DVABatch: Diversity-aware Multi-Entry Multi-Exit Batching for Efficient Processing of DNN Services on GPUs

Weihao Cui, Han Zhao, Quan Chen, Hao Wei, and Zirui Li, *Shanghai Jiao Tong University;* Deze Zeng, *China University of Geosciences;* Chao Li and Minyi Guo, *Shanghai Jiao Tong University*

https://www.usenix.org/conference/atc22/presentation/cui

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the 2022 USENIX Annual Technical Conference is sponsored by

DVABatch: Diversity-aware Multi-Entry Multi-Exit Batching for Efficient Processing of DNN Services on GPUs

Weihao Cui*, Han Zhao*, Quan Chen*, Hao Wei*, Zirui Li*, Deze Zeng◇, Chao Li*, Minyi Guo*
*Shanghai Jiao Tong University, ◇China University of Geosciences

Abstract

The DNN inferences are often batched for better utilizing the hardware in existing DNN serving systems. However, DNN serving exhibits diversity in many aspects, such as input, operator, and load. The unawareness of these diversities results in inefficient processing. Our investigation shows that the inefficiency roots in the feature of the existing batching mechanism: one entry and one exit. Therefore, we propose **DVABatch**, a runtime batching system that enables the multi-entry multi-exit batching scheme. We first abstract three meta operations, new, stretch, and split, for adjusting the ongoing batch of queries to achieve the multi-entry multi-exit scheme. The meta operations could be used to form different scheduling logics for different diversities. To deliver the meta operations to an ongoing batch, we slice the DNN models into multiple stages. Each stage corresponds to one executor, which is managed by a state transition diagram. Compared with state-of-the-art solutions, our experimental results show that DVABatch reduces 46.4% average latency and achieves up to 2.12× throughput improvement.

1 Introduction

Deep neural networks (DNNs) [27, 37, 57] are widely used in intelligent services [1, 5, 6, 12]. Since user queries have stringent QoS in terms of end-to-end latency, dedicated accelerators like GPUs [10, 11] and NPUs [21] are used to speed up the DNN inferences. However, a single DNN query often cannot fully utilize these accelerators [17, 18, 43, 46, 65] (e.g., An Nvidia Titan RTX GPU has 72 SMs [10]). Therefore, emerging DNN serving systems (e.g., Clipper, Triton, TF-Serving) [9, 23–26, 51, 62, 63] batch queries for better taking advantage of the accelerators' parallelism. Queries that arrive in a given batch time window are organized into a batch, and an *executor* (process) is used by the DNN serving system to process the entire batch at a time. Such a batching policy uses the same batch size (*bs*) across a single inference process.

On GPUs, due to the single program multiple-data (SPMD) design, all the queries in a batch return at the same time. This batching pattern is referred to be the *single-entry single-exit batching pattern*. It works great for best-effort applications, and the services when the queries arrive in uniform intervals [22]. However, DNN serving scenarios show various diversities, and the single-entry single-exit batching pattern results in the long response latency of inference queries on GPUs. For instance, we find at least three types of diversities when serving DNN models, as shown in Figure 1.

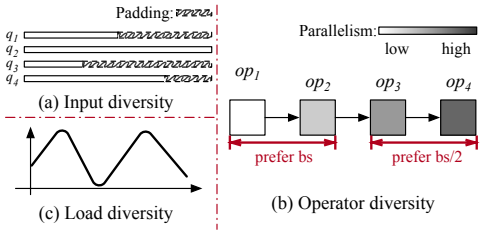

Figure 1: Serving diversities in real-world services.

Input Diversity. The inputs of user queries show high diversity (e.g., in natural language processing services). Short queries are all padded to the size of the longest query for batching. The benefit of batching may be negated by the wasted computation of the padded part.

Operator Diversity. While all the operators of a DNN model share the same batch size, they have different preferred batch sizes. An operator's preferred batch size is the smallest batch size that fully utilizes the current GPU. The hardware is not fully utilized if an operator's preferred batch size is larger than the used batch size. Otherwise, the processing time is increased unnecessarily.

Load Diversity. The service queries do not arrive in a uniform interval. In this case, the number of queries collected in a single batch time window varies. When the load bursts, a previous non-full batch results in the long latency of subsequent queries. In other words, hardware resources are wasted while the queries are waiting in the next batch time window.

The diversities result in inefficient processing of user queries (discussed in more detail in Section 3). The ineffi-

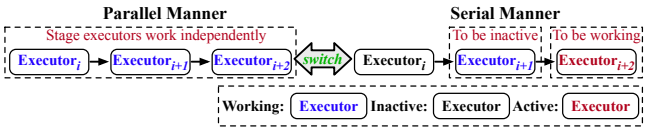

Figure 2: The work manner switch of stage executors.

ciency stems from the batching pattern of single-entry single-exit. To address the inefficiency, we, therefore, propose a *multi-entry multi-exit batching scheme* for DNN serving on GPUs. For instance, with the multi-entry multi-exit batching scheme, a short query can exit early without waiting for the entire batch to exit (input diversity), a batch can be split into smaller batches to execute an operator with a preferred smaller batch size (operator diversity), and the queries that arrive later can join an ongoing but non-full batch (load diversity).

It is nontrivial to implement the multi-entry multi-exit batching scheme for GPUs. *It necessitates the ability of the serving system to dynamically alter the batch size of an ongoing query batch.* Specifically, such a system should enable the joining of incoming queries into the ongoing batch and the splitting of an ongoing batch into smaller batches (The queries in the smaller batches could exit independently). *Moreover, it introduces extra complexity for designing executors in the DNN serving system.* With the multi-entry multi-exit scheme, the inference of batched queries is broken down into multiple stages, and each stage's execution requires one executor. Multiple executors have to coordinate with each other to ensure the validity of the query inference.

To this end, we propose **DVABatch** to enable the multi-entry multi-exit batching scheme effectively. DVABatch provides three meta operations, *new*, *stretch*, and *split*, to adjust the ongoing batch (Section 5). The *new* operation creates a new batch, just like the traditional batching strategy. The *stretch* operation adds new queries to the ongoing batch. The *split* operation breaks a running batch into multiple batches, which could be scheduled separately. Query batching can be done in a variety of ways using the three meta operations.

To deliver the meta operations to the stage executors, a *batch queue* that stores the batch information is added between adjacent stage executors, and a global *batch table* is utilized to record the to-be-performed meta operations at each stage (Section 5). When an executor completes its computation for a batch of queries, it verifies the to-be-performed meta operations for the next stage in the *batch table*. If a *split* or *stretch* operation is required, the executor applies the corresponding meta operation on the current batch and pushes new batches of queries into the batch queue of the next stage.

While multiple active executors run independently like a software pipeline, DVABatch should manage them properly. Otherwise, the naive parallel execution of the executors invalidates the execution due to data hazard and results in unnecessary long latency. For instance, the executor should run batches with different input sizes in parallel for the input diversity, and run the sub batches after the *split* meta operation sequentially for the operator diversity. DVABatch introduces a state transition diagram based solution to support the executors' complicated runtime scheduling (Section 6). Each executor has four states: *active*, *checking*, *working*, and *inactive*,. Through the state transition diagram, the work manners depicted in Figure 2 are both supported.

The main contributions of this paper are as follows.

- We propose a multi-entry multi-exit batching scheme for efficient DNN service processing on GPUs.
- We provide a general scheduling mechanism that leverages meta operations, and state transition diagram to create policies for different serving diversities.
- We implement DVABatch with Triton, a state-of-the-art DNN serving system. Our experimental results on Nvidia Titan RTX show that DVABatch reduces 46.4% average latency and achieves up to 2.12× throughput improvement for the involved serving diversities.

2 Related Work

Many systems have been proposed for efficient DNN inference [9, 25, 35]. Clipper [23], TF-Serving [51], Triton [9] adopted the traditional batch strategy that uses batch time window and the maximum allowed batch size. They treated the DNN model as an indivisible whole. They left the scheduling of inner operators to their supported backends. These works do not perceive the serving diversity and utilize the DNN operator scheduling for efficient processing.

