M.S. THESIS

Efficient Resource Scaling Policy in Inference Serving of Natural Language Generation Models

자연어 생성 모델 추론 서비스의 효율적인 자원 스케일링 정책

2022 년 7 월

서울대학교 대학원

컴퓨터 공학부

조 성 우

Efficient Resource Scaling Policy in Inference Serving of Natural Language Generation Models

자연어 생성 모델 추론 서비스의 효율적인 자원 스케일링 정책

지도교수 전 병 곤

이 논문을 공학석사학위논문으로 제출함

2022 년 7 월

서울대학교 대학원

컴퓨터 공학부

조 성 우

조성우의 공학석사 학위논문을 인준함

2022 년 7 월

Abstract

Though number of different types of Deep Neural Network (DNN) models are increasing, language generation model is still the most in demand. There is also an increasing demand for serving the pre-trained model. However, managing computing resources in serving Natural Language Generation (NLG) model is not a trivial problem, because requests and responses of each query is different due to a variety of environment. Moreover, it is even more challenging to decide scaling policy, which minimizes both violation of service level objective (SLO) and GPU resource usage. In this paper, we discuss the problem of using efficient GPU resources in serving language generation model, and propose a design a serving framework which supports fast and accurate scaling policy. We implemented an deep learning inference serving framework with policy and validated our system on the serving request query workloads.

Keywords: Deep Learning, Serving, NLP, NLG, GPU, Resource Management, Scaling, Inference Engine

Student Number: 2018-23073

Contents

Chapter 1

Introduction

While Deep Neural Network (DNN) models have rapidly grown [1, 2, 3, 4, 5], it's focus was mainly on serving and training the model. Especially in Natural Language Generation (NLG), the size of model parameter have grown, starting from BERT [1] to GPT-3 [4]. A lot of services intend to use these trained NLG models in many ways: Automatic response in chat-bot, smart assistance, text automatic generation, and so on.

Distributed training is essential to train those large DNN models quickly in shared GPU clusters. GPU resource management system and the job scheduling system are essential for efficient learning that saves costs. There were researches about resource management in shared GPU clusters [6, 7, 8]. These systems schedule training jobs and their resources to grow and to shrink dynamically. As such, the resource management, scheduling, and scaling system realted to DNN model training has been researched enough.

Though the resources used for model inference serving is less than the resources used for training models, GPU resource management system for serving model is needed as the demand increases. In traditional machine learning (ML), there are multiple Machine Learning as a Service (MLaaS) platforms such as Google Vertex AI, Amazon ML, and Microsoft Azure ML [9, 10, 11], which already support resource management system in ML.

However, it is difficult to use the mechanism or policy used in NLG model inference serving because it has different properties from the training and traditional machine learning service. In the case of training, if the input batch size does not change for a job, the same computation is always repeated, so dynamic resource change does not occur. Likewise, input query of MLaaS is simple, so the resource is changed by almost input query rate. In the case of NLP inference serving, if the request query changes, the amount of computation to response to the query may change for each. Therefore, despite some studies of training and MLaaS resource management, research on serving is still needed.

The use of input query throughput or latency, commonly used as Service Level Objectives (SLO) in previous studies, is not suitable for NLP inference serving as mentioned above. In other words, to allocate proper resources in resource management system, it would be more reasonable to use another metric, not query throughput or latency. Per-token latency(*input query latency / number of generated tokens*) is a proper metric for scaling NLP inference serving engine.

In this work, we designed and implemented an inference serving management system. User can create the inference serving engine using RESTful API and also choose the policy about SLO. After the inference serving engine is deployed, our system automatically scales the engine out and in according to input queries, not violating SLO. System collects both query latency and per-token latency to decide scaling and supports both based scaling policy. Per-token latency based scaling policy can achieve more flexible, fast and correct scaling.

And this scaling policy also generates less SLO violation than query latency based scaling policy.

Chapter 2

Background

2.1 Natural Language Generation Model

Natural Language Generation model, or NLG model, is one of the most popular AI which aimed at creating machines that can understand and produce human language. As NLG models have developed, number of parameters of the model is rapidly grown. Table 2.1 shows the number of parameters of recent NLG models.

There are several Deep learning serving engines which guarantee high performance for production environments. Tensorflow Serving [12], or TFX is a serving system for machine learning Tensorflow [13] models. TorchServe [14] is also a useful framework for serving and scaling PyTorch [15] models. NVIDIA Triton server [16] is an open-source inference serving software that supports various backends, such as Tensorflow, Pytorch, TensorRT [17] and so on.

Several existing NLG interference services use the various engines described above in production.

