Fast and Efficient Model Serving Using Multi-GPUs with Direct-Host-Access

Jinwoo Jeong
Ajou University
Suwon, Korea

Seungsu Baek
Ajou University
Suwon, Korea

Jeongseob Ahn
Ajou University
Suwon, Korea

Abstract

As deep learning (DL) inference has been widely adopted for building user-facing applications in many domains, it is increasingly important for DL inference servers to achieve high throughput while preserving bounded latency. DL inference requests can be immediately served if the corresponding model is already in the GPU memory. Otherwise, it needs to load the model from host to GPU, adding a significant delay to inference. This paper proposes *DeepPlan* to minimize inference latency while provisioning DL models from host to GPU in server environments. First, we take advantage of the `direct-host-access` facility provided by commodity GPUs, allowing access to particular layers of models in the host memory directly from GPU without loading. Second, we parallelize model transmission across multiple GPUs to reduce the time for loading models from host to GPU. We show that a single inference can achieve a 1.94× speedup compared with the state-of-the-art pipelining approach for `BERT-Base`. When deploying multiple BERT, RoBERTa, and GPT-2 instances on a DL inference serving system, `DeepPlan` shows a significant performance improvement compared to the pipelining technique and stable 99% tail latency.

CCS Concepts: • **Computer systems organization**; • **Software and its engineering → Software system structures**;

Keywords: DNN model serving, Direct-host-access, Parallel-transmission

ACM Reference Format:
Jinwoo Jeong, Seungsu Baek, and Jeongseob Ahn. 2023. Fast and Efficient Model Serving Using Multi-GPUs with Direct-Host-Access. In *Eighteenth European Conference on Computer Systems (EuroSys '23), May 9–12, 2023, Rome, Italy*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3552326.3567508

EuroSys '23, May 9–12, 2023, Rome, Italy

ACM ISBN 978-1-4503-9487-1/23/05…$15.00
https://doi.org/10.1145/3552326.3567508

1 Introduction

Due to the increasing demand to utilize deep neural networks (DNNs) in many user-facing applications, it is becoming increasingly important to provide deep learning (DL) inference with low latency [8, 13, 17, 27]. To serve incoming inference requests within the strict latency constraints (e.g., service level objectives), a straightforward approach is to cache models in the GPU memory, as depicted in Figure 1a. However, the downside of this approach is that inference servers need to be over-provisioned for the peak load, increasing the operation cost of servers. A promising way to reduce the cost of GPU servers is to allow the number of models to extend beyond the GPU memory limit [20], leading to fewer GPU servers. Once GPU memory becomes insufficient to add a new model, we can reclaim the GPU memory space occupied by an inactive model and load the active model. If an inference request arrives at a model not ready in the GPU memory, it starts loading the corresponding model to GPU on-demand [34, 37] (Figure 1b). The remaining challenge is to minimize the (cold-start) time for loading DL models to GPU memory, which significantly delays inference. For instance, loading a `BERT-Base` model takes 40ms if the model is available in host memory, while a single inference on the model cached in the GPU memory is complete within 9.35ms for NVIDIA V100.

A recent inspiring study presented populating model transmission per layer granularity [6], enabling inference to start before the entire model is loaded, as shown in Figure 1c. This approach hides the time for loading layers by overlapping it with the computation. Since DNN models comprise a sequence of layers, we can separate the inference computation layer-by-layer. Once the first layer is loaded, the inference starts immediately. While performing the inference on the first layer, it loads the next layer simultaneously. However, to make such pipelining technique effective, it is required that the computation time must be sufficiently longer than the loading time. Otherwise, the computation cannot proceed until the corresponding layer is completely loaded, called *pipeline stall*. Since recent DNN models such as BERT and GPT have large layers that take a substantial loading time, it is challenging to fully overlap such layer loading time with the computation.

In this study, we explore three techniques to minimize the performance impact of loading models: executing layers

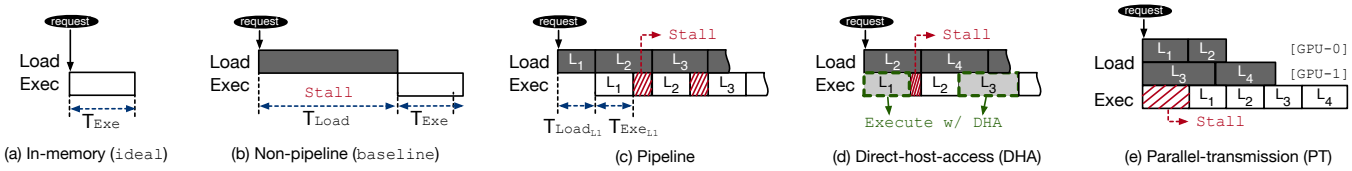

Figure 1. Previous model provisioning approaches (a, b, and c) and our approaches (d and e)

without loading, parallelizing model transmission with multiple GPUs, and automatically combining the two methods. First, we take advantage of the `direct-host-access` facility that allows GPU to perform the computation on layers residing in the host memory without loading layers to the local GPU memory. For example, in NVIDIA GPUs, memory allocated by `cudaHostAlloc` can be directly accessible from GPU through PCIe [33]. This is similar to traditional direct memory access (DMA). The direct-host-access facility has not been widely used because the computation capability can be limited by the narrow PCIe bandwidth. Interestingly, however, we observe that in particular layers, such as embedding layers of NLP and convolutional layers used in vision, direct-host-access shows faster execution time than *load-then-execute*, which loads layers to the GPU and then executes the layer computation.

This paper shows a novel use of direct-host-access to minimize DL inference latency while provisioning DL models from host to GPU. Figure 1d depicts our proposal at a high level. A straightforward approach is to replace the load-then-execute with direct-host-access by layer-by-layer performance comparison. However, such a simple approach does not take into account the pipelining effect, leading to sub-optimal performance. To this end, we introduce an algorithm that adaptively selects the execution method. We prevent direct-host-access to be applied for the layers whose stall time can be hidden by pipelining. This is because direct-host-access cannot be faster than load-then-execution where its load time can be hidden by pipelining.

Second, we parallelize the model loading by leveraging PCIe bandwidth to multiple GPUs. Figure 1e presents how the latency of loading the latter part of a model can be hidden in two GPUs. After partitioning models to the number of GPUs, each partition is transferred simultaneously to a GPU through individual PCIe lanes. We call this `parallel-transmission`. Like the pipeline scheme, we immediately start inferences once the first layer of the first partition is loaded. In this setting, there are two possible approaches for executing inferences. First, we can support inferences for distributed execution across GPUs. Although this approach is simple, it pays the cost of GPU-to-GPU communication while inferencing. Second, we can merge partitions into a single GPU to avoid distributed execution. While executing on the first partition, the remaining partitions are forwarded to the GPU, where the first partition is loaded. As modern multi-GPU servers support the additional high-speed interconnect (e.g., NVLink) across GPUs [23, 24], the transmission from host to GPU and the forwarding step between GPUs can also work in a pipelined manner. Due to this reason, we take the second approach in this study. In addition, we incorporate the direct-host-access facility into the first partition because the parallel transmission cannot reduce the stall time for the first partition.

The last piece of this study is to automate the decision on layer loading and execution. Since the number of layers in recent DNN models is steadily increasing, it poses a challenge for ML practitioners to decide whether to use direct-host-access by traversing all the layers manually. Also, the parallel-transmission scheme requires users to partition models by understanding the underlying hardware facilities such as PCIe and NVLink topology. To tackle this problem, we introduce a tool called *DeepPlan*, which automatically generates an inference execution plan for a given model, minimizing the inference latency to the model that is not cached in the GPU memory. We first conduct a performance profiling step for a given model when executing with the local GPU memory and host memory. Second, we determine the execution method for each layer by comparing the performance difference from direct-host-access with the stall time from the pipeline approach. Third, if we have multiple GPUs, we partition the model evenly to the number of GPUs by understanding the server hardware organization. Then, we override the execution method for all the partitions except for the first. Last, we coordinate to overlap the execution of direct-host-access with the transmission of layers to the GPU memory. Note that this procedure is a one-time process before deploying a model to a new kind of GPU server and is not required while inferencing.

We evaluate our proposed schemes on an AWS `p3.8xlarge` instance. This instance has four NVIDIA V100 GPUs with NVLink. With the guided execution from `DeepPlan`, we can significantly reduce the inference latency when the model is not pre-loaded into the GPU memory. It improves a single inference request for `BERT-Base` by 1.94× compared with the state-of-the-art scheme `PipeSwitch` [6]. The other models, including `ResNet`, `RoBERTa`, and `GPT-2`, also show a speedup of around 1.18~2.21× with `DeepPlan`. In server environments with realistic workloads [30] that run three

hours, DeepPlan reduces 99% tail latency significantly. Consequently, the goodput performance achieves around 99% while PipeSwitch shows 88%.

The rest of the paper is organized as follows. Section 2 presents the background of serving DNN models and the approaches this paper takes. Section 3 introduces the performance characteristics of model execution and model transmission. Section 4 describes the proposed design of DeepPlan, and Section 5 evaluates DeepPlan on a multi-GPU server with various DNN models. Section 6 discusses the prior studies, and Section 7 concludes the paper.

2 Background and Motivation

In this section, we provide background on the out-of-memory problem in provisioning a large number of models on commodity GPUs and how modern DL inference servers deal with the limited GPU memory. Then, we revisit serving DL models with direct host memory access and parallel transmission with multi-GPUs to accelerate model provisioning.