There are some prior research on improving operator scheduling. TensorFlow Fold [47], DyNet [49], and BatchMaker [31] focused on the runtime scheduling of operators for RNN. They are model-specific solutions, removing padding for RNNs. The RNN cells of the same type share the same parameter weights and are executed recursively [38]. These works relied on this property to remove the padding. The design is restricted to resolving the input diversity for RNN. It cannot be applied to other models with input diversity, e.g., attention-based models, and BatchMaker-like batch mechanism can be achieved through DVABatch's meta operations. Besides, LazyBatch [22] cared about the load fluctuation and proposed batching queries lazily. LazyBatch performed per-operator scheduling that incurs high scheduling overhead. It could not handle other diversities. In this work, we focus on resolving the problems caused by serving diversities in a holistic way.

There are also some prior works about ragged tensor for the input diversity [4, 29]. They generated the customized implementation of operators to remove the padding. However, operators like GEMM and convolution cannot be optimized through these works, which dominate the computation in DNN. These works are orthogonal to DVABatch and can be combined together to enable even lower latency.

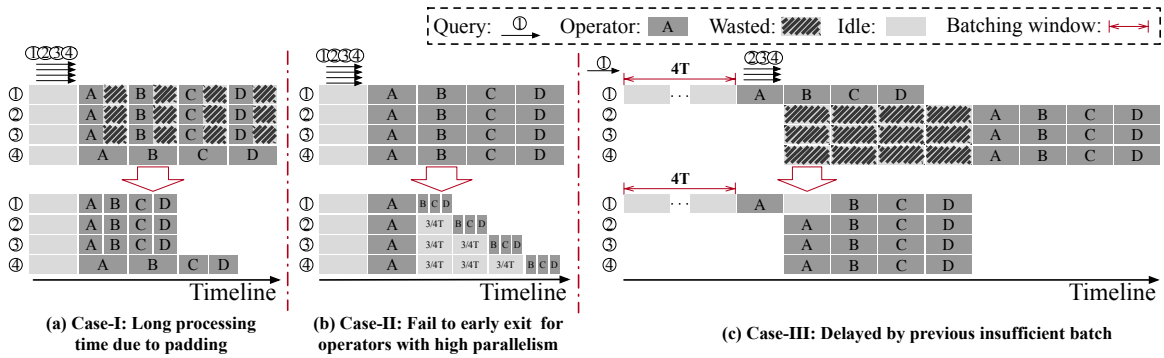

Figure 3: The long latency of user queries due to (Case-I) input diversity, (Case-II) operator diversity, and (Case-III) load diversity.

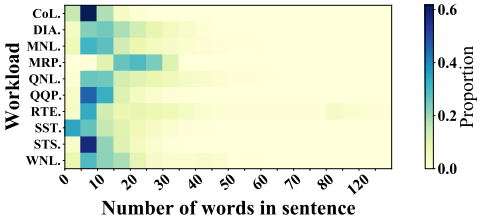

Figure 4: The sequence distribution of workloads in GLUE.

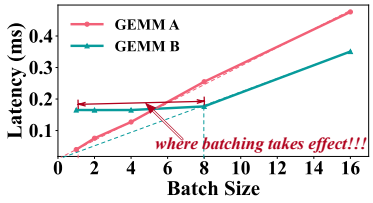

Figure 5: Latencies of two GEMM operators with different batch sizes on Titan RTX.

3 Background and Motivation

This section shows the long query latency problem due to the single-entry single-exit batching, and motivates the design of DVABatch. Figure 3 shows the three involved diversities. For simplicity of illustration, we assume that each operator completes in 1 time unit (T), and the batch size is 4. In this case, once 4 queries are received or the batch time window ends, the received queries are batched and issued to run.

3.1 Input diversity

Input diversity widely exists in DNN services. E.g., natural language processing services often process sentences of different lengths. Figure 4 shows the sequence length distribution in 10 workloads of the General Language Understanding Evaluation (GLUE) dataset [59]. As observed, most sentences have 5-20 words, but some have more than 100 words.

For these models, the input of short queries is padded to the same size as the input of the longest query so that they can be batched to run [28]. Case-I in Figure 3 shows the batching with input diversity. As shown in the upper part of Case-I, the hardware resources are wasted for the computation of extra paddings (the queries in a batch return simultaneously).

The lower part of Case-I shows a better batching strategy: the batch is divided into two smaller batches, one for short queries ①, ②, and ③, and one for the long query ④. In this way, queries ①, ②, and ③ return earlier, and the average latency is reduced by 37.5% (from $4T$ to $2.5T$). Note that the two batches may run in parallel before the short batch completes if 4 operators are required to fully utilize the GPU.

3.2 Operator diversity

For a DNN service, the operators often require different batch sizes to fully utilize the GPU. Figure 5 shows the latencies of two General Matrix Multiplication (GEMM) operators converted from two convolution operators of Resnet50 [37], with the shapes of $[bs*3136,576]\times[576,64]$ and $[bs*49,576]\times[576,512]$. GEMM operators dominate DNNs (occupying 86% of the computation time) [44].

As shown, the preferred batch sizes of *GEMM-A* and *GEMM-B* are 1 and 8, respectively. For *GEMM-A*, batching only increases its latency without improving the processing throughput. For *GEMM-B*, using a batch size smaller than 8 is not able to fully utilize a GPU (the processing time does not increase until the batch size is larger than 8).

Case-II in Figure 3 shows the batching with operator diversity. In Case-II, operator A prefers batch size 4, operator B, C, and D prefer batch size 1. The lower part of Case-II shows a better batching strategy: we can run operator A with batch size 4, split the batch into four smaller batches with batch size 1 at operator B, and run the small batches sequentially. In this way, query ①, ②, and ③ return earlier. The average latency can be reduced by 28.1% (from $4T$ to $2.875T$).

Many DNN models suffer from operator diversity. Figure 6 shows the processing time of the operators in Unet [56], a widely used image segmentation network with different batch

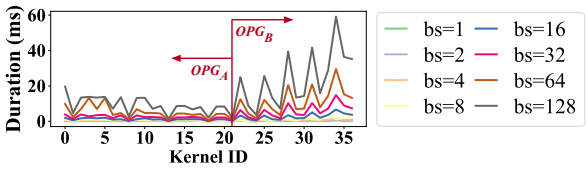

Figure 6: Operator duration of Unet on Titan RTX with variable batch sizes.

sizes on an Nvidia Titan RTX GPU. In the figure, the *x*-axis represents each operator's id in the executive order. As observed, most operators in the former part (OPG_A) benefit from large batch sizes (e.g., larger than 32), but the operators in the latter part (OPG_B) only benefit from small batch sizes (smaller than 8). Using a batch size larger than 8 for Unet, the operators in OPG_B have longer latency without throughput improvement. On the contrary, using a batch size smaller than 32, the operators in OPG_A do not fully utilize the GPU.

3.3 Load diversity

User queries do not arrive in a uniform interval, as end users may randomly submit their queries. The number of received queries in a single batch time window varies [30,34,36,41,55].

Case-III in Figure 3 presents the batching with the load diversity. The batch time window is $4T$, the operators prefer batch size 4. With the current batching policy (the upper part of Case-III), query ① starts to run alone after it waits for $4T$, and the GPU is not fully utilized. During the processing of query ①, three queries ②, ③, and ④ arrive, but they have to wait to be executed in the next batch.

The lower part of Case-III shows a better way to run the four queries: the first batch (query ①) waits for the second batch after the first operator *A* completes. Then, the two insufficient batches are merged into a new batch that fully utilizes the hardware. In this way, the average latency can be reduced by 34.4% (from $8T$ to $5.25T$).

3.4 Diversities among DNN services

The three types of diversities may exist in the same DNN services. Current batching policies with simple modifications cannot effectively handle them. In this case, designing a static batching policy is not able to fulfill the ever-changing diversities. A low-level batching mechanism that supports configuring the batching policy accordingly is required. Analyzing the three better batching cases in Figure 3, they share three requirements for the batching mechanism.

First of all, the mechanism should be able to interrupt an ongoing batch so that we can adjust the inappropriate batching decision. Second, the mechanism should be able to split a large batch into small batches. In this way, the small batches may run in a parallel manner (Case-I) or sequentially (Case-II). Third, the mechanism should be able to merge multiple insufficient batches. In this way, we can build a large batch to better utilize the hardware resource (Case-III).

