layerweaver* is to find a layer-wise schedule of execution that can finish all necessary computations for a given set of requests in the shortest time possible.

Deployment. *Layerweaver* is comprised of a request queue, scheduler, and NPU hardware for inference computing. It could be often integrated with a cloud load balancer that ❶ directs the proper amount of inference queries for each compute- and memory-intensive models to a particular NPU instance [16], [47]–[49]. Such load-balancers are important in existing systems as well since supplying an excessive number of queries to a single NPU instance can result in an unacceptable latency explosion. ❷ Once request queues of *Layerweaver* accepts the set of requests for each model, ❸ the host processor dequeues a certain number of requests from each queue and pass them to the scheduler. And then, ❹ the host processor invokes *Layerweaver* scheduler and performs the scheduling. ❺ Following the scheduling result, the host processor dispatches those scheduled layers to NPU by appending to activated instruction streams. ❻ Depending on the scheduling results, one of two request queues may have higher occupancy. In that case, it is reported to the cloud load balancer so that the load balancer can utilize this information for the future load distribution. Finally, ❼ NPU executes those instructions, and once it finishes handling a single batch of requests, it returns the result to the host processor. Note that *Layerweaver* employs a greedy scheduler (see the next section), and thus only needs information about the one next layer for each model.

Baseline NPU. There exist many different types of NPUs with their own unique architecture. However, *Layerweaver* is not really dependent on the specific NPU architecture. Throughout the paper, we assume a generic NPU that resembles many of the popular commercial/academic NPUs such as Google Cloud TPUv3 [17] or Intel NNP-I [18]. This generic NPU consists of compute units (PEs), and two shared on-chip scratchpad memory buffers, one for weights and one for activations. The host controls this NPU by issuing a stream of commands such as fetching data to on-chip memory or performing computation. Once the NPU finishes computation, it automatically frees the consumed weights, and stores the outcome at the specified location of the activation buffer. One important aspect is that the NPU processes computation and main memory accesses in a decoupled fashion. The NPU eagerly processes fetch commands from the host by performing weight transfers from its main memory to the on-chip weight buffer as long as the buffer has empty space. In a similar way, its processing unit eagerly processes computation commands from the host as long as its inputs (i.e., weights) are ready.

Supporting the layer-wise interweaving in this baseline NPU does not require an extension. In fact, the NPU does not even need to be aware that it is running layers from different models. The host can run *Layerweaver* scheduler to obtain the effective layer-wise interweaved schedule, and then simply issue commands corresponding to the obtained schedule. Unfortunately, we find that existing commercial NPUs do not yet expose the low-level API that enables the end-user to directly control the NPU’s scratchpad memory. However, it is reasonable to assume that such APIs are internally available [17], [20], [50]. In this case, we believe that the developer can readily utilize *Layerweaver* on the target NPU.

B. Greedy Scheduler

The main challenge of finding an optimal layer-wise scheduling is the enormous size of the scheduling search space. Brute-force approach naturally leads to a burst of computation cost. For example, to find the optimal schedule for a model set consisted of ResNet50 and BERT-base, which has 53 and 75 layers respectively, ${}_{128}\mathrm{C}_{53}$ ($\simeq 4 \times 10^{36}$) candidate schedules should be investigated, which is not feasible. Note that the previous work [51] uses simplified heuristics to manage the high search cost, while *Layerweaver* presents a way to formally calculate the exact idle time of each resource and maximize the total resource utilization.

To determine suitable execution order of layers for k different models within a practically short time, the scheduler adopts a greedy layer selection approach. It estimates computation and memory idle time incurred by each candidate layer then selects a layer showing the least idle time as the next scheduled layer. Here, the algorithm maintains a *candidate group* and only considers layers in the group to be scheduled next. For one model, a layer belongs to the group if and only if it is the first unscheduled layer of the model. Assuming that an inspection of a single potential candidate layer takes $O(1)$ time, the complexity for making a single scheduling decision is $O(k)$. Assuming that this process is repeated for N layers, the overall complexity is $O(kN)$ for N layers.

Profiling. Before launching the scheduler, *Layerweaver* requires to profile a DNN model to identify computation time, memory usage, and execution order of each layer. This profiling stage simply performs a few inference operations and then records information for each layer L of the model. Specifically, it records the computation time `L[comp]`, and the number of weights that this layer needs to fetch from the memory `L[size]`. Finally, such pairs are stored in a list M following the original execution order. Since most NPUs have a deterministic performance characteristic, offline profiling is sufficient.

Greedy Scheduler. Figure 7 shows the working process of the scheduler. We assumed NPU running *Layerweaver* has `BufSize` sized on-chip buffer and `MemBW` off-chip memory bandwidth. It maintains three auxiliary data structures during its run. The algorithm selects one layer from the candidate group and append it to the end of `curSchedule` every step, and this data structure is the outcome of *Layerweaver* once the algorithm completes. `indexWindow` represents the indexes of layers in the candidate group of each model to track the layer execution progress correctly. Lastly, `schedState` keeps several information representing the current schedule.

For each step the algorithm determines the next layer to be scheduled among candidates. First, the algorithm constructs `candidateGroup` from `indexWindow`. Then, for each candidate layer `UpdateSchedule` function computes how the NPU state changes if a candidate layer is scheduled as described in Section III-C. Updated schedule states are stored in `stateList`. Next, `Select` function examines all schedule states in `stateList` and estimates the idle time of each updated schedule state, which will be further elaborated in Section III-D. It selects the layer having the shortest idle time as the next scheduled layer, which is appended to `curSchedule`.

```
1  def Schedule($M_0$, ..., $M_{k-1}$):
2    totalSteps = $\sum_{i=0,\ldots,k-1}(len(M_i))$
3    curSchedule = [ ]
4    indexWindow = [0, ..., 0] # length k
5    schedState = [$t_m: 0, t_c: 0, l$ :[ ]]
6      for step in range(totalSteps):
7        # checks for pause
8        for i in range(k):
9          CheckEnd(indexWindow[i], $M_i$)
10       # one scheduling step
11       candidateGroup = [$M_i$[idx] for (i, idx) in
             enumerate(indexWindow)]
12       stateList = [ ]
13       for modelNum in range(k):
14         # schedule state update
15         newSchedState = UpdateSchedule(schedState,
               candidateGroup[modelNum])
16         stateList.append(newSchedState)
17       # layer selection
18       schedState, selectedIdx = Select(schedState,
             stateList, candidateGroup)
19       curSchedule.append(candidateGroup[selectedIdx])
20       indexWindow[selectedIdx]++
21   return curSchedule
```

Fig. 7. Greedy layer schedule algorithm.

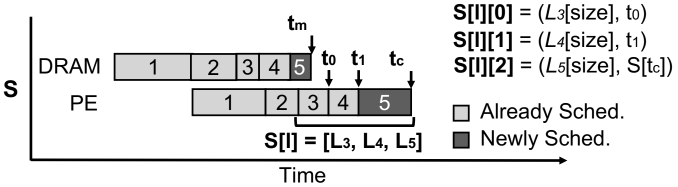


Fig. 8. Schedule state of an example schedule `S`.