2.1 DL Model Serving

With the wide adoption of deep learning on interactive online applications, DL inference servers play an important role in the quality of user experiences and hardware resource efficiency [8, 10, 13, 14, 29, 32]. As a result, the primary requirement of the inference servers is to satisfy the target latency while maximizing the system throughput. We describe three prior approaches in serving models. First, a naive approach is to keep models in the GPU memory always, as shown in Figure 1a. Once a DL inference request arrives at the system, it can be immediately served with low latency. In this setting, we can judiciously increase the number of models to share a single GPU with spatial sharing. For example, the NVIDIA Triton inference server provides the facility to deploy multiple models on a single GPU to increase the concurrency. At the peak load, such spatial sharing can fully utilize the hardware resource by executing multiple models in parallel. However, this approach results in the underutilization of GPUs on average and low load [13, 29].

Second, to achieve high resource efficiency, we can take the time-sharing technique on GPUs by multiplexing a large number of models as on-demand. Figure 1b presents an inference execution after loading the corresponding model to the GPU memory from host. Such a time-sharing approach can increase the model consolidation ratio by swapping inactive models out to the host memory and swapping the active requested models in the GPU memory. However, such on-demand model provisioning can impose an additional latency to inference due to loading a model to GPU via narrow PCIe bandwidth. This is called cold-start. Such considerable loading time is a major contributor to increasing the tail latency of inference.

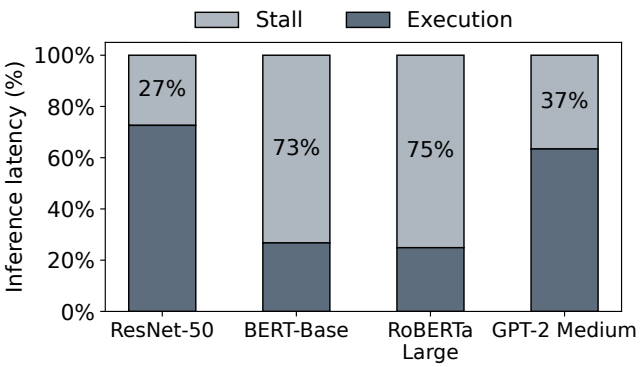

Figure 2. Decomposition of inference latency spent by pipelining [6]

Lastly, to remedy the performance overhead of loading models, Bai et al. introduced a pipelining technique to accelerate inference while provisioning models. Figure 1c shows the pipeline model transmission per layer granularity [6]. The inference computation does not need to wait until the whole set of layers is loaded into the GPU memory. Instead, it can start the computation as early as possible when the first layer becomes ready in the GPU. While performing the computation for the first layer, it loads the second layer simultaneously, hiding the load latency. However, it is required that the computation time is sufficiently longer than the loading time. Otherwise, the computation cannot proceed due to dependency.

To investigate the efficiency of the pipelining approach, we measure the inference execution time while loading the model to GPU. Figure 2 decomposes the inference latency into the GPU execution time and stall time for batch size 1. Even with the pipelining technique, inference performance is limited by the pipeline stalls. The inference execution is frequently stalled across all the models. For BERT and RoBERTa models, the stall time accounts for 73~75% because of large embedding layers. ResNet and GPT show less stall than BERT, but they still occupy around 27~37%.

2.2 Direct Host Memory Access from GPUs

To minimize the effect of the loading time when dealing with an inference request, we take a different approach to directly accessing the layers that resided in host memory without loading to the GPU memory, called direct-host-access. Commodity GPUs provide a facility for directly accessing the host memory (e.g., cudaHostAlloc). It is analogous to direct-memory-access (DMA) between the main memory and peripheral devices of the system.

Figure 3 shows the comparison between the explicit copy with cudaMemcpy and zero-copy with cudaHostAlloc. Although the explicit copy can utilize the abundant local memory bandwidth, the direct-host-access operation opens up

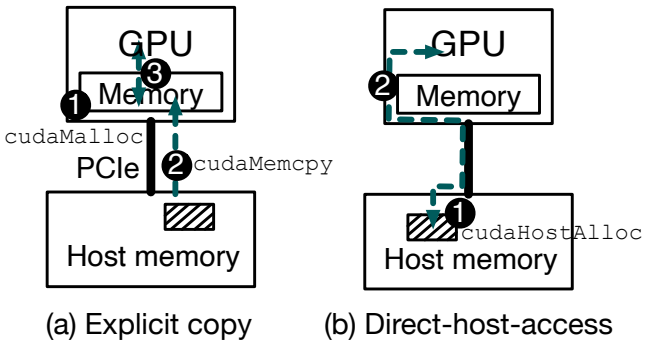

Figure 3. Two different execution types

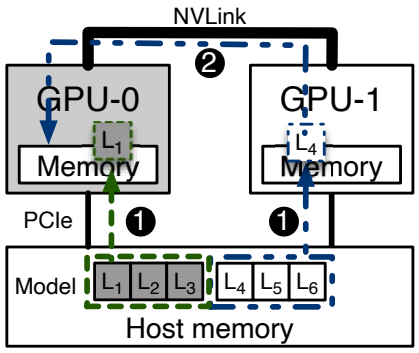

Figure 4. Model provisioning with two GPUs

new opportunities to serve layers (or models) on GPUs. First, it enables GPUs to serve models that reside in the host memory, not the GPU memory. Second, we can load only particular layers that show significant performance benefits on the GPU. Note that existing DL frameworks, such as TensorFlow [5], PyTorch [26], and TVM [7], and prior studies [6, 8, 13] do not leverage the direct-host-access facility.

2.3 Parallel Model Transmission with Multi-GPUs

As a single server supports multiple GPUs, it opens up a new opportunity to parallelize the model transmission across GPUs. Figure 4 presents distributing a partitioned DL model across two GPUs. While loading the first partition of the model to GPU-0, we can transmit the second partition to GPU-1 simultaneously (❶ of Figure 4). In this regard, we can support inferences for distributed execution (i.e., model parallelism) with the cost of GPU-to-GPU communication. However, this can pose additional latency even for in-memory executions and performance interference across GPUs.

Instead, we can merge the divided partitions into the GPU that has the first partition. Since modern GPUs support a high-speed interconnect such as NVLink for accelerating GPU-to-GPU communication [23], we can accelerate transferring the second partition to that GPU (❷ of Figure 4). Although the NVLink facility is widely used for multi-GPU DL training, we can repurpose the existing hardware feature for provisioning DL models. This study focuses on the latter approach merging partitions.

3 Performance Analysis for Model Execution and Provisioning Methods

This section analyzes the performance impacts of the (1) `direct-host-access` facility in executing layers and the (2) `parallel-transmission` scheme in loading models. Our experiments run on an NVIDIA V100 GPU connected through PCIe 3.0. The host CPU is Intel Xeon Gold 6230R. We use PyTorch v1.9 with popular DNN models.

3.1 Load-then-execute vs. Direct-host-access

There is a trade-off between the `load-then-execute` and `direct-host-access` approaches. Although the time for loading layers can account for a significant portion of the total inference time in load-then-execute, GPUs can accelerate the remaining computation with the local memory accesses. By contrast, the direct-host-access facility does not pay the cost of loading layers, but it slows down the inference execution due to host memory accesses through the PCIe interconnect. Figure 5 presents the measured execution time for `embedding`, `convolutional`, and `fully connected` layers by load-then-execute and direct-host-access, respectively. We select the layers from `BERT-Base` [9] and `ResNet-50` [18] models. The input for each layer is identical to batch size 1. For direct-host-access, we modified PyTorch to use `cudaHostAlloc` instead of loading the layers with `cudaMemcpy`.

Embedding layer: We first observe that direct-host-access can be an alternative to load-then-execute in embedding layers. Figure 5a exhibits that direct-host-access shows better performance than load-then-execute in two embedding layers from `BERT-Base`. When the size of embedding layers is pretty large (e.g., 89.4MB out of 417MB in `BERT-Base`), it contributes a significant portion (`hatched`) to the model loading time. Since a single inference request incurs only a small number of sparse memory accesses for the embedding layers, applying the load-then-execute scheme to the large embedding layers is not a cost-effective way. Instead, we advocate leaving such large embedding layers on the host memory. Although it increases the execution time for taking the embeddings, a considerable loading time can be eliminated, leading to the total latency improvement.

