TCB: Accelerating Transformer Inference Services with Request Concatenation

Boqian Fu
m5242101@u-aizu.ac.jp
The University of Aizu
Aizuwakamatsu, Japan

Fahao Chen
d8232101@u-aizu.ac.jp
The University of Aizu
Aizuwakamatsu, Japan

Peng Li
pengli@u-aizu.ac.jp
The University of Aizu
Aizuwakamatsu, Japan

Deze Zeng
deze@cug.edu.cn
China University of Geoscience
Wuhan, China

ABSTRACT

Transformer has dominated the field of natural language processing because of its strong capability in learning from sequential input data. In recent years, various computing and networking optimizations have been proposed for improving transformer training efficiency. However, transformer inference, as the core of many AI services, has been seldom studied. A key challenge of transformer inference is variable-length input. In order to align these input, existing work has proposed batching schemes by padding zeros, which unfortunately introduces significant computational redundancy. Moreover, existing transformer inference studies are separated from the whole serving system, where both request batching and request scheduling are critical and they have complex interaction. To fill the research gap, we propose TCB, a **T**ransformer inference system with a novel **C**oncat**B**atching scheme as well as a jointly designed online scheduling algorithm. ConcatBatching minimizes computational redundancy by concatenating multiple requests, so that batch rows can be aligned with reduced padded zeros. Moreover, we conduct a systemic study by designing an online request scheduling algorithm aware of ConcatBatching. This scheduling algorithm needs no future request information and has provable theoretical guarantee. Experimental results show that TCB can significantly outperform state-of-the-art.

KEYWORDS

Transformers, inference, scheduling, online algorithm

ACM Reference Format:
Boqian Fu, Fahao Chen, Peng Li, and Deze Zeng. 2022. TCB: Accelerating Transformer Inference Services with Request Concatenation. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Transformer, as a kind of emerging deep learning models [12, 20, 27, 32, 35], has shown great power in applications related to natural language processing (NLP). It overwhelms competitors, e.g., Recurrent Neural Networks (RNNs) [7, 17], in both accuracy and training efficiency, thanks to its unique self-attention mechanism. Transformer ignites not only the machine learning community but also system research. There has been many existing work that improves transformer's training efficiency by exploiting its unique computation and communication patterns. For example, Zhang et al. [37] propose to reduce training overhead by adaptively dropping some unimportant layers. Chen et al. [6] find that a significant computation cost of the transformer comes from attention operations and propose an attention-aware pruning design to remove some weights. The communication efficiency of transformer training has been studied by [18], which points out that a key bottleneck of transformer's training is the data movement and proposes an optimization based on dataflow analysis.

Despite great efforts at optimizing transformer training, little attention is devoted to accelerating transformer inference, which plays a critical role in NLP-related serving systems. These serving systems, which usually reside in cloud, receive inference requests from end users and should make responses with low latency. However, existing transformer inference methods are not efficient, mainly because of input of variable lengths. The variable-length input is very common in NLP-related services. For example, language translation services receive requests in the form of sentences, and obviously these sentences are with different lengths. A common optimization to increase inference efficiency is to batch several requests, which are further described as multi-dimensional tensors that can be processed by GPUs with high efficiency. When batching, short requests need to be zero-padded so that they can align with long ones in the same batch. An example is shown in Fig. 1(a), where the batch row length is fixed and shorter requests are padded with zeros.

This is the default batching scheme adopted by popular machine learning platforms (e.g., PyTorch), and it is referred to as NaiveBatching in this paper. Zero-padding wastes GPU memory and brings extra computation. To reduce the overhead incurred by zero-padding, Fang et al. [14] have proposed a length-aware batching scheme, called TurboBatching, which reorders requests according to their lengths and batches the ones of similar length, as shown in Fig. 1(b). Such a length-aware batching scheme can significantly reduce padded zeros, only if it is fed by a sufficient number of requests with similar length. However, the requests received by serving systems could be quite different in length, which makes the length-aware batching scheme cannot always play its full power. For example, TurboBatching has low GPU utilization on several

Conference'17, July 2017, Washington, DC, USA
2022. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

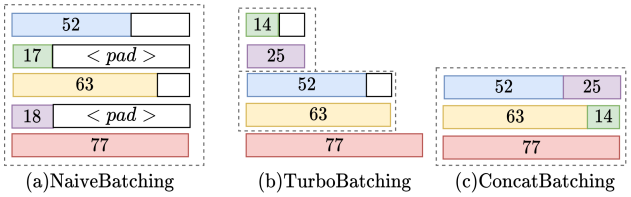


Figure 1: Different batching schemes.

datasets, e.g., ParaCrawl [3] and DIA [33], whose workloads are highly variable in length.

Another issue of existing work is the lack of request scheduling policies, with tight integration with request batching schemes. Request scheduling policies determine which requests should be served first, while batching schemes decide how they can be served. They are separately studied in [11, 24, 38] and [8, 10, 14], respectively. Request scheduling and batching have complex interaction, which unfortunately has not been well exploited by existing work.

In this paper, we propose TCB, a fast **T**ransformer inference system that integrates a novel batching scheme called **C**oncat**B**atching and an online request scheduling algorithm with theoretical guarantee. As illustrated in Fig. 1(c), the proposed ConcatBatching scheme concatenates several requests so that they can be accommodated within a single batch with less padded zeros. Compared to other two batching schemes, ConcatBatching is more flexible and is able to handle requests with arbitrary length distributions.

Although ConcatBatching is promising, we need to address two challenges to make it work correctly and efficiently in practice. First, existing transformer inference systems do not support request concatenation, so we need to modify the inference algorithm to distinguish requests concatenated in the batch row, so that we can get correct inference results. Second, serving system efficiency is also affected by request scheduling algorithms, which determine when and which requests should be batched and sent to GPU. We desire an algorithm that can fully exploit the potential capacity of ConcatBatching, while being aware of request deadlines to guarantee a certain level of service quality.

TCB conquers both challenges with a modular design that consists of a runtime supporting ConcatBatching and a plugable scheduler module. Different from traditional transformer inference, TCB runtime customizes the positional encoding and the self-attention operation to guarantee the correctness of inference results. Moreover, by carefully examining the inference process, we find that there exists some redundant computation brought by ConcatBatching. Thus, we propose an enhancement to eliminate the redundancy, so that the inference can be further speed up. Based on this runtime, we study a request scheduling problem with the objective of maximizing the number of requests responded by their deadlines. An online algorithm has been designed and implemented in the scheduler module. We also derive its theoretical bound. Thanks to the joint design of ConcatBatching runtime and the scheduler, TCB can significantly outperform existing works, as demonstrated by our experimental results .

The rest of this paper are organized as follows. We introduce the background and related work in Section 2. TCB system overview is given in Section 3, followed by inference engine design and scheduling algorithm in Section 4 and Section 5, respectively. Finally, we evaluate TCB system in Section 6 and conclude our work in Section 7.

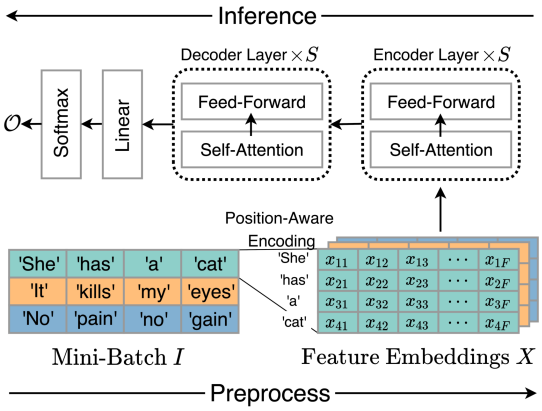


Figure 2: Transformer-based language model computation process.