At the beginning of every step, `CheckEnd` function checks `indexWindow[]` to see if scheduling of any model is completed (i.e., all of its layers are scheduled). If so, the scheduler is paused and scheduled layers up to this point (as recorded in `curSchedule`) are dispatched to NPU. Just before the completion of execution, the scheduler is awoken and continues from where it left off. `CheckEnd` then probes the request queue in search of any remaining workload. If there is nothing left, the scheduler is terminated. If there are additional requests queued, they are appended to existing requests of the same model, if any. And then the scheduler resumes.

C. Maintaining and Updating Schedule State

This section provides intricate detail of `UpdateSchedule` (Line 15 in Figure 7). The function explores the effect of scheduling a layer in the `candidateGroup` on the overall schedule and pass the information to `select`.

Concept of Schedule State. *Layerweaver* maintains the state for a specific schedule `S`. This state consists of three elements: compute completion timestamp `S[t`$_\mathrm{c}$`]`, communication completion timestamp `S[t`$_\mathrm{m}$`]`, and a list `S[l]` containing information about already scheduled layers that are expected to finish in a time interval (`S[t`$_\mathrm{m}$`]`, `S[t`$_\mathrm{c}$`]`]. Specifically, the list `S[l]` consists of pairs where each entry (`S[l][j].size`, `S[l][j].completion`) represents the on-chip memory usage and the completion time of the `j`th layer in the list, respectively. Figure 8 illustrates an example schedule and elements com-

```
1  # called by UpdateSchedule
2  def ScheduleMemFetch(S, L):
3    sizeToFetch = L[size]
4    remainingBuf = BufSize - $\sum_j$S[l][j].size
5    curTime = S[$t_m$]
6    if sizeToFetch <= remainingBuf:
7      return curTime + sizeToFetch / MemBW
8    else:
9      curTime = curTime + remainingBuf / MemBW
10     sizeToFetch -= remainingBuf
11     for j in range(len(S[l])):
12       if curTime < S[l][j].completion:
13         curTime = S[l][j].completion
14       if sizeToFetch < S[l][j].size:
15         return curTime + sizeToFetch / MemBW
16       else:
17         sizeToFetch -= S[l][j].size
18         curTime = curTime + S[l][j].size / MemBW
```

Fig. 9. Scheduling Memory Fetch for Layer L on Schedule S. BufSize represents the on-chip buffer capacity, and MemBW represents the system's memory bandwidth.

posing its state. Below, we discuss how scheduling a specific layer L changes the each element of the schedule state.

Scheduling Memory Fetch. Figure 9 shows the process of scheduling the memory fetch represented in pseudocode. If the layer L's memory size L[size] is smaller than the amount of available on-chip memory at time $S[t_m]$, the memory fetch can simply start at $S[t_m]$ and finish at $S[t_m]$ + L[size]/MemBW (Line 7). This case is shown in Figure 10(a). However, if the amount of available on-chip memory at time $S[t_m]$ is not sufficient, this becomes trickier, as shown in Figure 10(b). First, a portion of L[size] that fits the currently remaining on-chip memory capacity is scheduled (Line 9). Then, the remaining amount (i.e., sizeToFetch in Line 10) is scheduled when the computation for a layer in list S[l] completes and frees the on-chip memory. For this purpose, the code iterates over list S[l]. For each iteration, the code checks if the jth layer in the list has completed by the time that previous fetch has completed (Line 12). If so, it immediately schedules a fetch for the freed amount (Line 14-18). If not, it waits until this layer completes and then schedules a fetch (Line 16-18). This process is repeated until L[size] amount of data is fetched. The returned value of the algorithm is $S'[t_m]$.

Scheduling Computation. Once the memory fetch ends, the computation for layer L can be scheduled. Here, there are two different cases. In the first case (Figure 11(a)), the previous schedule's computation has not ended by the time that memory fetch completes (i.e., $S[t_c] > S'[t_m]$). In this case, computation is scheduled on time $S[t_c]$ and completes at $S[t_c]$ + L[comp]. On the other hand, if the previous schedule's computation has ended before the time that memory fetch for the current layer completes (i.e., $S[t_c] < S'[t_m]$ as shown in Figure 11(b)), the compute needs to start on time $S'[t_m]$ (since it is dependent on the fetched memory), and completes at $S'[t_m]$ + L[comp]. The following equation summarizes the process of obtaining $S'[t_c]$.

$$S'[t_c] = \max(S[t_c], S'[t_m]) + L[comp]$$

Updating S[l]. Finally, S[l] needs to be updated accordingly. To obtain S'[l], all layer j in S[l] whose completion time is before $S'[t_m]$ are excluded. Then, the current layer L is appended to S[l].

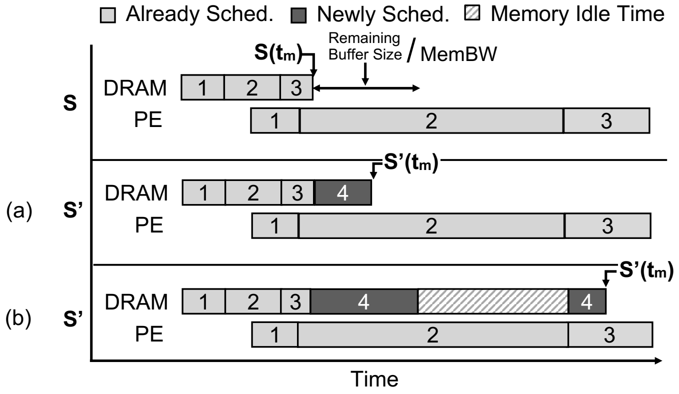


Fig. 10. Visualization of memory fetch scheduling.

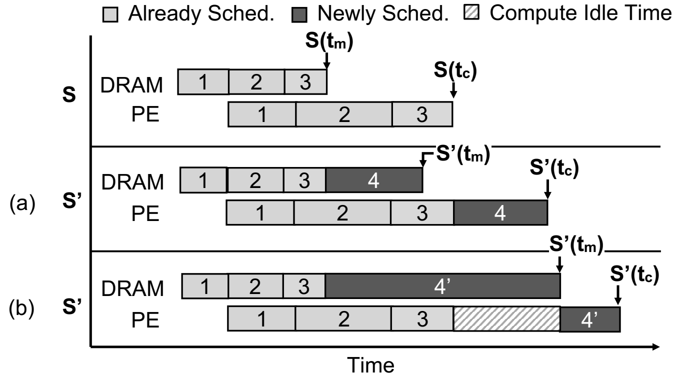


Fig. 11. Visualization of computation scheduling.

D. Selecting a Layer to Schedule

The goal of *Layerweaver* scheduler is clear: maximize both the PE utilization, and the DRAM bandwidth utilization (i.e., bandwidth utilization of off-chip memory links). To effectively achieve this goal, the scheduler should execute *the layer that incurs the shortest compute or memory idle time*. Here, we explain how *Layerweaver* estimates the amount of compute or memory idle time incurred if layer L is scheduled following the current schedule S by Select.

