Efficient Adaptive Batching of DNN Inference Services for Improved Latency

Osama Khan*, Junyeol Yu†, Yeonjae Kim†, Euiseong Seo†
*Dept. of AI System Engineering, Sungkyunkwan University, Suwon, Republic of Korea
Email: khan980@g.skku.edu
†Dept. of Computer Science and Engineering, Sungkyunkwan University, Suwon, Republic of Korea
Emails: {junyeol.yu, yeonjae.kim, euiseong}@skku.edu

***Abstract*—With the rising success of deep neural network (DNN) applications, GPU servers are increasingly utilized to provide DNN inference services. Batching, in which multiple incoming requests are grouped together for simultaneous processing, is a common practice for the DNN inference serving systems to enhance GPU utilization and thus lower the service cost. While batching enhances GPUs' efficiency, it can extend the end-to-end latency for each individual request. This delay is especially noticeable when requests are infrequently incoming to form a complete batch. On the other hand, when the request rate is high, a consistently small batch size may not fully harness the GPU's capacity. Handling requests one by one might decrease this latency but also raise energy consumption. Therefore, it's crucial to strike a balance between energy efficiency and latency, optimizing the batch size to ensure timely processing without compromising performance under varying request rates. Many existing solutions aim to fully use user-set deadlines, which, even though they maximize, can heighten perceived latency, particularly in larger pipelines. In this paper, we introduce a dynamic batching strategy tailored to adapt batch sizes in real-time for varying request rates. Our method aims to balance the time taken to assemble a batch and the time required to execute it, ensuring efficient energy consumption and prompt response times. By dynamically adjusting batch size relative to the current request rate, we strive to ensure that the collection duration for a batch coincides with its execution duration. This will minimize the latency as much as possible while ensuring we utilize the batching to provide an efficient solution. We built our approach using Triton, a platform for AI model deployment and execution. Our evaluation with four different DNN models shows that our technique achieved marginally close latency as the batching deployed by the Triton inference server and improved the energy efficiency by up to 5%.**

***Index Terms*—inference, batching**

I. Introduction

DNN inference applications are surging in popularity, leading to a heightened reliance on GPUs to meet the demands of such services. These GPUs excel in handling vast parallel operations, making them indispensable in inference applications. To harness their full potential in situations with multiple service requests, the batching technique has been introduced. By grouping multiple requests for simultaneous processing, batching boosts GPU utilization and efficiency [1].

When using batching, the choice of batch size becomes crucial in influencing both latency and the efficient use of resources. For example, a large batch size during periods of low demand can lead to increased latency because the server has to wait until a predefined number of requests are accumulated. On the other hand, because a small batch size underutilizes GPU's capability during peak times with high load, the reduced throughput from a small batch size may significantly retard the response time. It might seem advantageous to avoid the use of a predefined batch size and to repeatedly process all accumulated requests whenever the GPU is available. However, this approach diminishes the servers' energy efficiency, measured in requests per joule. Without batching, the parallel processing advantages of GPUs remain untapped. Consequently, adopting an adaptive strategy that dynamically adjusts batch size according to the prevailing request rate becomes essential for efficient inference services.

Several methods have been proposed to adapt batching for inference tasks with different goals. These methods either aim to maximize the throughput [1]–[3] or improve energy efficiency [4], [5]. However, many of these methods tend to fully utilize the allowable SLO (service-level objective) slack. This can affect the user experience as most of these inference services are interconnected and if each service consistently operates at the upper end of its SLO limit, the aggregate latency can escalate significantly, adversely affecting the overall latency experienced by the user. Moreover, certain approaches necessitate layer-level profiling and optimization [1], [4], which is unfeasible when preserving the model's structural integrity is crucial.

This paper presents an adaptive batching scheme aimed at reducing latency while ensuring efficient batching. In contrast to existing research, our approach offers a model-agnostic solution that adjusts batch sizes without additional overhead, prioritizing energy efficiency and user experience. Initially, we focus on separating the process of computing the optimal batch size for the incoming request rate from the actual dispatching process, i.e., combining the requests into a particular batch and sending it to the model. By separating these two processes, our goal is to eliminate the potential for computational delays in request handling caused by frequent calculations of batch size.

Consequently, our approach is made up of two parts: BatchMonitor and BatchDispatcher. The BatchMontior phase constantly assesses the rate of incoming requests and determines the optimal batch size that aligns with this rate.

Once this batch size is found for the given request rate, it is passed to the BatchDispatcher, which combines the requests into this particular batch size. Therefore, it is the job of the BatchMonitor to find the batch size for the given request rate and pass it to the BatchDispatcher.