2 BACKGROUND AND RELATED WORK

2.1 Transformer Model

Transformer has been widely studied for language processing [5, 28–30]. It adopts an encoder-decoder architecture, which takes a sentence of multiple words (also called tokens) as input and generates a task-related result, such as a translated sentence, as output. Typically, transformer inference is conducted in a mini-batch manner, which feeds multiple sentences to the transformer model for inference simultaneously. A typical transformer inference process is illustrated in Fig. 2. In the preprocessing stage, a mini-batch of sentences, denoted by I, are first transferred into feature embeddings X, where each word is represented by a feature vector. The word position information can also embeded via positional encoding [32] as

$$PE(pos, 2e) = \sin(pos/10000^{2e/d_{model}}) \quad (1)$$

$$PE(pos, 2e+1) = \cos(pos/10000^{(2e+1)/d_{model}}) \quad (2)$$

where pos is the position of the word in the sentence, e−th dimension of sinusoidal positional encoding corresponds to a sinusoid, and d_{model} is the dimension of the feature embeddings. In the inference stage, these embeddings are sent to the encoder and decoder, each of which contains S layers. Each encoder or decoder layer includes the self-attention computation, followed by a feed-forward network to enhance model performance. Decoder outputs go through a final linear layer and activation function to generate final results.

The core of transformer is the self-attention computation. Given feature embeddings X, the first step of self-attention computation is to create the query matrix Q, the key matrix K, and the value matrix V as follows.

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V, \quad (3)$$

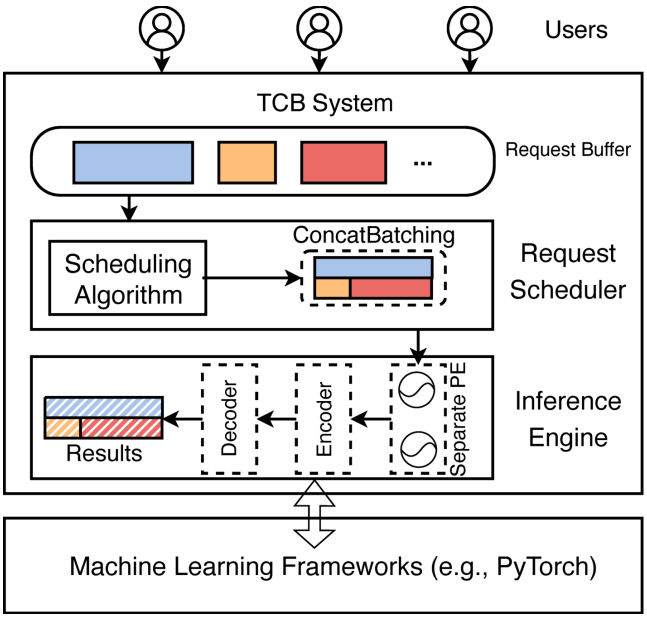


Figure 3: System overview

where W^Q, W^K and W^V are weight matrices. The second step is to compute the self-attention as:

$$\text{Att}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V, \tag{4}$$

where $\sqrt{d}$ is a scaled parameter.

A plethora of works [16, 23, 25, 31, 34] have been proposed to improve transformer computation efficiency. At the hardware level, ELSA [16] has been proposed as a specialized accelerator for approximate self-attention computing. Moreover, Softermax, proposed by [31], is an accelerator that is able to reduce computational cost of the softmax operation. At the software level, Yan et al. [34] reconstruct self-attention computation to avoid the linear conversion of keys and values, which brings a significant speedup. Li et al. [23] propose a token-level pipeline algorithm to improve the efficiency of the training transformer-based language models. Meanwhile, Narayanan et al. [25] introduce a novel interleaved pipelining mechanism to improve the throughput for training large-scale transformer-based models. A unique feature of transformer for language processing is variable-length input. Fang et al. [14] have proposed TurboTransformer, an online serving system that adopts a length-aware dynamic programming batching mechanism to mitigate the redundant computations. Despite the improvement brought by TurboTransformer, it heavily relies on the assumption that arrived requests have similar lengths and and it does not involve a dedicated scheduling algorithm for online services.

2.2 Online Request Scheduling

In practical serving systems, inference requests arrive in an online manner. Therefore, designing online scheduling mechanisms has a profound implication for guaranteeing the system performance and quality of service. Many online scheduling algorithms have been proposed to solve various problems in machine learning systems [8–10, 13, 15, 19, 21, 22, 36]. For example, Li et al. [22] introduce AutoDeep, which strategically decides the device placements and cloud configurations for online inference tasks through Bayesian Optimization and deep reinforcement learning. RIBBON is introduced by [21], which also adopts a Bayesian Optimization-driven strategy to employ heterogeneous cloud computing instances to minimize inference costs. A novel ahead-of-time (AoT) scheduling has been proposed to improve inference speed [19]. Prema [9] pursuits to balance multiple serving metrics, such as latency, fairness, and service level agreement (SLA), through a predictive scheduling algorithm under the preemptive NPUs environment. Choi et al. [8] propose LazyBatching, an SLA-aware DNN inference system that adopts fine-grained scheduling and batching. However, these works are inefficient for transformer-based inference tasks since they ignore properties of transformers with variable-length input.

3 SYSTEM OVERVIEW

A system overview of TCB is shown in Figure 3. TCB resides between machine learning frameworks (e.g., TensorFlow [2] or PyTorch [26]) and user applications. It receives inference requests, usually in the form of sentences, from user applications and process them using two core modules: a request scheduler and a customized inference engine. Each request contains a sentence to be processed, associated with an arriving time and a desired response deadline. When GPU is idle, the scheduler packs some received requests into a batch and sends it to the inference engine. A request scheduling algorithm is designed to decide which requests should be batched together, by jointly considering request concatenation, request deadlines and running speed of inference engine. The inference engine is responsible for running the transformer inference computation in GPUs. Since multiple requests are concatenated, the default inference algorithm adopted by PyTorch or TensorFlow would generate wrong results. Therefore, we customize the self-attention computation process to guarantee inference correctness. In addition, we have examined and identified the computation redundancy due to request concatenation and proposed a "slotted" design that is able to reduce such redundancy to further accelerate the inference computation. The design details of inference engine and request scheduler are given in the following sections.

4 CONCATBATCHING INFERENCE ENGINE

In this section, we first present a customized inference engine design that supports arbitrary concatenation of requests in a batch, which is referred to as *pure ConcatBatching*. Then, we further reduce computation redundancy by proposing a *slotted ConcatBatching* scheme, where a batch is divided into several slots and requests can be concatenated within each slot. Conceptually, the difference between two schemes can be illustrated by Fig. 4.

4.1 Pure ConcatBatching

The TCB inference engine enabling pure ConcatBatching follows traditional transformer inference process, as shown in Fig. 2, but it customizes the following two key components, which are crucial for generating correct inference results.