A multi-entry multi-exit batching scheme fulfills all three requirements, and has the potential to achieve better batching, together with appropriate batching policies.

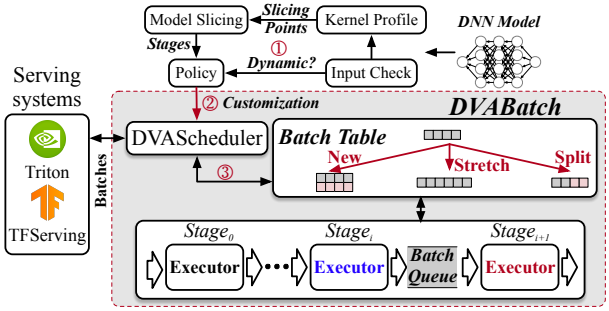

Figure 7: Design of DVABatch.

4 Design of DVABatch

We therefore design and propose DVABatch to resolve the long latency problem due to the serving diversities.

4.1 Overview

Figure 7 illustrates DVABatch's methodology. DVABatch enables the multi-entry multi-exit batching scheme for the upper-level DNN serving systems (e.g., Triton, TF-Serving).

In general, in order to support the multi-entry and multi-exit batching, DVABatch slices a DNN model into multiple fine-grained stages, and each stage has multiple adjacent operators. The queries are able to join a batch at the beginning of a stage, and exit from a batch at the end of a stage. Based on the stages, DVABatch designs and implements batching policies that manage the batching operation of each stage, based on the real-time diversities. DVABatch supports three meta-operations *new*, *stretch*, *split* for adjusting the batching. The *new* operation creates a new batch, *stretch* adds new queries to the ongoing batch, and *split* breaks a running batch into multiple smaller batches. Various batching policies can be implemented based on the meta-operations.

As shown in Figure 7, DVABatch comprises a *batch table*, *stage executor*s, *batch queue*s, and *DVAScheduler*.

- The *batch table* records the running status of the ongoing batches. It supports three meta operations for adjusting the batches at each stage.
- A *stage executor* is a process that is responsible for the corresponding stage's execution. DVABatch utilizes a state transition diagram for executor management.
- There is a *batch queue* between the two adjacent stages, for transmitting the batch information. A stage executor pulls

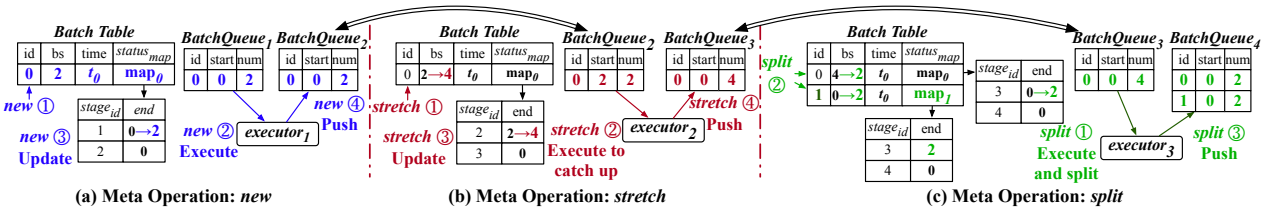

Figure 8: Implementing the meta operations with batch table and batch queues between stages.

out batches from its previous *batch queue* for execution, and pushes batches into the *batch queue* of the next stage.

- *DVAScheduler* provides diversity-aware scheduling using various batching policies implemented with the three meta operations. The policies can be customized according to the serving diversity.

4.2 The Serving Workflow with DVABatch

Figure 7 also shows DVABatch's serving workflow for the involved diversities. The steps for using DVABatch to serve a DNN service are as follows:

① DVABatch checks the input data pattern of the service, and profiles it to obtain the diversity patterns. Based on the profiling, the DNN model is sliced into stages automatically. A diversity-aware policy is generated for the DNN model.

② Each stage executor loads the corresponding stage of the model, and DVAScheduler uses the scheduling logic in the policy to schedule the accepted queries.

③ If a specific condition defined in the DVAScheduler is satisfied, the batch table is instructed to adjust the ongoing batch by *new, stretch, split* operations accordingly.

For a new DNN service, DVABatch handles input diversity before operator diversity. This is because handling input diversity directly reduces computation while handling operator diversity better schedules computation.

5 Enabling Multi-entry Multi-exit Scheme

In this section, we propose the abstraction of meta operations and how we achieve the multi-entry multi-exit scheme.

5.1 Defining the Meta Operations

As stated before, to achieve low latency query scheduling, the batched queries should be able to join and exit the batching system in several forms: **Multi-entry.** When a new batch of queries arrives, it can interrupt an ongoing batch, catch up with the progress of the interrupted one, then be merged into a single larger batch. In addition, the new incoming batch also can join the processing by co-running with the ongoing batches in different stage executors without pausing the ongoing one. **Multi-exit.** If some queries inside a batch need to exit early, we need to split the batch into several batches and allow them to exit execution independently.

DVABatch abstracts three meta scheduling operations inside the DVAScheduler: *new*, *stretch*, and *split*, for supporting the multi-entry multi-exit scheme. With *new*, the new incoming queries are organized into a new batch. The batch created by *new* operation could co-run with the previous batches; With *stretch*, an ongoing batch is stretched with new incoming queries. At a specific stage, these queries are merged into the ongoing batch for processing; With *split*, a large ongoing batch is split into several smaller batches to be processed separately. The three meta operations can be used to form complicated scheduling logic when necessary.

5.2 Implementing the Meta Operations

It is challenging to support the meta operations, as a batch runs in multiple stages. The meta operations should be performed based on the stages' execution status. In general, DVABatch tracks the stage status of the batches, and notifies the meta operations to the corresponding stage executors.

5.2.1 Batch Table and Batch Queues

DVABatch uses a *batch table* to track the processing status of all the batches on a GPU. The batch table is updated by the DVAScheduler through the meta operations.

As shown in Figure 8, a row in the batch table records the status of an ongoing batch. In a row, *id* is the batch's identifier, *bs* is its current batch size, *time* is the timestamp the batch is created, $status_{map}$ is a map that records the number of completed queries in each stage of the batch. For instance, the first row of $status_{map}$ in Figure 8(a) means stage 1 has completed 2 queries in its batch. We need $status_{map}$ because the latter queries of a *stretched* batch should catch up with the ongoing queries. With $status_{map}$, the executors could get the right number of queries to execute after *stretch* operation.

After the current stage executor completes its execution, it notifies the subsequent stage executor to run. DVABatch maintains *batch queues* between adjacent stage executors to trigger such execution. In a row of a batch queue, *id* is the batch's identifier, *start* is the id of the to-be-processed query, and *num* is the number of to-be-processed queries in the batch.

The stage executor pulls a batch from its batch queue, and processes the queries accordingly. For instance, $executor_1$ in Figure 8(a) will run query 0 to query 1 ($start = 0$, $num = 2$) in the current batch. Once the execution completes, the stage executor updates the row of the processed batch in batch table, and pushes an item into the next batch queue.

5.2.2 Handling Meta Operations

Based on batch table and batch queues, Figure 8 shows an example that three meta operations are performed on the same batch, $batch_0$. In the example, we first *new* the batch $batch_0$ with 2 queries (Figure 8(a)); Then, we *stretch* the batch $batch_0$ with another 2 new queries while it is already processed by $executor_2$ (Figure 8(b)); Last, we *split* the batch $batch_0$ into 2 smaller batches at the third stage (Figure 8(c)).

Handling *new*. Once $batch_0$ is received, ① a *new* operation is instructed, and a new item is added to the batch table. Meanwhile, an item is pushed to the first stage executor's batch queue ($BatchQueue_1$). ② the executor of the first stage (hereinafter, we refer to the executor of $stage_i$ as $executor_i$) is notified to obtain the item and perform the execution. Once $executor_1$ completes, it ③ updates $status_{map}$ in the batch table, and ④ pushes an item into $BatchQueue_2$.