BatchMonitor selects the batch size that will meet the throughput demand of the current request rate. For example, at a request rate of five requests per second, it identifies a batch size with a throughput of five requests per second. This ensures that a batch's processing time matches that particular batch's formation time at a given request rate. In other words, a batch size with a throughput below five requests per second will make the execution time of the batch a bottleneck, as the number of requests that can be formed becomes more than the number of requests that is processed by the current batch size. Conversely, a batch size exceeding this throughput risk increases the queueing time of the request as the time to create that batch also increases. Therefore, by equating the request rate with batch processing speed, we aim to eliminate the latency from long queueing time under a small request rate and long compute time under a heavy request rate. Finally, once the optimal size is discerned, it's relayed to the BatchDispatcher, which then collates requests into this preferred batch size.

We integrated our method with NVIDIA's Triton Server, an AI model-serving platform compatible with various deep learning and machine learning frameworks. For our assessment, we employed Triton's standard procedures. Specifically, Triton offers two primary methods: one where the maximum possible batch size is selected under the timing constraint and another where the requests are served as they arrive.

The remainder of the paper is as follows. Section II presents the background and motivation. Section III proposes our approach, and Section IV evaluates it. Section V introduces the related work, and Section VI concludes our research.

II. Background and Motivation

In batching, multiple inputs are combined together into a single input. The batched input has a larger dimension than the single input, providing more opportunities for GPUs to utilize their large number of computational cores simultaneously. As a result, batching helps increase the overall throughput [6] or energy efficiency of these GPUs [5].

However, the benefits of batching come at the expense of increased latency [2]. The trade-off between the benefits obtained from batching and the increased latency is controlled by the batch size we aim to form. In the case of inference applications, if the rate of incoming requests to these applications is slow, it will take more time to form a large batch size. On the other hand, if the rate is high and the batch size we selected is small, we will lose the opportunity to fully utilize the GPUs.

Therefore, instead of directly selecting a batch size, given the time constraint on the latency of the request, a timeout is usually set when creating the batch size [1]–[3], [5]. The timeout puts an upper bound on the wait time so the system can form the maximum possible batch size. The timeout is usually set according to the SLO constraint of the inference application, such that after executing the batch, none of the requests in that batch violate the SLO.

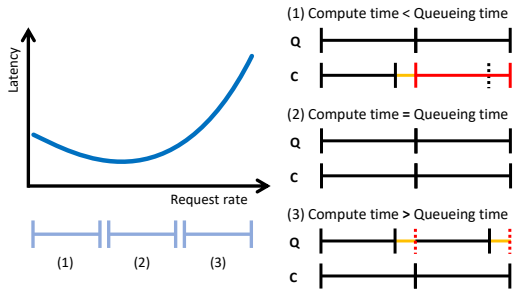


Fig. 1: Latency as a function of request rate for batch size N, highlighting three distinct cases based on the relationship between the queueing time (Q) and compute time (C).

While having a timeout ensures that we process the maximum possible batch size under the SLO for a given request rate, utilizing the available slack time created by a longer SLO than the execution time increases the perceived latency in inference applications, where they are part of large interconnected services. In these situations, if every inference service tends to utilize its SLO, the final cumulative latency increases. Furthermore, this negatively affects the user experience. Therefore, it is essential to create a batch based on the incoming request rate that will minimize the latency as much as possible while utilizing the benefits of batching.

III. Our Approach

Batching plays a pivotal role in optimizing GPU utilization. However, it's essential to modulate the batch size based on the influx of requests at any given moment. For instance, smaller batch sizes are more favorable during periods of low request rates. Choosing larger batch sizes in such situations could inadvertently increase the average latency for early-arriving requests. On the other hand, larger batch sizes should be employed during times of high request rates to enhance the system's throughput and maximize resource utilization.

We introduce an adaptive scheduling framework that allows efficient batching while keeping the latency as small as possible. Our approach is divided into two distinct components: BatchMonitor and the BatchDispatcher. BatchMonitor continuously assesses the rate of incoming requests. After computing the request rate, it then determines an optimal batch size, which will keep the latency to a minimum at the current request rate. BatchMonitor then passes this information to the BatchDispatcher which runs independently of the BatchMonitor. BatchDispatcher initially assembles a batch of size one. However, once notified of the recommended batch size from BatchMonitor, it swiftly adjusts, forming batches that align with the suggested size for processing.

A. Computation of Preferred Batch Size

To compute the optimal batch size for a given request rate, we need first to understand the relation between the batch size and the latency under different request rates. Figure 1 shows the situation where forming a batch size of N creates three different situations.

Compute time is less than queueing time. This suggests that our batch size is excessively large. With a slow incoming requests, we are left waiting to accumulate enough to form the desired batch, leading to prolonged queueing times.

