BenchCouncil Transactions on Benchmarks, Standards and Evaluations 3 (2023) 100151

Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

journal homepage: www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/

Full length article

Characterizing and understanding deep neural network batching systems on GPUs

Feng Yu [a,b], Hao Zhang [c], Ao Chen [a,b], Xueying Wang [d], Xiaoxia Liang [e], Sheng Wang [c], Guangli Li [a,b,f,*], Huimin Cui [a,b], Xiaobing Feng [a,b]

[a] State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, China
[b] University of Chinese Academy of Sciences, China
[c] China Mobile Research Institute, China
[d] Beijing University of Posts and Telecommunications, China
[e] Intel Corporation, China
[f] UNSW Sydney, Australia

ARTICLE INFO

Keywords:
Deep learning systems
Dynamic batching
Neural networks
Performance characterization

ABSTRACT

As neural network inference demands are ever-increasing in intelligent applications, the performance optimization of model serving becomes a challenging problem. Dynamic batching is an important feature of contemporary deep learning serving systems, which combines multiple requests of model inference and executes them together to improve the system's throughput. However, the behavior characteristics of each part in deep neural network batching systems as well as their performance impact on different model structures are still unknown. In this paper, we characterize the batching system by leveraging three representative deep neural networks on GPUs, performing a systematic analysis of the performance effects from the request batching module, model slicing module, and stage reorchestrating module. Based on experimental results, several insights and recommendations are offered to facilitate the system design and optimization for deep learning serving.

1. Introduction

As the demand for deep learning algorithms based on deep neural networks (DNNs) continues to increase, serving systems [1–3], which provide DNN training and inference as services to users on computing platforms, are sparking interest in both academia and industry. Given the user's real-time response desire, achieving low-latency inference becomes a fundamental prerequisite in these serving systems. To effectively handle model inference requests, dynamic batching plays a crucial role in existing serving systems for improving the system throughput by leveraging parallelism and locality between batched inputs. Unlike training, where all training inputs are available before training starts, inference presents a different challenge as input arrives at the serving system over time, and its arrival rate depends on the popularity of the deployed models. Therefore, inference batching must carefully balance the trade-off between latency and throughput. For instance, larger batch sizes may improve throughput but introduce longer waits for the scheduler to accumulate a sufficiently large input batch and thus increase latency, whereas smaller batch sizes may reduce latency but at the cost of lower throughput.

Traditional deep learning serving systems represented by Triton [1] and TensorFlow-Serving [2] relied on configuring the model-allowed maximum batch size (MAX-BS), which limits the input that can be batched, and the batching time window (TW), which indicates the longest wait time for inputs for combining a batch, as hyper-parameters. Unfortunately, these statically configured serving systems lack the flexibility to dynamically adjust server traffic to accommodate varying loads, leading to sub-optimal performance. For instance, during periods of low-load inference request traffic, employing a large time window results in over-provisioning, as queued requests within the window increase the average response time. Conversely, in server congestion scenarios, larger batch time windows and batch sizes may prove advantageous. Traditional serving systems lack the capability of interrupting ongoing batches to serve new arriving requests. Recently, multi-entry multi-exit batching systems, e.g., DVABatch [3], have arisen, which adopt sub-graphs as the scheduling granularity and introduce several meta-operations to improve the system throughput.

* Correspondence to: Institute of Computing Technology, Chinese Academy of Sciences, 100190 Beijing, China.
E-mail addresses: yufeng@ict.ac.cn (F. Yu), zhanghao@chinamobile.com (H. Zhang), chenao23s@ict.ac.cn (A. Chen), wangxueying@bupt.edu.cn (X. Wang), xiaoxia.liang@intel.com (X. Liang), wangshengyjy@chinamobile.com (S. Wang), liguangli@ict.ac.cn (G. Li), cuihm@ict.ac.cn (H. Cui), fxb@ict.ac.cn (X. Feng).

https://doi.org/10.1016/j.tbench.2024.100151
Received 2 November 2023; Received in revised form 3 January 2024; Accepted 6 January 2024
Available online 13 January 2024

DNN serving systems are intrinsically intricate, influenced by numerous factors encompassing neural network models, load levels, and model slicing patterns, among others. However, existing studies predominantly focus on a localized perspective of batching systems, without providing a comprehensive characterization and understanding of batch behavior. As such, this paper aims to reveal the intricacies of batching behavior in DNN serving systems, offering valuable insights into resource management and system design, particularly concerning typical neural network models and workloads encountered by service providers. Meanwhile, we underscore the limitations of existing serving system batching techniques while presenting innovative optimization avenues for serving system developers.

To characterize the batch behavior within DNN serving systems, we perform a comprehensive systematic evaluation on a GPU platform. We conducted experimental evaluations using representative DNN models from three different domains, including ResNet [4] for image classification, BERT [5] for natural language processing (NLP), and LinkNet [6] for image segmentation. As described in Section 3.2, ResNet has low utilization of computing resources, BERT can saturate system resources even with small batches, and LinkNet is memory-bounded. Leveraging these three representative models, we conduct an in-depth investigation into the behaviors of the request batching, model slicing, and stage reorchestrating within batched serving systems. Regarding the request batching module, we initially examine the impact of batch size on system throughput and request average latency (Section 4.1), followed by a comprehensive exploration of hyperparameters, specifically the MAX-BS and TW, and their relationship with system throughput (Sections 4.2 and 4.3). For the model slicing module, we discuss the influence of slicing positions and the number of stages on system throughput (Sections 5.1 and 5.2), respectively. For the stage reorchestrating module, we analyze the correlation between reorchestrating strategies and system throughput across varying workloads and network models (Section 6.1). Subsequently, we conduct a comprehensive analysis of meta-operations in multi-entry multi-exit systems, including split and stretch operations (Sections 6.2 and 6.3). Based on observations, we present potential application scenarios, along with insights for various research directions (Section 7). Our contribution can be summarized as follows:

- We perform a comprehensive analysis of batching behavior within deep learning serving systems on GPUs by leveraging three representative neural network models from different application scenarios.
- We characterize the effects of batch sizes and hyperparameters on the behavior of the request batching module, explore different slicing patterns associated with batching within the model slicing module and analyze the influence of stage reorchestrating strategies and meta-operations on the behavior of the reorchestrating module.
- Based on experimental studies, we provide several insights and recommendations to facilitate the system design and optimization for deep learning serving. We hope that these observations could pave the road for developing high-efficiency deep neural network batching systems.

2. DNN batching serving systems

2.1. Meta-operations

In traditional DNN serving systems, such as Triton, the batch size remains constant until the inference is completed, as depicted in the upper part of Fig. 1. In such a design, the next batch can only be launched for execution after the ongoing batch inference is completed, and the requests in the batch cannot be exited early, resulting in longer response latency [7]. To support requests being able to exit or join the serving system, DVABatch abstract the two actions of request exit and join into meta-operations, namely split and stretch operations. Fig. 1 shows how meta-operations can reduce average latency. To simplify the explanation, we assume that each operator completes in 1 time unit (T) and the MAX-BS is 4. In this case, once 4 requests are received or the batching time window ends, the received requests will be batched and issued for execution.

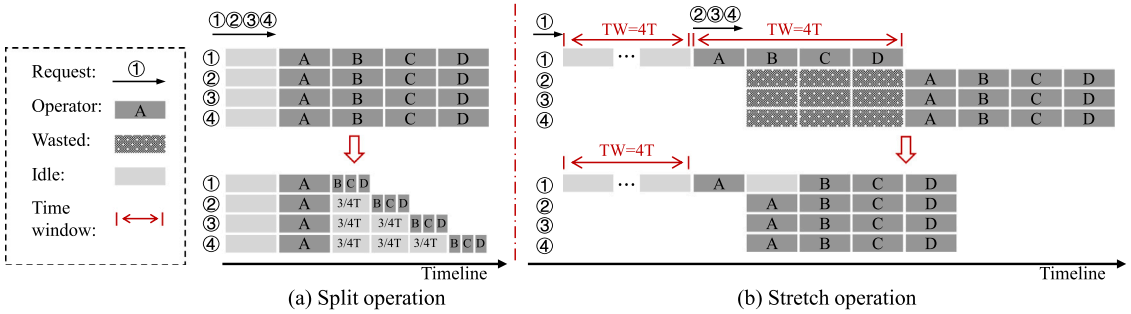
Through the split operation, a large ongoing batch is split into several smaller batches for individual processing, which makes it easier for some queries in the batch to exit early. Fig. 2 shows the execution time of two convolution operations in Resnet under different batch sizes. Convolution operations dominate DNNs (accounting for 86% of the computation time) [8]. As shown in the figure, the preferred batch sizes of Convolution-A and Convolution-B are 4 and 1, respectively. For Convolution-A, using a batch size smaller than 4 cannot fully utilize the GPU (the processing time starts to increase only when the batch size is greater than 4). For Convolution-B, batching will only increase its execution time without improving processing throughput. Fig. 1(a) shows how the split operation can reduce average latency, where operator A has a preferred batch size of 4, operators B, C, and D have a preferred batch size of 1, and the received requests have been batched and are ready to be issued for execution. In Triton, (upper half of Fig. 1(a)), the requests in the batch start processing at the same time and end at the same time. The lower half of Fig. 1(a) shows the split operation, i.e., operator A executes the full batch, then splits the batch into four smaller batches with a batch size of 1 at operator B, and executes these small batches in sequence. In this way, Requests ①, ②, and ③ can exit earlier. The average latency can be reduced by 28.1% (from 4 T to 2.875 T).