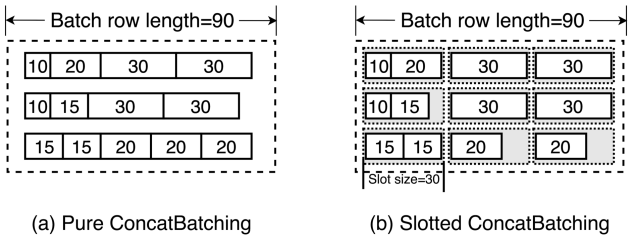


Figure 4: The difference between pure ConcatBatching and slotted ConcatBatching.

4.1.1 Separate positional encoding. Position or order of words (or tokens) in a sentence is critical for transformer to understand the language. Positional encoding (PE) [32] has been proposed to incorporate word order into the transformer model. The default positional encoding scheme in traditional transformer treats all words in a batch row as a single sentence and encodes them accordingly, as shown in Fig. 5(a). This default scheme cannot be directly applied in TCB, because words of different sentences concatenated in a batch row have no order relationship. Therefore, we customize the positional encoding in TCB by encoding these sentences separately. The separate positional encoding can be illustrated in Fig. 5(b).

4.1.2 Customized self-attention. Self-attention is the most critical computation in the transformer inference. TCB customizes self-attention computation to guarantee correct inference results. For easy understanding, we use a single batch row to present the self-attention computation process, which is illustrated in Fig. 6. We suppose that m requests are concatenated in this row, and corresponding feature embeddings are denoted by $\mathcal{X} = [X_1, X_2, ..., X_m]$. Similar with traditional self-attention computation, we transfer $\mathcal{X}$ into query, key and value matrices as $\mathcal{Q} = \mathcal{X} \cdot W^Q = [Q_1, Q_2, ..., Q_m]$, $\mathcal{K} = \mathcal{X} \cdot W^K = [K_1, K_2, ..., K_m]$ and $\mathcal{V} = \mathcal{X} \cdot W^V = [V_1, V_2, ..., V_m]$, respectively.

Note that $\mathcal{X}$ contains several parts and X_i corresponds to a request. Accordingly, matrices $\mathcal{Q}$, $\mathcal{K}$ and $\mathcal{V}$ have similar structures. After that, we compute the self-attention as follows.

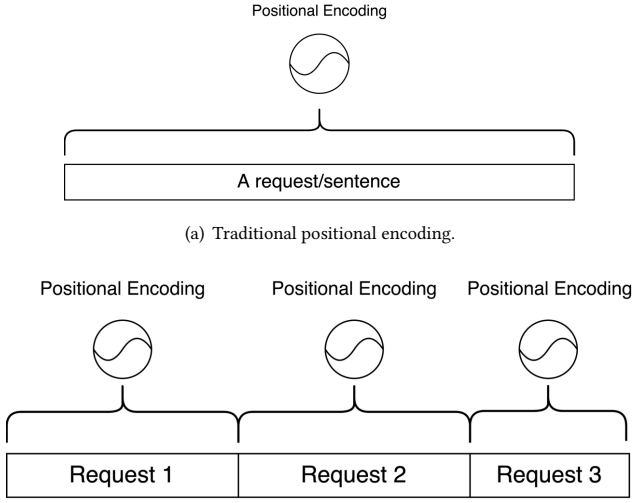
$$\text{Att\_CB}(\mathcal{Q}, \mathcal{K}, \mathcal{V}) = \text{softmax}\left(\frac{\mathcal{Q}\mathcal{K}^T}{\sqrt{d}} + \mathcal{M}\right)\mathcal{V} \tag{5}$$

where $\mathcal{M}$ is a mask matrix defined as follows:

$$\mathcal{M} = \begin{cases} 0, & \forall \text{entries at } Q_i K_i^T, 1 \le i \le m; \\ -\infty, & \text{otherwise.} \end{cases} \tag{6}$$

The major difference between the customized self-attention (5) and the original one (4) is that we use a special matrix $\mathcal{M}$ to mask redundant parts introduced by request concatenation. This masking operation is necessary to guarantee correct inference. We explain the reason by showing the details of score matrix $\frac{\mathcal{Q}\mathcal{K}^T}{\sqrt{d}}$ as follows.

$$\frac{\mathcal{Q}\mathcal{K}^T}{\sqrt{d}} = \begin{pmatrix} Q_1K_1^T & Q_1K_2^T & \cdots & Q_1K_m^T \\ Q_2K_1^T & Q_2K_2^T & \cdots & Q_2K_m^T \\ \cdots & \cdots & \cdots & \cdots \\ Q_mK_1^T & Q_mK_2^T & \cdots & Q_mK_m^T \end{pmatrix} / \sqrt{d} \tag{7}$$



(a) Traditional positional encoding.

(b) TCB's separate positional encoding.

Figure 5: TCB's separate positional encoding for a batch row.

Since we care about intra-sentence attention, not the inter-sentence one, only the main diagonal entries, e.g., $Q_iK_i^T$, in the above matrix is useful for generating final results. Therefore, we conduct a mask operation that changes all off-diagonal entries to $-\infty$, so that they are eliminated from the next softmax operation.

4.2 Slotted ConcatBatching

To motivate the design of slotted ConcatBatching, we first take a look at the final step of the self-attention computation in Fig. 6, where the matrix $\mathcal{A}$ is multiplied by the value matrix $\mathcal{V}$. We notice that many entries in matrix $\mathcal{A}$ are redundant, but they are still involved in the subsequent softmax and multiplication operations. Although our mask matrix $\mathcal{M}$ can eliminate their negative influence on the final result, they occupy extra GPU memory spaces and waste GPU computing resources.

4.2.1 A new self-attention operation aware of slots. To further improve the inference efficiency, we propose a new self-attention operation that divides matrices $\mathcal{Q}$, $\mathcal{K}$, and $\mathcal{V}$ into several slots, each of which represents a sentence or a group of sentences. We let $\mathcal{Q}_i$, $\mathcal{K}_i$, and $\mathcal{V}_i$ to denote each slot. The new self-attention operation can be formally described as

$$\text{Att\_CB\_S}(\mathcal{Q}, \mathcal{K}, \mathcal{V}) = \text{Concat}\left\{\text{softmax}\left(\frac{\mathcal{Q}_i\mathcal{K}_i^T}{\sqrt{d}} + \mathcal{M}_i\right)\mathcal{V}_i, \forall i\right\}. \tag{8}$$

The corresponding computing process is illustrated in Fig. 7. We omit the generation of $\mathcal{Q}$, $\mathcal{K}$, and $\mathcal{V}$ because it is similar with that in Fig. 6. These slots can be computed by GPU in parallel, with reduced redundancy compared with Fig. 6. Finally, we use a Concat() function to combine results to generate a single output. Note that multiple short requests can be concatenated in each slot if they do not exceed the slot length limit, just like what we have