Convolutional layer: Figure 5b shows three different sizes of convolutional layers used in `ResNet-50`. For small- and medium-size layers, the performance difference between the two approaches is negligible. In such a case, we propose to use direct-host-access. It allows us to save the loading time and then utilize the time to load subsequent layers of

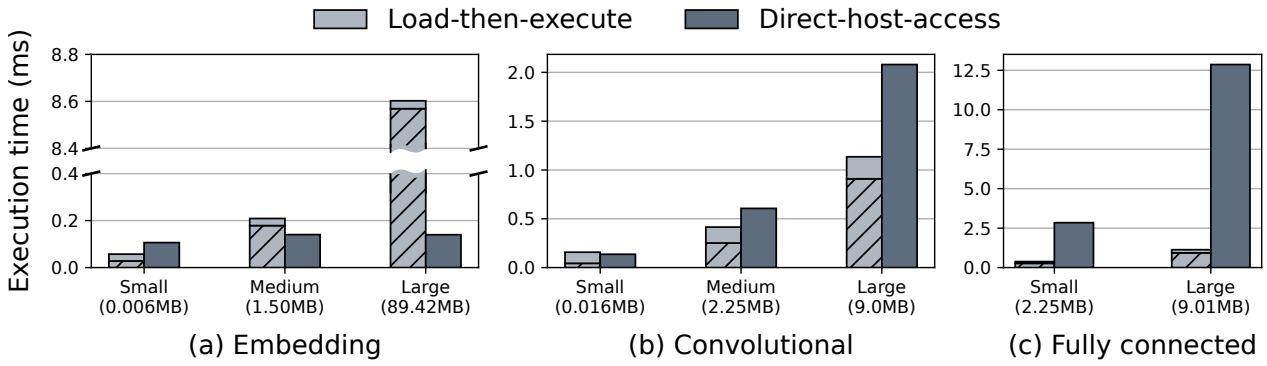

Figure 5. Layer performance comparison (In `load-then-execute`, the lower hatched part indicates the loading time and the upper part presents the execution time.)

models in advance. However, as the size of the convolutional layer increases, the performance gap is widening. In direct-host-access, the memory access through PCIe becomes a performance bottleneck. Thus, we need to decide whether to use either direct-host-access or load-then-execute by understanding the performance benefits. CNN models place the small convolutional layers in the front of models, and the size of convolutional layers is steadily increasing toward the back of models. We can handle the front convolutional layers of the model with direct-host-access while loading large convolutional layers simultaneously. Then, we can hide the time of loading large convolutional layers.

Fully-connected layer: Figure 5c presents the performance of fully connected (FC) layers with load-then-execute and direct-host-access. For both small and large sizes, load-then-execute outperforms direct-host-access. Unlike the convolutional layers, even the small layer exhibits a large amount of memory access. Therefore, direct-host-access for fully connected layers can affect the execution time negatively. Specifically, the self-attention of BERT-Base computes the key, value, and query for each token of an input sequence. As a result, the FC layers are iteratively reused for completing all the sequences. Due to this memory reuse characteristic, load-then-execute can amortize the cost of loading the layers while direct-host-access pays the memory access cost through PCIe every time it accesses. Note that data accessed through direct-host-access is not copied to the GPU memory. We can conclude that load-then-execute is much faster than direct-host-access due to the dense and reuse memory access characteristics.

Other layers: Although Figure 5 does not include the performance results for batch normalization (`BatchNorm`) and layer normalization (`LayerNorm`), these layers are also frequently used in CNN and Transformer models. For `BatchNorm`, direct-host-access shows better performance than load-then-execute. However, for `LayerNorm`, the opposite is shown.

Figure 5		Load	Direct-host-access
(a) Embedding	Medium (1.50MB)	24,580	18,267
	Large (89.42MB)	1,465,112	18,459
(b) Convolutional	Medium (2.25MB)	36,869	65,891
	Large (9.0MB)	147,465	273,487
(c) Fully connected	Small (2.25MB)	36,920	446,276
	Large (9.01MB)	147,660	1,765,787

Table 1. Comparison for the number of PCIe events: `load` vs. `direct-host-access`

Changes in the Number of PCIe Accesses: To further understand the performance difference between load-then-execute and direct-host-access, we measure the number of PCIe accesses between the two execution methods. We utilize the hardware performance counters (`PCIeRdCur`) for profiling layers used in Figure 5. To accurately profile the events, we insert the measurement code in libTorch with the PCM library[1]. Table 1 compares the number of PCIe accesses when loading the layer and using direct-host-access.

When loading a layer, the number of PCIe accesses is proportional to the size of the layer. Since the payload size in transferring through PCIe is 64B (cache-line size), the number of PCIe accesses is that the layer size is divided by 64B. For embedding layers, the direct-host-access scheme can significantly reduce the number of PCIe accesses compared with loading the entire layer because all the embeddings are not used in the inference phase. On the other hand, in the convolutional and fully-connected layers, we observe a different behavior, as depicted in Figure 5. The direct-host-access scheme shows more PCIe accesses than loading the layer. This is expected because the convolutional and linear layers reuse the tensor in the host memory to compute the output. In such layers, the load-then-execute scheme can amortize the cost of loading the entire parameters of layers.

[1]https://github.com/opcm/pcm

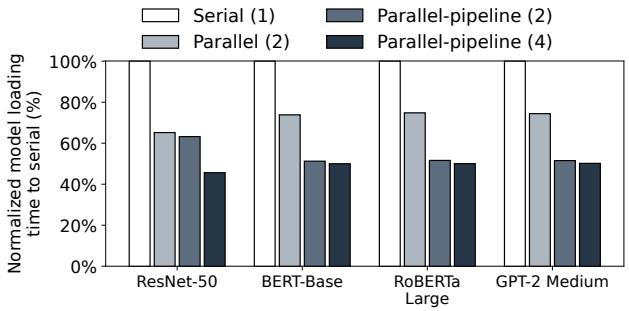

Figure 6. Model loading time: `serial` vs. `parallel` (The numbers in parentheses indicate the number of used GPUs.)

Figure 6	Average PCIe bandwidth (GB/s)		
	Serial (1)	Parallel-pipeline (2)	Parallel-pipeline (4)
ResNet-50	9.10	9.13	7.01
BERT-Base	10.87	10.67	5.89
RoBERTa-Large	10.94	10.75	6.01
GPT-2 Medium	11.52	11.32	5.96

Table 2. Average PCIe bandwidth when using serial and parallel transmissions

3.2 Model Transmission: Serial vs. Parallel

Beyond single GPUs, we evaluate the effectiveness of parallel model transmission through the individual PCIe lanes attached to each GPU. Figure 6 presents the completion time loading each model from host memory to a target GPU. We divide models into two partitions evenly in terms of size. First, the `serial` approach transfers the models *directly* from the host to one GPU in the server. Second, in the case of `parallel`, we transfer two partitions of a model to each GPU in parallel. Then, we forward the second partition to the target GPU through NVLink *indirectly*. This `parallel` scheme reduces the transfer time by around 30~45% compared to `serial`. Third, we add the pipeline feature to the `parallel` mode, called `parallel-pipeline`. Once the first layer of the second partition is loaded, it is immediately transferred to the target GPU. This further reduces the time for model transmission by almost half in transformer models. For `ResNet`, it reduces the loading time by about 40%.

When using four GPUs, the parallel approach shows a small performance benefit for the transformer models. Table 2 shows the average PCIe bandwidth across GPUs. Our machine is with PCIe 3.0, providing up to 15.75GB/s theoretically. With four GPUs, the bandwidth reduces almost by half. This degradation comes from the PCIe contention by GPUs attached to the same PCIe switch [36]. In modern multi-GPU servers, there are eight GPUs, and every two GPUs share the same PCIe switch [16]. In other words, even with two GPUs, the performance improvement depends on which two GPUs are used for parallelization. When parallelizing the model transmission, we need to understand such PCIe topology.

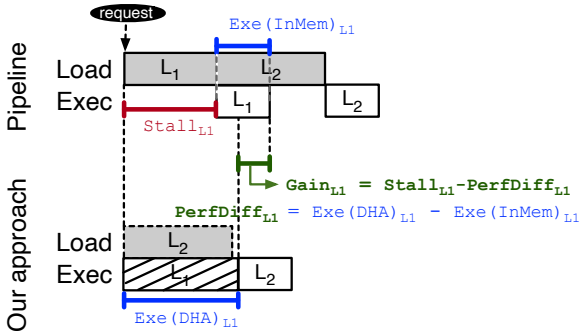

Figure 7. Acceleration of L_1 execution with `direct-host-access`

In `ResNet-50`, we observe that the PCIe is not effectively utilized due to the transmission of a large number of small layers. The PCIe contention with four GPUs is less severe than other transformer models.

4 DeepPlan

Based on the performance characterization in the previous section, we introduce our two designs to reduce the pipeline stall with direct-host-access on particular layers of models and parallel model transmission with multiple GPUs. These two approaches accelerate inference while provisioning models from host to GPU memory. Last, we propose a tool, called *DeepPlan*, which automatically generates an inference execution plan for server environments by incorporating the two techniques.

4.1 Leveraging Direct-Host-Access

Our approach is to take advantage of the direct-host-access (DHA) facility to replace the load-then-execute part of the pipeline scheme. We can apply direct-host-access to the layers that exhibit better performance than load-then-execute through the layer-by-layer performance comparison. The direct-host-access facility brings two advantages. Figure 7 and 8 show the two cases of how direct-host-access can effectively reduce the stall time. First, as depicted in Figure 7, the pipeline execution for L_1 is stalled due to the dependency. On the other hand, we can avoid the stall time by changing the execution of L_1 with direct-host-access (`hatched box`). In addition, we can start loading the following layer L_2 (`dotted box`). We define the performance difference between direct-host-access and in-memory executions as $PerfDiff_{L_n} = Exe(DHA)_{L_n} - Exe(InMem)_{L_n}$. If $PerfDiff_{L_n} > 0$, we represent the performance gain as $Gain_{L_n} = Stall_{L_n} - PerfDiff_{L_n}$. In this example, direct-host-access for L_1 is shorter than that of load-then-execute. It means there is a performance gain ($Gain_{L_1} > 0$). Such a performance characteristic can be observed in embedding layers of transformer models and convolutional layers of vision models (see Section 3.1).