Handling *stretch*. ① As 2 new queries are added into $batch_0$ with the *stretch* operation at stage 2, bs of $batch_0$ in the batch table is changed from 2 to 4. Because $batch_0$ is *stretch*ed to 4 queries while being processed by $executor_2$, the executor does not push an item into $BatchQueue_3$, but only updates $status_{map}$. ② A new item (a batch with $id = 0, start = 2, num = 2$) is pushed into $BatchQueue_1$, so that the newly added queries can catch up with the progress of the current batch. ③ Once the new queries catch up, $executor_2$ updates $status_{map}$ and ④ pushes a merged batch into $BatchQueue_3$ (a batch with $start = 0$ and $num = 4$). The *stretch* operation is only performed on the latest batch stored in the batch table.

Handling *split*. When $batch_0$ goes to stage 3, ① $executor_3$ pulls $batch_0$ from the batch queue and runs it with $bs = 4$. After that, $executor_3$ is instructed with a *split* operation. ② The original batch is split into two batches in the batch table. ③ Lastly, the generated batches are all pushed into $BatchQueue_4$ one by one. The *split* operation can happen when a new batch of queries comes or during the execution of an ongoing batch.

The meta operations co-exist without conflict. Although *split* happens at any stage, a potential *split* operation can be known when generating the batch by *new* and *stretch*. We disable further *stretch* for the batch that we perceive *split* operation. The batches generated by *split* inherit this property.

6 Managing Stage Executors

In this section, we present the way the stage executors are organized to process the batches.

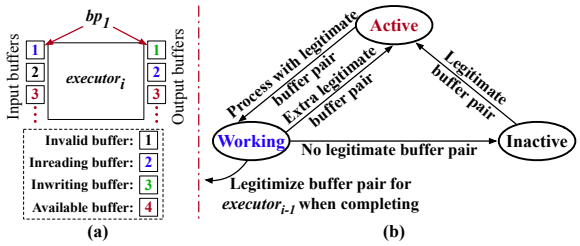

Figure 9: **(a)** Stage executor processes batches with multiple buffers; **(b)** Traditional state transition diagram of executor$_i$.

6.1 Processing with Multiple Buffers

With multiple stages, the executors of adjacent stages have "producer-consumer" relationships. In this case, there are data hazards on the buffers between stages.

Figure 9(a) shows the way the stages are connected. Multiple buffers are used because multiple batches may be active concurrently. An executor needs to obtain an input-output buffer pair before it can process a batch. The output buffer of $executor_i$ is also the input buffer of $executor_{i+1}$. If $executor_i$ is using a buffer pair bp, there is a Write-After-Read hazard on bp's input buffer, as $executor_{i-1}$ may write to the very buffer before $executor_i$ reads the data. Similarly, there is a Read-After-Write hazard on bp's output buffer.

For ensuring the execution correctness, a buffer can be in the *available*, *invalid*, *inreading*, or *inwriting* state. A buffer is *invalid* when it cannot be used as an input buffer currently. It is *inreading*/*inwriting* when it is used as an input/output buffer for a batch's execution. It is *available* when it is not used by any executor. Figure 9(a) shows an example that $executor_i$ is using the first buffer pair bp_1, and $executor_{i+1}$ is using the output buffer of the second buffer pair bp_2 as its input.

- The input buffer of bp_1 is in *inreading* state and the output buffer of bp_1 is in *inwriting* state.
- The input buffer of bp_2 is in *invalid* state because the output buffer of bp_2 is currently used by $executor_{i+1}$ for execution. $executor_i$ cannot use it to run a new batch.
- The third buffer pair are both in *available* states.

A stage executor can run a batch only when it successfully obtains a *legitimate* buffer pair. A buffer pair is legitimate, when both the input and output buffer are *available*, or the input buffer is *available* but the output buffer is *invalid*. This is because an *invalid* buffer can be used as the output buffer of an executor, but cannot be used as the input buffer of the later stages.

6.2 State Transition of the Executors

Based on the buffer states, there are some traditional ways to create a naive state transition rule for the stage executors to run as a pipeline. For instance, a stage executor can be in three states: *active, working, inactive*, and the states change

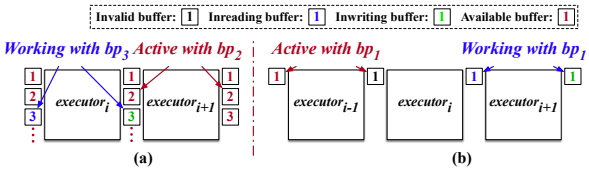

Figure 10: **(a)** Inconsistency of buffer pairs; **(b)** Fail to run in serial manner with single buffer pair.

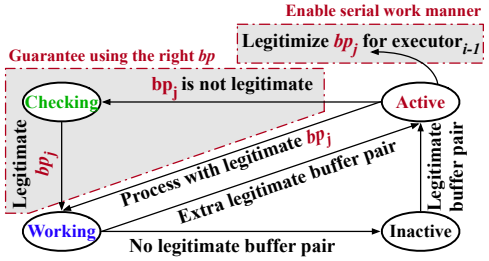

Figure 11: State transition diagram of executor$_i$ in DVABatch.

according to the diagram in Figure 9(b).

However, these traditional state transition rules assume all the stage executors use the buffer pairs in some fixed order (e.g., ID order). **While the traditional rules work well for the single-entry single-exit pipeline, they will encounter the validity problem for the multi-entry multi-exit pipeline.**

Specifically, the requirement of meta operations *stretch* and *split* invalids the above traditional transition rules. *stretch* merges the outputs from different buffer pairs into a single one, and *split* may split the output into multiple buffer pairs. That means some stage executors may use more buffer pairs than others, and the access order of these buffer pairs is scrambled. As shown in Figure 10(a), while $executor_i$ is using bp_3 for execution, $executor_i$ is active with bp_2. Such inconsistency invalidates the traditional state transition rule.

On the other hand, stage executors need to run multiple batches in parallel for load diversity and input diversity, and run batches sequentially for operator diversity (Section 3). The traditional transition rule supports parallel manner well but fails to satisfy the requirement of serial manner. As shown in Figure 10(b), the stage executors always run in parallel even if only one buffer pair is used. Because $executor_i$ legitimizes bp_1 for $executor_{i-1}$ after it completes the execution, $executor_{i-1}$ is *active* with bp_1, when $executor_{i+1}$ is in *working* state with bp_1. In this case, $executor_{i-1}$ is able to run in parallel with $executor_{i+1}$.

We therefore design the transition diagram in Figure 11. Suppose $executor_i$ is in the *active* state with a legitimate buffer pair currently. **Once $executor_i$ pulls out a batch with buffer pair bp_j, it checks the states of bp_j instead of starting execution directly.** If bp_j is legitimate, $executor_i$ enters *working* state and starts the execution. Meanwhile, the input buffer and output buffer of bp_j enter *in-reading* and *in-writing* states respectively. **If bp_j is not legitimate, $executor_i$ enters *checking* state, waiting for bp_j to become legitimate.** When $executor_i$ completes a batch with bp_j, the input buffer of bp_j enters *invalid* state, and the output buffer of bp_j restores to its previous state. If $executor_i$ gets another legitimate buffer pair, it enters *active* state. Otherwise, it enters *inactive* state. The last stage executor always re-enters *active* state directly.

Note that, after $executor_i$ re-enters *active* state with any legitimate buffer pair, $executor_i$ updates the j-th input buffer of $executor_{i-1}$ to *available* state. It denotes that $executor_{i-1}$ can use bp_j now.

In short, we first add a *checking* state for the stage executor to guarantee using the right buffer pair. We also move the buffer pair legitimation for $executor_{i-1}$ after $executor_i$ enters *active* state. In this case, while DVABatch only configures one available buffer pair, all stage executors stay in *inactive* state until the current batch is executed by the last stage executor. Therefore, the serial work manner is supported.

6.3 Implementing the Transition Diagram

We implement the transition diagram of stage executors through CUDA synchronization APIs for the correctness guarantee. Each stage executor is bound to a CUDA *stream* for parallel execution, and each buffer pair is equipped with a CUDA *event* to enforce its legitimation. Upon finishing launching the CUDA functions with a buffer pair, the stage executor performs a *record* operation with the buffer pair's *event* on its *stream*. After that, if the corresponding buffer pair is legitimate, the other stage executor requires synchronization with the *event* on its own *stream* before using the buffer pair to avoid data hazards on GPUs. In order to deliver the best performance, the stage executor calls *cudaStreamWaitEvent* instead of explicit synchronizations on the host side.