Through the stretch operation, new incoming queries are added to the ongoing batch to form a larger batch, thereby utilizing the hardware computing power. Fig. 1(b) shows how the stretch operation can reduce average latency, where the batching time window is 4 T and the operator preferred batch size is 4. In Triton (i.e., the upper half of Fig. 1(b)), Request ① starts running individually after waiting for a time window, leaving the GPU underutilized. During the processing of Request ①, Requests ②, ③, and ④ arrive, but they must wait to be executed in the next batch. The lower half of Fig. 1(b) shows the stretch operation, where the first batch (containing only Request ①) waits for the second batch after completing operator A, and then the two insufficient batches are merged into a new large batch to fully utilize the hardware. In this way, the average latency can be reduced by 34.4% (from 8 T to 5.25 T).

2.2. Major components

In this section, we introduce three major components of contemporary DNN batching serving systems, including a request batching module, a model slicing module, and a stage reorchestrating module. These three components are ubiquitously present in DNN batching serving systems, such as Triton [1], DVABatch [3], Ebird [9], and LazyBatching [7], among others.

Request Batching Module (RBM). The serving system initiates by placing end-users' requests into a request queue. The RBM subsequently organizes these requests into batches, based on two hyperparameters: MAX-BS and TW. These formed batches are placed in a batch queue for the request processing module to utilize. Fig. 3 illustrates the behavior of different configurations of RBM under the circumstance where request R_i enters the request queue at time t_i. RBM with configuration 0 efficiently aggregates two requests within a specified time window to form a batch. Conversely, RBM with configuration 1 can accumulate three requests during the same time window, resulting in a batch of size equal to MAX-BS. However, RBM with configuration 2, although also capable of collecting three requests within the designated time window, is constrained by MAX-BS, leading to the creation of a reduced-size batch of 2.



(a) Split operation (b) Stretch operation

Fig. 1. Illustration of how meta-operations address the long latency problem of user requests. Split operation enables requests to exit early when encountering operators with high parallelism. Stretch operation enables the merging of multiple insufficient batches to reduce waiting time and fully utilize hardware.

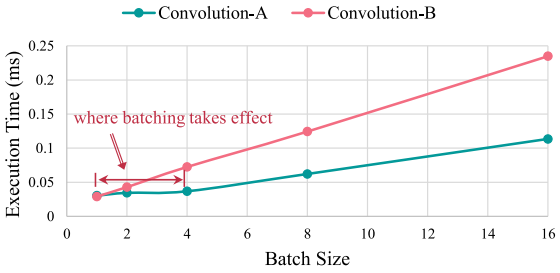


Fig. 2. Execution time of two convolution operators from ResNet with different batch sizes on A100.

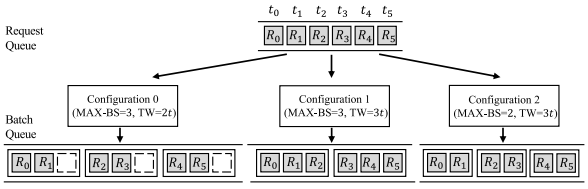


Fig. 3. The behavior of the request batching module under various parameter configurations, namely, the maximum allowed batch size (MAX-BS) and the time window (TW).

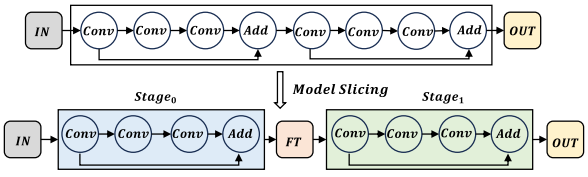


Fig. 4. Diagram of model slicing, where IN/OUT are input/output tensors, and FT are feature tensors.

Model Slicing Module (MSM). To support interruptible batch execution, the serving system needs to slice the models during deployment, which includes determining the slice positions and the number of stages formed after slicing [3]. Fig. 4 provides an example of graph slicing, where slicing occurs after the first Add operation and only once, resulting in two stages with identical graph structures. Batching serving systems frequently employ stage performance models to guide meta-operation decisions, making model slicing critical for system throughput due to its direct influence on stage determination.

Stage Reorchestrating Module (SRM). The stage reorchestrating module typically employs a reorchestrating strategy involving split and stretch operations to control batch and stage execution. The split operation is employed to split large batches into multiple smaller sub-batches, enabling the early completion of smaller sub-batches without waiting for the entire large batch, thus reducing the average request

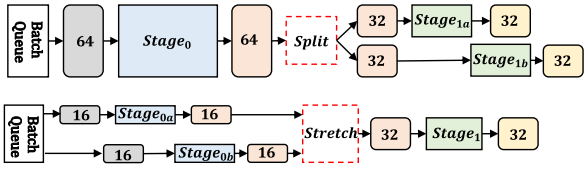


Fig. 5. Diagram of split and stretch operations, where the numbers inside the rounded rectangles represent batch sizes.

latency. As illustrated in Fig. 5, the split operation divides a batch of size 64 from the $Stage_0$ output into two sub-batches of size 32 each, which are then processed sequentially by two stage instances ($Stage_{1a}$ and $Stage_{1b}$). On the other hand, the stretch operation is used to merge multiple small sub-batches into a larger batch, harnessing hardware parallelism to enhance throughput. As shown in Fig. 5, when a new batch arrives, the current batch is undergoing inference in $Stage_{0a}$. Once the current batch completes the inference in $Stage_{0a}$, SRM passes the new batch to $Stage_{0b}$ for processing. Subsequently, the stretch operation increases the batch size from 16 to a larger batch of size 32, combining the outputs from these two stage instances for $Stage_1$ inference.

3. Experimental setup

3.1. Hardware and software setting

Table 1 lists the setups of the experiments. In this paper, we characterize and analyze the dynamic batching with two serving systems, Triton Inference Server (version 22.05) [1] and DVABatch (main branch) [3], on a high-performance platform that integrates Intel Xeon CPUs and an NVIDIA A100 GPU. As the latency of a DNN model/operator varies with DNN frameworks or compilers [10–12], we employ TensorRT (version 8.2.3) [13] as the inference engine for both of these serving systems to provide SOTA operator performance. Additionally, We use the NVIDIA Triton client [14], which employs an approach similar to MLPerf [15] for generating workloads with arrival times that conform to a uniform distribution. The client uses the HTTP protocol to send requests and sets the QoS target to 200 ms. Regarding DVABatch, we set request rates corresponding to 1/4, 3/5, and 9/10 of the peak throughput as low, medium, and high loads. For ease of experimentation, we align the request rate with the number of client threads, which is 64. Leveraging NVIDIA's Model Analyzer tool [16], we ascertain the maximum throughput attainable by the serving system for specific models. Specifically, the Model Analyzer indicates peak throughputs for ResNet, BERT, and LinkNet as 4288, 1088, and 3264 in DVABatch, respectively.

Table 1
Evaluation specifications.

Hardware		CPU: Intel Xeon Gold 6248 GPU: NVIDIA A100
OS & Driver		Ubuntu: 18.04.2 (kernel 5.4.0-72) GPU Driver: 515.43.04
Software	Client	NVIDIA Triton Client: v22.05
	Server	NVIDIA Triton inference server: v22.05 DVABatch: main branch
	Inference engine	TensorRT: v8.2.3

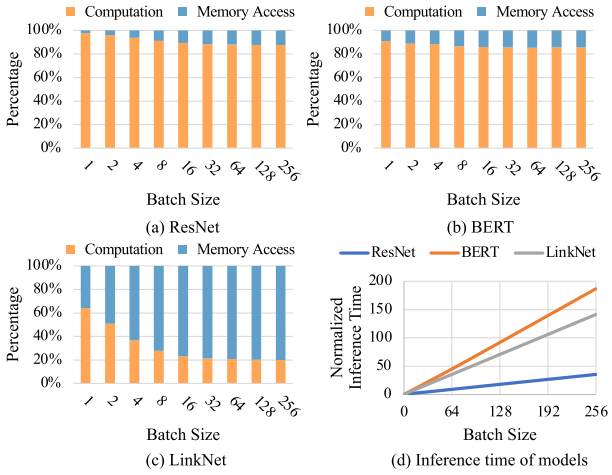


Fig. 6. Performance of three benchmarked models.