Model	Developer	Parameter Size
BERT-Base [1]	Google	110M
BERT-Large [1]	Google	340M
GPT2 / GPT2-XL [2]	OpenAI	1.5B
Chinchilla [18]	DeepMind	70B
LaMDA [19]	Google	137B
GPT3 [4]	OpenAI	175B
OPT [20]	Meta	175B
Gopher [21]	DeepMind	280B
Megatron Turing [22]	NVIDIA	530B
PaLM [23]	Google	540B

Table 2.1 Recent Language model parameter size.

- Text Summarization: For given any text, the task is to generate a summary, which contains its main information and is shorter in length.

- Dialogue: A model intends to converse with a human, especially chat-bot.

- Creative writing: A model generates storytelling or poetry with given text as a topic or material.

- Image captioning: For given image, a model makes a verbal description of the contents of the image

- Machine Translation: For given text from one source language, a model automatically translates to the target language.

2.2 Scaling Inference Engine in Kubernetes Cluster

In general, a kubernetes [24] cluster is used for deep learning interference serving. Kubernetes cluster is a group of nodes that execute containerized programs, which is more lightweight and flexible than virtual machines. `Pod` is a group of containerized programs, with shared storage, network, and computing resources of node, and the smallest deployable units that kubernetes can create and manage. When deploying inference serving engine in kubernetes cluster, each engine runs in a pod unit using node resources. `Deployment` is a group of same pod, and provides declarative updates for pods. In order to manage and scale in/out the pods of same interference engine, it is necessary to manage it as deployment, not as each pod. `Service` is an abstraction method to expose executing application in pod group to network service. `Kube-proxy` manages network connections across multiple nodes in shared cluster and maintain network rules.

Figure 2.2 describes that inference engine is deployed on the kubernetes clsuter. There are three inference engine is executed in a pod, and three pods are grouped as a deployment. Service gathers the endpoints of inference engine to single service endpoint, and kube-proxy manages and update IP table of service when the number of pods is changed. When a user requests input queries to the service endpoint, service properly distributes requests to each inference engine. Pod scaler adjusts the number of pods in a deployment when triggered by specific condition.

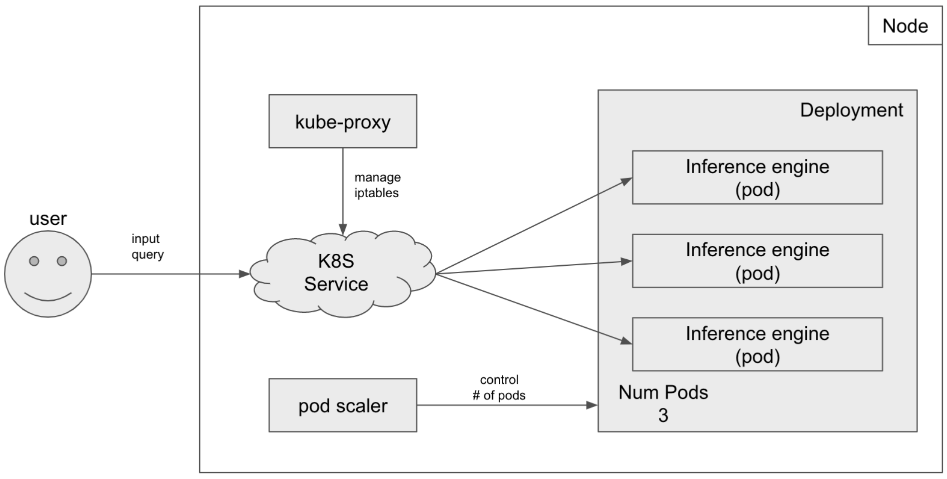

Figure 2.1 Kubernetes proxy to inference serving engine.

Chapter 3

Related Work

3.1 Scaling in Machine Learning Inference Serving

In traditional Machine Learning (ML), ML-as-a-service (MLaaS) is widely adopted for inference serving. There have been studies about resource management system and autoscaling. Jennings et al. [25] and Lorido-Botran et al. [26] investigate a variety of resource management systems, which Gujarati et al. [27] focuses service level agreement (SLA) aware autoscaling in a production of large scale ML-as-a-service platform. Though these works targeted to serve general ML model, polices in these works are not suitable for NLP model inference serving.

3.2 Model-less Inference Serving

InFaas [28] is a model-less inference serving system that efficiently navigates the proper amount of resources. Resource requirements of serving application are diverse due to the application purpose. For example, in a face recognition model inference service, social media application needs high accuracy with low cost, where as the visual guidance application asks high accuracy, low latency, but is willing to pay high cost for service. Even an application with the same function can have different requirements: InFaas selects a model, hardware, and model optimizations for each application requirements.