7 Scheduling Policies of Serving Diversities

In this section, we present the way to deal with the serving diversities using DVABatch. First of all, we identify the existing diversities in a DNN model and divide the model into several stages. Then, at runtime, DVABatch is able to schedule the batches of the model appropriately.

7.1 Identifying Diversities and Slicing Models

For a DNN service, We identify the existing diversities and slice the model offline, by checking the input patterns and profiling the model with several different batch sizes (e.g., 1, 2, 4, 8, 16, 32, 64) using the tools provided by Nvidia [7, 8].

All the models are considered to have load diversities, as the load pattern is often determined by the end-users. The model that accepts inputs with different shapes (dynamic dimension

```
1  //stage executors run within a while loop
2  void Run():
3    Batch& inBatch = BatchQueue.Get();
4    CheckBuffer(inBatch);//Check buffer pair
5    Execute(inBatch);
6    Schedule(inBatch, outBatches);
7    for (auto& batch : outBatches):
8      nextBatchQueue.Push(batch)
9      getBuffer(); //get legitimate buffer pair
10   updatePrExecutor();//update preceding executor
11 //call Schedule to perform meta operations
12 void Schedule(Batch& inBatch, vector<Batch>&
       outBatches):
13   BatchTable.update(inBatch);
14   if userDefined1:
15     outBatches = BatchTable.New(inBatch);
16   else if userDefined2:
17     outBatches = BatchTable.Stretch(inBatch);
18   else if userDefined3:
19     outBatches = BatchTable.Split(inBatch);
20   else:
21     outBatches.Copy(inBatch);
```

Figure 12: Creating scheduling policies with meta operations.

except for batch size) has input diversity. During the profiling, we obtain the preferred batch size of each operator, as shown in Figure 5. When the operators have different preferred batch sizes, the model has operator diversity.

Once the diversities are identified, DVABatch slices the DNN models into stages. If a model is sliced into N_{st} stages, the operators are time-evenly assigned to the stages in the topological order for simplicity. It is non-trivial to theoretically identify the optimal N_{st}. If N_{st} is too small, the opportunity for batch scheduling is limited. Otherwise, if N_{st} is too large, the fine-grained stages incur heavy scheduling overhead. Moreover, unlike Pipedream [48], the model slicing in DVABatch can also be tight with diversities. E.g., DVABatch considers the operators' preferred batch size and slices a model at specific operators for operator diversity. We currently determine the optimal N_{st} through profiling. It can be done in 10 minutes (8 tries) for emerging benchmarks on each type of GPU. It is worth noting that the model slicing does not conflict with the compilation techniques like kernel fusion [64]. We slice the model after it is already optimized by the DNN compilers.

7.2 Defining Policies with Meta Operations

Figure 12 shows the interface provided by DVABatch to define batch scheduling policies with the meta operations. Each stage executor runs in a while loop (*Line 2-11*) to execute the batches (*Line 3, 6, 8* stated in Section 5, *Line 4, 5, 9, 11* stated in Section 6). The stage executor calls *Schedule*() to schedule the batches at *Line 5*. Inside *Schedule()*, the stage executor updates the batch table, and performs meta operations if user-defined conditions are satisfied accordingly.

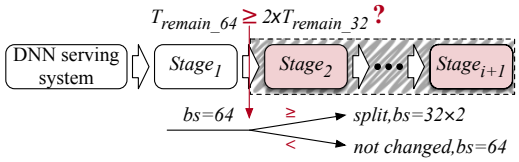

Figure 13: Scheduling according to the preferred batch size.

The policy is implemented outside user models, and does not require modification to the model. We then show the policies defined to handle the three diversities.

Policy I for input diversity. The input diversity requires running multiple small batches in parallel. While accepting a batch of queries from the upper-level serving systems, DVAScheduler clusters the queries according to their input sizes. The queries with similar input sizes are batched and padded to the same size for processing. DVABatch processes these batches in parallel for better utilizing the hardware parallelism. As the batches run in parallel, the scheduler prefers a smaller batch time window instead of generating the batches as large as possible. Practically, we set the batch time window to be the duration of the first stage with the largest allowed batch size bs_{max}. The number of active queries in the software pipeline does not exceed bs_{max}.

Policy II for operator diversity. Figure 13 shows the way DVABatch schedules the next stage when a stage completes. Assume the current batch size (denoted by bs) is 64 for $stage_1$ in the figure. The DVAscheduler compares the processing time of all the remaining stages with different batch sizes. Let T_{remain_i} represent the time needed when batch size is i. In Figure 13, if $T_{remain\_64} \geq 2 \times T_{remain\_32}$, the large batch is split into two batches with $bs = 32$. The two smaller batches run in the serial manner, as the hardware is already fully utilized with $bs = 32$. The duration of each stage with the different batch sizes is already profiled offline.

Policy III for load diversity. At load diversity, a latter batch should be able to join a previous batch, if it does not result in the QoS violation of the previous batch. In this case, DVABatch uses *stretch* operation to enlarge the batch at runtime. We set a time threshold T_{comp_wait} to eliminate the possible QoS violation. If a batch is already processed for T_{comp_wait}, no *stretch* operation is allowed on this batch.

Load diversity is widespread. For input diversity, as the new batches are allowed to enter the software pipeline independently, it already resolves the load diversity. For operator diversity, Policy III and Policy II work together due to the co-existence of *stretch* and *split*, as stated in Section 5.2.2.

8 Evaluation of DVABatch

In this section, we evaluate the performance of DVABatch in reducing the latencies of DNN services.

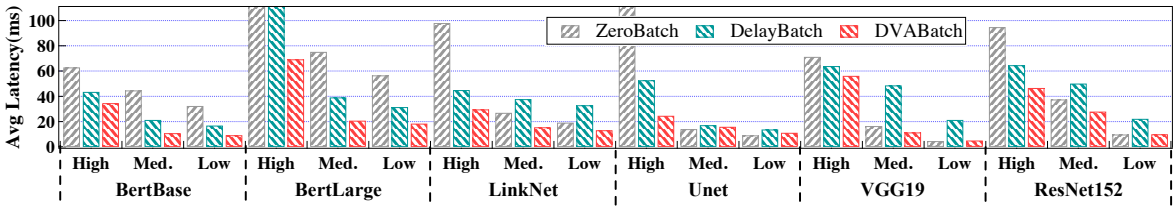

Figure 14: Average latencies of six DNN services at low, medium, high load with ZeroBatch, DelayBatch, and DVABatch.

Table 1: Evaluation specifications.

Hardware	CPU: Intel Xeon E5-2620, GPU: Nvidia Titan RTX
OS & Driver	Ubuntu: 18.04.6 (kernel 4.15.0); GPU Driver: 470.57
Software	CUDA: 11.4; TensorRT: 8.03; Triton 21.10
Benchmarks	Unet [56]; LinkNet [16];BertBase; BertLarge [27]; VGG19 [57]; ResNet152 [37]
Dataset	GLUE [59]

8.1 Experiment Setup

We implement the prototype of DVABatch with 5k lines of C++ codes as a runtime backend for Triton [9], a DNN serving system from Nvidia. As the latency of a DNN model/operator varies with DNN frameworks [14, 19, 20, 39, 52] or compilers, we use TensorRT [13] to provide SOTA operator performance. DVABatch relies on Triton to batch the accepted queries. However, Triton sends the batched queries to DVABatch in an asynchronous fashion, and DVABatch enables the multi-entry multi-exit scheme for it. We also modify the model loading logic to load multiple stages (each stage is a sub-model) for a single DNN service.

Table 1 lists the setups of the experiments. We perform all the experiments on a machine that equips an Nvidia Turing Titan RTX (Titan RTX) GPU. We use six representative image processing and natural language processing DNN models as the benchmarks. All models experience load diversity. Besides, *BertBase* and *BertLarge* show input diversity, *LinkNet* and *Unet* show operator diversity.