3.2. Benchmarked deep neural networks

Incumbent Internet giants have been offering services for tasks such as image classification, natural language processing, and image segmentation, exemplified by Google Cloud Vision AI [17,18], Microsoft Azure Text Analytics [19,20], and Amazon Rekognition [21,22]. This study focuses on these AI domains, employing benchmark network architectures: ResNet [4], BERT [5], and LinkNet [6] for experimental evaluation. Specifically, the study utilizes Torchvision's resnet152 [23], HuggingFace's bert-base-uncased [24], and LinkNet from Purdue University's e-Lab project.

Fig. 6 visually illustrates the end-to-end inference latency of these neural network models for different batch sizes, while also presenting a detailed breakdown of time allocation for computation and memory access. We observe that for the ResNet and BERT models, computation time takes the lead (see Fig. 6(a) and (b)), whereas in the LinkNet model, memory access time predominates (see Fig. 6(c)). Furthermore, Fig. 6(d) illustrates the relationship between inference time and batch size. It is evident from the figure that as the batch size increases, the time of the ResNet model increases relatively slowly, whereas the time of the BERT model experiences a sharp rise. In other words, under small batch sizes, ResNet exhibits lower resource utilization, while BERT potentially leads to system resource saturation.

In summary, ResNet and BERT primarily emphasize computational resources, with ResNet demonstrating efficient resource utilization under small batch sizes, while BERT's resource demands quickly saturate the system. In contrast, LinkNet places a stronger focus on memory access, making it more memory-bound compared to the other models.

3.3. Evaluation metrics

The evaluation metrics include:

- Latency, defined as the average time taken by the serving system to process a query, encompassing both the waiting time and the inference time for the query.
- Throughput, defined as the average number of queries processed by the system per second.
- Inference time, defined as the time required for a DNN model to perform inferences on input data. Unlike request latency, inference time does not encompass the waiting time associated with the request.

Since batching is a technique employed to enhance the throughput of the serving system, in this paper, we will use "system performance" interchangeably with system throughput.

4. Analysis of request batching

4.1. Performance of different batch sizes

Popular DNN serving systems such as Triton support batch execution of multiple requests. In this experiment, as the serving system receives batched inputs that are already formed, it does not wait for them to be collected; thus, we set the time window to 0. Fig. 7 presents the throughput of batching for three typical networks across various batch sizes. Additionally, we demonstrate the benefits of batching in reducing request latency, indicated by the blue line in the corresponding figure.

Finding 1. *In general, the system's throughput can be enhanced by increasing the batch size while meeting QoS requirements.* Observing Fig. 7, it becomes apparent that as the batch size increases, effective throughput rapidly rises, amortizing the inference cost and significantly reducing the request latency. This phenomenon occurs because larger batch sizes increase the computational workload required for inference, allowing better saturation of the GPU's computational resources, thereby achieving higher throughput.

Finding 2. *Enlarging the batch size does not always lead to an improvement for the system throughput.* Once a specific threshold for batch size is exceeded, GPU resources are fully utilized, and further increasing the batch size may lead to request latency exceeding users' expected response time, without yielding additional enhancements in throughput.

4.2. Effects of max batch size settings

We evaluate Triton's performance across various workloads by running three typical neural networks under different MAX-BS configurations. In this experiment, for ResNet, BERT, and LinkNet, the time windows are set to 500 μs, 10 μs, and 10 μs, respectively, aligning with the observations presented in Section 4.3. Our corresponding results are presented in Tables 2, 3, and 4. In these tables, "Collected-BS" represents the batch size formed by the batcher, and "Latency" denotes the average request latency (in milliseconds).

Finding 3. *For ResNet and BERT models, enlarging the MAX-BS parameter has the potential to improve the system throughput.* We observe that as MAX-BS gradually increases, the batch size formed by the RBM also increases correspondingly. For both ResNet and BERT models, Triton's throughput steadily increases with the increasing values of MAX-BS, eventually plateauing, regardless of the workload. In situations where MAX-BS is configured with a smaller value, such as 1, it may lead to a substantial number of requests being blocked in the queue. This occurrence stems from Triton's operational design, where a new batch will initiate execution only upon the completion of the preceding batch. To mitigate this, we can increase MAX-BS to maximize the batch size per execution, thereby reducing the average wait time for requests. However, as MAX-BS increases to a certain extent, although the batch scheduler can form larger batches to amortize inference overhead, it also leads to longer waiting times for requests in the queue.

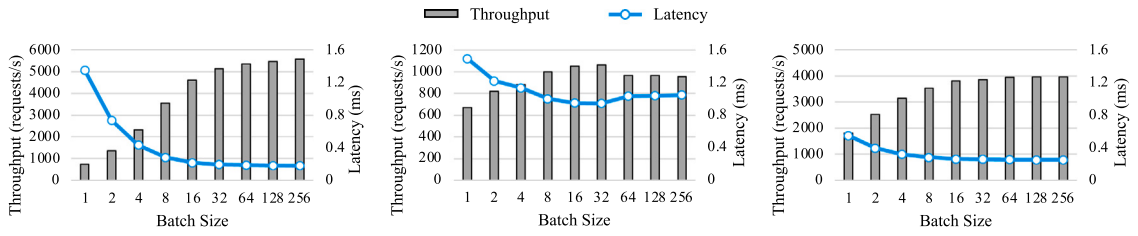


Fig. 7. Effect of batching on throughput and latency of batched execution as a function of batch size. For this experiment, we assume that the batched inputs are already formed, without waiting for them to be collected.

Table 2
Performance of Triton with varying MAX-BS for ResNet model across three workloads.

MAX-BS	Low load			Medium load			High load		
	Throughput	Latency	Collected-BS	Throughput	Latency	Collected-BS	Throughput	Latency	Collected-BS
1	211.40	406.73	1	212.76	410.08	1	212.37	411.79	1
2	414.65	214.52	2	415.43	218.64	2	413.89	220.13	2
4	798.72	109.98	4	810.27	114.45	4	801.31	116.01	4
8	1082.94	8.33	6	1475.22	63.29	8	1476.97	63.67	8
16	1083.26	8.32	6	2356.09	38.30	16	2485.64	37.93	16
32	1082.79	8.34	6	2604.12	13.35	22	3458.38	27.19	32
64	1083.20	8.38	6	2606.72	13.63	22	3463.90	27.28	47

Table 3
Performance of Triton with varying MAX-BS for the BERT model across three workloads.

MAX-BS	Low load			Medium load			High load		
	Throughput	Latency	Collected-BS	Throughput	Latency	Collected-BS	Throughput	Latency	Collected-BS
1	319.33	2.83	1	664.29	64.48	1	663.88	91.08	1
2	319.37	2.95	1	702.70	4.17	1	797.73	74.48	2
4	319.33	2.83	1	700.26	7.56	2	889.94	65.50	4
8	319.32	2.75	1	700.13	13.19	4	1002.18	37.82	8
16	319.30	2.93	1	696.00	22.00	3	1000.02	50.39	13
32	319.32	2.83	1	695.00	22.00	3	989.57	53.73	18
64	319.32	2.84	1	697.06	22.41	5	1003.20	48.88	23

Table 4
Performance of Triton with varying MAX-BS for the LinkNet model across three workloads.

MAX-BS	Low load			Medium load			High load		
	Throughput	Latency	Collected-BS	Throughput	Latency	Collected-BS	Throughput	Latency	Collected-BS
1	830.63	1.37	1	1977.77	1.80	1	2044.28	30.63	1
2	830.34	1.47	1	1976.69	2.52	2	2453.50	25.37	2
4	830.27	1.45	1	1975.77	4.15	3	2541.31	24.58	4
8	830.18	1.55	1	1974.40	8.74	5	2570.41	24.44	8
16	830.31	1.61	1	1967.48	16.67	10	2195.27	28.68	13
32	830.31	1.75	1	1964.68	24.64	16	2006.76	31.41	21
64	829.64	2.26	1	1966.96	23.27	16	1947.94	32.37	20

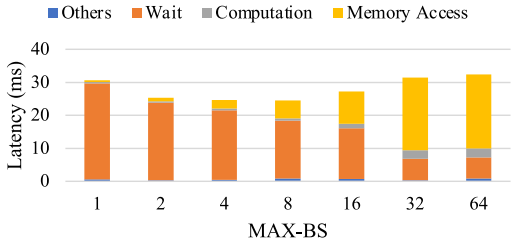


Fig. 8. Performance breakdown with different MAX-BS for the LinkNet model.