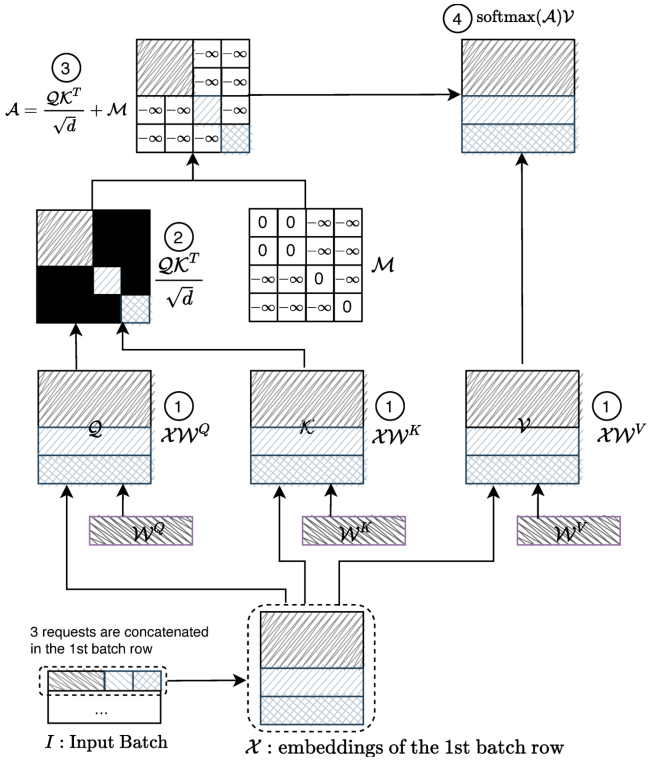


Figure 6: Customized self-attention for request concatenation. Step 1: The input feature matrix X is transferred into Q, K, and V matrices; Step 2: A score matrix is calculated by taking the dot product of Q and K^T; Step 3: The redundant parts of the score matrix are masked; Step 4: A softmax operation is conducted, followed by the multiplication with V.

done for pure request concatenation. The benefit of slotted request concatenation is to remove the redundancy among slots, which suggests that the redundancy may exist within each slot.

Slotted request concatenation brings additional challenge about slot length, which constrains the longest requests we can deal with in each batch. Moreover, slot length affects request scheduling and we will study it in the next section.

4.2.2 Early memory cleaning. During inference, a batch, no matter with or without request concatenation, and its intermediate data are kept in GPU memory until inference results are generated. After obtaining all results, we clean the GPU memory by removing data related to the current batch and load the next batch.

An important observation in our experiments is that inference results of requests in a batch are generated at different time because the decoder is an auto-regressive model. A straightforward idea to improve GPU memory efficiency is to remove the data of requests whose results are generated, instead of waiting the completion of whole batch inference. In such a way, GPU memory cleaning overlaps with inference process, bringing additional performance improvement.

Unfortunately, this idea does not work under ConcatBatching. The memory management operations, e.g., allocating or deleting,

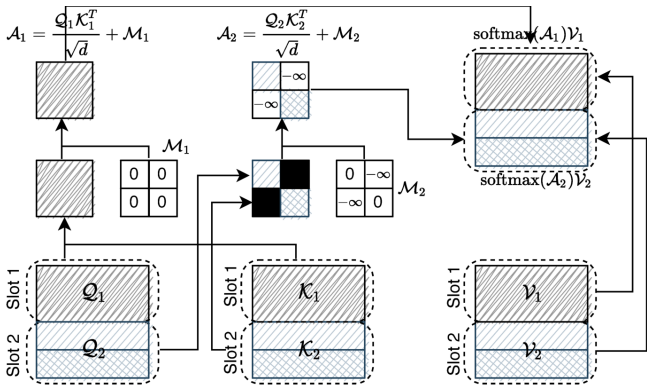


Figure 7: Slotted self-attention for request concatenation. The generation of Q, K, and V matrices is similar with that in Fig. 6 and we omit it for space saving. Different slots can run self-attention computation in parallel.

should be conducted in units of tensors. In ConcatBatching, request data do not aligned and we cannot separate the ones whose results are generated.

Slotted ConcatBatching provides the chances of early memory cleaning. Slots are independent and they can be easily separated into different tensors. We remove data of slots after generating their inference results. Meanwhile, released GPU memory can be allocated to the next batch, so that its data loading can overlap with the current batch's inference.

5 REQUEST SCHEDULING

The scheduler receives requests from user applications, packs them into batches and sends batches to the inference engine. The core of this module is a scheduling algorithm that decides which requests should be packed into a batch. In this section, we first formulate a scheduling problem and then present an algorithm with theoretical performance guarantee. Note that this is a pluggable module and the algorithm can be flexibly replaced with other ones with different goals.

5.1 Problem Statement

We consider a set N of requests that arrive in different time. Each request $n \in N$ is associated with an arriving time a_n, a deadline d_n, and a sentence of length l_n. Each request can only be scheduled after its arrival and before its deadline. The TCB's running time can be divided into slots, each of which corresponds the inference time of a batch. Each batch has B rows and each row can accommodate at most L words. Note that B and L are system parameters that can be adjusted. By fixing B and L, all batches have the same workloads and their inference time is similar.

We define a variable x_n^{tk} to denote whether request $n \in N$ is put into the k-th row of the batch scheduled at time t. We define the utility value of request n as $v_n \triangleq \frac{1}{l_n}$. A request fails if it is not scheduled before its deadline, and its utility value is zero. With the objective of maximizing the total utility of scheduled requests, we

Algorithm 1 Online Deadline-Aware Scheduling Algorithm (DAS)

function DAS(N_t, L)
$H_{tk} \leftarrow \emptyset$, for $k = 0, 1, ..., B-1$
for each batch row $k = 0, 1, ..., B-1$ **do**
  if $\sum_{n \in N_t} l_n \leq L$ **then**
    Put N_t into the current batch row;
  else
    Sort requests in N_t according to their utility in a non-increasing order to generate a sequence $\widetilde{N}_t$;
    Compute s_{tk}, so that the first s_{tk} request in sequence $\widetilde{N}_t$ can saturate the current batch row;
    $\widetilde{N}_t^U \leftarrow$ the first p_{tk} tasks in $\widetilde{N}_t$, where $p_{tk} \leq s_{tk}$;
    Put $\widetilde{N}_t^U$ into the current batch row;
    $\widetilde{N}_t^D \leftarrow$ tasks in $N_t - \widetilde{N}_t^U$ whose utility is no less than $q\bar{v}(\widetilde{N}_t^U)$;
    Sort requests in $\widetilde{N}_t^D$ according to deadlines and greedily put them into the current batch;
    if $\widetilde{N}_t^D$ cannot saturate the current batch **then**
      Greedily put rest requests in $\widetilde{N}_t - \widetilde{N}_t^U - \widetilde{N}_t^D$;
    end if
  end if
end for

formulate a scheduling problem as follows.

$$\max \sum_{n \in N} v_n \Big(\sum_{t \in T} \sum_{k=0}^{B-1} x_n^{tk}\Big) \tag{9}$$

$$\text{s.t.} \sum_t \sum_k x_n^{tk} \leq 1, \forall n \in N, \tag{10}$$

$$\sum_n l_n x_n^{tk} \leq L, \forall k \in [0, B-1], t \in T, \tag{11}$$

$$x_n^{tk} = 0, \forall n \in N \text{ and } t \notin [a_n, d_n], \tag{12}$$

$$x_n^{tk} \in \{0, 1\}, \forall k \in [0, B-1], n \in N, t \in T. \tag{13}$$

Constraint (10) indicates that each request can be processed once at most. Constraint (11) means that the total length of requests concatenated in each row cannot exceed the limit L. The final constraint (12) represents that task n can not be scheduled out of its available time interval $[a_n, d_n]$. The above formulation is a mixed-integer linear programming, which is in general NP-hard, even with information of all requests. In practice, we have no knowledge about future requests, which motivates us to design an online heuristic algorithm.