Decoupling Distance. For a schedule S, its decoupling distance is defined as $S[t_c]$ - $S[t_m]$. And maintaining an appropriate decoupling distance is important. If this distance is too large, it means that the prefetch is happening far before the fetched values are actually used resulting in memory idle time as on-chip buffers have a finite size (Figure 10(b)). On the other hand, if this value is too small, it means that the prefetch is happening right before the fetched values are used, leading to compute idle time (Figure 11(b)). In what follows we cover both cases in greater details.

Compute Idle Time. This occurs when the decoupling distance (i.e., $S[t_c]$ - $S[t_m]$) is smaller than L[size] / MemBW. In this case, memory fetch cannot finish within the decoupling distance, and PEs should wait until the memory fetch is completed (Figure 11(b)). The idle time can be computed as follows.

$$L[size] / MemBW - (S[t_c] - S[t_m])$$

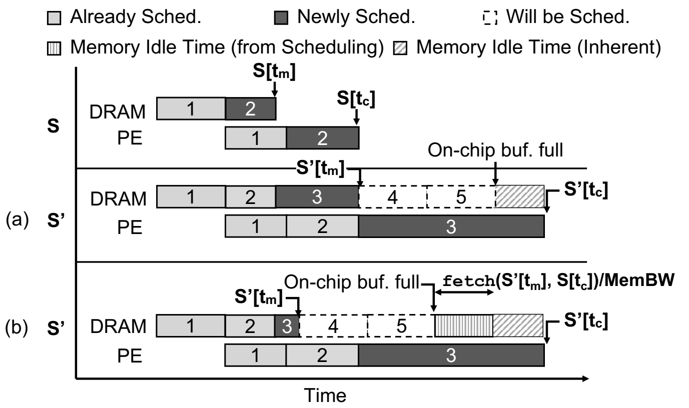


Fig. 12. Illustration of the memory idle time.

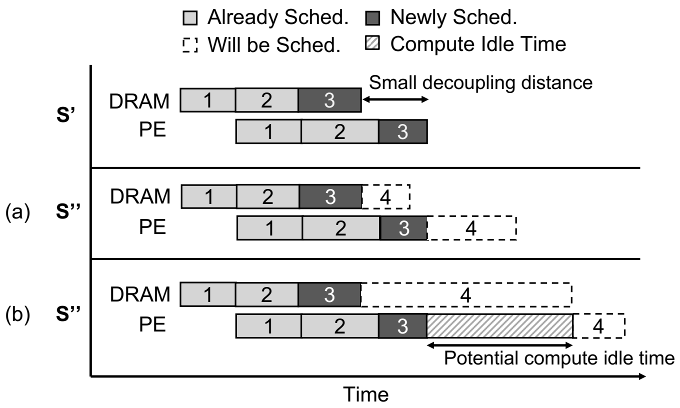


Fig. 13. Illustration of the potential compute idle time.

Memory Idle Time. Identifying the amount of memory idle time incurred from a scheduling decision is trickier. This is because the timeline of the current schedule does not actually show the memory idle time. Figure 12(a) shows an example case illustrating a scheduling decision that incurs large memory idle time. In this case, a layer L with the very large compute time (i.e., `L[comp]`) is scheduled. Unfortunately, the resulting decoupling distance is so large and exceeds the amount of time it takes to completely fill the on-chip buffer with the assumed system memory bandwidth. In such a case, regardless of a layer scheduled following the current layer, the hatched area of the timeline (i.e., `L[comp] - (BufSize - L[size]) / MemBW`) remains as memory idle time. However, one should note that this is **not** the result of the scheduling decision. Rather, this is a layer's inherent characteristic because such a memory idle time occurs regardless of the point that this layer is scheduled.

Still, this does not mean that one can schedule such a layer anywhere without any implication. Figure 12(b) shows a more general case where S'[t_m] is not equal to S[t_c]. In this case, the same memory idle time exists. However, the situation is worse here, because more fetch operations (for the next layers) will be scheduled in a time interval (S'[t_m], S[t_c]). By the time S[t_c], the amount of available on-chip buffer may be much lower than that of Figure 12(a) (i.e., `BufSize - L[size]`). Say that the amount of memory fetch operations that will be scheduled in a time interval (S'[t_m], S[t_c]) is fetch(S'[t_m], S[t_c]). fetch(S'[t_m], S[t_c]) can be computed by inspecting `S'[l]` and such a process is similar to Line 10-17 in Figure 9. In this case, the remaining on-chip buffer at time S[t_c] is BufSize - L[size]-fetch(S'[t_m], S[t_c]). As a result, the resulting memory idle time is L[comp] - (BufSize - (L[size] + fetch(S'[t_m], S[t_c]))) / MemBW. However, note that `L[comp] - (BufSize - L[size]) / MemBW`) is not the result of the scheduling decision. The additional amount of memory idle time resulting from the scheduling decision is as follows.

$$\mathtt{fetch}(\mathtt{S'}[\mathtt{t}_\mathtt{m}],\ \mathtt{S}[\mathtt{t}_\mathtt{c}])\ /\ \mathtt{MemBW}$$

Potential Compute Idle Time. There is an additional implication of a scheduling decision. A scheduling decision can potentially incur a compute idle time in the future, depending on the next layer that is scheduled. Figure 13 illustrates this case. As shown in the figure, the scheduling of a layer L resulted in a relatively small decoupling distance. This does not incur a compute idle time when the next scheduled layer's `L[size]` is small, as shown in Figure 13(a). However, if candidate layers for the next scheduling step have large memory time, it ends up occurring a compute idle time, as shown in Figure 13(b). To avoid such a potential compute idle time, it is better to maintain the decoupling distance that is at least as large as the time it takes to fetch the data for the largest layer (i.e., maximum of `L[size] / MemBW` for all L in currently running models). In this case, the amount of potential compute idle time for a scheduling decision is as follows.

$$\mathtt{L[size]}_{\mathrm{max}}\ /\ \mathtt{MemBW}\ -\ (\mathtt{S'}[\mathtt{t}_\mathtt{c}]\ -\ \mathtt{S'}[\mathtt{t}_\mathtt{m}])$$

Layer Selection. Given a set of candidate layers to schedule, *Layerweaver* computes the total idle time that each scheduling decision incurs. Then, the one that incurs the minimum total idle time is selected and scheduled. In a case where multiple candidate layers incur zero total idle time, one that does not incur inherent memory idle time (i.e., `L[comp] - (BufSize - L[size]) / MemBW`) < 0) is selected. Finally, for further tie-breaking, one with the largest decoupling distance (i.e., S'[t_c] - S'[t_m]) is selected. One drawback of this greedy policy is that it can potentially incur starvation. If a model contains a highly unbalanced layer (e.g., very large memory usage with a very little compute), this layer is likely to be never selected by our scheduler despite the model has layers with more favorable characteristics following that unbalanced layer.

Starvation Prevention. To avoid starvation, *Layerweaver* sometimes schedules the layer that yields a longer total idle time. Specifically, the first case is where all candidates are incurring compute idle time (excluding potential ones). This indicates that all candidate layers are memory-intensive. This is not the steady-state behavior since *Layerweaver* only targets scenarios where there exist at least one or more compute-intensive models. However, if a memory-intensive layer from the compute-intensive model is not scheduled, *Layerweaver* will continue to encounter memory-intensive candidates, and the circumstance can persist. In this case, *Layerweaver* selects a layer from the compute-intensive model regardless of the exact size of compute idle time it incurs. By doing so, a candidate layer from the compute-intensive model will eventually be a compute-intensive one and effectively continue operation.