We compare DVABatch with two batching policies: the default scheduling policy with batch time window $T_{window} = 0$ (ZeroBatch for short), and one with an optimized T_{window} (DelayBatch for short). The optimized T_{window} of DelayBatch is tuned for supporting the max peak throughput [2]. In all experiments, the maximum allowed batch size bs_{max} is 64 and the QoS target is 200ms to support a high load. Current production DNN serving systems (e.g., Triton [9], Clipper [23], TFServing [51]) all use the above batch time window and batch-size-based batching mechanism [22].

The load used for evaluation is generated using the method in MLperf [55], and the arrival time pattern satisfies the Poisson distribution [55]. We obtain the performance of the benchmarks at low, medium, and high loads. For a benchmark, we use 1/4, 3/5, 9/10 of its peak throughput as low load, medium load, high load. The three load levels are obtained by feeding each benchmark with a stepping load [3] in Section 8.3. For BertBase and BertLarge, we use the workloads in GLUE [59] to simulate the sequence length distributions of real-world services. By default, the RTE workload of GLUE is used for evaluating BertBase and BertLarge. Other workloads are evaluated in Section 8.4.

8.2 Reducing Average Latency

Figure 14 shows the average latencies of all benchmarks with ZeroBatch, DelayBatch, and DVABatch at low, medium, high loads. DVABatch reduces the average latency of the benchmarks by 16.1%/39.0%/57.7% compared with ZeroBatch, 35.4%/47.3%/48.5% compared with DelayBatch on average at low, medium, and high loads, respectively. DVABatch reduces the average latency in all cases with the multi-entry multi-exit batching scheme. Meanwhile, DelayBatch shows lower average latency at high load compared with ZeroBatch, and ZeroBatch achieves lower latencies at low load. It is because DelayBatch has an optimized batch time window for peak throughput, and ZeroBatch does not introduce latency due to the batch time window at low load.

BertBase and BertLarge show both input diversity and load diversity. The latency reduction of DVABatch is comparatively high at all loads compared with the two baselines. This is because DVABatch perceives the input diversity, and splits the large batches into small batches to reduce the extra computation due to padding in all cases. DVABatch processes the small batches in the form of a software pipeline to accelerate the computation, which further reduces the latency.

LinkNet and Unet show both operator diversity and load diversity. At low load, DVABatch and ZeroBatch reduce latency due to the smaller batch window, compared with DelayBatch. At high load, DVABatch performs much better than ZeroBatch and DelayBatch. When the load is high, the batch received from the upper-level serving system has more queries. There is a higher opportunity that DVABatch can split a large batch into batches with the preferred batch size. Operators do not use a batch size larger than their preferred batch sizes. Some queries can exit the batching early with shorter latency.

VGG19 and ResNet152 only show load diversity. At low load, DVABatch achieves equivalent latency performance compared with ZeroBatch, and reduces the average latency

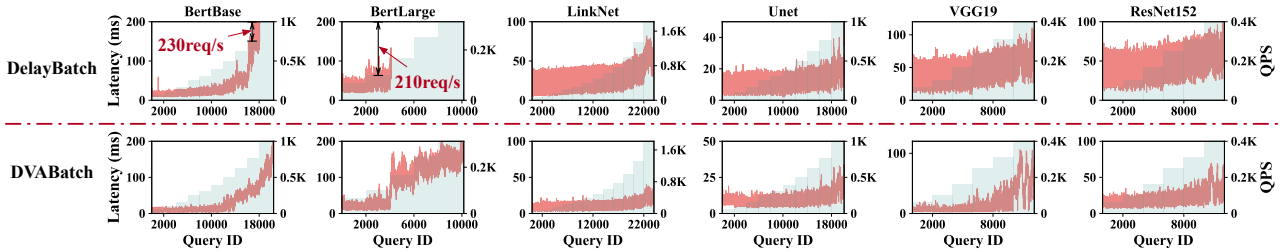

Figure 15: Latencies and peak throughput of DelayBatch and DVABatch fed with stepping load.

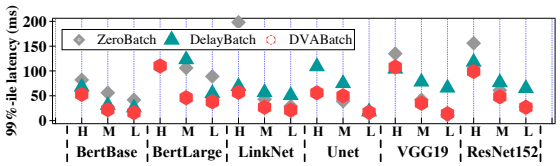

Figure 16: 99%-ile latencies of six DNN services at three loads with ZeroBatch, DelayBatch, and DVABatch.

compared with DelayBatch. This is reasonable because DVABatch and the baselines are all processing the queries with $bs = 1$ in this case. At high load, DVABatch and DelayBatch both perform better than ZeroBatch. The latency reduction benefits from the reasonable batching parameters. With DVABatch, early arrived queries are executed in advance, and latter queries are also considered to be merged with the former batch. Then, their response latencies are all reduced.

Comparison with Limited Solutions. While we do not compare DVABatch with BatchMaker [31] and LazyBatch [22] directly in this section, DVABatch outperforms them for the evaluated benchmarks. BatchMaker cannot handle input diversity for BertBase and BertLarge, as it is RNN-specific for input diversity and Bert-like models are not based on RNN cells. In this case, BatchMaker performs the same as the two baselines presented in Figure 14. LazyBatch only handles load diversity and performs per-operator scheduling. DVABatch degenerates to LazyBatch while evaluating VGG19 and ResNet152, if we slice the model into the granularity of operators and only enable *stretch* operation. However, experiments in Section 8.7 indicates this incurs severe performance degradation. Meanwhile, per-operator model slicing for LazyBatch cannot be implemented with TensorRT, as it is against the compilation techniques like kernel fusion. Compared with them, DVABatch achieves better performance for all evaluated benchmarks by promising a multi-entry multi-exit batch scheme with minimal runtime scheduling overhead.

Comparison of Tail Latency. Other than the average latency, Figure 16 shows the 99% latencies of all benchmarks. DVABatch reduces the 99%-ile latencies by 16.9%/27.4%/53.7% compared with ZeroBatch, and 45.2%/45.1%/29.2% compared with DelayBatch. In terms of tail latency, DelayBatch has better performance at high load, ZeroBatch performs better than DelayBatch at low load. The multi-entry multi-exit design allows DVABatch to maintain consistent low tail latency at varying loads.

8.3 Robustness at Stepping Load

In this experiment, we evaluate the robustness of DVABatch in handling dynamic loads. Similar to prior work [3, 40], we use stepping load to obtain the peak load supported by DVABatch. We only compare DVABatch with DelayBatch in the following section, as ZeroBatch always shows poor performance at high load.

The stepping load is generated as follows. At first, the load is low (66 queries per second), and we gradually increase the load for every 2000 queries. After 30,000 queries, the load is increased to 4000 queries per second (QPS). We use the corresponding highest load under the constrain, that the latency is shorter than the QoS target 200*ms*, as the peak throughput.

Figure 15 shows the latencies of the benchmarks at stepping loads. In each subfigure, the *x*-axis represents the query ID in the issuing order. The left *y*-axis represents the latency of each query, and the right *y*-axis represents the load.

As observed, all the benchmarks have lower latency with DVABatch than with DelayBatch in all cases. For BertBase and BertLarge, DVABatch improves the peak throughput, because it eliminates the computation wasted for the padded inputs. On average, DVABatch increases 46.81% peak throughput for BertBase, 1.37× peak throughput for BertLarge. For operator diversity and load diversity, DVABatch has not impact on the computation. In that case, the peak throughput of DVABatch is limited by the hardware capacity. DVABatch maintains the same peak throughput as DelayBatch.

8.4 Impact of Input Distributions

The effectiveness of DVABatch for input diversities is affected by how different the inputs are. In this experiment, we show the performance of DVABatch when different workloads in GLUE [59] are used as the inputs of Bert and BertLarge. We use the same stepping load in Section 8.3.

Figure 17 shows the supported peak loads of BertBase and BertLarge with different workloads in GLUE. As observed,

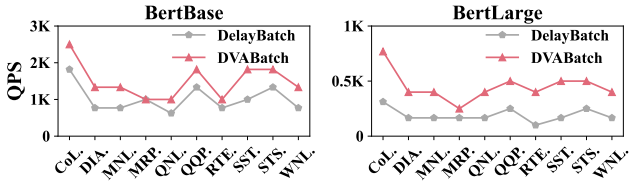

Figure 17: Peak load supported by DVABatch for BertBase/BertLarge with different workloads.