Compute time equals queueing time. The batch of size N is processed in the time it takes to assemble it. As soon as we have a full batch ready, it's processed immediately, achieving the minimum latency.

Compute time surpasses queueing time. In this case, although requests pour in quickly, forming batches swiftly, the compute time remains unchanged. This means requests often have to wait their turn while the current batch is processed, turning compute time into a bottleneck.

When expressed mathematically, this relationship can be depicted as follows: Let's denote the batch size as B and the compute time for a batch of size B as T_b. The appropriate request rate for the batch size B, represented as R_b, is determined when the time taken for B number of requests to arrive matches with T_b. If the queueing time for the batch size B is given as Q_b, and if the requests come in slowly such that $Q_b > T_b$, then the minimum latency is T_b, while the maximum latency equals T_b+Q_b. Conversely, if requests come in swiftly and $Q_b < T_b$, the minimum latency is calculated as $T_b + a$, and the maximum latency is $T_b + Q_b + a$. Here, a represents the additional waiting time due to requests arriving during the computation of the preceding batch, causing a to gradually increase as time progresses. In the unique situation where $Q_b = T_b$, both minimum and maximum latency equate to $T_b + Q_b$. In all these situations, since T_b, the computing time for batch size B, remains consistent, it's evident that the average latency is minimized when Q_b aligns with T_b. Based on this, the correlation between the time taken for B requests to be received and the compute time of batch size B can be expressed as

$$R_b \leq B/T_b. \tag{1}$$

Therefore, for a certain request rate, we will choose the batch size that will make equation 1 true.

Algorithm 1 shows how BatchMonitor selects the preferred batch size and updates the variable inside the BatchDispatcher. The algorithm starts by initializing the request count to zero, the window to one second, and by loading the regression model to predict the execution time for a given batch size *(line 1-4)*. We select a linear regression model to predict the time for a given batch size as the batch execution time shows a linear relation with the increasing size of batch [5]. The BatchMonitor then starts a while loop and continuously updates the variable for the preferred batch size. A new loop is created during which we go through all the batch sizes *(line 7-15)*. Once the batch size that satisfies the equation 1 is found, the BatchMonitor breaks the loop and after updating the preferred batch size variable, goes to sleep for the remaining time window. The variable for request count is updated independently of the BatchMonitor, so even if the BatchMonitor goes to sleep, it does not affect the request count variable.

Algorithm 1 Computing Preferred Batch Size

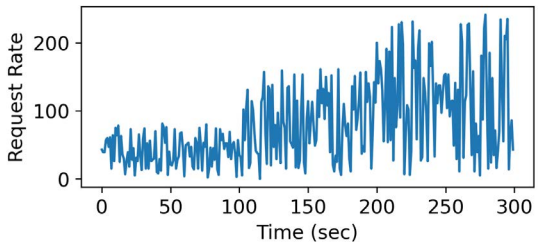
```
1:  request_count ← 0
2:  t ← 1                                  ▷ Time window in seconds
3:  pbs ← 1                                ▷ Preferred Batch Size
4:  regression                             ▷ Regression model
5:  while True do
6:      batch_size ← 2
7:      while batch_size ≤ max_batch_size do
8:          batch_time ← regression.predict(batch_size)
9:          throughput ← batch_size ÷ batch_time
10:         if request_count < throughput then
11:             pbs ← batch_size
12:             break
13:         end if
14:         batch_size ← batch_size + 1
15:     end while
16:     if batch_size == max_batch_size + 1 then
17:         pbs ← batch_size − 1
18:     end if
19:     request_count ← 0
20:     sleep(t)
21: end while
```

IV. Evaluation

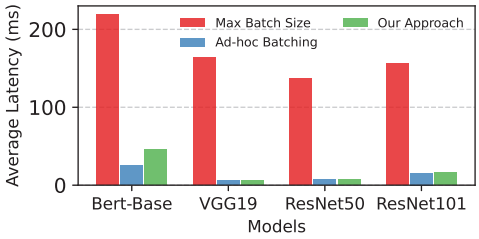
We integrated our approach with NVIDIA's Triton Server, an AI model-serving platform supporting a wide array of deep learning frameworks. We further enhanced Triton by introducing BatchMonitor and BatchDispatcher components. For the comparison, we choose four different models: BertBase, VGG19, ResNet50, and ResNet101.

We compared our method with two different approaches. The first approach creates a max batch size possible under the given deadline. Since this method keeps waiting until the edge, we create the largest batch size. However, the latency of this approach is the highest, as this method keeps waiting until the deadline is about to be crossed. The second approach, ad-hoc batching, is the opposite of the first. In this approach, we serve the requests as they come. If multiple request arrives at the same time, we create a batch and serve them. This approach ensures a minimum latency, but the energy efficiency is small. Figure 2 illustrates latency and energy efficiency distinctions, measured in "Requests per joule," between our approach, Ad-hoc batching, and the maximum batch size technique. For this evaluation, we use synthetic data. The data has the request rate range from one all the way up to 240 mimicking the real-world environment.