Similarly, there is also a case where all candidate layers are incurring memory idle time. For a similar reason, *Layerweaver* selects a layer from the memory-intensive model. We find that this is sufficient to avoid starvation, considering that all other cases (e.g., some candidate layers are compute-intensive while the others are memory-intensive) cannot starve a single model.

E. Discussion

Scheduling Cost. Our scheduler has $O(kN)$ complexity, where k is the number of models, and N is the number of layers to interweave at a single scheduler invocation. We measured the latency of our scheduling algorithm using a single core of Intel i7-7700K CPU @ 4.20GHz. For two models, the measured scheduler throughput is about 15 layers/μs. Considering that the average latency of a single layer in our evaluated workloads (w/ batch size = 1) on our evaluated NPUs (see Section IV-A) ranges from 4.7us to 221us, the time spent on scheduling is much smaller than the time spent on DNN models. Furthermore, such scheduling happens off-critical path most of the time using the host CPU. We also verified that the scheduling time scales linearly with the number of models and the number of layers as expected.

Scheduling Granularity. Section III-A explains that *Layerweaver* generates a schedule by interweaving layers from each model. By layer, we meant basic building blocks of neural networks such as convolutional layer and FC (dense) layer. Both TensorFlow Keras and PyTorch list a set of supported layers in their documentation [52], [53]. However, this is just an example. In fact, the minimum scheduling granularity of *Layerweaver* is tied with the granularity of instruction that a target NPU supports. For example, if the host processor utilizes a single instruction for a sequence of layers (layer fusion), *Layerweaver* can interleave the schedule at that granularity. On the other hand, if the host utilizes finer-grained instructions (e.g., different instructions for matrix multiplication and the followed elementwise activation), *Layerweaver* can operate at that granularity. In general, *Layerweaver* can perform better with the finer-grained instructions. For platforms that only support very coarse-grained instructions, *Layerweaver* can be implemented in hardware by extending the NPU design. For example, AI-MT [51] utilizes hardware extensions to enable fine-grained data movements as well as computation. Doing so allows them to minimize the on-chip storage requirements for the weight buffer.

Context Switch Overhead. To minimize the context switch overhead (i.e., the overhead of executing a layer from one model then executing another layer from a different model), *Layerweaver* requires an activation buffer that can house activations for two models. By doing so, even when executing a layer from a different model, no extra off-chip data transfer needs to happen. Each model's activation buffer is sized to fit the largest activation size of the model for the given batch size. In our evaluation, the model that required the largest activation buffer size was MobileNetV2 (2.2MB) and ResNet50 (1.5MB) for single-batch inference. In other words, *Layerweaver* requires an additional 1.53MB activation buffer. The storage overhead becomes larger as the batch size increases, but the relative overhead remains the same since the original activation buffer size increases at the same time. Table I shows the storage overhead of multi-model execution on two different evaluated NPU configurations.

TABLE I
NPU CONFIGURATION PARAMETERS

Compute-centric architecture [18]	
Peak throughput	92 TOP/s
# of PEs	12 × 4096 (12 ICE)
PE operating frequency	927 MHz
Memory BW	68 GB/s (12 ICE)
On-chip SRAM	48 MB (Weight) / 2.3 MB (ACT) 1.5 MB (Extra ACT Buffer)
Memory-centric architecture [17], [55]	
Peak throughput	22.5 TOP/s
# of PEs	128 × 128 (1 MXU)
PE operating frequency	700 MHz
Memory BW	225 GB/s (1 MXU)
On-chip SRAM	48 MB (Weight) / 36 MB (ACT) 24 MB (Extra ACT Buffer)
Common parameters	
Arithmetic precision	16 bits
Dataflow	Weight-stationary

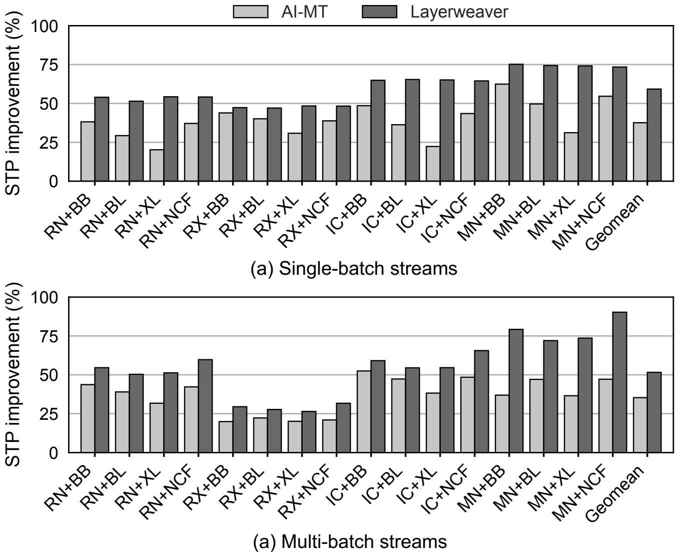
Failsafe Mechanism. *Layerweaver* gets very limited or no benefit at all when all tasks are compute-intensive or memory-intensive at the same time. In such a case, it is impossible to achieve decent performance improvement. For example, if all workloads are compute-intensive, the scheduling decision does not really make a difference. All choices will incur memory idle time, and there will be near-zero compute idle time. For this reason, if *Layerweaver* detects that all provided workloads are compute-intensive or memory-intensive during the profiling run, *Layerweaver* is disabled.

IV. Evaluation

A. Methodology

Simulation Setup. To estimate the computation cycles of NPU, we used MAESTRO [54], which analytically estimates computation cycles with various architectural parameters. We set two types of the NPU, NNP-I-like compute-centric NPU [18], and TPUv3-like memory-centric NPU [17], [55]. For dataflow we use weight stationary dataflow [56]. The detailed parameters are summarized in Table I. We built a custom simulator to model the layer-wise execution behavior. We estimated computation cycles from MAESTRO. To estimate memory behavior, we calculate the data transfer time from off-chip DRAM for each layer using its tensor dimensions.

Workloads. We select four popular DNNs for each of the two workload groups: compute-intensive and memory-intensive. Thus, the total number of DNN pairs taking one from each group is 16. The four compute-intensive models are InceptionV3 (IC) [44], MobileNetV2 (MN) [45], ResNet50 (RN) [22], and ResNeXt50 (RX) [21]. For memory-intensive models, we use BERT-base (BB), BERT-large (BL) [24], NCF (NCF) [26], and XLNet (XL) [25].