Chapter 4

Observation

The concept of service level objective is needed when explaining the scaling of NLG inference serving. Service Level Objective (SLO) is the most important element of service level agreements (SLA), which is the consensus between a customer and a service provider. SLOs commonly include input query throughput, accuracy, frequency, response time or latency, availability. SLOs also can be measured based on acheivement levels that can represented by menas, rates or percentiles.

4.1 Various Input Queries Violates SLOs

In this section, we explain why scaling is difficult only with the existing SLO measurement and propose a new suitable SLO measurement metric for NLG inference serving.

In general inference serving, input query is not very diverse. For example in image recognition inference service, the size and type of input image is fixed

Option	Explanation
`prompt`	Input text passed to the language generation engine.
`max tokens`	Maximum number of tokens to be generated by engine.
`min tokens`	Minimum number of tokens to be generated by engine.
`choices`	Number of choices to generate by engine.
`no repeat ngram`	Engine blocks to generate repeated n-gram.
`temperature`	Distribution temperature to decide which token to generate. Only one of temperature and top p should be utilised.
`top p`	Distribution top p to decide which token to generate. Only one of temperature and top p should be utilised.
`top k`	Decide which token to generate in k largest element of the candidates along a given vocabulary.
`stop`	Engine stops to generate when it generate stopping words.

Table 4.1 Various options of input query for natural language generation model.

Option		Query Type	
		Simple	Complex
# of tokens in prompt		34	82
# of generated tokens	mean	30	62
	min	10	15
	max	34	64
choices		1	4

Table 4.2 Options and stats of input query to observe.

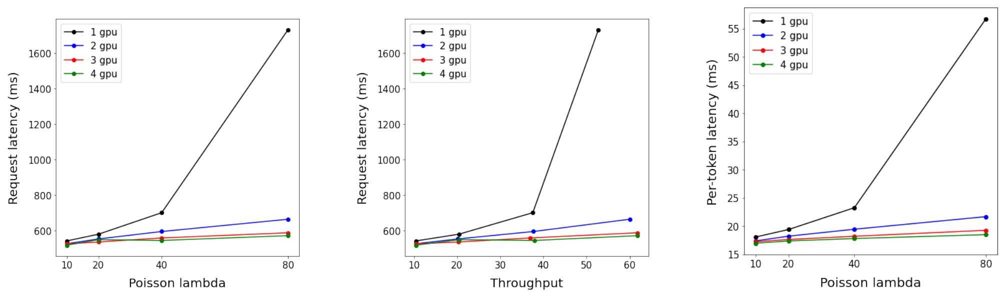

Figure 4.1 Latency of simple query using GPT2-XL model inference engine.

Left: Input rate and request latency graph.

Middle: Output throughput and request latency graph.

Right: Input rate and per-token latency graph.

and also the output is the same. In this case, query latency is according to the input rate which is arrival interval from user to engine.

However, natural language generation inference service can receive a various kinds of queries. Table 4.1 shows that several options of input query. In various options, especially `max tokens`, `min tokens`, `choices`, and `stop` take control the engine to decide the number of tokens to generate. Naturally, the number of tokens to be made by engine is proportional to the latency of the query. Furthermore, even if the number of token to generate is not enough, if the number of input tokens (`prompt`) is large then this causes engine use a lot of computing resources.

Table 4.2 describes two different type of input query for using observation experiment. A simple query only request 1 choice, while a complex query requests 4 choices for each input. Moreover, number of input tokens for a complex query is x2.4 larger than a simple query, and the number of generated tokens is x8 more than a simple one. Both input query types look the same, but the engine computation for both query is quite different.

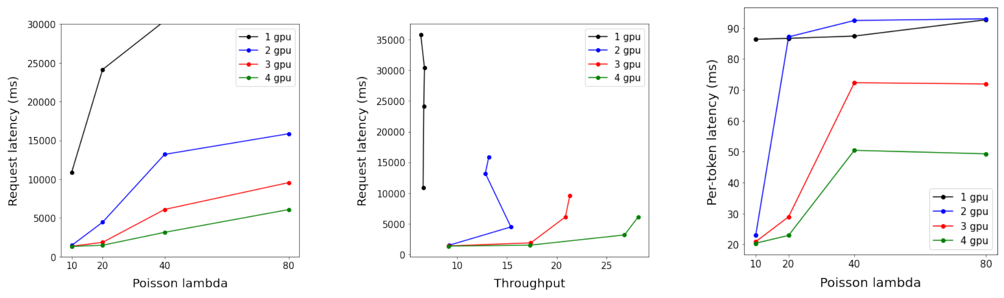

Figure 4.2 Latency of complex query using GPT2-XL model inference engine.
Left: Input rate and request latency graph.
Middle: Output throughput and request latency graph.
Right: Input rate and per-token latency graph.