5.2 Scheduling Algorithm for Pure ConcatBatching

A straightforward idea to maximize total utility is to always schedule requests with higher utility. However, requests are also constrained by their deadlines, and some urgent ones with lower utility may loss scheduling chances. Therefore, we are motivated to design an algorithm by jointly considering utility values and deadlines.

The proposed scheduling algorithm, called DAS, is described as a function whose pseudo codes are shown in Algorithm 1. This function is invoked at the beginning of each time slot t. Fed by a set N_t of waiting tasks, this function returns a batch H_t of requests that are sent to GPU for inference in time slot t. Note that requests that have arrived but been not scheduled by their deadlines are excluded from N_t.

For each batch row k, this function initializes H_{tk} as an empty set and then chooses requests. If all requests in N_t can be accommodated into the current batch row, i.e., $\sum_{n \in N_t} l_n \leq L$, we put them into H_{tk} and return it to finish this function. Otherwise, we need to select a subset of requests from N_t, which could be more challenging since we need to jointly consider request utility and deadlines, as motivated above. We sort requests in N_t according to their utility values in a non-increasing order to generate a sequence $\widetilde{N}_t$. The first s_{tk} requests in $\widetilde{N}_t$ can be accommodated in the current batch row. We divide $\widetilde{N}_t$ into three parts, each of which represents different preference on utility or deadlines, as illustrated in Fig. 8. The first part, denoted by $\widetilde{N}_t^U$, consists of the first p_{tk} tasks in $\widetilde{N}_t$, where $p_{tk} = \eta s_{tk}$ and $\eta \in (0, 1)$ is a system parameter. We also call $\widetilde{N}_t^U$ a utility-dominant set since we choose them because of their high utility values. Note that η is a tunable system parameter and $p_{tk} \leq s_{tk}$.

The second part, which is denoted by $\widetilde{N}_t^D$, includes rest requests whose utility is no less than $q\bar{v}$, where $\bar{v}$ is the average utility of requests in $\widetilde{N}_t^U$ and $q \in (0, 1)$ is another tunable system parameter. The $\widetilde{N}_t^D$ is referred to deadline-aware set, since we choose requests from it by a deadline-preference strategy. To guarantee the total utility, we let $\eta + q = 1$. Specifically, we sort requests in $\widetilde{N}_t^D$ according to their deadlines and greedily choose the ones with closer deadlines. If $\widetilde{N}_t^D$ can saturate the current batch, the algorithm returns. Otherwise, we continue to pick requests from the rest set, i.e., $\widetilde{N}_t - \widetilde{N}_t^U - \widetilde{N}_t^D$.

Theorem 5.1. *Algorithm 1 is $\frac{\eta q}{\eta q + 1}$-competitive.*

Proof. Some key steps of proving this theorem are as follows. Due to the space limitation, we put the complete proof in our technical report [1].

Step 1: Dual problem setup. According to Fenchel duality [4], we can write the dual problem of the primal problem (9) as follows:

$$\begin{aligned}\mathcal{D}(\lambda) &= \max_{x \in \mathbf{X}} \sum_{n,t,k} x_n^{tk} \lambda_n^{tk} - \min_{x \in \mathbf{X}} \Big\{ \sum_{n,t,k} x_n^{tk} \lambda_n^{tk} - \sum_n v_n \Big(\sum_{t,k} x_n^{tk}\Big) \Big\} \\ &= \max_{x \in \mathbf{X}} \Big\{ \sum_{n,t,k} v_n x_n^{tk} - \sum_{n,t,k} x_n^{tk} \lambda_n^{tk} \Big\} + \max_{x \in \mathbf{X}} \sum_{n,t,k} x_n^{tk} \lambda_n^{tk} \qquad (14) \\ &\leq \sum_{n: v_n \geq \lambda_n} (v_n - \lambda_n) + \max_{x \in \mathbf{X}} \sum_{n,t,k} x_n^{tk} \lambda_n \qquad (15)\end{aligned}$$

where $v_n = \frac{1}{l_n}$, as defined in Section 5.1, $\mathbf{X}$ is the domain of x limited by constraints (10)-(12), and λ is the dual variable. Since x_n^{tk} is a binary variable and $\sum_{t,k} x_n^{tk} \leq 1$, we can set $\lambda_n^{tk} = \lambda_n$ for all t and k.

We have (15) since $\sum_{n: v_n < \lambda_n} (v_n - \lambda_n) \sum_{t,k} x_n^{tk} \leq 0$ and $\sum_{t,k} x_n^{tk} \leq 1$.

Step 2: Problem conversion. To obtain the competitive ratio α, we need to prove:

$$ALG \geq \alpha OPT, \tag{16}$$

where ALG is the solution of DAS and OPT is the optimal solution. However, its hard to obtain the optimal solution due to the NP-hardness of primal problem. Thanks to the dual property that each feasible solution λ of the dual problem gives an upper bound of the optimal solution, i.e., $OPT \leq \mathcal{D}(\lambda)$, we can converse the above problem to find the α as:

$$\mathcal{D}(\lambda) \leq \frac{1}{\alpha} ALG \tag{17}$$

Thereby, we converse the proof problem to find the α to satisfy (17).

Step 3: Scaling proof. Now we introduce how to find the relationship between $\mathcal{D}(\lambda)$ and ALG. The key idea is setting the suitable dual parameter λ to build the relationship between $\mathcal{D}(\lambda)$ and ALG. Concisely, we set $\lambda_n \leq v_n$ for each task n in H and $\lambda_n \geq v_n$ for tasks not in H, where H is chosen task set by Algorithm 1. By doing this, we can replace $\sum_{n:v_n \geq \lambda_n}(v_n - \lambda_n)$ by $\sum_{n \in H}(v_n - \lambda_n)$ and build the relationship between $\mathcal{D}(\lambda)$ and ALG. Before introducing the specific settings of λ, we first give some necessary definitions.

As defined in Algorithm 1, we denote the the utility-dominant task set and deadline-aware task set in k-th row at time slot t by $\tilde{N}^U_{tk}$ and $\tilde{N}^D_{tk}$, respectively. Given a solution H_{tk} from Algorithm 1, we define $H_{tk}[1] = \tilde{N}^U_{tk} \cap H_{tk}$ and $H_{tk}[2] = H_{tk} - H_{tk}[1]$, for all k and t. We have $p_{tk} = |H_{tk}[1]|$, and we define $j_{tk} = |H_{tk}[2]|$. Then, we set the dual variables λ according to three cases as follows:

(1) For $n \in H_{tk}[1], \forall t, k$, we set $\lambda_n = 0$;
(2) For $n \in H_{tk}[2], \forall t, k$, we set $\lambda_n = \min\{v_n, \hat{\lambda}\}$, where $\hat{\lambda} = \max\{\lambda_n | n \in \tilde{N}^D_{tk} \text{ and } n \notin H_{tk}\}$;
(3) For $n \notin \cup_{t,k} H_{tk}$, we set $\lambda_n = v_n$.