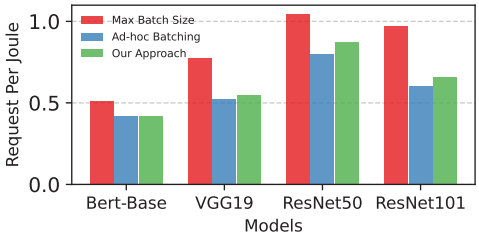
The result shows that our approach's latencies for Bert-Base, VGG19, ResNet50, and ResNet101 were 45.78ms, 6.11ms,



(a) Time Series Data



(b) Latency



(c) Energy

Fig. 2: (a) Request rate over a span of 300 seconds. (b) Average latency for different DNN models using three batching techniques: Max Batch Size, Ad-hoc Batching, and our approach. (c) Energy efficiency, measured as requests per joule

7.29ms, and 9.67ms, outpacing the max batch size technique by 14 times. Meanwhile, energy efficiency under our approach ranged from 0.46 to 0.97, offering a 5% edge over Ad-hoc batching. Unlike creating the maximum batch size possible, our approach's dynamic batch size adjustments optimize both speed and energy, underscoring its superior system efficiency.

V. Related Work

Batching is a widely used technique to leverage the parallel computational prowess of GPUs for inference tasks. Clipper introduced an adaptive approach, adjusting batch sizes based on SLO breaches, but struggles with varying request rates [2]. Mark and LazyBatching have their own methods to optimize GPU efficiency; the former sets strict rules on response and waiting times, while the latter operates at the layer-level, though it faces challenges with certain model types like RNNs [1], [3]. On serverless platforms, BATCH employs adaptive batching to decrease the instances needed for multiple requests by modeling request rates and corresponding latencies [7]. BatchDVFS and EAIS target energy efficiency in different ways: the former balances power consumption with batch size and GPU frequency, while the latter uses regression models to optimize energy efficiency within SLO bounds, though the latter's method can be time-intensive [5], [6]. Despite these techniques successfully optimizing throughput or energy, they often use up the entire slack of the SLO. This can raise overall response times when these services are part of a pipeline, impacting user experience. Contrarily, our approach offers a model-agnostic, simple, and efficient way to adapt batch sizes in alignment with the request rate, bypassing the complexities of traditional methods.

VI. Conclusion

Batching is an important technique to utilize the GPU potential and increase energy efficiency. However, the fluctuations in inference request rates introduce a trade-off between the benefits that can be gained from batching and the latency. While utilizing the slack time fully to increase the batch size ensures a large batch size, it affects the latency, especially in cases where the inference services are chained to each other. Therefore, in these situations, It is essential to reduce the overall latency as much as possible for a smoother user experience. In this paper, we develop an adaptive batching that allows us to create a batch size depending on the request rate. Our key motivation is that for every request rate, there exists a batch size that results in minimum latency. Choosing this batch size can ensure that we meet the minimum latency while also utilizing the benefits of batching. Our evaluation with four different DNN models shows that our technique achieved almost similar latency as the Ad-hoc batching and improved the energy efficiency by up to 5%.

Acknowledgement

This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2023-RS-2023-00258649) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation).

References

[1] Y. Choi, Y. Kim, and M. Rhu, "Lazy batching: An SLA-aware batching system for cloud machine learning inference," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 493–506.

[2] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A Low-Latency online prediction serving system," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 613–627.

[3] C. Zhang, M. Yu, W. Wang, and F. Yan, "MArk: Exploiting cloud services for Cost-Effective,SLO-Aware machine learning inference serving," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 1049–1062.

[4] C. Yao, W. Liu, Z. Liu, L. Yan, S. Hu, and W. Tang, "Eali: Energy-aware layer-level scheduling for convolutional neural network inference services on gpus," *Neurocomputing*, vol. 507, pp. 265–281, 2022.

[5] C. Yao, W. Liu, W. Tang, and S. Hu, "Eais: Energy-aware adaptive scheduling for cnn inference on high-performance gpus," *Future Generation Computer Systems*, vol. 130, pp. 253–268, 2022.

[6] S. M. Nabavinejad, S. Reda, and M. Ebrahimi, "Coordinated batching and dvfs for dnn inference on gpu accelerators," *IEEE transactions on parallel and distributed systems*, vol. 33, no. 10, pp. 2496–2508, 2022.

[7] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Batch: Machine learning inference serving on serverless platforms with adaptive batching," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.