(a) Single-batch streams

(a) Multi-batch streams

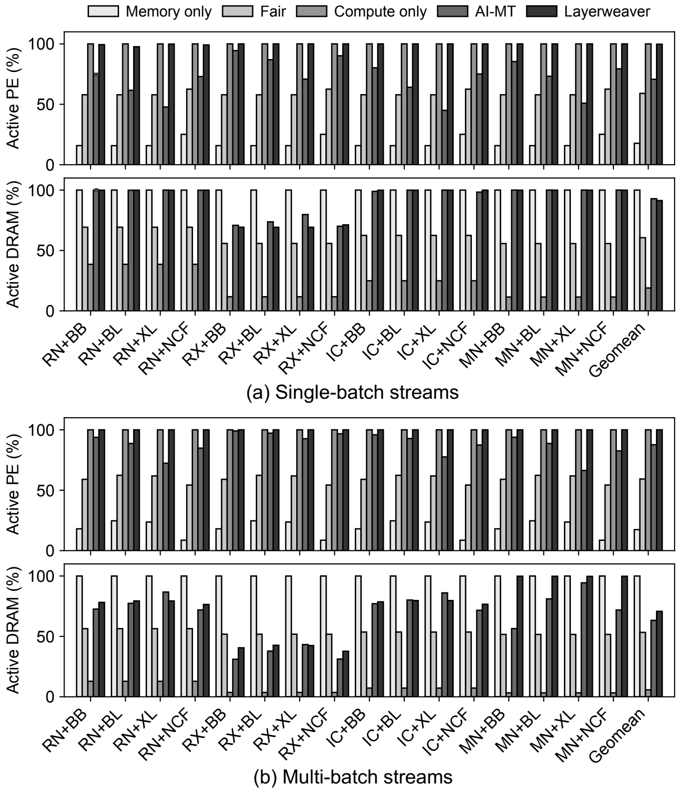
Fig. 14. System throughput (STP) improvement on (a) a memory-centric NPU for single-batch streams and (b) a compute-centric NPU for multi-batch streams (# streams=2). Higher is better.

B. Evaluation Scenarios

We extend the single-stream scenario of MLPerf inference [27] benchmark to support multiple different kinds of inference requests. As stated in Section II, *Layerweaver* can enhance various kinds of NPUs when there is an inherent mismatch between their compute-memory capabilities and arithmetic intensities of the DNNs being served.

Schedulers. For evaluation we compare *Layerweaver* with three baseline schedulers as well as a (nearly) concurrent work AI-MT [51]. The three baseline schedulers include: 1) scheduling only computation-intensive models (*Compute-only*), 2) scheduling only memory-intensive models (*Memory-only*), 3) scheduling both computation-intensive and memory-intensive models by bisecting the cycles equally and allocating each half to a specific model (*Fair*). Note that those baselines used for our evaluation substantially outperform layer-wise double buffering [32], which is used as a baseline scheme of AI-MT. We also carefully model the features of AI-MT, including memory block prefetch, compute block merging, and priority mechanism. AI-MT requires setting two user-defined thresholds, and through extensive 2D parameter sweeping we use the setting that yields the best overall throughput for our workloads. All schedulers are run on the same hardware configuration specified in Table I.

Metric. We evaluate *Layerweaver* by measuring the system throughput (STP) [57], which is a common metric to quantify the performance of multiple workloads running on an NPU. To compute this metric, each query from a model gets the weight that is proportional to its latency in standalone execution. Then, we compare the weighted queries per second using various schedulers. For example, assume that model A takes 5ms and model B 10ms to process a single query. Suppose Scheduler 1 processes four queries of model A within 20ms and Scheduler 2 processes two queries of model B within the same interval. Then, their STPs are equal according to our metric.



(a) Single-batch streams

(b) Multi-batch streams

Fig. 15. Portion of active cycles on (a) a memory-centric NPU for single-batch streams and (b) a compute-centric NPU for multi-batch streams (# streams=2).

Single-Batch Streams with Memory-centric NPU. This scenario reflects a case where a single NPU is asked to handle multiple tasks simultaneously while achieving the maximum throughput. In this scenario, whenever the NPU completes a query for a model, the next query for the same model is immediately dispatched. Such a single-batch inference is popular for time series or real-time data. For such data, achieving better throughput in this scenario results in a better processing rate (e.g., frames per second, sensor frequency, etc.). *Layerweaver* can resolve a severe resource under-utilization problem that a memory-centric NPU experiences on single-batch workloads (Figure 4 in Section II-C).

Multi-Batch Streams with Compute-centric NPU. This scenario is analogous to the previous scenario except that a request is batched with more than one inputs. For evaluation, we use the batch size of 16 unless specified otherwise. This scenario may correspond to a case where the NPU is required to run multiple models, and each model is invoked with inputs collected from multiple sources at a regular interval (e.g., autonomous driving, edge computing). *Layerweaver* can alleviate the resource under-utilization problem of a compute-centric NPU on multi-batch workloads.

C. Results

Throughput. Figure 14(a) shows the system throughput (STP) over various model combinations. *Layerweaver* improves the system throughput on memory-centric NPU for single-batch streams (# streams = 2) by 60.1% on average (up to 75.2%) compared to the three baselines. Note that all three baseline

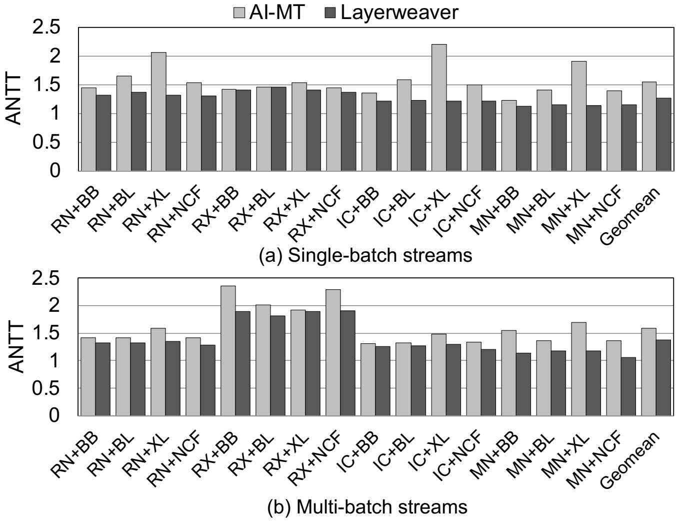


Fig. 16. Average Normalized Turnaround Time (ANTT) with (a) single-batch streams and (b) multi-batch streams (# streams=2). Lower is better.

schedulers (Section IV-B) have the same system throughput because they simply execute different combinations of two models with no layer-wise interweaving. Similarly, Figure 14(b) shows that *Layerweaver* improves the system throughput on compute-centric NPU for multi-batch streams (# streams = 2) by 53.9% on average (up to 90.2%). These results translate to 21.6% and 16.3% higher geomean throughput than AI-MT for single- and multi-batch streams, respectively. (A more detailed analysis is to be presented later in this section.) *Layerweaver* effectively interweaves layers to achieve a much higher resource utilization, and hence substantial throughput gains.

Utilization of PE Cycles and DRAM Bandwidth. Figure 15(a) shows the portion of active cycles for PEs and memory on the single-batch streams scenario, and Figure 15(b) shows the same on the multi-batch streams scenario. On average, *Layerweaver* achieves 99.7% and 91.3% utilization of PE cycles and DRAM bandwidth, respectively, for the single-batch streams scenario, and 99.9% and 70.7% for the multi-batch streams scenario. The baseline schedulers share the same resource under-utilization problem. The *Compute-only* and *Memory-only* schedulers end up with a low utilization for either DRAM bandwidth or PE cycles. Even if two DNNs are time-multiplexed at a model granularity (i.e., Fair), they end up with a mediocre level of utilization for both resources.