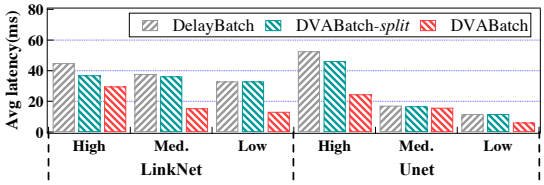

Figure 18: Comparison of *split* operation for LinkNet/Unet.

DVABatch improves the peak throughput by 46.8% for BertBase, 1.37× for BertLarge compared with DelayBatch on average. DVABatch brings different throughput improvements for different datasets. The more imbalanced the sequence distribution is, the higher the workload's input diversity is.

In general,DVABatch works better for workloads with higher input diversity, as DVABatch can eliminate more unnecessary padding. For instance, while the input diversity of SST-2 is higher than that of MRPC, the performance gap between DelayBatch and DVABatch for SST-2 workload is much larger than that for MRPC. The more imbalanced the sequence distribution is, the higher throughput improvement DVABatch achieves.

8.5 Effectiveness of *split* operation

In this experiment, we evaluate the effectiveness of *split* for load diversity. Figure 18 shows the average latencies of LinkNet and Unet with DVABatch, DVABatch-*split*, and DelayBatch. DVABatch-*split* is a variant of DVABatch that only *split* is enabled in DVABatch.

As shown, DVABatch-*split* reduces the latencies most at high load for LinkNet and Unet. Compared with DelayBatch, DVABatch-*split* reduces average latency by 15.0% at high load. This is because DVABatch-*split* rarely has the choice to split a batch at low and medium loads.

We can also find that DVABatch-*split* brings different improvements for the two DNN services. The difference originates from the operator diversity and load diversity. We look into the processing details and find that DVABatch-*split* generates larger batches for LinkNet than Unet. Half of the batches generated for LinkNet are with $bs > 50$ and Unet are with $bs > 30$. In this case, DVABatch-*split* has more chances to identify the preferred batch size for LinkNet than Unet.

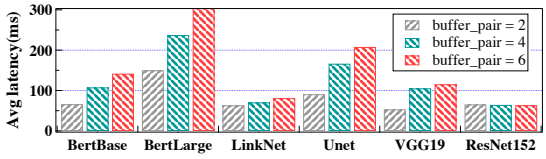

Figure 19: The average latencies of DVABatch with different number of buffer pairs under peak load.

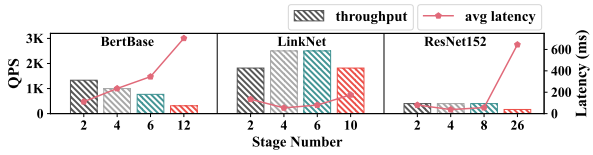

Figure 20: The average latencies and peak throughput of the benchmarks when sliced into different numbers of stages.

8.6 Impacts of the Number of Buffer Pairs

This experiment evaluates the impact of the number of buffer pairs used in DVABatch. Figure 19 presents the average latencies of DVABatch with different numbers of buffer pairs at the peak load. As shown, the average latencies are always the lowest with two buffer pairs for all benchmarks.

Therefore, two buffer pairs are already enough to preserve the execution validity and enable the work manner switch. More buffer pairs degrade the performance. Each buffer pair uses a set of CUDA [50] synchronization data structures to guarantee scheduling correctness. When two buffer pairs are used, DVABatch only needs to switch between them. However, managing many buffer pairs requires extra FIFO queues to transmit these data structures. Too many buffer pairs incur a high scheduling overhead.

8.7 Impacts of the Stage Numbers

In this experiment, we investigate the number of stages on the performance of DVABatch. We use BertBase, LinkNet, and ResNet152 as the representative benchmarks with the three types of diversities, respectively, due to the limited space.

Figure 20 shows the throughputs and average latencies of the benchmarks with different stage numbers. In the figure, the left *y*-axis represents the peak throughput and the right *y*-axis represents the corresponding average latencies.

As observed, the best stage number is 2 for BertBase, 4 for LinkNet and ResNet152 in terms of latency and throughput, respectively. For BertBase, the accepted batch is *split* into two batches in most cases. DVABatch avoids generating too many small batches to reduce scheduling overhead. Therefore, 2 stages are enough for BertBase. LinkNet and ResNet152 require more stages to enable *stretch* and *split*. If the number of stages is too large (e.g., 20), managing queues between stages incur a high overhead for all benchmarks.

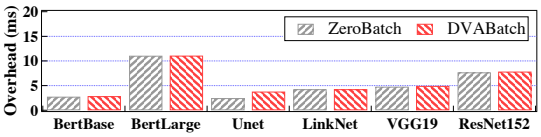

Figure 21: Close-loop latencies of ZeroBatch and DVABatch.

8.8 Scheduling Overhead

As mentioned in Section 7.1, the profiling for model slicing needs to be done for a single time on each type of target GPU, and completed in 10 minutes. To measure the runtime overhead introduced by DVABatch, we run the six benchmarks in a close loop, and compare the end-to-end latencies of the queries with ZeroBatch. In this experiment, we set the batch time window of DVABatch to 0. Figure 21 shows the experimental results.

As observed, the average latency overhead is 0.29ms, and DVABatch achieves almost the same latency compared with ZeroBatch. DVABatch has a low overhead because it does not interrupt the execution of stage executors at the extremely low load, and the two executors overlap the overhead of CPU-GPU synchronization for each stage. As depicted in Figure 12, *Line*5 is usually an asynchronous operation on GPU and the scheduling operation in *Line*4 is overlapped by the execution. Moreover, the model slicing in DVABatch does not invalidate the optimization of DNN compilers.

DVABatch needs extra global memory (buffer pairs) to avoid read-write hazards while maintaining the software pipeline, which takes 200 MB of space on average.

9 Discussion

9.1 Implication for Future DNN Inference

Omnipresent Diversity. Readers can find that all the mentioned diversities are caused by dynamic attributes (dynamic input, operator, load). Existing DNNs may have a dynamic architecture in depth, width, and routing [32,33,45,53,58,60]. As more dynamic attributes emerge, diversity spreads across new DNNs.

Intra-model Scheduling. DVABatch performs fine-grained scheduling within the DNN models. As DNNs grow larger and show more diversity, the execution of DNNs cannot be treated as a single function call. Large models [15,54] like Bert are being deployed on multiple machines. MoE [42] models activate different paths for different inputs. Intra-model scheduling is a trend for future DNN inference.

9.2 Flexibility

DVABatch is flexible to other diversities. For instance., in the services with early-exiting and layer-skip[42,53,55] models, the inference returns at early stages or skips some stages when the intermediate results satisfy a predefined threshold. Users can modify the user-defined condition in DVABatch to check the intermediate results during batch inference. If some queries meet the predefined threshold, users then utilize the *split* operation for them to exit the ongoing batch without executing the rest layers. Then, the layer skip mechanism is implemented by the holistic design of multi-entry and multi-exit in DVABatch.

9.3 Limitations

DVABatch targets on efficient batch processing of models with diversities. It performs the same as the traditional batch policy for the models without any diversity. The CV models commonly crop the images to the same size before processing. Then input diversity does not exist in these models. Because the same blocks share the favored batch size, models with repetitive blocks like ResNet-50 and Bert do not show operator diversity. Load diversity also vanishes when the queries arrive with a uniform load. In these cases, DVABatch has no opportunity to achieve a better batch scheme. But as stated in Section 9.1, more and more model are showing diversities due to dynamic attributes. As long as a more efficient batch scheme is available for the diversities, DVABatch takes effect through its holistic design, even on platforms like CPUs [61].

10 Conclusion

In this work, we utilize the multi-entry multi-exit scheme to resolve the long latency problem due to serving diversity in existing DNN serving systems. We dig out the root inefficiency of the existing batching policy when facing serving diversities. Therefore, we propose DVABatch runtime batching system. Firstly, DVABatch divides the DNN models into stages and abstracts three meta operations to support the multi-entry multi-exit scheme. Secondly, DVABatch introduces a state transition diagram to manage the execution of stages. And then, DVABatch conducts diversity-aware batch scheduling with the meta operations for the incoming batch of queries. Overall, DVABatch achieves 46.4% average latency reduction and up to 2.12× throughput improvement for involved diversities, compared with state-of-art solutions.