Finding 4. *MAX-BS mainly influences the queue wait time, the data transmission time, and the computation time.* For the LinkNet model, Triton exhibits similar behavior to ResNet and BERT models under medium to low workloads. However, under high workloads, Triton's observed throughput initially increases but then decreases as MAX-BS values continue to grow. To further analyze this behavior, we provide a decomposition graph of request latency for different MAX-BS values, as shown in Fig. 8. Fig. 8 shows that as the batch size gradually increases, the waiting time decreases, but memory access time increases due to the growing data volume. In Triton, excessive batch sizes cause a significant increase in request latency, as the memory access time for a request equals that of the entire batch.

4.3. Effects of batching time window

Fig. 9 depicts the influence of time windows on system throughput. In this experiment, for ResNet, BERT, and LinkNet, we configure MAX-BS as 64, 16, and 8, respectively, based on the observations in Section 4.2. The x-axis represents the request rate (the number of client requests sent per second), while the y-axis signifies the system throughput. In addition, the positions of symbols L, M, and H correspond to low, medium, and high rates, respectively.

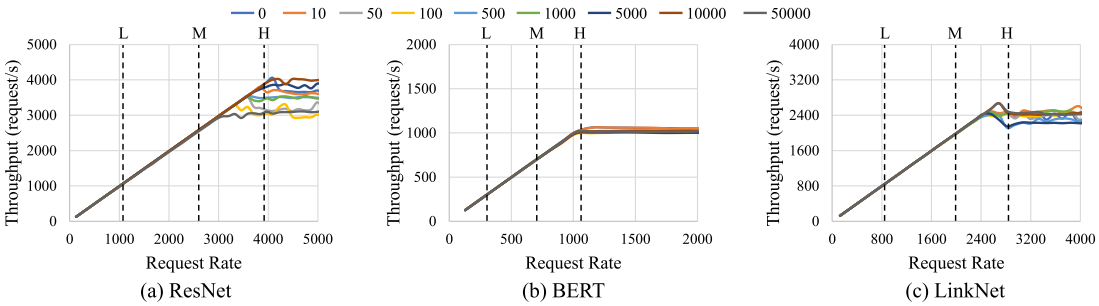


Fig. 9. Effects of time windows on Triton throughput with different request rates. At low and medium loads, the impact of the time window on system throughput is limited, indicating a linear correlation between system throughput and request rate. At high loads, BERT exhibits lower sensitivity to the time window compared to ResNet and LinkNet.

Finding 5. *Under medium to low workloads, the effect of time windows on the system throughput is limited.* As the request rate increases, system throughput exhibits linear growth. This behavior stems from the fact that, under medium to low workloads, the RBM collects a relatively small number of requests. Elevating the request rate can augment the quantity of requests gathered by the RBM, consequently amplifying the system throughput. Nonetheless, once the request rate surpasses a certain threshold, further increases do not contribute to enhanced throughput. This is because, when the request rate surpasses the system's processing capability, requests will be blocked in the queue, resulting in heightened latency.

Finding 6. *Under high workloads, for models that do not fully utilize the resources, time windows impact the system throughput through waiting times and batch sizes.* When subjected to high workloads, Triton's peak throughput for the BERT model exhibits minimal variance across different time windows. Since the BERT model saturates the system's resources with small batches, necessitating requests to wait in the batch queue until resources become available. The time window serves as a parameter for regulating the waiting time of requests in the request queue and the size of batches formed. Selecting an appropriate time window size can enhance the overall throughput of the system. A shorter window reduces queue wait times but limits batch size, underutilizing hardware. Conversely, a longer window extends waits but yields larger batches, maximizing hardware utilization. Therefore, compared to models like BERT, the impact of the time window is more pronounced for models that underutilize resources, such as ResNet and LinkNet.

5. Analysis of model slicing

5.1. Effects of slice positions

In this section, we slice the model into two subgraphs (i.e., stages) and investigate the impact of varying the slicing position on system throughput. We use a slicing ratio to denote the slicing position, specifically, the percentage of the total network compute time allocated to $Stage_0$, that is, $\frac{Stage_0}{Stage_0+Stage_1}$. The evaluation results are presented in Fig. 10, with the x-axis denoting the slicing ratio and the y-axis representing throughput.

Finding 7. *The choice of slice positions has an impact on both the model's computation time and memory access time.* Fig. 11 illustrates the breakdown of end-to-end model inference time at various slice positions. In Fig. 11, there are slight variations in computation time at different slice positions. This phenomenon is attributed to the fact that model slice disrupts operator fusion and other optimizations within the graph. Furthermore, we observe that memory access time at different slice positions is closely related to the model's architecture, specifically, it is influenced by the volume of data exchanged between stages. In addition, although Fig. 11 shows that the model inference time is the lowest when the model is not sliced (i.e., the slicing ratio is 100%), this also implies that meta-operations cannot be applied, so the system throughput is not necessarily optimal, as shown in Table 5.

Finding 8. *When selecting slice points, the computation time, the model structure, and the memory access time are important factors that need to be considered.* Fig. 10 demonstrates the impact of slice points on system throughput. "Naive Batching" refers to a system devoid of meta-operations and pipelined execution. The x-axis represents the slice ratio, and the y-axis represents throughput. Fig. 10 clearly indicates that slice points significantly influence the performance of pipelining execution systems by affecting pipeline balance. We also note that for ResNet, optimal throughput is achieved when slicing occurs in the model's middle, while for LinkNet, it is more advantageous towards the model's end. This variation is attributed to data transfer costs between stages, as depicted in Fig. 11.

5.2. Effects of stage counts

In this experiment, we employ the PipeDream [25] tool to slice the model into several stages with approximately equal execution time, aligning with the experimental methodology of the DVABatch. Fig. 12 illustrates the impact of the number of stages on system throughput, where the x-axis represents the number of stages, and the y-axis represents throughput.

Finding 9. *The optimal number of stages is typically small and, in most cases is not equal to 1.* As the number of stages increases, system throughput experiences a brief increase followed by a gradual decline. In contrast to schemes without model slicing, multi-stage designs support batch interruptions to leverage meta-operations for enhanced system throughput. However, increasing the number of stages introduces additional system overhead, such as synchronization costs between stages, resulting in finer scheduling granularity that undermines graph optimizations like layout selection and operator fusion, subsequently reducing system throughput. Additionally, we observe that pipelined execution is highly sensitive to the number of stages; as the stage count increases, system throughput deteriorates rapidly. Increasing the number of pipelined stages can enhance system throughput, but surpassing a specific threshold may reduce throughput due to resource contention.

6. Analysis of stage reorchestrating

6.1. Effects of reorchestrating strategies

Reorchestrating strategy is a method that prescribes execution in batches or stages, aimed at enhancing system throughput. In batching serving systems, reorchestrating strategy manages batch execution through meta-operations while also determining whether pipelining execution of stage instances is permissible. Stretch, split and pipeline execution are mutually independent, thereby allowing users to configure strategies to determine how these three operations are employed. Table 5 presents the system throughput of eight strategies for the model under various workloads. It can be observed from this table that the impact of strategies on system performance is limited in medium and

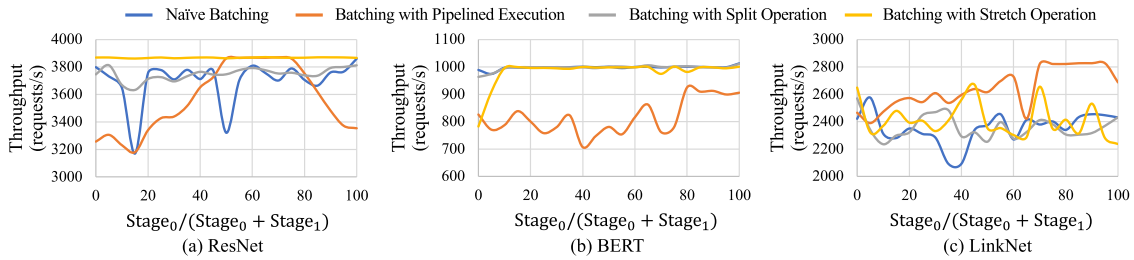
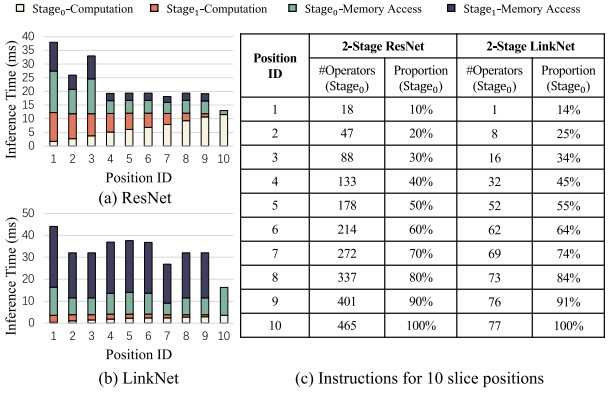


Fig. 10. Effects of slice position on the system's throughput. The optimal slicing position is related to the model structure. For example, the optimal slicing position for ResNet models is in the middle, while the optimal slicing position for LinkNet models is at the end. Pipeline parallel execution is highly sensitive to the selection of the slicing position. BERT models are not suitable for pipeline parallel execution.