In contrast, *Layerweaver* improves the resource utilization of both PE cycles and DRAM BW by scheduling layers in a way that minimizes the resource idle time. Note that *Layerweaver* does not fully utilize DRAM bandwidth in some cases (e.g., combinations containing RX in Figure 15(a) and some combinations in Figure 15(b)). This is due to an inherent imbalance between compute and memory time leading to memory idle time (i.e., `L[comp]` - (`BufSize` - `L[size]`) / `MemBW`) as discussed with Figure 12 in Section III-D. This problem can be alleviated by increasing the on-chip buffer size (`BufSize`) or adding more PEs to reduce compute time (`L[comp]`).

Impact on Single Request Latency. Figure 16 presents the average normalized turnaround time (ANTT) for both single- and multi-batch streams. This metric is an average of latency slowdown compared to standalone execution for each model. By executing multiple, heterogeneous requests concurrently, *Layerweaver* trades the latency of an individual request for higher system throughput. The ANTTs of *Layerweaver* are 1.27 and 1.36 for single- and multi-batch streams, respectively, normalized to the standalone execution time. This result is much more favorable than AI-MT [51], whose ANTTs are 1.55 and 1.58. Furthermore, *Layerweaver* has much smaller variations of ANTT than AI-MT to demonstrate more robust performance. We also checked the geomean of maximum slowdown (i.e., 100% tail) for each workload on both AI-MT and *Layerweaver*. AI-MT exhibited 1.89× maximum slowdown and *Layerweaver* exhibited 1.40× maximum slowdown on single-batch streams. Similarly, AI-MT showed 1.90× maximum slowdown and *Layerweaver* exhibited 1.61× maximum slowdown on multi-batch streams. As expected, *Layerweaver* has much less impact on tail latency than AI-MT. Note that *Layerweaver*'s near-ideal schedule that fully utilizes both DRAM bandwidth and PE still incurs a certain level of slowdown. This is inevitable in cases where a single model was already utilizing more than 50% of the resources. In such a case, an interleaving of two models can never achieve more than 2× speedup, and thus the slowdown is unavoidable.

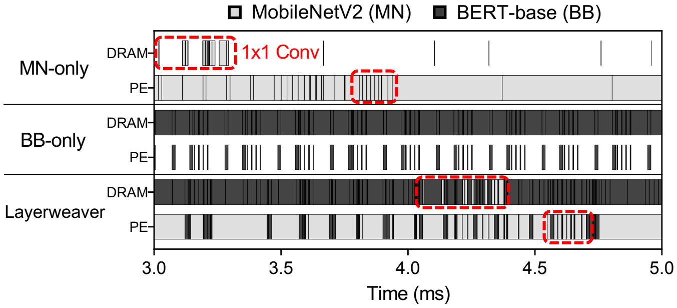


Fig. 17. Timeline analysis on BERT-base (BB) and MobileNetV2 (MN). Multi-batch streams (# streams = 2) is run on memory-centric device.

Timeline Analysis. As a case study, we present an execution timeline visualizing the busy cycles for PEs and DRAM channels in Figure 17 while executing two DNN models (MN and BB). Standalone execution leads to under-utilization of either compute or off-chip DRAM bandwidth resources. However, *Layerweaver* can successfully balance the memory and compute resource usage to minimize the resource idle time. One interesting observation is that *Layerweaver* can effectively handle memory-intensive layers in a compute-intensive model. The boxed area in red in the figure corresponds to the 1×1 and grouped convolution layers with very large input channels in MN [23]. These layers are memory-intensive although the whole model is known to be compute-intensive. In such a case, our scheduler prioritizes the compute-intensive model (MN) as described in Section III-D so that MN can quickly move to compute-intensive layers to prevent starvation of the model and improve overall resource utilization.

In-depth Comparison to AI-MT. AI-MT [51] is a very recent work that also utilizes time-multiplexed multi-DNN execution to improve system throughput. It aims to balance both compute

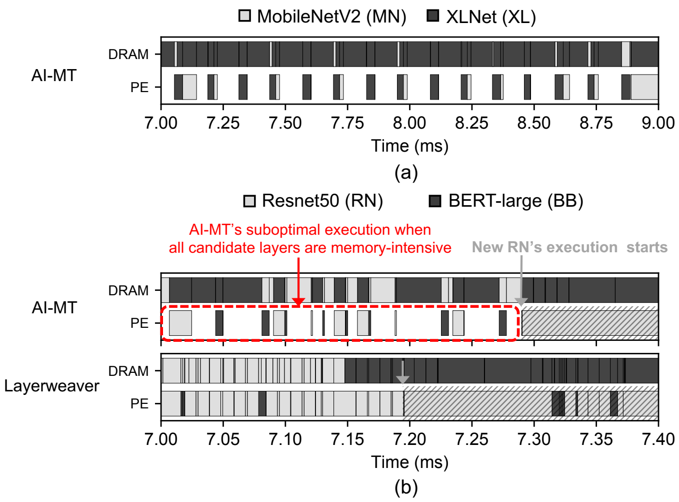


Fig. 18. Timeline analysis demonstrating (a) a case that AI-MT shows PE under-utilization with MobileNetV2 (MN) and XLNet (XL). And (b) shows a ResNet50 (RN) and BERT-large (BL) case that AI-MT suffer from the starvation originated from memory-intensive layers in compute-intensive models (e.g. FC layers in RN).

and memory resource usage by maintaining the decoupling distance within a desired level. One limitation of AI-MT is that its performance largely depends on the choice of two user-defined thresholds. Even with careful selection of those parameters via extensive 2D parameter sweeping, we find that *Layerweaver* consistently outperforms AI-MT across all workloads as shown in Figure 14 and Figure 16.

Figure 18(a) shows a case where AI-MT execution results in the PE under-utilization (MN+XL in Figure 14(b)). When AI-MT scheduler finds that the system has little on-chip memory space left, it blindly selects a layer with the least compute time, with an intention of avoiding the memory idle time that can occur from scheduling a layer with a long compute time. Unfortunately, in many cases, such a layer is the one from a memory-intensive model, which also consumes a large amount of on-chip memory space, and thus does not lower the on-chip memory usage. Thus, the problem persists and the system suffers from PE under-utilization since the scheduler favors layers from memory-intensive models. In contrast, *Layerweaver* achieves high performance for all cases by estimating the cost of a particular scheduling decision in a more formal way and directly aiming to minimize the resource idle time.

Figure 18(b) presents another scenario where AI-MT fails to make good scheduling decisions. This is a case where all layers in `candidateGroup` are all memory-intensive, and there exists a sufficient decoupling distance. The particular timeline shown in the figure is from the RN+BL workload where a compute-intensive model (RN) has a memory-intensive layer (e.g., FC or 1x1 Conv). In this case, AI-MT randomly selects a layer to execute. Unfortunately, every time AI-MT schedules a layer from BERT-large (BL) in this situation, a large amount of PE under-utilization happens. On the other hand, *Layerweaver* intelligently chooses to execute the memory-intensive layer from a compute-intensive model, expecting the compute-intensive layer to follow the current memory-intensive layer (explained in the "Starvation Prevention" paragraph in Section III-D). As a result, *Layerweaver* can avoid the unnecessary PE idle time that AI-MT suffers from.