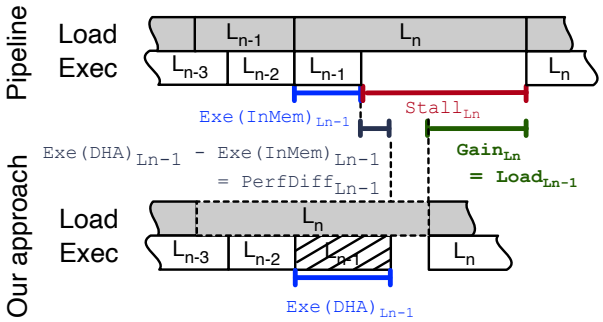

Figure 8. Stall reduction of loading L_n execution with `direct-host-access`

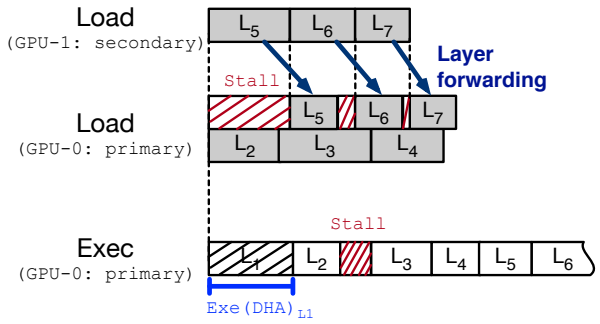

Figure 9. Cooperative `parallel-transmission` with `direct-host-access` to accelerate model provisioning

Second, we apply direct-host-access to the layers that can reduce the pipeline stalls of the following layers. Figure 8 illustrates that the pipeline execution does not entirely hide the loading time of L_n. However, we have the opportunity to reduce the stall time of L_n by changing the execution method for previous layers (L_{n-1}, L_{n-2}, ...). Once we execute L_{n-1} with direct-host-access (`hatched box`), the loading time of L_{n-1} disappears. Even though direct-host-access can increase the execution time of a given layer, it can advance the loading of the following layers early by eliminating the loading time itself. Then, we can use the saved time to start loading L_n (`dotted box`) early, leading to the stall time reduction. The amount of performance gain ($Gain_{L_n}$) is the eliminated loading time of L_{n-1}.

However, since changing the execution of L_{n-1} to direct-host-access cannot eliminate the stall time of L_n, we can additionally utilize L_{n-2} to reduce the remaining stall time further. Once a previous layer cannot eliminate the stall time of a given layer, we attempt to leverage another previous layer until the stall time can be eliminated or there are no previous layers. When selecting a previous layer, we choose the layer that shows the smallest $PerfDiff$ first among the earlier layers. For example, instead of L_{n-1}, we can utilize the L_{n-2} layer if $PerfDiff_{L_{n-2}} < PerfDiff_{L_{n-1}}$. The rationale behind is that the smaller the performance difference between the two execution methods, the more the stall time of subsequent layers is reduced while minimizing the negative performance impact on the target previous layer.

4.2 Leveraging Parallel Model Transmission

To further reduce the stall time of loading models, we parallelize the model transmission by using multiple GPUs in the same server. Figure 9 depicts the advantage of our `parallel-transmission` scheme with two GPUs. Note that our approach is not limited to two GPUs. For the parallelization, we take advantage of the `map-reduce` concept. In the `map` phase, we divide a given model into the number of GPUs participating in the parallel transmission. Each partition is copied to the corresponding GPU through individual PCIe lanes in parallel. While transferring the first partition from L_2 to L_4 to GPU-0, the second partition from L_5 to L_7 is loaded to GPU-1 simultaneously. This example shows that L_1 is chosen for direct-host-access.

Suppose the GPU that holds the first partition is the primary GPU, and the others are secondary GPUs. Then, in the `reduce` phase, each secondary GPU forwards its own partition to the primary GPU through NVLink. In that example, the layers (L_5 to L_7) on GPU-1 are migrated to GPU-0. As NVLink provides an additional path and high-speed interconnects to PCIe, we can overlap the `reduce` phase with the transmission for the first partition. While copying L_3 from host to GPU-0 through PCIe, it transfers L_5 from GPU-1 to GPU-0 through NVLink. It can effectively eliminate the stall time, as shown in Figure 9. To support the parallel-transmission scheme, we reserve a small amount of memory for storing layers temporarily on each GPU.

In the case of the first partition, the parallel-transmission does not reduce the stall time at all. Instead, we still have an opportunity to utilize our direct-host-access approach to the first partition to reduce the pipeline stalls. These two techniques can complement each other.

Meanwhile, as discussed in Section 3.2, GPUs attached to the same PCIe switch do not show the performance benefit for parallel-transmission due to the PCIe bandwidth contention from the host. Thus, it is required to utilize GPUs attached to the different PCIe switches. Also, the parallel-transmission scheme can incur performance interference to secondary GPUs serving other models. Fortunately, however, we do not see that the interference becomes severe (Section 5).

4.3 Generating Execution Plans Automatically

To apply direct-host-access to layers of models, it is required for ML practitioners to profile the performance of models per layer granularity and then extract the inference execution plan by understanding the performance benefit of direct-host-access for each layer. Also, the number of GPUs participating in the parallel-transmission depends on the underlying hardware where the models are served. As modern DNN models and GPU servers are becoming diverse and

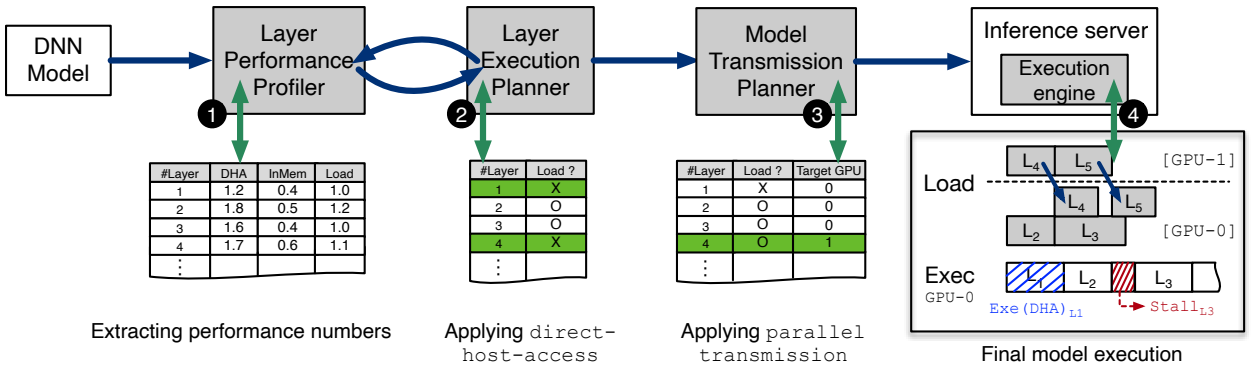

Figure 10. DeepPlan: system support for generating model execution plans

complex, we introduce a tool that automatically generates the inference execution plan for a given server environment.

This section presents the design of our proposed tool, called DeepPlan. Figure 10 shows the overall process of how it works. DeepPlan takes a pre-trained DL model as an input. Note that it does not require any intervention from ML practitioners. ❶ For a given model, it profiles the runtime performance of individual layers. From a pre-run, we can measure the execution time for two methods, DHA and in-memory, and the loading time for each layer. ❷ The profiled information will be given to our layer execution planner as inputs, and it identifies which layer has the pipeline stall and attempts to reduce the stall with direct-host-access. This is an iterative process until all the layers are examined. In this example, layer 1 (L_1) and 4 (L_4) are selected for direct-host-access. ❸ If the server has multiple GPUs, it applies the parallel-transmission scheme across the GPUs by understanding the PCIe interconnect. In this example, it overrides the execution method for L_4 to be loaded to the other GPU. Note that we do not statically assign the GPU. This is an example to help readers understand. ❹ Once DeepPlan generates the inference execution plan, it is ready to be deployed into the serving systems. Since L_1 is executed with direct-host-access, we start loading L_2 on GPU-0 to reduce the stall time. At the same time, L_4 is loaded to GPU-1 and then forwarded to GPU-0 in the pipelined manner. Consequently, our proposed techniques can accelerate the inference execution upon a cold-start.

4.3.1 Performance Profiling of Individual Layers.

For a given model, the first step is to collect the performance statistics from load-then-execute and direct-host-access by performing inferences on an actual system where the model will be deployed. Although we can have an analytic model to avoid the profiling step for each model, such a pre-run is practical and provides a robust decision for diverse server environments. Note that this procedure is a *one-time* process before deploying a model. From this pre-run, we construct the performance table, including the pipeline stall for each layer, depicted in Figure 10. These are fed into our plan generator. We discuss the performance overhead of profiling in Section 5.2.