Table 5
Effects of reorchestrating strategies on the system's throughput. At low and medium loads, the system throughput is hardly affected by variations with different reorchestrating strategy. Under high loads, however, the performance of the reorchestrating strategy differs among different types of models.

Strategy	Stretch	Split	Pipeline	ResNet			BERT			LinkNet		
				Low	Medium	High	Low	Medium	High	Low	Medium	High
I	0	0	0	1083.14	2608.62	3455.97	319.32	698.61	999.95	830.10	1974.27	2315.73
II	0	0	1	1081.07	2603.97	3867.68	319.33	692.98	749.00	830.02	1974.22	2774.40
III	0	1	0	1083.07	2607.60	3087.69	319.34	697.34	999.47	829.97	1974.36	2409.07
IV	0	1	1	1081.04	2604.87	3740.58	319.35	695.63	997.73	829.98	1974.86	2351.10
V	1	0	0	1081.12	2605.03	3854.02	319.30	696.19	995.43	830.03	1974.49	2368.43
VI	1	0	1	1080.92	2602.08	3866.82	319.33	686.51	752.64	829.91	1973.39	2764.35
VII	1	1	0	1080.89	2604.82	3756.22	319.34	695.99	993.90	829.94	1974.31	2322.81
VIII	1	1	1	1080.88	2602.44	3867.37	319.33	681.77	737.95	829.99	1974.06	2746.51



Position ID	2-Stage ResNet		2-Stage LinkNet	
	#Operators ($Stage_0$)	Proportion ($Stage_0$)	#Operators ($Stage_0$)	Proportion ($Stage_0$)
1	18	10%	1	14%
2	47	20%	8	25%
3	88	30%	16	34%
4	133	40%	32	45%
5	178	50%	52	55%
6	214	60%	62	64%
7	272	70%	69	74%
8	337	80%	73	84%
9	401	90%	76	91%
10	465	100%	77	100%

(c) Instructions for 10 slice positions

Fig. 11. Performance breakdown of 2-stage models with different slice positions. The impact of slicing positions on memory access time is notably significant and correlates with the model structure.

low-load scenarios. Under high load, the impact of strategies varies depending on the model type.

Finding 10. *Pipelined parallel execution is suitable for models with unsaturated computational resources or those encountering memory access bottlenecks. Stretch operations enhance the utilization of system computational resources, and split operations are effective for models bottlenecked by memory access.* For the ResNet model, strategies involving pipelined execution or stretch operations effectively improve resource utilization, thereby enhancing system throughput. However, for strategies that only involve split operations, performance decreases due to the sequential execution of the sub-batches, which prolongs request completion times. For models with saturated system resources, such as BERT, pipelined execution exacerbates resource contention and leads to performance degradation. In contrast, strategies incorporating meta-operations prevent performance degradation because the timing of operations is based on stage-specific performance models. For models with memory access bottlenecks, such as LinkNet, strategies involving pipelined execution effectively hide memory latency and enhance the system's throughput, while stretch operations only marginally reduce average computational time. Furthermore, split operations enable requests to finish in advance, enhancing system throughput by eliminating the need to wait for the entire batch to complete.

6.2. Performance analysis on split operations

The split operation allows requests to exit early, reducing average latency, but the resulting sub-batches may suffer from lower resource utilization, potentially reducing throughput. Therefore, the timing of split operations is a critical factor affecting system throughput. In this section, we explore the impact of the slice position, initial batch size for split, and the number of final sub-batches on the effectiveness of split operations. The evaluation results are presented in Figs. 13 and 14, where the x-axis represents the slice ratio, and the y-axis represents the speedup achieved by split operations compared to naive batching. We divide the model into two stages, $Stage_0$ and $Stage_1$, and the slice ratio refers to the percentage of the total network compute time allocated to $Stage_0$.

Finding 11. *Split operations yield more pronounced acceleration when occurring earlier (i.e., with lower slice ratios).* Observing Figs. 13 and 14, it is evident that split operations achieve their optimal effects with lower slice ratios. As the slice ratio increases, split operations gradually degrade into graph batching. This is because the benefits of split operations stem from the reduced average latency during the execution of sub-batches in the $Stage_1$ sequence. Therefore, a higher percentage of time allocated to $Stage_1$, the primary contributor to performance gains, implies greater potential benefits from split operations.

Finding 12. *Split operations are effective for the system under large batches, and the larger the batch to be divided, the greater the performance gain of split operations achieved.* Examining Fig. 13, it becomes apparent that, given a fixed slice ratio (e.g., 5%), split operations yield higher benefits as the batch size to be divided increases. Large batches may lead to resource contention due to the system's limited resources. Split operations mitigate resource competition by subdividing large batches into smaller sub-batches, thereby reducing average latency. Smaller sub-batches, on the other hand, are often unable to fully utilize hardware resources, and split operations further decrease hardware resource utilization, consequently reducing system throughput. Additionally,

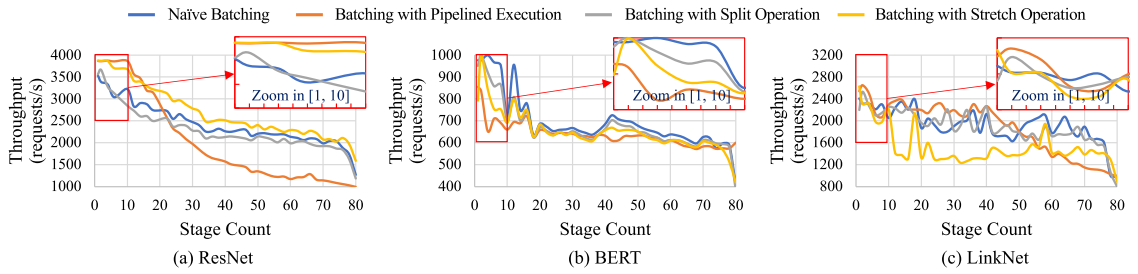


Fig. 12. Effects of stage counts on the system's throughput. As the number of stages increases, the system throughput initially experiences a transient increase, followed by a gradual decline.

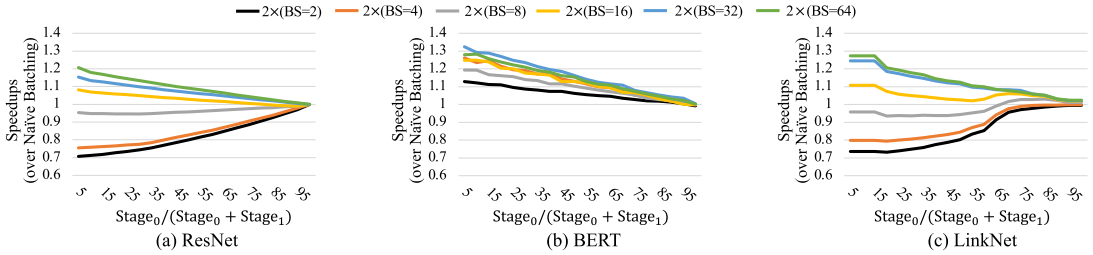


Fig. 13. Effects of split operations on batching process across varying batch sizes, where the legend "2×(BS = N)" represents dividing a batch of size N into two sub-batches, each of size $N/2$.

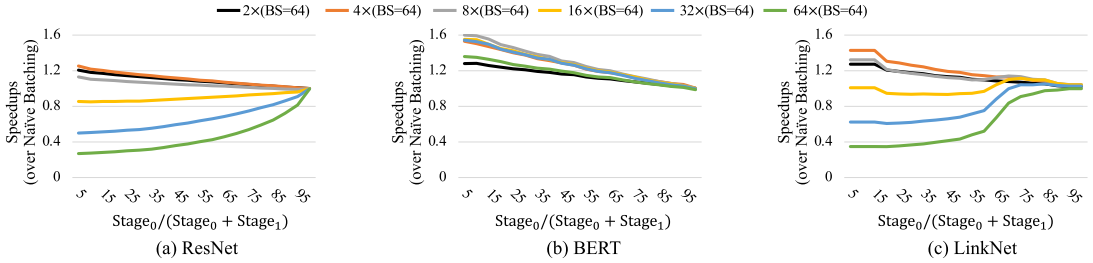


Fig. 14. Effects of split operations on batching process across varying sub-batch counts, where the legend "n×(BS = 64)" represents dividing a batch of size 64 into n sub-batches, each of size $64/n$.