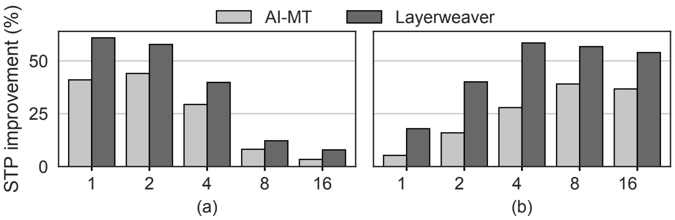


Fig. 19. Average system throughput (STP) on (a) memory-centric NPU and (b) compute-centric NPU for multi-batch streams (stream # = 2) scenario. Various batch size from 1 to 16 is used to demonstrate its sensitivity. The bold label denotes the selected batch size for workload-specific evaluation (Figure 14).

Sensitivity to Changes in Workload Characteristics. Figure 19 shows the average throughput improvement in N-batch streams for memory-centric and compute-centric NPUs. The setup is similar to Figure 14 except that we vary batch sizes for each NPU and report the geomean throughput improvement for each case. Changing the batch size affects the arithmetic intensity of the model. And both *Layerweaver* or AI-MT [51] benefit the most when one of the models is compute-intensive, and the other is memory-intensive in that particular NPU. However, depending on batch sizes, both models can be *relatively* compute-intensive or memory-intensive as implied in Figure 3. For example, Figure 19(a) shows that the larger batch size makes both workloads to be compute-intensive and lower the benefits of multi-model scheduling on memory-centric NPU. On the other hand, the smaller batch size makes both models to be memory-intensive on compute-centric NPU and lower the benefits of *Layerweaver* or AI-MT. In all cases, *Layerweaver* outperforms AI-MT by a significant margin.

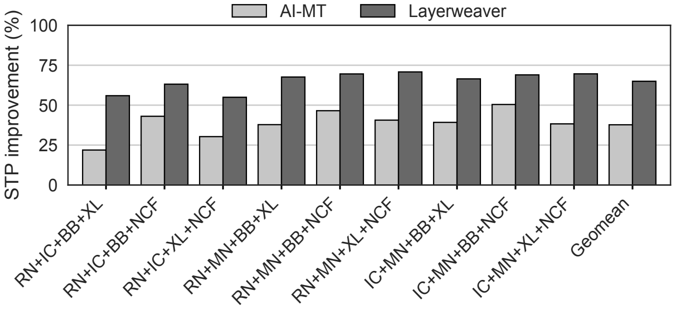


Fig. 20. Single-batch streams (# streams = 4) system throughput (STP) on a compute-centric NPU.

***Layerweaver* with More Than Two Models.** *Layerweaver* can also be utilized with more than two models. Here, we evaluate a case where four models (two compute-bound, two memory-bound) are deployed. Specifically, we assume a single-batch streams scenario on compute-centric NPU device. To make the number of combinations manageable, we do not include ResNeXt50 and BERT-large for this experiment to obtain 9 combinations (instead of 36) as shown in Figure 20. It shows that *Layerweaver* can achieve substantial speedups for these workloads as well. *Layerweaver* demonstrates 27.2% higher throughput improvement than AI-MT. However, we observe that utilizing *Layerweaver* for more than two models does not bring much additional benefit. Compared to the case of simply utilizing the schedule that stitches two-model-interweaved schedules (e.g., RN + BB schedule followed by IC + XL schedule), the four-model-interweaved schedule resulted in the small geomean STP gain (i.e., <1 % average). As long as one model is memory-intensive and the other is compute-intensive, *Layerweaver* almost fully utilizes resources just with two models.

V. Related work

Task Multi-tasking on Accelerators. Minimizing performance interference caused by multi-tasking on GPU has been previously studied [58]–[60]. Prophet [58] and Baymax [59] acknowledge that resource contention by task co-location and PCIe bandwidth contention caused by data transfers is crucial for multi-tasking performance. Thus, they identify the task co-location performance model to improve compute utilization of GPUs. In contrast, *Layerweaver* leverages different workload characteristics of DNN models to balance resource utilization on the emerging NPU hardware. AI-MT [51] proposes a TPU extension to support time-multiplexed multi-model execution for optimizing throughput. However, its scheduling algorithm largely relies on heuristics requiring fine tuning of two user-defined thresholds, which is burdensome and suboptimal. We quantitatively compared the quality of scheduling to demonstrate *Layerweaver* substantially outperforms AI-MT without dedicated hardware support or parameter tuning (Section IV-C).

Priority-based Task Preemption. To satisfy the latency constraints of high-priority inference tasks, preemption-based approaches have been proposed [61]–[64]. PREMA [61] introduces an effective preemption mechanism that considers the task size and its priority to balance throughput and latency, and a preemptible NPU architecture holding metadata for task switching. TimeGraph [62] proposes a real-time device-driver level scheduler based on two-priority policy using GPU resource usage. Tanasic et al. [64] devise two preemption mechanisms using context switch and GPU SM draining, respectively, to reduce the performance overhead of preemption. These proposals target to improve QoS for the latency of requests but not increase system throughput. Instead, *Layerweaver* demonstrates throughput improvement by co-scheduling multiple heterogeneous DNN models.

DNN Serving System Optimization. TensorFlow serving [15] is a production-grade DNN serving system for a serving system. Also, other serving systems such as SageMaker [65], Google AI platform [66], and Azure Machine Learning [67] offer separate online and offline services that automatically scale models based on their load. Clipper [13] targets a low-latency prediction serving system on top of various machine learning frameworks using caching, batching, adaptive model selection. Based on Clipper, Pretzel [14] improves the inference latency by optimizing the serving pipelines. Since these inference serving systems use the only coarse-grained (e.g., model, input batch) scheduling, the results can be suboptimal compared to *Layerweaver*, which exploits fine-grained scheduling at a layer granularity across different models.

VI. Conclusion

This paper presents *Layerweaver*, a DNN inference serving system with a novel multi-model scheduler, which eliminates temporal waste in compute and memory bandwidth resources via layer-wise time-multiplexing of two or more DNN models with different characteristics. The scheduling algorithm follows the concept of divide-and-conquer and finds a near-optimal schedule that minimizes the temporal waste. Our evaluation of *Layerweaver* on 16 pairs of compute- and memory-intensive DNN models demonstrates an average of 60.1% and 53.9% improvement in system throughput for single- and multi-batch streams, respectively, compared to the baseline decoupled execution with no overlap between models. This improvement is attributed to an increase in temporal utilization of PEs and DRAM channels via minimizing resource idle time.

Acknowledgment

This work is supported by Samsung Advanced Institute of Technology and the National Research Foundation of Korea grant funded by the Ministry of Science, ICT & Future Planning (PE Class Heterogeneous High Performance Computer Development, NRF-2016M3C4A7952587). Jae W. Lee is the corresponding author.