Note that for each task $n \in \cup_{t,k} H_{tk}$, it has $v_n \geq \lambda_n$. For task n in $H_{tk}[2]$, it has $\lambda_n = 0$ when $\tilde{N}^D_{tk} - H_{tk} = \emptyset$, i.e., all tasks in deadline-aware set have been chosen. Moreover, we denote $\lambda^{tk}_{\max} = \max_n\{\lambda_n | n \in \tilde{N}_{tk}\}$ to represent the maximum of λ for available tasks for time slot t and row k, where $\tilde{N}_{tk}$ is the sorted available tasks set in k-th row at time slot t. Based on the given λ, the dual problem (15) can be rewritten as:

$$\begin{aligned}
\mathcal{D}(\lambda) &\leq \sum_{n:v_n \geq \lambda_n} (v_n - \lambda_n) + \max_{x \in \mathbf{X}} \sum_{n,t,k} x^{tk}_n \lambda_n \\
&= V - \sum_{n \in H} \lambda_n + \sum_{t,k} \frac{s_{tk}}{|H_{tk}|} |H_{tk}| \lambda^{tk}_{\max} \\
&= V + \sum_{t,k} \sum_{n \in H_{tk}} \left(\frac{s_{tk}}{|H_{tk}|} \lambda^{tk}_{\max} - \lambda_n\right)
\end{aligned} \tag{18}$$

where s_{tk} represents the first s_{tk} tasks in $\tilde{N}_{tk}$ can be accommodated in the current batch row. Since $\tilde{N}_{tk}$ is sorted by $v_n = \frac{1}{l_n}$ non-increasingly, s_{tk} means the maximum number of tasks in $\tilde{N}_{tk}$ can be accommodated. Moreover, $V = \sum_{n \in H} v_n$ is the total utility obtained from Algorithm 1. Then we let $\beta_{tk} = \frac{s_{tk}}{|H_{tk}|}$ and $G_{tk} = \sum_{n \in H_{tk}} (\beta_{tk} \lambda^{tk}_{\max} - \lambda_n)$ for analyzing the relationship between G_{tk} and V_{tk}, where $V_{tk} = \sum_{n \in H_{tk}} v_n$. According to the setting of λ, we can prove $\lambda^{tk}_{\max} \leq \bar{v}(\tilde{N}^U_{tk})$. The proof details can be found in our technical report [1].

According to the relationship between s_{tk} and $|\tilde{N}^U_{tk}| + |\tilde{N}^D_{tk}|$, we have $\frac{G_{tk}}{V_{tk}} \leq \frac{s_{tk}}{p_{tk} q} = \frac{1}{\eta q}$, where $\eta = \frac{p_{tk}}{s_{tk}}$ is the tunable system parameter. Then we have:

$$\begin{aligned}
\mathcal{D}(\lambda) &\leq V + \sum_{t,k} \sum_{n \in H_{tk}} \left(\frac{s_{tk}}{|H_{tk}|} \lambda^{tk}_{\max} - \lambda_n\right) \\
&= (1 + \frac{1}{\eta q}) V = (\frac{\eta q + 1}{\eta q}) ALG
\end{aligned} \tag{19}$$

which completes the proof. □

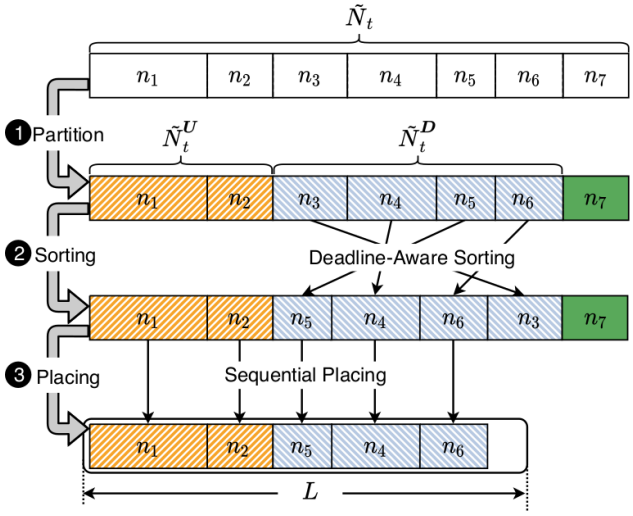


Figure 8: Three cases when scheduling tasks in Algorithm 1

Note that η and q are two tunable system parameters. When $\eta = q = \frac{1}{2}$, we get the $\frac{1}{5}$-competitive ratio.

5.3 Scheduling Algorithm for Slotted ConcatBatching

Compared to pure request concatenation, the algorithm design for slotted request concatenation is more challenging because batching decisions are affected by slot size, which becomes a new optimization dimension. The pure request concatenation can be treated as a special case whose slot size is equal to the batch length L. A smaller slot can eliminate more computational redundancy, but can accommodate less requests since the ones larger than the slot would be discarded. In this section, we propose a simple but effective heuristic algorithm for slotted request concatenation, whose pseudo codes are shown in Algorithm 2. We first invoke Algorithm 1 to obtain a set of candidate tasks, i.e., $\{H_{tk}\}$, for each batch row.

The slot size is defined as the largest length of requests in the utility-dominant set H^U, as shown in lines 3 - 4. Therefore, no request in $\tilde{N}^U_t$ would be discarded because of slot size limit. After deciding the slot size, we greedily put requests into slots.

6 PERFORMANCE EVALUATION

In this section, we first introduce our experimental settings. We then study overall performance and the influence of critical system modules and parameters.

Algorithm 2 Slotted Online Deadline-Aware Scheduling Algorithm

1: **Function** Slotted_DAS(N_t, L)
2: $\{H_{tk}\} \leftarrow$ Invoke DAS(N_t, L);
3: Generate the utility-dominant set H^U by selecting the first p_{tk} tasks in $\{H_{tk}\}$;
4: Compute the slot size m according to the largest length of tasks in H^U;
5: Divide each row of the batch into multiple slots based on the slot size z;
6: **for** each batch row $k = 0, 1, ..., B-1$ **do**
7: Put tasks in H_{tk} into slots greedily;
8: **end for**

6.1 Experimental Settings

We conduct performance evaluation on an AWS instance p3.2xlarge with an NVIDIA Tesla V100 GPU, a Intel Xeon E5-2686v4 CPU, and 61 GiB memory. We use a Seq2Seq encoder-decoder model [32], which uses 3 encoders and 3 decoders with a hidden dimension of 3072 ($d_{model} = 3072$). The number of self-attention heads is 8. The maximum sentence length that can be processed is 400 words. We compare TCB with two baselines.

- TNB (Transformer with NaiveBatching): TNB uses the default setting of PyTorch [26]. Requests of different lengths are batched together. Each batch line is dedicated to a request, and short requests are padded with zeros to aligned with the longest one in the batch, as illustrated in Fig. 1(a).
- TTB (Transformer with TurboBatching): TTB uses the batching scheme adopted by TurboTransformer [14], which batches requests of similar length to reduce padded zeros, as shown in Fig. 1(b). Note that a complete implementation of TurboTransformer contains many computational and memory optimization schemes, which are orthogonal to our work. They can be also applied in TCB for further performance improvement. Here we mainly focus on studying request batching schemes and its influence to inference efficiency.

Note that both TNB and TTB are not associated with any scheduling algorithms for online services in existing work. For fair comparison, we feed them with the same scheduling results generated by the DAS algorithm proposed in Section 5.2.