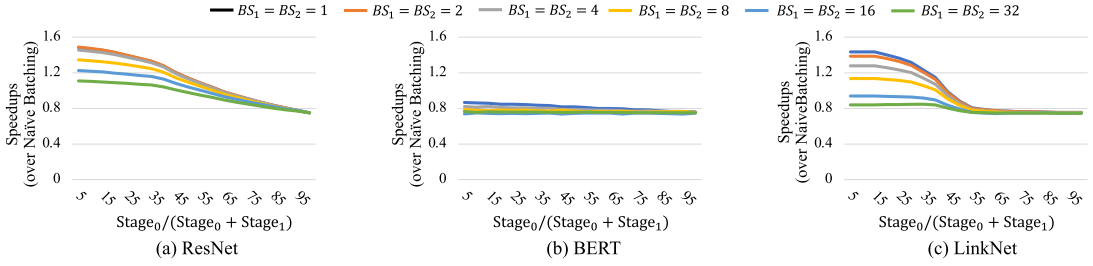


Fig. 15. Effects of stretch operations on the batching process across varying batch sizes.

since the BERT model saturates system resources with small batches, split operations result in acceleration across various batch sizes.

Finding 13. *The optimal number of sub-batches could be guided by the stage's performance model and does not follow the "more is better" principle.* Split operation is applicable to the scenario where resource utilization is saturated, that is, batching only increases its inference time without improving the processing throughput, such as Convolution-B in Fig. 2. Consequently, the split operation can split the original batch into several sequentially executed sub-batches to reduce the average latency, as shown in Fig. 1 (a). Observing the speedups of the split operation for the slice ratio of 5% in Fig. 14, we can find that the optimal number of sub-batches is not 64 (i.e., the green line), that is, the number of sub-batches is not the more the better. This is because when the sub-batch size reaches a certain threshold, further reducing the batch size will lead to insufficient hardware resource utilization due to the small batch size, which does not meet the premise of using the split operation, and thus leads to the ineffectiveness of the split operation or even negative effects. For this reason, we recommend that the timing of using the split operation should be referenced to the curve of the execution time of the stage with the batch size (such as Fig. 2), that is, the performance model.

6.3. Performance analysis on stretch operations

The stretch operation enhances system throughput by consolidating multiple small batches into a larger batch to fully exploit hardware resources. Fig. 5 illustrates the stretch operation process: when a new

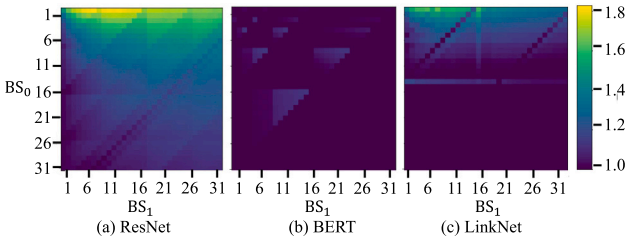


Fig. 16. Effects of various batch combinations on the effectiveness of stretch operations, where BS_0 and BS_1 denote two batches arriving subsequently. The intensity of color shading indicates the acceleration effect of stretch operations. For clarity, we also use 1 to represent negative effects. Stretch operations have a significant effect on ResNet models.

batch (BS_1) arrives, the current batch (BS_0) is in the middle of inference at $Stage_{0a}$. The stretch operation first completes the inference of BS_0 at $Stage_{0a}$, then proceeds to perform inference on BS_1 at $Stage_{0b}$, and finally merges them into a larger batch for $Stage_1$ inference. While stretch operations maximize computational resources by forming larger batches, they introduce waiting time during the batching process. This section analyzes the impact of the sizes and combinations of BS_0 and BS_1, as well as slice positions, on the effectiveness of stretch operations. For ease of analysis, we consider the scenario where BS_0 has just started execution at stage 0 and BS_1 arrives as our target scenario. Fig. 15 illustrates the impact of slice positions on the effectiveness of stretch operations, where the x-axis represents the slice ratio, and the y-axis represents the speedups over not using stretch operations. Fig. 16 illustrates the influence of various combinations of BS_0 and BS_1 on stretch operation efficacy. The brightness of the color signifies speedup levels relative to non-stretch operation scenarios, with instances of negative effect (speedups less than 1) marked as 1 for clarity.

Finding 14. *Stretch operations yield more significant acceleration when performed earlier.* As seen in Fig. 15, stretch operations are more likely to achieve noticeable acceleration when the slice ratio is low. This is because a lower slice ratio implies less waiting time, and with a higher proportion in stage 1 when system resources are not saturated, batch processing benefits more. As the slice ratio increases, the acceleration ratio of stretch operations tends to converge to 0.75. This is because when the slice ratio approaches 100, the proportion of stage 1 becomes nearly zero. Due to the necessity to wait for BS_1 to finish at stage 0, BS_0 cannot exit prematurely, resulting in an average delay of approximately 1.75 times that of batch processing.

Finding 15. *Applying stretch operations can usually enhance the system throughput under small batches.* As shown in Fig. 15, stretch operations exhibit noticeable acceleration when merging small batches, as the system cannot efficiently utilize hardware resources with small batches. However, for models like BERT, the system resources are already saturated with small batches, making stretch operations unsuitable for such models.

Finding 16. *Stretch operations are more suitable for ResNet-like models when computational resources are not saturated, and they are influenced by waiting time and batch processing gains.* Fig. 16 shows that in ResNet models, stretch operations generally have an acceleration effect, particularly when BS_0 is small, as it reduces the waiting time for BS_1 execution and still improves resource utilization after merging. However, for BERT models, stretch operations have minimal acceleration as the system resources are already saturated with small batches. For LinkNet models, stretch operations produce acceleration only with specific batch combinations. Thus, systems should decide whether to adopt stretch operations based on stage-specific performance models.

7. Discussion

In this work, an in-depth analysis and appraisal of the DNN batching serving system were undertaken, offering significant findings. This section gives the application scenarios and potential inspirations based on these findings.

7.1. Serving system configuration

In existing serving systems, the configuration of hyperparameters is a critical factor that affects the effectiveness of batching. However, there is a lack of comprehensive analysis and guidance on the configuration of these hyperparameters. This work fills this gap by providing insights into the impact of hyperparameters on batching effectiveness. Model deployment personnel can use the Finding 3 to configure MAX-BS to a larger value when deploying computationally intensive models. This will help to improve hardware resource utilization by forming larger batches. Furthermore, Finding 5 suggests that deployment personnel need not overly focus on the time window under medium to low loads. Serving system developers can adhere to the recommendations in Finding 9 for setting the number of stages. With these findings and suggestions in place, users of the serving system can more easily obtain appropriate parameter settings without undergoing complex, tedious, and time-consuming experiments and adjustments, thereby accelerating the application of the serving system.

7.2. DNN system optimization

We explore the potential directions outlined by the findings in this paper for promoting optimization and design of DNN serving systems. We first analyzed the relationship between the hyperparameters in the request batching module and the batching effect, and revealed the constituents of request latency (Findings 3–6). This provides a foundation for researchers to design adaptive parameter tuning systems for serving systems. Considering that workloads in practical scenarios often exhibit burstiness [26], and the inference serving time is deterministic [27], we can fit the collected request arrival traces to a Markov arrival process [28] at runtime to capture the burstiness. Based on the components of latency and deterministic inference time, we design a parameter tuner. The tuner determines optimal hyperparameter configurations based on the arrival process and QoS, maximizing throughput while meeting the QoS. Furthermore, we discover that in the model slicing module, the selection of slicing positions should consider computation time, model structure, and memory access time. Additionally, the number of stages correlates with runtime synchronization overhead. The aforementioned analysis offers possibilities for researchers to automatically determine optimal slicing positions and the number of stages. This inspires researchers to design an profiler to obtain computation time and access time under different slicing locations. Then, they can model the inference process under different stage reorchestrating strategies and query arrival processes, subsequently automatically determining the optimal slicing positions and stage numbers based on the performance model. Lastly, we examine the stage reorchestrating module and find that the conditions for utilizing pipelined execution and meta-operations should consider model characteristics and stage performance models. This insight guides researchers designing multi-tenant serving systems to execute computation- intensive stages and memory access-intensive stages in a pipelined manner to fully utilize hardware resources. Concurrently, performance models of stages inform the execution of meta-operations and resource allocation for the stages.