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] P. Li, "The DQN model based on the dual network for direct marketing," in *Proceedings of the 2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2018, pp. 1088–1093.

[3] L. M. Matos, P. Cortez, R. Mendes, and A. Moreau, "Using deep learning for mobile marketing user conversion prediction," in *Proceedings of the 2019 International Joint Conference on Neural Networks (IJCNN)*, 2019, pp. 1–8.

[4] U. Gupta, C. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang, "The architectural implications of facebook's DNN-based personalized recommendation," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 488–501.

[5] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018, pp. 620–629.

[6] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. Association for Computing Machinery, 2014, p. 701–710.

[7] A. Grover and J. Leskovec, "Node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. Association for Computing Machinery, 2016, pp. 855—864.

[8] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, L. Tang, and J. Mars, "Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015, p. 223–238.

[9] Google, "Google now," http://www.google.com/landing/now/.

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

[11] M. Yan, A. Li, M. Kalakrishnan, and P. Pastor, "Learning probabilistic multi-modal actor models for vision-based robotic grasping," in *Proceedings of the 2019 International Conference on Robotics and Automation (ICRA)*, May 2019, pp. 4804–4810.

[12] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "Deepthings: Distributed adaptive deep learning inference on resource-constrained IoT edge clusters," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2348–2359, 2018.

[13] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[14] Y. Lee, A. Scolari, B.-G. Chun, M. D. Santambrogio, M. Weimer, and M. Interlandi, "PRETZEL: Opening the black box of machine learning prediction serving systems," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2018, pp. 611–626.

[15] "TensorFlow serving models," https://www.tensorflow.org/tfx/guide/serving.

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

[17] "Cloud TPU at Google Cloud." https://cloud.google.com/tpu/docs/system-architecture.

[18] O. Wechsler, M. Behar, and B. Daga, "Spring Hill (NNP-I 1000), Intel's Data Center Inference Chip," in *A Symposium on High Performance Chips (Hot Chips)*, 2019.

[19] Cambricon, "Cambricon MLU100," http://www.cambricon.com.

[20] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[21] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," *CoRR:1611.05431*, 2016.

[22] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[23] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," *CoRR:1801.04381*, 2018.

[24] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *CoRR:1810.04805*, 2018.

[25] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. Salakhutdinov, and Q. V. Le, "XLNet: Generalized autoregressive pretraining for language understanding," *CoRR:1906.08237*, 2019.

[26] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T. Chua, "Neural collaborative filtering," *CoRR:1708.05031*, 2017.

[27] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "MLPerf inference: A benchmarking methodology for machine learning inference systems," in *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*. Association for Computing Machinery, 2020.

[28] M. Gao, X. Yang, J. Pu, M. Horowitz, and C. Kozyrakis, "TANGRAM: Optimized coarse-grained dataflow for scalable NN accelerators," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[29] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, S. G. Tell, Y. Zhang, W. J. Dally, J. Emer, C. T. Gray, B. Khailany, and S. W. Keckler, "Simba: Scaling deep-learning inference with multi-chip-module-based architecture," in *Proceedings of the 52nd Annual International Symposium on Microarchitecture (MICRO)*, 2019.

[30] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, and A. Raghunathan, "SCALEDEEP: A scalable compute architecture for learning and evaluating deep networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[31] 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," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[32] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2015.

[33] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *Proceedings of the 49th Annual International Symposium on Microarchitecture (MICRO)*, 2016.

[34] Y. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[35] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[36] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[37] T. J. Ham, S. J. Jung, S. Kim, Y. H. Oh, Y. Park, Y. Song, J.-H. Park, S. Lee, K. Park, J. W. Lee, and D.-K. Jeong, "A^3: Accelerating attention mechanisms in neural networks with approximation," in *Proceedings of the 26th IEEE International Symposium on High Performance Computer Architecture*, ser. HPCA, 2020.

[38] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual International Symposium on Microarchitecture (MICRO)*, 2014.

[39] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang, "Djinn and tonic: DNN as a service and its implications for future warehouse scale computers," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, p. 27–40.

[40] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017, p. 615–629.

[41] "Samsung Exynos 9 Series 9820," https://www.tensorflow.org/tfx/guide/serving.

[42] H. Li, K. Ota, and M. Dong, "Learning IoT in edge: Deep learning for the internet of things with edge computing," *IEEE Network*, vol. 32, no. 1, pp. 96–101, 2018.

[43] Google Cloud, "Edge TPU: Run inference at the edge," https://cloud.google.com/edge-tpu.

[44] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," 2015.

[45] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[46] Habana, "Habana Goya," https://habana.ai.

[47] "AWS Elastic Load Balancing," https://aws.amazon.com/ko/elasticloadbalancing.

[48] "Load Balancing on IBM Cloud," https://www.ibm.com/cloud/learn/load-balancing.

[49] "Kubeflow: The Machine Learning Toolkit for Kubernetes," https://www.kubeflow.org.

[50] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An instruction set architecture for neural networks," in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, 2016.

[51] E. Baek, D. Kwon, and J. Kim, "A multi-neural network acceleration architecture," in *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 940–953.

[52] "TensorFlow Core v2.3.0. Keras API docs." https://www.tensorflow.org/api_docs/python/tf/keras/layers, 2020.

[53] "PyTorch 1.6.0 documentation." https://pytorch.org/docs/stable/nn.html, 2020.

[54] 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," in *Proceedings of the 52nd Annual International Symposium on Microarchitecture (MICRO)*, 2019.

[55] Y. Wang, G.-Y. Wei, and D. Brooks, "A systematic methodology for analysis of deep learning hardware and software platforms," in *Proceedings of Machine Learning and Systems (MLSys)*, 2020, pp. 30–43.

[56] G. Zhou, J. Zhou, and H. Lin, "Research on NVIDIA deep learning accelerator," in *Proceedings of the 12th International Conference on Anti-counterfeiting, Security, and Identification (ASID)*, 2018.

[57] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, 2008.

[58] 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," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[59] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[60] P. Yu and M. Chowdhury, "Fine-grained GPU sharing primitives for deep learning applications," in *Proceedings of Machine Learning and Systems 2020 (MLSys)*, 2020, pp. 98–111.

[61] Y. Choi and M. Rhu, "PREMA: A predictive multi-task scheduling algorithm for preemptible neural processing units," in *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 220–233.

[62] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "Timegraph: GPU scheduling for real-time multi-tasking environments," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (ATC)*, 2011.

[63] G. A. Elliott, B. C. Ward, and J. H. Anderson, "Gpusync: A framework for real-time GPU management," in *Proceedings of the 2013 IEEE 34th Real-Time Systems Symposium (RTSS)*, 2013, pp. 33–44.

[64] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on GPUs," in *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 193–204.

[65] Amazon, "Amazon SageMaker," https://aws.amazon.com/sagemaker/.

[66] Google, "Google AI platform," https://cloud.google.com/ai-platform.

[67] Microsoft, "Microsoft azure machine learning," https://docs.microsoft.com/en-us/azure/machine-learning/.