6.2 Results

6.2.1 Performance under different request rates. We randomly generate requests with 3−100 tokens according to a normal distribution and they arrive as a Possion process. The batch size is set to 64 for TNB and TCB. We follow [14] to use a dynamic programming method to determine the batch size for TTB. As shown in Fig. 9, system utility increases as arriving rate grows for all systems. The utility of TNB and TTB has no big change when there are more than 350 requests/second, which means that both systems have achieved their maximum processing capability. This phenomenon is also called system saturation. TCB has shown larger capacity by handling 450 requests/second. After saturation, total utility of TCB is higher than TNB and TTB by 2.20**x** and 1.29**x**, respectively. We also show system throughput in Fig. 10, where we have similar observation that TCB always outperforms other two systems. The maximum performance gaps with TNB and TTB are about 2.22**x** and 1.48**x**, respectively.

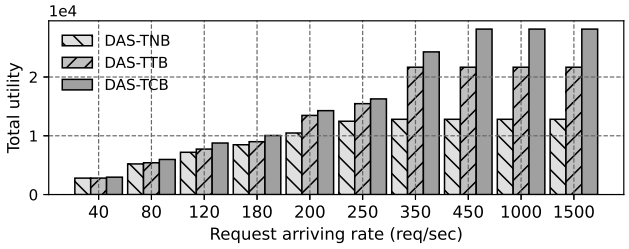


Figure 9: Utility under different request rates (input length 3-100, average 20, variance 20, batch size 64).

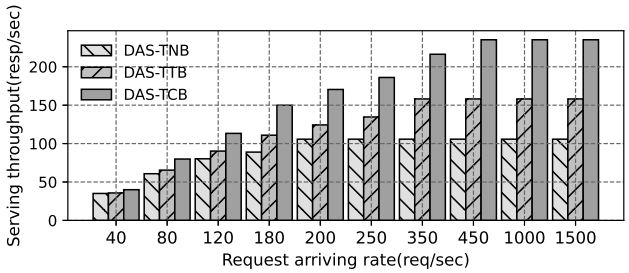


Figure 10: Serving throughput under different request rates (input length 3-100, average 20, variance 20, batch size 64).

6.2.2 Study of inference engine efficiency. We then study system performance under a simple first-come-first-served (FCFS) scheduling policy. This set of experiments eliminate the influence of our designed scheduling algorithm, so that we can better show the benefits of request concatenation adopted by TCB's inference engine. We first set the variance of request length to 20 and show results in Fig. 11. Thanks to request concatenation, the maximum throughput of TCB is higher than TNB and TTB by 3.33**x** and 1.52**x**, respectively. We then change the request length variance to 100 and show results in Fig. 12. The maximum performance gap between TCB and TTB increases to 1.72**x**. That is because higher variance means that incoming requests show more variability in length and it would be harder for TTB to find sufficient number of requests with similar lengths. Moreover, in both figures, all systems are saturated at lower request arriving rates, compared to Fig. 10 using our DAS algorithm. It demonstrates the benefits of our scheduling algorithms, which will be further studied in later experiments.

6.2.3 Speedup of slotted ConcatBatching. We first set the batch size as 10 and measure the average batch inference time of TCB with pure and slotted ConcatBatching, respectively. The speedup under different number of slots is shown in Fig. 13. Note that pure ConcatBatching can be treated as a special case of the slotted scheme with a single slot and the corresponding speedup is 1. As we use more slots, we can obtain at most about 1.18x speedup. We then increase batch size to 32 and show results in Fig. 14. The maximum speedup is 2.31**x** when there are 7 slots. There is no big performance growth

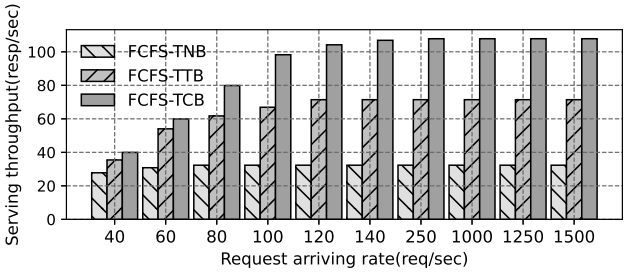


Figure 11: Serving throughput under different request rates when using FCFS (input length 3-100, average 20, variance 20, batch size 64).

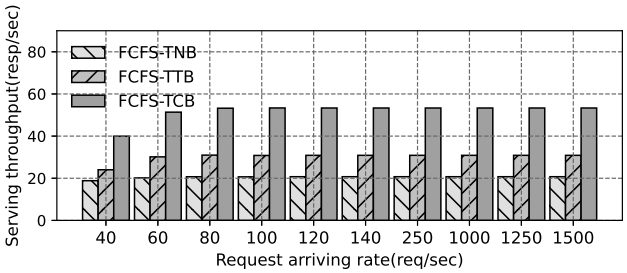


Figure 12: Serving throughput under different request rates when using FCFS (input length 3-100, average 20, variance 100, batch size 64).

when we further increase the number of slots. The results demonstrate that slotted ConcatBatching can reduce more redundancy under larger batch size.

6.2.4 Influence of different scheduling algorithms. To show the superiority of our proposed DAS algorithm, we equip TCB with three popular scheduling algorithms, i.e., short-job-first (SJF), first-come-first-served (FCFS) and deadline-early-first (DEF). The results under different batch sizes are shown in Fig. 15(a). Note that we use the same TCB inference engine for all algorithms. The utility increases as the batch size grows for all algorithms because larger batch size can accommodate more requests. DAS-TCB outperforms others at all batch sizes. We then fix the batch size to 16 and study the affect of request length variances. As shown in Fig. 15(b), DAS-TCB has obvious improvement in utility compared with other scheduling algorithms, which demonstrates that DAS-TCB is aware of requests of variable lengths and makes better scheduling. We finally change the batch row length, i.e., the parameter L, and show results of different scheduling algorithms in Fig. 15(c). DAS-TCB has about 40% higher utility than SJF-TCB and more than others.

6.2.5 Overhead of DAS algorithm. We finally evaluate the overhead of DAS algorithm by measuring its running time and show the ratio to a single batch inference time in Fig. 16. As request arriving rate increases, the ratio grows because DAS needs to sort and schedule more requests. However, the ratio is only 2% when there are 400 requests/seconds, which already saturate the GPU.

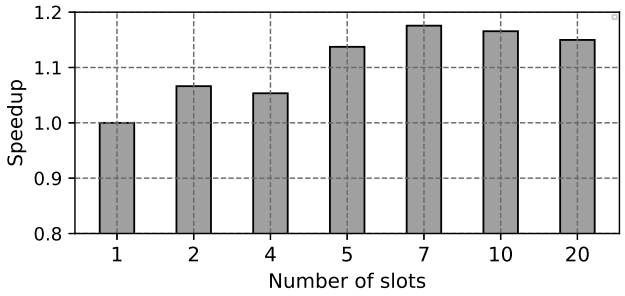


Figure 13: Speedup of slotted ConcatBatching. (batch size 10, length 400)

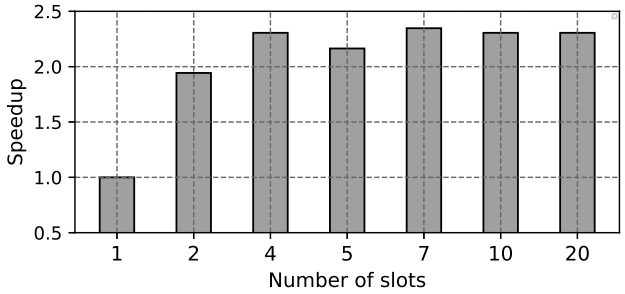


Figure 14: Speedup of slotted ConcatBatching. (batch size 32, length 400)

7 CONCLUSION

This paper targets on improving the efficiency of transformer inference systems. We have identified two major weaknesses of existing work. First, there exists high computation redundancy in existing request batching schemes. Second, request scheduling and batching is separate, missing the joint optimization chances. This paper fills this gap by developing TCB, a transformer inference system that integrates two novel techniques. We propose ConcatBatching, as a new batching scheme that concatenates requests in a batch to reduce computation redundancy. A online request scheduling algorithm aware of the ConcatBatching is designed to maximize the utility of TCB and its theoretical bound is derived. Extensive experiments have been conducted to evaluate TCB and show its superiority over state-of-the-art.