7.3. DNN application development

The findings in this paper also have implications for neural network application developers. Findings 1 and 2 indicate that the performance improvements achieved through batching techniques primarily arise from the efficient utilization of hardware computing resources, particularly when larger batch sizes are employed. Therefore, in the design of neural network models, efforts should be made to reduce the proportion

of memory access time. This hints at the importance for application developers to use lightweight operators whenever possible, such as employing depth-wise convolution operations in place of naive convolutions, and adopting quantization techniques to reduce memory access time. Finding 8 indicates that the position of model slices affects data flow and tensor lifecycle management. Long-lived tensors occupy memory resources for extended periods, increasing memory consumption and limiting the number of batching requests. Thus, DNN application developers should avoid designing long-lived tensors. Finding 9 suggests that model slicing may impact graph optimization techniques like operator fusion. Therefore, our advice to model designers is to construct network models using small, reusable blocks as much as possible to minimize the impact on graph optimization techniques such as operator fusion.

7.4. Impact on large language models

In various applications, the significance of language generation tasks has escalated, sparking heightened interest in optimizing serving systems via batching techniques. Orca represents the inaugural adaptation of DVABatch tailored for Large Language Models (LLMs). A pivotal insight of Orca posits that Transformer-based generative models function iteratively, so the batching should focus on iterations rather than individual requests. Consequently, Orca aligns DVABatch stages with LLM iterations and supports batching arbitrary requests by executing the iterations in a batch that are in prefill and decode states separately. In this study, we conducted a comprehensive evaluation of the DVABatch system, yielding several critical insights.

BERT and Transformer models differ in terms of task objectives and output layers. Transformer is a sequence processing model that uses SoftMax for probability distribution computation at the output layer, while BERT focuses on learning language representations from text data, which is typically used to generate context-related word embeddings. However, they are both implemented based on multiple stacked transformer layers (i.e., including attention layers and forward feedback layers). In this paper, we discover that BERT can saturate hardware resources even with small batch sizes. Furthermore, serving systems utilizing pipeline parallelism exhibit lower throughput when confronted with the BERT model compared to naive serving systems. Consequently, this insight suggests that designers of LLM serving systems should refrain from employing pipeline parallelism on a single GPU platform.

Given that LLMs typically operate iteratively, and the behavioral characteristics during the prefill and decode phases exhibit significant differences [29,30], this constitutes the most prominent distinction between LLM and BERT. Researchers can leverage the findings of this paper and integrate the unique features of LLM to design serving systems effectively. In this context, we propose two potential research directions and offer possible solutions to stimulate further scholarly discourse. Findings 4 and 6 elucidate that the queue's waiting time markedly impacts the serving system, primarily due to the unpredictable request distribution. In LLM, the arrival time distribution and iteration count remain indeterminate. Hence, researchers may formulate a multi-feedback queue scheduler for handling unknown arrival times [31] and develop a compact model consistent with LLM to forecast request iteration counts [32], facilitating batch processing of requests with analogous iteration counts to minimize latency. Finding 10 suggests that the design and use of meta-operations should align with model characteristics, offering insights for researchers in designing new meta-operations for LLM serving systems. This prompted researchers to develop new meta-operations that couple multiple iterations in decoding states with a iteration in prefill states, utilizing weight data reuse to reduce memory access and thereby improve system throughput [33].

In future work, we will augment the characterization of batching behavior within the LLM serving system and undertake a more profound exploration based on the aforementioned two research directions.

7.5. Multi-GPU platforms

In existing DNN serving system designs, the batching module and the inference engine module are independently designed, encompassing serving systems such as Triton, DVABatch, and Orca. In contemporary DNN serving systems, tensor parallelism and pipeline parallelism are commonly utilized for inference services across multiple GPUs [34], primarily within the confines of the inference engine module. While this paper focuses on the batching system, insights into the design of the inference engine remain beneficial. For instance, in Section 5, we highlight that the selection of model slicing positions is associated with the model structure, which can provide guidance for the design of the pipeline stages of the pipeline parallelism paradigm in the execution engine layer. Furthermore, this work clarifies existing DNN serving system designs, laying the groundwork for future collaborative designs between the batching system and the inference engine. For example, considering a machine equipped with two GPU cards (GPU_1 and GPU_2) using the pipeline parallelism paradigm—where GPU_1 handles the front portion of the model and GPU_2 manages the rear portion. Assuming two batches of varying sizes, A and B (with A having a larger batch size than B), arrive sequentially. Orca would first execute A on GPU_1 (front portion of the model) followed by A on GPU_2 (rear portion) while simultaneously processing B on GPU_1 (front portion). Due to A's larger batch size compared to B, a bubble occurs on GPU_1. If the batching system layer can perceive that the execution engine layer uses the pipeline parallelism paradigm, it can reduce the occurrence of bubbles by dividing the requests into finer granularities.

8. Related work

Dynamic Batching. In the realm of model training, researchers focused on adjusting batch sizes to strike a balance between training efficiency and model generalization [35–37]. In the training phase, all input data is available, allowing for the efficient collection of multiple samples without latency. However, in the inference phase, since the ML serving systems receive input at different times, and batching system needs to balance latency and throughput, which poses challenges. Therefore, our paper focuses on analyzing batching techniques in the inference phase.

Regarding batching techniques during model inference, there are three primary types, as delineated in prior studies [7,38]: static batching, dynamic batching, and application-specific batching. Static batching, as exemplified by systems such as Triton and TensorFlow-Serving, relied on two critical hyperparameters: the model-allowed maximum batch size and the time window, which govern request batching behavior. In a static batching system, new batches can only be executed after the current batch inference is done, causing longer request wait times.

Therefore, researchers have proposed dynamic batching, allowing batch size modification during the inference process, with some typical serving systems including LazyBatching [7] and DVAbatch [3]. In dynamic batching techniques, models are sliced into different subgraphs to support the addition of new requests and the early exit of old requests. LazyBatching slices the model at the granularity of operators and employs a QoS-aware slack time prediction algorithm to delay request processing, creating larger batches. DVABatch, built upon LazyBatching, uses subgraphs as the slice granularity and introduces stretch and split operations to adapt to different application scenarios.

Furthermore, there have been batching techniques tailored for specific applications. As the number of iterations varies for different requests in the generation model, Orca [39] introduces iteration-level batching, i.e., considering whether to incorporate new iterations or early exit the iteration from the batch. In applications involving diverse sequence lengths, researchers explored strategies for concatenating requests into larger inputs [40] or adopting finer-grained grouping techniques [41] to improve performance.

Despite numerous DNN serving system batching techniques, their applicability and operational contexts remain unclear. Additionally, these methods often target specific modules, such as static batching for request batching module and dynamic batching for stage reorchestrating modules. This work delivers a holistic assessment of the influence of parameter configurations, model slicing strategies, and stage reorchestrating strategies on batching serving systems across diverse models and workloads. To the best of our knowledge, this is the first study that comprehensively evaluates and analyzes DNN batching serving system.

Serving Systems. In serving systems, batch processing was often considered in conjunction with factors such as resource allocation and QoS. Various approaches were devised to employ adaptive strategies, enhancing efficiency and ensuring equitable resource distribution to fulfill users' inference demands. DyBatch [42] adjusted batch sizes based on device workloads and task requisites to uphold fairness. Nanily [43] dynamically allocated computational resources, aiming to meet QoS requirements while optimizing resource utilization. Ebird [9, 44] excelled in performance maximization across fluctuating workloads. In the design of serving systems, batching techniques typically need to be collaboratively designed with other optimization techniques. This study contributes to a better understanding of batching techniques for developers and lays the foundation for designing superior serving systems.

9. Conclusion

Optimizing and deploying DNN serving systems lay in understanding the behavior of batching throughout the entire system. In this paper, we characterized the behavior of the request batching module, model slicing module, and stage reorchestrating module, in deep neural network batching systems on GPUs, by using three representative models. Based on experimental results, several meaningful insights and findings are provided for future research to further enhance deep learning serving systems.

CRediT authorship contribution statement

Feng Yu: Writing – original draft, Methodology, Conceptualization. **Hao Zhang:** Software, Formal analysis. **Ao Chen:** Validation, Investigation. **Xueying Wang:** Validation, Formal analysis. **Xiaoxia Liang:** Software, Investigation. **Sheng Wang:** Validation, Investigation. **Guangli Li:** Project administration, Methodology, Conceptualization. **Huimin Cui:** Supervision, Methodology. **Xiaobing Feng:** Supervision, Methodology.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (62232015, 62090024, 62302479), the China Postdoctoral Science Foundation (2023M733566), and the Innovation Funding of ICT, CAS, China (E361010).

References

[1] N. Inc., NVIDIA triton inference server, 2023, URL https://docs.nvidia.com/deeplearning/triton-inference-server/, Accessed: August, 2023.

[2] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, J. Soyke, Tensorflow-serving: Flexible, high-performance ml serving, 2017, arXiv preprint arXiv:1712.06139.