Acknowledgment

This work is partially sponsored by the National Natural Science Foundation of China (62022057, 61832006, 61872240, 62172375), Shanghai international science and technology collaboration project (No.21510713600), and Open Research Projects of Zhejiang Lab (No. 2021KE0AB02). Quan Chen,and Deze Zeng are the corresponding authors.

References

[1] https://aws.amazon.com/machine-learning/.

[2] Delayed batching in triton. https://github.com/triton-inference-server/server/blob/v2.14.0/docs/model_configuration.md#delayed-batching.

[3] Edit load patterns to model virtual user activities. https://docs.microsoft.com/en-us/visualstudio/test/edit-load-patterns-to-model-virtual-user-activities?view=vs-2022.

[4] Effective transformer. https://github.com/bytedance/effective_transformer.

[5] Google translate. https://translate.google.com.

[6] Microsoft xiaoice. http://www.msxiaobing.com/.

[7] Nvidia nsight compute. https://docs.nvidia.com/nsight-compute/NsightCompute/index.html.

[8] Nvidia nsight system. https://developer.nvidia.com/nsight-systems.

[9] Nvidia triton inference server. hhttps://github.com/NVIDIA/triton-inference-server.

[10] Nvidia turing gpu architecture whitepaper. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[11] Nvidia v100 tensor core gpu. https://www.nvidia.com/en-us/data-center/v100/.

[12] Siri. https://www.apple.com/siri/.

[13] Tensorrt. https://developer.nvidia.com/tensorrt, 2021.

[14] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[15] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[16] Abhishek Chaurasia and Eugenio Culurciello. Linknet: Exploiting encoder representations for efficient semantic segmentation. In *2017 IEEE Visual Communications and Image Processing (VCIP)*, pages 1–4. IEEE, 2017.

[17] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In *Proceedings of the 22th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 17–32, 2017.

[18] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. *ACM SIGPLAN Notices*, 51(4):681–696, 2016.

[19] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[20] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[21] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE, 2014.

[22] Yujeong Choi, Yunseong Kim, and Minsoo Rhu. Lazybatching: An sla-aware batching system for cloud machine learning inference. *arXiv preprint arXiv:2010.13103*, 2020.

[23] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, 2017.

[24] Weihao Cui, Quan Chen, Han Zhao, Mengze Wei, Xiaoxin Tang, and Minyi Guo. E^2bird: Enhanced elastic batch for improving responsiveness and throughput of deep learning services. *IEEE Trans. Parallel Distributed Syst.*, 32(6):1307–1321, 2021.

[25] Weihao Cui, Mengze Wei, Quan Chen, Xiaoxin Tang, Jingwen Leng, Li Li, and Mingyi Guo. Ebird: Elastic batch for improving responsiveness and throughput of deep learning services. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pages 497–505. IEEE, 2019.

[26] Weihao Cui, Han Zhao, Quan Chen, Ningxin Zheng, Jingwen Leng, Jieru Zhao, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. Enable simultaneous DNN services based on deterministic operator overlap and precise latency prediction. In *SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, 2021*, pages 15:1–15:15. ACM, 2021.

[27] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[28] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbotransformers: an efficient gpu serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 389–402, 2021.

[29] Pratik Fegade, Tianqi Chen, Phillip B Gibbons, and Todd C Mowry. The cora tensor compiler: Compilation for ragged tensors with minimal padding. *arXiv preprint arXiv:2110.10221*, 2021.

[30] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[31] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[32] Yue Guan, Zhengyi Li, Jingwen Leng, Zhouhan Lin, and Minyi Guo. Transkimmer: Transformer learns to layer-wise skim. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022*, pages 7275–7286. Association for Computational Linguistics, 2022.

[33] Cong Guo, Yuxian Qiu, Jingwen Leng, Xiaotian Gao, Chen Zhang, Yunxin Liu, Fan Yang, Yuhao Zhu, and Minyi Guo. SQuant: On-the-fly data-free quantization via diagonal hessian approximation. In *International Conference on Learning Representations*, 2022.

[34] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S Lee, David Brooks, and Carole-Jean Wu. Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 982–995. IEEE, 2020.

[35] Johann Hauswald, Yiping Kang, Michael A Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G Dreslinski, Jason Mars, and Lingjia Tang. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 27–40. IEEE, 2015.

[36] Johann Hauswald, Michael A Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, et al. 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*, pages 223–238, 2015.

[37] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[38] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[39] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678, 2014.

[40] Zhen Ming Jiang and Ahmed E Hassan. A survey on load testing of large-scale software systems. *IEEE Transactions on Software Engineering*, 41(11):1091–1118, 2015.

[41] Harshad Kasture and Daniel Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2016.

[42] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[43] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 661–670, 2014.

[44] Xiaqing Li, Guangyan Zhang, H Howie Huang, Zhufan Wang, and Weimin Zheng. Performance analysis of gpu-based convolutional neural networks. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 67–76. IEEE, 2016.

[45] Lanlan Liu and Jia Deng. Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[46] Zihan Liu, Jingwen Leng, Zhihui Zhang, Quan Chen, Chao Li, and Minyi Guo. VELTAIR: towards high-performance multi-tenant deep learning services via adaptive compilation and scheduling. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2022*, pages 388–401. ACM, 2022.

[47] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181*, 2017.

[48] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019*, pages 1–15. ACM, 2019.

[49] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, et al. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017.

[50] CUDA Nvidia. Nvidia cuda c programming guide. *Nvidia Corporation*, 120(18):8, 2011.

[51] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139*, 2017.

[52] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[53] Yuxian Qiu, Jingwen Leng, Cong Guo, Quan Chen, Chao Li, Minyi Guo, and Yuhao Zhu. Adversarial defense through network profiling based path extraction. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019*, pages 4777–4786. Computer Vision Foundation / IEEE, 2019.

[54] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*, 2019.

[55] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 446–459. IEEE, 2020.

[56] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[57] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[58] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. Branchynet: Fast inference via early exiting from deep neural networks. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 2464–2469. IEEE, 2016.

[59] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.

[60] Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E Gonzalez. Skipnet: Learning dynamic routing in convolutional networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 409–424, 2018.

[61] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. Deepcpu: Serving rnn-based deep learning models 10x faster. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 951–965, 2018.

[62] Wei Zhang, Quan Chen, Ningxin Zheng, Weihao Cui, Kaihua Fu, and Minyi Guo. Toward qos-awareness and

improved utilization of spatial multitasking gpus. *IEEE Trans. Computers*, 71(4):866–879, 2022.

[63] Wei Zhang, Weihao Cui, Kaihua Fu, Quan Chen, Daniel Edward Mawhirter, Bo Wu, Chao Li, and Minyi Guo. Laius: Towards latency awareness and improved utilization of spatial multitasking accelerators in datacenters. In *Proceedings of the ACM International Conference on Supercomputing, ICS 2019*, pages 58–68. ACM, 2019.

[64] Han Zhao, Weihao Cui, Quan Chen, Youtao Zhang, Yanchao Lu, Chao Li, Jingwen Leng, and Minyi Guo. Tacker: Tensor-cuda core kernel fusion for improving the GPU utilization while ensuring qos. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2022*, pages 800–813. IEEE, 2022.

[65] Han Zhao, Weihao Cui, Quan Chen, Jieru Zhao, Jingwen Leng, and Minyi Guo. Exploiting intra-sm parallelism in gpus via persistent and elastic blocks. In *39th IEEE International Conference on Computer Design, ICCD 2021*, pages 290–298. IEEE, 2021.

A Artifact Appendix

Abstract

This artifact provides a prototype of DVABatch implemented as a runtime backend for Triton Inference Server

Scope

This artifact is licensed with Apache 2.0

Hosting

The code is available on Github https://github.com/sjtu-epcc/DVABatch.git.

Requirements

Hardware Requirements.

1. *CPU: Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz*
2. *Memroy: 252G*
3. *NVIDIA TitanRTX*

Software Requirements.

1. *Ubuntu 18.04.6 (Kernel 4.15.0)*
2. *GPU Driver: 460.39*
3. *CUDA 11.3*
4. *CUDNN 8.2*
5. *TensorRT 8.0.3.4*
6. *RapidJSON*
7. *Cmake 3.17*