Algorithm 1 Generating a layer execution plan

```
1:  for each layer i in 1, ..., n do
2:      if Stall_{L_i} > 0 then
3:          # Step 1: Make a list of candidate layers by sorting from
4:          #          PerfDiff in ascending order
5:          sortedLayers ← SortByPerfDiff(L_1 to L_i)
6:          for each sorted layer j in 1, ..., i do
7:              # Step 2: Check whether L_j can contribute to reducing
8:              #          stall of L_i
9:              if Stall_{L_i} < PerfDiff_{L_j} then break
10:             # Step 3: Change for L_j to use direct-host-access and
11:             #          update the stall time of L_i
12:             ChangeToDirectHostAccess(L_j)
13:             Stall_{L_i} ← (Stall_{L_i} − Load_{L_j} − PerfDiff_{L_j})
14:             # Step 4: Update pipeline execution when the stall time
15:             #          is eliminated
16:             if Stall_{L_i} ≤ 0 then
17:                 UpdatePipelineExecutionFrom(L_j)
18:                 break
```

4.3.2 Layer Execution Planning.

Given a profiled layer-wise performance information, Algorithm 1 describes how our planner decides whether individual layers need to be loaded to the GPU memory or can be kept in the host memory for direct-host-access. The goal is to reduce the stall time for each layer. In Step 1, we find candidate layers that are placed before the given layer L_i, and direct-host-access is not applied yet. To visit the most effective layer to reduce the stall, the selected layers from L_1 to L_i are sorted by the performance difference between the two execution methods in ascending order. Again, the smaller the difference, the more the stall time can be reduced. While traversing the sorted layers (`line 6`), we attempt to minimize the stall time of L_i. In Step 2, we determine whether the previous

layer L_j can contribute to reducing the stall time. If $Stall_{L_i}$ is larger than the performance difference $Exe(DHA)_{L_j} - Exe(InMem)_{L_j}$, it indicates that we can reduce the stall by changing the execution type for L_j to direct-host-access. It allows us to start loading layer L_i in advance. Otherwise, we cannot reduce the stall of L_i so that we move on to the next layer L_{i+1} (*break* at `line 9`). Since the previous layers are sorted by $PerfDiff$, we do not need to check the next sorted layer L_{j+1}. In Step 3, we record the changed decision for L_j and update the stall time to reflect the performance gain by omitting the load time $Load_{L_j}$ and by the changed execution time with direct-host-access $PerfDiff_{L_j}$. In Step 4, we check whether the stall time is eliminated. If so, we profile the performance of the new execution plan to get the correct stall time for the subsequent layers and move on to the next layer L_{i+1}. Otherwise, we go to Step 2 to reduce the remaining stall of L_i by the next sorted layer L_{j+1}.

4.3.3 Model Transmission Planning. In this stage, we equally partition a given model into the number of GPUs participating in the parallel-transmission. To select the GPUs, we need to understand how GPUs are laid out in the PCIe switches of the server to avoid bandwidth contention while loading in parallel. Next, we check whether the selected GPUs are connected through NVLink. If not, we do not enable the parallel-transmission. Although we could support the inference execution across GPUs, this option has not been considered due to the performance interference to other inference requests.

Once we decide to apply this parallel-transmission, we change the execution method for layers belonging to the second and later partitions to be loaded. While planning, we do not designate the GPUs for parallel-transmission. The number of secondary GPUs depends on the number of PCIe switches. In a `p3.8xlarge` instance, there are two PCIe switches, and each of them serves two GPUs. `DeepPlan` guides us to use up to two GPUs out of four for the parallel-transmission at once.

4.3.4 Model Execution Coordination. Once the inference execution plan is made, the last step is coordinating the layer load and execution timing. This is a relatively simple task. Our execution engine is an extension of PyTorch v1.9. We use two separate GPU streams to load layers while executing layers with direct-host-access simultaneously. The *load stream* copies selected layers sequentially according to the plan. If the execution plan is enabled with parallel-transmission, we create an additional *migration stream* on each secondary GPU for migrating the partitions to the primary GPU. To optimize data transfers from the host to GPU, we allocate the pinned memory for the layers. For layers executed by direct-host-access, we add a new memory allocator in PyTorch that allocates memory through `cudaHostAlloc`.

Due to the dependency between the load and computation for layers, the *execution stream* needs to check whether the corresponding layer is completely loaded. To simplify the synchronization, the *load stream* and *migration stream* insert a CUDA event through `cudaEventRecord()` after transmitting each layer. Then, the *execution stream* simply identifies the event status through `cudaStreamWaitEvent()`. Note that if the layer is executed by direct-host-access, we can skip the dependency check process.

5 Evaluation

5.1 Experimental Setup

We implement the layer profiler and execution planner as a standalone Python tool. The execution engine provides five execution options: Baseline (non-pipeline), PipeSwitch [6], and our three designs of `DeepPlan`, direct-host-access (`DHA`) parallel transmission (`PT`), and integration of the two (`PT+DHA`). To evaluate DL inference performance, we use a `p3.8xlarge` instance in AWS which has four NVIDIA V100 (16GB) GPUs with NVSwitch. The server instance has a Xeon E5-2686 v4 CPU with 32 virtual CPUs and 244GB of memory. We use PyTorch v1.9, CUDA 11.3, and a set of representative pre-trained DNN models: `ResNet-50` and `ResNet-101` from TorchVision [1] and `BERT-Base`, `BERT-Large`, `RoBERTa-Base`, `RoBERTa-Large`, `GPT-2`, and `GPT-2 Medium` from Transformers [35]. We use a synthetic dataset for all the benchmark inputs. `ResNet` uses 224×224 RGB images. The sequence length for `BERT` and `RoBERTa` models is 384 while `GPT-2` is 1,204. Our artifact is available at https://github.com/csl-ajou/DeepPlan.

5.2 Inference Performance

In this section, we evaluate the performance of a single inference request when models are not loaded in GPU memory. Then, we assess the performance impact of DeepPlan in serving scenarios where GPU memory is not sufficient to serve a number of models.

Single inference with batch size 1. The inference requests upon the model not resided in the GPU memory are significantly delayed due to the time spent on loading layers of models from the host to the GPU memory. We first evaluate the inference latency for a single batch. Figure 11 exhibits the relative speedup of our three designs of `DeepPlan` and the state-of-the-art pipeline execution called `PipeSwitch` [6] normalized to `Baseline`, which loads the model from the host memory to the GPU memory and then executes the inference when handling a single inference request. We report the inference latency averaged on 100 runs with error bars.

① **Single GPU:** `DeepPlan` (`DHA`) outperforms `PipeSwitch` across all the models we evaluate. For `ResNet-50` and `101`, the speedup improvement is 1.01~1.03× over `PipeSwitch`. Since the pipelining approach reduces the stall time effectively for the image classification models, the performance improvement is not significant. Table 3a shows part of the

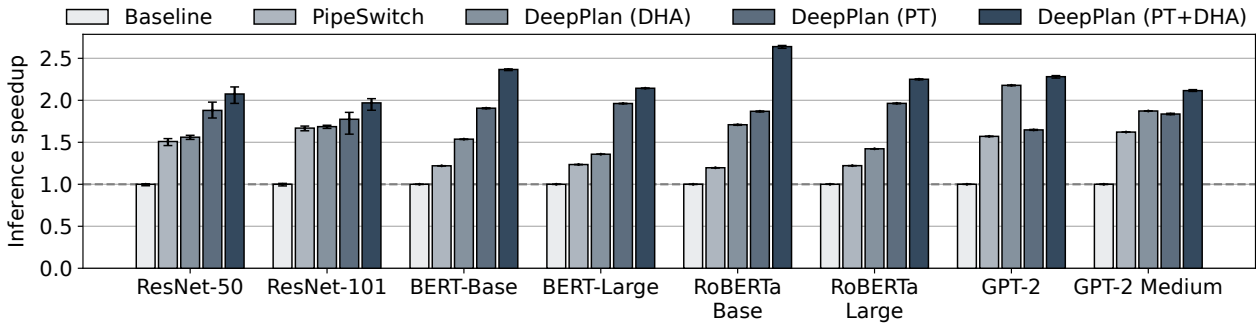

Figure 11. Performance comparison of DeepPlan and previous studies (batch size: 1)

	Layer #: Name						
	63: BN	64: ReLU	65: Conv	66: BN	67: ReLU	68: Conv	69: BN
Initial approach	X	X	X	X	X	O	X
DeepPlan (DHA)	X	X	O	X	X	O	X

(a) ResNet-101: layers of a middle part

	Layer #: Name				
	0: Emb	1: Emb	2: LN	3: FC	4: FC
Initial approach	X	X	O	O	O
DeepPlan (DHA)	X	O	O	O	O

(b) GPT-2: front first 5 layers

Table 3. Part of generated execution plans (O: load, X: direct-host-access) [Conv: Convolutional, BN: BatchNorm, Emb: Embedding, LN: LayerNorm, FC: Full connected]

execution plans for DeepPlan (DHA). According to the layer-by-layer performance comparison (the Initial-approach row), direct-host-access for the 65th convolutional layer shows better performance than load-then-execute so that DeepPlan initially decides not to load the layer on GPU. However, by considering the pipeline effect, it makes a different decision. This is because the loading time of the convolutional layer can be hidden by the computation of precedent layers.

For transformer models, the speedup is around 1.10~1.43× compared to PipeSwitch. DeepPlan (DHA) accelerates the execution for embedding layers with direct-host-access as shown in Table 3b. While executing the embedding layers without loading, it can simultaneously load fully connected layers behind the embeddings. Therefore, it can reduce the stall time of loading fully connected layers. In GPT-2 with a larger sequence length, the pipelining technique can effectively reduce the stalls because the computation time is relatively longer than BERT and RoBERTa models.