[3] W. Cui, H. Zhao, Q. Chen, H. Wei, Z. Li, D. Zeng, C. Li, M. Guo, DVABatch: Diversity-aware Multi-Entry Multi-Exit batching for efficient processing of DNN services on GPUs, in: 2022 USENIX Annual Technical Conference, 2022, pp. 183–198.

[4] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 770–778.

[5] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, Bert: Pre-training of deep bidirectional transformers for language understanding, 2018, arXiv preprint arXiv:1810.04805.

[6] A. Chaurasia, E. Culurciello, Linknet: Exploiting encoder representations for efficient semantic segmentation, 2017, arXiv preprint arXiv:1707.03718.

[7] Y. Choi, Y. Kim, M. Rhu, Lazy batching: An SLA-aware batching system for cloud machine learning inference, in: International Symposium on High-Performance Computer Architecture, 2021, pp. 493–506.

[8] X. Li, G. Zhang, H.H. Huang, Z. Wang, W. Zheng, Performance analysis of GPU-based convolutional neural networks, in: 2016 45th International Conference on Parallel Processing, ICPP, IEEE, 2016, pp. 67–76.

[9] W. Cui, M. Wei, Q. Chen, X. Tang, J. Leng, L. Li, M. Guo, Ebird: Elastic batch for improving responsiveness and throughput of deep learning services, in: International Conference on Computer Design, 2019, pp. 497–505.

[10] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, et al., TVM: An automated End-to-End optimizing compiler for deep learning, in: 13th USENIX Symposium on Operating Systems Design and Implementation, 2018, pp. 578–594.

[11] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al., Pytorch: An imperative style, high-performance deep learning library, Adv. Neural Inf. Process. Syst. 32 (2019).

[12] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, T. Darrell, Caffe: Convolutional architecture for fast feature embedding, in: Proceedings of the 22nd ACM International Conference on Multimedia, 2014, pp. 675–678.

[13] N. Inc., NVIDIA tensorRT, 2021, URL: https://developer.nvidia.com/tensorrt, Accessed on 2023-09-04.

[14] N. inc, Triton client libraries and examples, 2021, URL: https://github.com/triton-inference-server/client, Accessed on 2023-09-04.

[15] V.J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, et al., Mlperf inference benchmark, in: International Symposium on Computer Architecture, 2020, pp. 446–459.

[16] D. Yastremsky, Maximizing deep learning infer-ence performance with NVIDIA model analyzer, 2020, URL https://developer.nvidia.com/blog/maximizing-deep-learning-inference-performance-with-nvidia-model-analyzer, Accessed: August, 2023.

[17] S.V. Saavedra, A.L. Uribe, Google cloud vision and its application in image processing using a raspberry Pi, in: Colombian Conference on Computing, Springer, 2022, pp. 102–113.

[18] D. Avinash, J.A. Kumar, R. Chandansingh, Use of AI in cloud-based certificate authentication for travel concession, in: Mobile Computing and Sustainable Informatics: Proceedings of ICMCSI 2023, Springer, 2023, pp. 349–361.

[19] A. Satapathi, A. Mishra, Build a multilanguage text translator using azure cognitive services, in: Developing Cloud-Native Solutions with Microsoft Azure and. NET: Build Highly Scalable Solutions for the Enterprise, Springer, 2022, pp. 231–248.

[20] H.-M. Sormunen, Enhancing customer feedback processing with machine learning in Microsoft Azure, 2022.

[21] M. Singh, Single stage facial recognition based on YOLOv5, in: 2022 International Conference on INnovations in Intelligent SysTems and Applications, INISTA, IEEE, 2022, pp. 1–6.

[22] T. Leonor Estévez Dorantes, D. Bertani Hernández, A. León Reyes, C. Elena Miranda Medina, Development of a powerful facial recognition system through an API using ESP32-Cam and amazon rekognition service as tools offered by industry 5.0, in: 2022 the 5th International Conference on Machine Vision and Applications, ICMVA, 2022, pp. 76–81.

[23] S. Marcel, Y. Rodriguez, Torchvision the machine-vision package of torch, in: Proceedings of the 18th ACM International Conference on Multimedia, 2010, pp. 1485–1488.

[24] S.M. Jain, Hugging face, in: Introduction to Transformers for NLP: With the Hugging Face Library and Models to Solve Problems, Springer, 2022, pp. 51–67.

[25] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N.R. Devanur, G.R. Ganger, P.B. Gibbons, M. Zaharia, PipeDream: Generalized pipeline parallelism for DNN training, in: Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP '19, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450368735, 2019, pp. 1–15, http://dx.doi.org/10.1145/3341301.3359646.

[26] A. Ali, R. Pinciroli, F. Yan, E. Smirni, Batch: machine learning inference serving on serverless platforms with adaptive batching, in: International Conference for High Performance Computing, Networking, Storage and Analysis, 2020, pp. 1–15.

[27] F. Yan, O. Ruwase, Y. He, E. Smirni, SERF: Efficient scheduling for fast deep neural network serving via judicious parallelism, in: SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2016, pp. 300–311.

[28] M.F. Neuts, A versatile Markovian point process, J. Appl. Probab. 16 (4) (1979) 764–779.
[29] K. Hong, G. Dai, J. Xu, Q. Mao, X. Li, J. Liu, K. Chen, H. Dong, Y. Wang, FlashDecoding++: Faster large language model inference on GPUs, 2023, arXiv preprint arXiv:2311.01282.
[30] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C.H. Yu, J. Gonzalez, H. Zhang, I. Stoica, Efficient memory management for large language model serving with pagedattention, in: Proceedings of the 29th Symposium on Operating Systems Principles, 2023, pp. 611–626.
[31] B. Wu, Y. Zhong, Z. Zhang, G. Huang, X. Liu, X. Jin, Fast distributed inference serving for large language models, 2023, arXiv preprint arXiv:2305.05920.
[32] Q. Su, C. Giannoula, G. Pekhimenko, The synergy of speculative decoding and batching in serving large language models, 2023, arXiv preprint arXiv:2310.18813.
[33] A. Agrawal, A. Panwar, J. Mohan, N. Kwatra, B.S. Gulavani, R. Ramjee, SARATHI: Efficient LLM inference by piggybacking decodes with chunked prefills, 2023, arXiv preprint arXiv:2308.16369.
[34] X. Miao, C. Shi, J. Duan, X. Xi, D. Lin, B. Cui, Z. Jia, SpotServe: Serving generative large language models on preemptible instances, 2023, arXiv preprint arXiv:2311.15566.
[35] A. Devarakonda, M. Naumov, M. Garland, Adabatch: Adaptive batch sizes for training deep neural networks, 2017, arXiv preprint arXiv:1712.02029.
[36] A. Lydia, S. Francis, Adagrad—an optimizer for stochastic gradient descent, Int. J. Inf. Comput. Sci. 6 (5) (2019) 566–568.
[37] M. Zaheer, S. Reddi, D. Sachan, S. Kale, S. Kumar, Adaptive methods for nonconvex optimization, Adv. Neural Inf. Process. Syst. 31 (2018).
[38] E.L. Cade Daniel, R. Liaw, How continuous batching enables 23x throughput in LLM inference while reducing p50 latency, 2023, URL: https://www.anyscale.com/blog/continuous-batching-llm-inference, Accessed on 2023-09-04.
[39] G.-I. Yu, J.S. Jeong, G.-W. Kim, S. Kim, B.-G. Chun, Orca: A distributed serving system for transformer-based generative models, in: USENIX Symposium on Operating Systems Design and Implementation, 2022, pp. 521–538.
[40] B. Fu, F. Chen, P. Li, D. Zeng, TCB: Accelerating transformer inference services with request concatenation, in: Proceedings of the 51st International Conference on Parallel Processing, 2022, pp. 1–11.
[41] Y. Zhai, C. Jiang, L. Wang, X. Jia, S. Zhang, Z. Chen, X. Liu, Y. Zhu, ByteTransformer: A high-performance transformer boosted for variable-length inputs, in: International Parallel and Distributed Processing Symposium, 2023, pp. 344–355.
[42] S. Zhang, W. Li, C. Wang, Z. Tari, A.Y. Zomaya, DyBatch: Efficient batching and fair scheduling for deep learning inference on time-sharing devices, in: International Symposium on Cluster, Cloud and Internet Computing, 2020, pp. 609–618.
[43] X. Tang, P. Wang, Q. Liu, W. Wang, J. Han, Nanily: A qos-aware scheduling for dnn inference workload in clouds, in: International Conference on High Performance Computing and Communications, 2019, pp. 2395–2402.
[44] W. Cui, Q. Chen, H. Zhao, M. Wei, X. Tang, M. Guo, E2bird: Enhanced elastic batch for improving responsiveness and throughput of deep learning services, IEEE Trans. Parallel Distrib. Syst. 32 (6) (2020) 1307–1321.