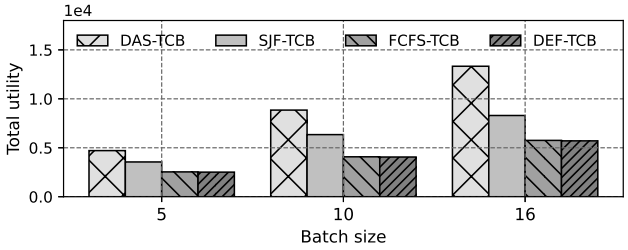
ACKNOWLEDGMENTS

This research was supported by JSPS KAKENHI (No. 21H03424), NSF of China (Grant No.62172375), Open Research Projects of Zhejiang Lab (No. 2021KE0AB02). Peng Li is the corresponding author.

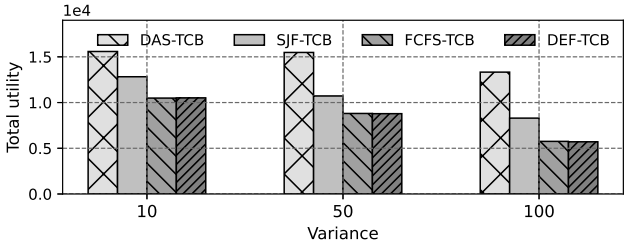
REFERENCES

[1] 2022. Technical Report. https://www.dropbox.com/s/ynhefllui4e8vgm/Technical_Report_TCB.pdf?dl=0

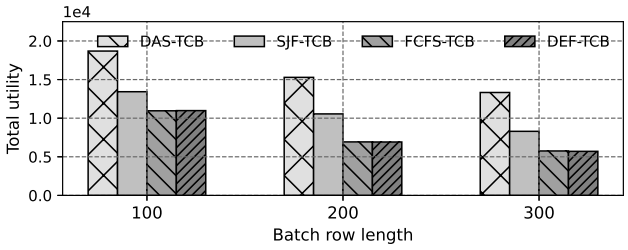
[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. {TensorFlow}: A System for {Large-Scale} Machine Learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.



(a) Utility under different batch sizes.



(b) Utility under different variances.



(c) Utility under different input lengths

Figure 15: Utility under different batch sizes, variances, and input lengths.

Figure 16: The ratio of DAS running time and single batch inference time.

[3] Marta Bañón, Pinzhen Chen, Barry Haddow, Kenneth Heafield, Hieu Hoang, Miquel Esplà-Gomis, Mikel L Forcada, Amir Kamran, Faheem Kirefu, Philipp Koehn, et al. 2020. ParaCrawl: Web-scale acquisition of parallel corpora. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 4555–4567.

[4] Dimitri P. Bertsekas. 1999. *Nonlinear Programming, 2nd Edition.* Athna Scientific, 1 Chestnut St, Ste 222, Nashua NH 03060, USA.

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[6] Shiyang Chen, Shaoyi Huang, Santosh Pandey, Bingbing Li, Guang R Gao, Long Zheng, Caiwen Ding, and Hang Liu. 2021. ET: re-thinking self-attention for transformer models on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–18.

[7] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259* (2014).

[8] Yujeong Choi, Yunseong Kim, and Minsoo Rhu. 2021. Lazy Batching: An SLA-aware batching system for cloud machine learning inference. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 493–506.

[9] Yujeong Choi and Minsoo Rhu. 2020. Prema: A predictive multi-task scheduling algorithm for preemptible neural processing units. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 220–233.

[10] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A {Low-Latency} Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 613–627.

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

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

[13] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. 2020. Gslice: controlled spatial sharing of gpus for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 492–506.

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

[15] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 443–462.

[16] Tae Jun Ham, Yejin Lee, Seong Hoon Seo, Soosung Kim, Hyunji Choi, Sung Jun Jung, and Jae W Lee. 2021. ELSA: Hardware-Software co-design for efficient, lightweight self-attention mechanism in neural networks. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 692–705.

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

[18] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. 2021. Data movement is all you need: A case study on optimizing transformers. *Proceedings of Machine Learning and Systems* 3 (2021), 711–732.

[19] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. 2020. Nimble: Lightweight and parallel gpu task scheduling for deep learning. *Advances in Neural Information Processing Systems* 33 (2020), 8343–8354.

[20] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2019. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942* (2019).

[21] Baolin Li, Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, Karen Gettings, and Devesh Tiwari. 2021. RIBBON: cost-effective and qos-aware deep learning model inference using a diverse pool of cloud computing instances. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[22] Yang Li, Zhenhua Han, Quanlu Zhang, Zhenhua Li, and Haisheng Tan. 2020. Automating cloud deployment for deep learning inference of real-time online services. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 1668–1677.

[23] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. 2021. Terapipe: Token-level pipeline parallelism for training large-scale language models. In *International Conference on Machine Learning*. PMLR, 6543–6552.

[24] Daniel Mendoza, Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. Interference-aware scheduling for inference serving. In *Proceedings of the 1st Workshop on Machine Learning and Systems*. 80–88.

[25] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on GPU clusters using megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[26] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019), 8026–8037.

[27] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

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

[29] EF Tjong Kim Sang and F De Meulder. 2003. Introduction to the CoNLL-2003 Shared Task: Language-Independent Named Entity Recognition. In *Proceedings of CoNLL-2003, Edmonton, Canada*. Morgan Kaufman Publishers, 142–145.

[30] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*. 1631–1642.

[31] Jacob R Stevens, Rangharajan Venkatesan, Steve Dai, Brucek Khailany, and Anand Raghunathan. 2021. Softermax: Hardware/Software Co-Design of an Efficient Softmax for Transformers. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 469–474.

[32] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

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

[34] Yu Yan, Jiusheng Chen, Weizhen Qi, Nikhil Bhendawade, Yeyun Gong, Nan Duan, and Ruofei Zhang. 2021. El-attention: Memory efficient lossless attention for generation. In *International Conference on Machine Learning*. PMLR, 11648–11658.

[35] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. 2019. Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems* 32 (2019).

[36] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. {MArk}: Exploiting Cloud Services for {Cost-Effective},{SLO-Aware} Machine Learning Inference Serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 1049–1062.

[37] Minjia Zhang and Yuxiong He. 2020. Accelerating training of transformer-based language models with progressive layer dropping. *Advances in Neural Information Processing Systems* 33 (2020), 14011–14023.

[38] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, et al. 2022. AStitch: enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 359–373.