② **Dual GPUs:** To evaluate the parallel-transmission scheme, DeepPlan (PT), we use two GPUs in the server. First, our PT shows better performance than PipeSwitch for all the models. For ResNet-50, BERT, and RoBERTa, PT presents a more reduced execution time than DHA. It improves the inference latency by 1.09~1.44× compared to DHA. With PT, we observe that the stalls incurred in the latter part of the models are eliminated. On the other hand, both parallel-transmission (PT) and direct-host-access (DHA) show a similar improvement for ResNet-101. In GPT-2 models, the performance improvement is not shown. PT loads all the layers rather than leaving certain layers in the host. Thus, it cannot reduce the stall time for embedding, convolutional, and layer normalization layers in the first partition.

	PipeSwitch (1)	PT+DHA (1)	PT+DHA (2)
ResNet-50	12.03	8.93	11.97
ResNet-101	19.85	17.71	21.19
BERT-Base	40.51	20.88	30.45
BERT-Large	122.37	70.56	108.16
RoBERTa-Base	45.86	20.83	34.48
RoBERTa-Large	129.58	70.26	107.87
GPT-2	48.41	33.38	35.98
GPT-2 Medium	134.10	101.83	112.71

Table 4. Increased inference execution time (milliseconds) from parallel-transmission

Second, we can further decrease the inference execution time across all the models by integrating direct-host-access on top of parallel transmission (PT+DHA). The individual techniques complement each other to reduce the stall time. While direct-host-access reduces the stall time for the first partition, parallel-transmission hides the loading time of the remaining partitions. For RoBERTa-Base, it improves the inference latency by 2.21× compared to PipeSwitch. Also, BERT-Base and BERT-Large models present a 1.94× and 1.74× speedup, respectively.

Interference from parallel-transmission. If two GPUs perform the parallel-transmission (PT) simultaneously, they can interfere with each other. Table 4 presents the performance interference effects on the two GPUs. The numbers

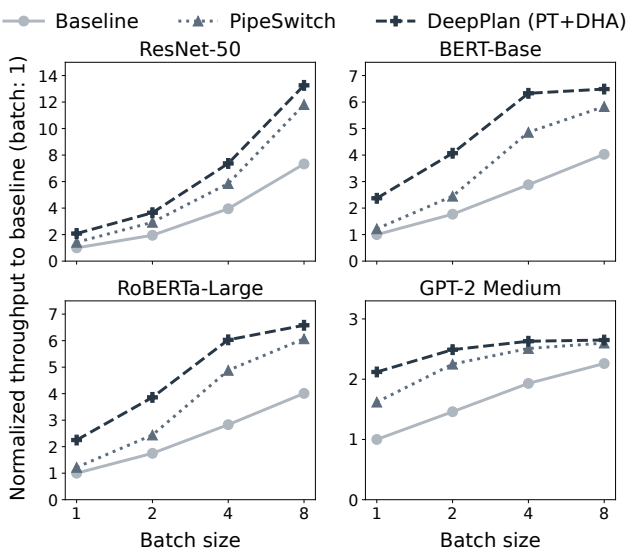

Figure 12. Throughput improvement with batching 1 to 8

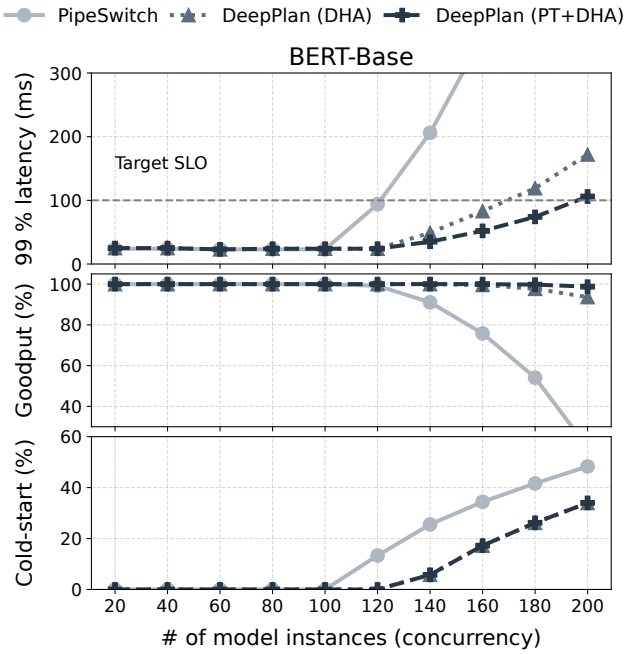

Figure 13. 99% latency (top), goodput rate (middle), and cold-start rate (bottom) while increasing the number of instances beyond GPU memory limit

in parentheses indicate the number of DL instances dealing with the cold-starts. `PT+DHA(1)` shows the performance when there is no interference whereas `PT+DHA(2)` is configured for each GPU to run the inference with DHA at the same time. For `PT+DHA(2)`, we average the execution time from the two GPUs. Each GPU runs the same model depicted in the first column. Although the performance of `PT+DHA` is affected when the two GPUs handle the cold-starts simultaneously, it is still faster than `PipeSwitch`.

Batching inference. We perform a performance sensitivity study by varying the batch size from 1 to 8 to see the extensibility of our DeepPlan. Figure 12 presents the throughput improvement by batching and compares performance with `Baseline` and `PipeSwitch`. We normalize the throughput to `Baseline` with batch size 1. For all the models, `DeepPlan (PT+DHA)` still achieves the best throughput. In `ResNet-50`, our `PT+DHA` shows 1.12~1.26× throughput improvement over `PipeSwitch` for all batch sizes. For `BERT-Base`, `RoBERTa-Large`, and `GPT-2 Medium`, as the batch size increases, the throughput differences between `DeepPlan (PT+DHA)` and `PipeSwitch` become narrow. This is because batching increases the computation time that can have more opportunity to overlap the pipeline stalls than non batching. Note that batching is not recommended for the most latency-sensitive workloads while provisioning models due to the cold-start latency violating the strict target latency (SLO) [34, 37].

	Profiling time			
	DHA	In-memory	Layer load	Total
ResNet-50	2.28s	0.44s	1.20s	3.92s
BERT-Base	7.99s	0.41s	4.00s	12.40s
RoBERTa-Large	63.61s	0.95s	11.31s	75.87s
GPT-2 Medium	28.1s	1.69s	11.02s	40.81s

Table 5. Time (seconds) spent in profiling models with 10 iterations

Profiling cost. The profiling phase is required for `DeepPlan` to determine the layer execution method. It profiles the time for executing layers with direct-host-access and in-memory settings and also for loading layers from host to GPU. To attain stable results in profiling, we measure the time with several iterations. We empirically set the number of iterations. Table 5 presents the time spent in profiling the models with 10 iterations. The last column (`Total`) is the sum of the three costs. The profiling cost depends on the size of the models and the execution time of the models. Note that this is only required once before deploying models.

5.3 Performance of Serving Models with DeepPlan

In this section, we evaluate tail latency and goodput for both synthetic and real-world workloads. Goodput is the number of requests satisfying the target SLO. For this evaluation, we use all four GPUs in the server. As Clockwork [13], our execution engine is designed for each GPU to run one inference request at a time. `PipeSwitch` deals with the cold-starts for each GPU individually. For the parallel-transmission, DeepPlan guides us to use up to two GPUs per model in the given server. Each of the two GPUs can run the model transmission simultaneously, as explained in the previous section, even though it can cause PCIe contention.

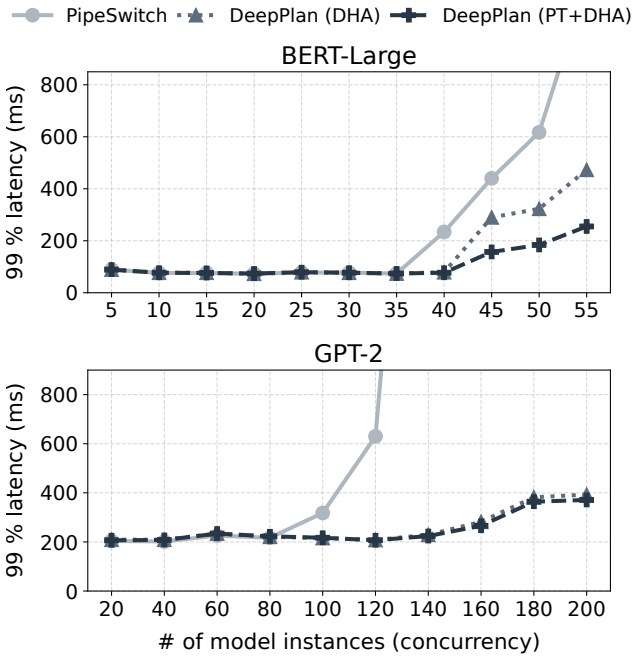

Figure 14. 99% latency for BERT-Large and GPT-2 (The requests per second are set to 30 and 90, respectively)

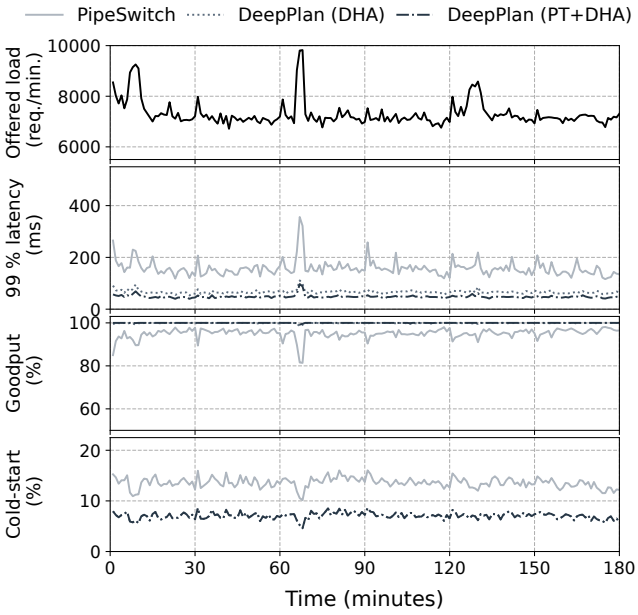

Figure 15. Performance of real-world trace (3 hours)

5.3.1 Synthetic workloads. Figure 13 exhibits 99% latency, goodput, and cold-start of BERT-Base while increasing the number of instances concurrently running on the GPUs. Each instance mimics a model corresponding to a different user or service. After warming up the instances presented on the x-axis, we measure performance for 1,000 requests. For generating a realistic request arrival pattern, we use Poisson distributions [13, 14, 27], which are widely used to evaluate interactive web services. We maintain 100 requests per second, which are randomly distributed across all the instances. For example, at concurrency 100 (x-axis), each instance serves approximately 1 request per second. We increase the number of models (concurrency) by 20.

At concurrency 120, PipeSwitch starts to increase 99% latency significantly. On the other hand, our DeepPlan (DHA) exhibits stable 99% latency until concurrency 160. In the case of DeepPlan (PT+DHA), we can serve up to 180 instances while satisfying the target SLO (100ms). As a result, PT+DHA can achieve stable goodput performance by concurrency 180. At concurrency 180, our PT+DHA improves goodput by 1.84× compared to PipeSwitch. When having a relatively tight target SLO such as 50ms, at concurrency 120, PipeSwitch starts violating the SLO, increasing the 99% latency to around 94ms. On the other hand, DeepPlan (PT+DHA) shows that it can handle requests within 35ms even at concurrency 140.

DeepPlan can reduce the required GPU memory space for models by placing selected layers such as embedding on the host memory. PipeSwitch can keep up to 100 instances across the four GPUs while DeepPlan serves 24 instances more. As shown in the bottom of Figure 13, our approach can delay occurring the cold-start. While serving more than 100 (124 in DeepPlan) instances, the number of cold-start is increasing. To evict an instance due to the lack of GPU memory, we select the least recently used instance.

Figure 14 presents 99% latency for BERT-Large and GPT-2. We use 30 and 90 requests per second for BERT-Large and GPT-2, respectively because they spend more execution time and cold-start latency than BERT-Base in inferencing a request. Our DeepPlan significantly improves the tail latency over PipeSwitch. In GPT-2, however, the latency gap between DHA and PT+DHA is not noticeable. This is because PT+DHA has a narrow lead over DHA for a single batch as shown in Figure 11. We also evaluated the other models and observed a similar improvement trend over PipeSwitch. Due to the limited space, we omit their results in this paper.

5.3.2 Real-world workloads. We evaluate our DeepPlan by replaying a real-world workload trace of Microsoft Azure Functions (MAF) [30]. Like the previous study [13], we deal with an Azure function invocation as an DL inference request. By scaling down the trace for our evaluation environment, we extract unique function IDs from the trace and map each function to a DL model. It represents a diverse range of workloads such as heavy sustained requests, fluctuations in request rates, and spikes in requests. We deploy three types of DL models, including BERT-Base, RoBERTa-Base, and GPT-2, on the server. The number of instances follows about a 4:4:1 ratio. By replaying 3 hours of the MAF trace in real-time, we measure the performance of 99% latency, goodput, and cold-start.

Figure 15 exhibits performance of PipeSwitch and two designs of DeepPlan for the real-world trace. We present the offered load to across models as time goes by at the

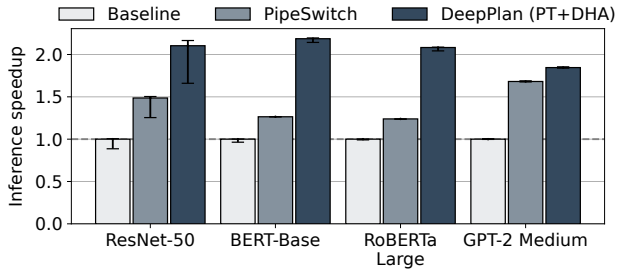

Figure 16. Inference speedup on a different system equipped with two NVIDIA RTX A5000 GPUs with PCIe 4.0

top of the figure. To stress the inference server, we set the requests per second to 150. Our DHA and PT+DHA show improved 99% latency and stable goodput results compared to PipeSwitch. In many cases, DeepPlan handles inferences under 100ms while PipeSwitch takes more than 150ms. The goodput result is based on when configuring the target SLO to 100ms. The two designs of DeepPlan achieve 98~99% goodput while PipeSwitch is around 81~98%. With PipeSwitch, we need to have more GPUs or less load (e.g., requests per second) to satisfy the target latency. Meanwhile, we observe a few latency spikes in the 9th and 67th minutes, even with DeepPlan. Note that such a phenomenon does not persist.

5.4 Inference Performance with PCIe 4.0

In addition to the inference performance shown in Figure 11, we include an additional evaluation to see the reproducibility for generating the execution plan by DeepPlan on a different system. According to GPU computational capabilities, the execution plan can be changed between load-then-execute and direct-host-access. Since the server has two NVIDIA RTX A5000 GPUs with NVLink, we can utilize the parallel-transmission feature. Note that the GPUs are attached to the system through PCIe 4.0.

Figure 16 presents the relative speedup for batch size 1 and shows the similar improvement trend observed in Section 5.2. Although the newer generation of PCIe can reduce the stall time for transferring models from host to a GPU, our two designs of DeepPlan still show improved performance.

6 Related Work

There have been significant efforts throughout architecture, system, and ML community to improve the performance of latency from hardware to software optimization.

DL serving systems. The primary design goal of building DL serving systems in practice is to meet a strict latency requirement (e.g., SLO) and then maximize the system throughput under the given SLO budget. Prior studies mainly focused on improving resource utilization by sharing GPUs timely and spatially with sophisticated scheduling, placing, and coordinating inference queries while not violating the target SLOs [8, 10, 14, 29, 32]. However, such previous work did not consider the case where DL models need to be loaded in the GPU memory while serving inferences. Also, most of the open-source serving systems, such as TF Serving [25], TorchServe [3], and Triton [2], delegate control of loading and unloading models on GPUs to developers.

On the other hand, we can find a different approach, allowing the number of DL models beyond the GPU memory limit. However, a new challenge is to address the *cold-start* problem [34, 37] when loading models from host to GPU. When building cloud-based applications such as AWS Lambda, we need to minimize the performance impact of the cold-start latency, affecting the quality of user experiences. If you leverage the cloud DNN serving systems, Amazon SageMaker or Google AI Platform, it is known that the inference latency suffers from the cold-start latency. Recently, Bai et al. introduced pipelining model transmission to hide the latency of loading models from the host to the GPU memory [6]. Compared with that, DeepPlan takes a different approach, accelerating the model serving with the direct-host-access facility [4]. To the best of our knowledge, there are no prior studies leveraging the zero-copy approach for inference. Instead of loading layers, we choose the best placement strategy for each layer and load selected layers by understanding the performance critically in given models.

Lightweight inference. Pruning and quantization have been widely explored to make the computation lightweight and reduce the size of models [15]. TVM optimizes given DL models to a specific hardware by fusing operators, making models lightweight [7]. By leveraging TVM, Clockwork implements a runtime system serving DL models [13]. Also, the hardware community has introduced specialized DL accelerators such as Brainwave [12] and TPU [20, 21], which are tailored for particular layer operations such as convolutional or fully connected layers with different floating-point formats [11, 19]. By sacrificing the inference accuracy, Zhang et al. introduced a model switching scheme from complex to lightweight models at high load [38]. However, such solutions are orthogonal to the space DeepPlan explored.

New hardware features. Min et al. [22] leveraged the direct-host-access scheme manually for training large graph networks, while DeepPlan automatically generates efficient execution plans for DL inferences. DeepPlan considers the performance benefits of the pipeline approach and opportunistically applies the direct-host-access scheme to reduce the pipeline stall. In addition, this study shows that direct-host-access can have a synergy effect with the parallel transmission for accelerating model provisioning. As the NVLink facility is introduced for accelerating GPU-to-GPU communication in multi-GPU workloads, modern GPU servers have supported the hardware feature by default [23, 24]. Even desktop GPUs such as NVIDIA RTX 3090 can exploit the high-speed GPU interconnect with NV Bridge. In this study, we leveraged NVLink to make the model provisioning fast.

7 Conclusions and Future Work

This paper introduced DeepPlan, an inference execution planner that minimizes the performance penalty when loading models from host to GPU. Our main technical contribution is to exploit the performance benefits of the *direct-host-access* and *parallel-transmission* schemes for accelerating inference performance while provisioning models from host to GPU. We achieved significant speedup for inferences across all the models and showed the effectiveness of reducing tail latency and improving throughput in model serving systems.

In future work, we envision how our approach can be utilized in other cases. For instance, DeepPlan can allow inferences to models which are not fit in single GPU memory. Since the model size is steadily growing, it is increasingly challenging to cache a model entirely on GPU. Although we may use multiple GPUs for hosting a large model enabled with pipeline parallelism [28], DeepPlan can be a cost-effective alternative for such large models.

In addition, we anticipate that DeepPlan can be extended for mixture of experts (MoE) [31]. In MoE models, all the layers of the model are not required for a given input because each input needs to take an expert. Once we are able to identify the required expert for a given forward pass, DeepPlan could effectively reduce the time spent of transferring models.

Acknowledgments

We thank our shepherd Tim Harris and anonymous reviewers for their valuable comments and feedback. This work was supported by Institute of Information & communication Technology Planning & Evaluation(IITP) grant (No. 2020-0-00844, Development of Lightweight System Software Technology for Resource Management and Control of Edge Server Systems) and the ITRC(Information Technology Research Center) support program(2021-0-02051). Both grants are funded by the Ministry of Science and ICT, Korea.

References

[1] 2017. TorchVision. https://pytorch.org/vision.

[2] 2018. NVIDIA Triton. https://developer.nvidia.com/nvidia-triton-inference-server.

[3] 2020. TorchServe. https://pytorch.org/serve.

[4] 2021. NVIDIA CUDA Toolkit Documentation (v11.3.0). https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/.

[5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[6] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[8] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*.

[10] Aditya Dhakal, Sameer G Kulkarni, and K. K. Ramakrishnan. 2020. GSLICE: Controlled Spatial Sharing of GPUs for a Scalable Inference Platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC)*.

[11] Mario Drumond, Tao Lin, Martin Jaggi, and Babak Falsafi. 2018. Training DNNs with Hybrid Block Floating Point. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NeurIPS)*.

[12] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*.

[13] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[14] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S. Lee, David Brooks, and Carole-Jean Wu. 2020. DeepRecSys: A System for Optimizing End-to-End at-Scale Neural Recommendation Inference. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*.

[15] Song Han, Huizi Mao, and William J Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In *Proceedings of the 4th International Conference on Learning Representations (ICLR)*.

[16] Mark Harris. 2017. NVIDIA DGX-1: The Fastest Deep Learning System. Available at https://developer.nvidia.com/blog/dgx-1-fastest-deep-learning-system/.

[17] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[19] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized Neural Networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NeurIPS)*.

[20] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. 2021. Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product. In *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*.

[21] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*.

[22] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. 2021. Large Graph Convolutional Network Training with GPU-Oriented Data Communication Architecture. *Proc. VLDB Endow.* 14, 11 (jul 2021), 14 pages.

[23] NVIDIA. 2019. NVIDIA DGX-2: The World's Most Powerful AI System for the Most Complex AI Challenges. Available at https://www.nvidia.com/en-us/data-center/dgx-2/.

[24] NVIDIA. 2020. NVIDIA HGX A100: The Most Powerful End-to-End AI Supercomputing Platform. Available at https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/HGX/a100-80gb-hgx-a100-datasheet-us-nvidia-1485640-r6-web.pdf.

[25] Christopher Olston, Fangwei Li, Jeremiah Harmsen, Jordan Soyke, Kiril Gorovoy, Li Lao, Noah Fiedel, Sukriti Ramesh, and Vinu Rajashekhar. 2017. TensorFlow-Serving: Flexible, High-Performance ML Serving. In *Workshop on ML Systems at NIPS 2017*.

[26] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems (NeurIPS)*.

[27] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. 2020. MLPerf Inference Benchmark. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*.

[28] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*.

[29] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*.

[30] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (ATC)*.

[31] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. In *Proceedings of the 5th International Conference on Learning Representations (ICLR)*.

[32] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: A GPU Cluster Engine for Accelerating DNN-Based Video Analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*.

[33] Hans-Nikolai Vießmann and Sven-Bodo Scholz. 2020. Effective Host-GPU Memory Management Through Code Generation. In *Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages (IFL)*.

[34] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*.

[35] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*.

[36] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[37] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (ATC)*.

[38] Jeff Zhang, Sameh Elnikety, Shuayb Zarar, Atul Gupta, and Siddharth Garg. 2020. Model-Switching: Dealing with Fluctuating Workloads in Machine-Learning-as-a-Service Systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*.

A Artifact Appendix

A.1 Abstract

Our artifact includes (1) the `DeepPlan` tool generating the inference execution plans for given DNN models, (2) the libTorch execution engine guided by the generated plans, and (3) the DL inference server prototype powered by our libTorch engine. We also include the DNN workloads used in our paper and script files to set up and test our artifact.

A.2 Description & Requirements

A.2.1 How to access. The DeepPlan artifact is available at https://github.com/csl-ajou/DeepPlan. The `README.md` in the repository includes all the required steps and detailed instructions to build, run, and reproduce our results.

A.2.2 Hardware dependencies. To evaluate the functionality of our proposed schemes, a multi-GPU server (at least two more GPUs) is required. This paper used an AWS `p3.8xlarge` instance equipped with four NVIDIA V100 GPUs connected through NVLinks[2]. Specifically, the DeepPlan (PT) scheme needs two more GPUs to parallelize the model transmission across GPUs. On the other hand, DeepPlan (DHA) is still effective in a single GPU system.

A.2.3 Software dependencies. We evaluated our schemes on the Ubuntu 18.04 distribution and the required software packages are CUDA 11.3, cuDNN 8.2.1, ProtoBuf 3.11.4, PyTorch 1.9, Boost 1.65, and TBB 2017 U7.

A.2.4 Benchmarks. We used representative pre-trained DNN models: `ResNet-50`, `ResNet-101`, `BERT-Base`, `BERT-Large`, `RoBERTa-Base`, `RoBERTa-Large`, `GPT-2`, and `GPT-2 Medium`. The benchmark inputs are a synthetic dataset.

A.3 Set-up

Please refer to the `README.md` file in the artifact repositories for detailed environment set-up instructions and how to run the experiments to reproduce the key results.

A.4 Evaluation workflow

A.4.1 Major Claims. We claim that DeepPlan minimizes the performance overhead of inferencing when models are not ready in the GPU memory. By alleviating the performance penalty induced by model transmission from host to GPU memory, DeepPlan is able to achieve high throughput (goodput) while preserving the bounded latency (SLO).

C1: When executing a DL inference request on a GPU, we suggested that all the layers for a given DL model are not needed to be in the GPU memory. DeepPlan (DHA) enables GPUs to perform the computation on layers in the host memory, eliminating the time for transmitting the layers to the GPU memory. This is proven by the experiment (E1) described in Section 5.2, whose results are illustrated in Figure 11. The DeepPlan (DHA) legend represents the relative speedup from the baseline.

C2: To accelerate the model transmission time from host to GPU, we claim that leveraging multiple GPUs can parallelize the model transmission through individual PCIe lanes of each GPU. This claim is supported by the experiment (E2) described in Section 5.2, and the DeepPlan (PT) in Figure 11 exhibits performance improvement compared to the baseline and DeepPlan (DHA). DeepPlan (PT+DHA) presents the performance numbers when integrating DHA and PT schemes.

C3: By leveraging the proposed schemes in DL model serving systems, we can improve the throughput (and goodput) while preserving the bounded latency of DL models. This is validated in both synthetic and realistic server workloads described in Section 5.3 and proven by the experiments (E3) and (E4), respectively.

A.4.2 Experiments. Detailed instructions on how to prepare the environment used by this paper and how to reproduce the results are found in `README.md`.

Experiment (E1): [DeepPlan (DHA)] [30 minutes] This experiment produces the results of DeepPlan (DHA) in Figure 11. We provide a script file, `scripts/fig10/run.sh`, to automate the steps. It runs a single inference with batch size 1 for our baseline and the pipeline scheme [6] to measure the latency. For DeepPlan (DHA), it generates an execution plan, runs the inference with the plan, and produces a graph file.

Experiment (E2): [DeepPlan (PT)] [30 minutes] This experiment produces the results of DeepPlan (PT) in Figure 11. For the parallel-transmission scheme, the server system needs to have two more GPUs at least. Like above, the `scripts/fig10/run.sh` file helps to run, measure, and produce the latency result of a single inference with batch size 1. In addition, we include the latency numbers when integrating the two schemes, DeepPlan (PT+DHA), in Figure 11.

Experiment (E3): [DeepPlan Server (Synthetic)] [4 hours] This experiment produces the results in Figure 13 and 14. We perform this experiment on a four-GPU server in an AWS instance. This experiment measures the 99% latency, goodput, and cold-start for three workloads, `BERT-Base`, `BERT-Large`, and `GPT-2`, while increasing the number of model instances concurrently running on the GPUs. The provided script files, `scripts/fig12/run.sh` and `scripts/fig13/run.sh`, produce graphs by automating the required steps. Note that Figure 13 and 14 are the same type of evaluation but different benchmark models. Figure 14 omits the result of goodput and cold-start ratio.

Experiment (E4) [DeepPlan Server (real-world)] [9 hours] This experiment is similar to Experiment (E3), but it runs with a real-world trace derived from Microsoft Azure Functions [30]. We scale down the traces for a four-GPU server environment. The adjusted traces can be found in our artifact repository and `scripts/fig14/run.sh` replays the traces and produces graphs in Figure 15.

[2]We tested our code on two NVIDIA A5000 GPUs with NVLink.

A.5 Notes on Reusability

To leverage `DeepPlan` on different hardware environments, it is required to tune the request rate so that the inference server can show reasonable throughput numbers while satisfying the target latency.