Figure 4.1, 4.2 shows an experiment result to observe. Simple and complex query in the experiment used the one described above, in Table 4.2. For experiment, we generate input trace of simple and complex query, each trace has 2000 input queries, and input query follows the Poisson probability distribution with four different lambdas(10, 20, 40, 80). We deploy GPT2-XL model inference engine in single node from 1 GPU to 4 NVIDIA T4 GPUs, and we fix the amount of GPU resources used by the engine for each trace. For each trace, we measure request latency and generated token latency (per-token latency).

In a simple query experiment (Figure 4.1), all three graphs are similar tendency. Comparing the left and right graphs, it can be seen that per-token latencies or request latencies have the same characteristics. The middle figure shows that engine have reached a limit on the throughput, when it only uses 1 GPU resource in computing high lambda of poisson distribution, and all the other requests in small lambdas are not a problem. For simple query input, 1 GPU is up to 40 lambda, and 2 4GPU is not difficult to serve in all cases in this experiment.

Furthermore, we discover that result of complex query is quite different to the previous result. (Figure 4.2) Once, it seems to radiate only at the engine with 1 GPU resource in left figure. The middle graph indicates the limitation of engine where each resource is used. An engine using 1 GPU resource cannot handle any throughput, and with 2, 3, and 4GPU, the limit is 13, 20, and 30 series/sec throughput, respectively. This means that the limitations of the engine cannot be sufficiently known by simply measuring the request latency. However, measurement of per-token latency at the right figure represents the limitation, too.

From these experimental results, we discover that per-token latency expresses the limitations of the engine better than request latency. The following section describes the scaling policy that uses per-token latency to enable the NLG inference engine to have efficient resources according to the input query.

Chapter 5

Scaling Mechanism and Policy

5.1 Horizontal Pod Scaling Mechanism

We define scaling mechanism of NLG inference serving in kubernetes cluster. In kubernetes cluster, Horizontal Pod Autoscaler (HPA) automatically updates a resource of deployment. HPA appropriately adjusts the number of pods in deployment using a user-defined metric. The following algorithm describes how HPA calculates the number of proper pods of inference serving engine for deployment.

```
 Desired number of pods
= ceil[current number of pods
         * ( current metric value / threshold )]
```

Listing 5.1 Autoscaling algorithm

HPA polls the current metric value every polling time and determines how many pods are most appropriate according to the autoscaling algorithm. If HPA determines that the number of pods needs to change, it adjusts the number of pods in the target deployment, and kubernetes cluster notices that the deployment's configuration has been updated and either executes new pods or stop working pods.

5.2 Per-Token Latency Based Policy

We propose per-token latency based scaling policy, which is a fast and correct resource scaling policy. We investigated the necessity of scaling using per-token latency through observation at the previous chapter. In this section, we argue in two ways the per-token latency based policy is better than request latency based one.

First of all, we compare the two policies from the perspective of scale in and out. In fact, the two policies are not very different for simple query input. In the case of request latency-based policy, the time it takes for the scaler to receive the metric is exactly as long as the request latency. On the contrary, the pod scaler recieves whenever an output choice is created, with per-token latency based policy.

Furthermore, per-token latency based policy is more flexible to both scale in and out. In fact, the measurement of per-token latency metric has its upper bound depending on the combination of inference serving engine and model. As seen in Chapter 4, per-token latency does not diverge no matter how fast any input query comes into the engine. Therefore, in per-token latency based policy, as fewer pods run and stop more frequently, the process of finding the optimal resource proceeds with less delay. This characteristic limits the number of pods that change at once when scaling in/out. Therefore, per-token latency based policy adjusts the number of pods little by little, quickly and accurately.

Chapter 6

System Design

In this chapter, we design inference serving system that supports fast and correct scaling policy.

6.1 System Architecture

In this section, we describe the system architecture of our inference serving system. The components of our system consists of management server, horizontal pod scaler and metric server. Figure 6.1 shows the whole architecture of system.

The management server controls the overall objects of the kubernetes clus-

function	usage
READ	`GET http://host:port/deployment/$MODEL_ID/`
DELETE	`DELETE http://host:port/deployment/$MODEL_ID/`
CREATE	`POST http://host:port/deployment/`

Table 6.1 APIs for inference management server.

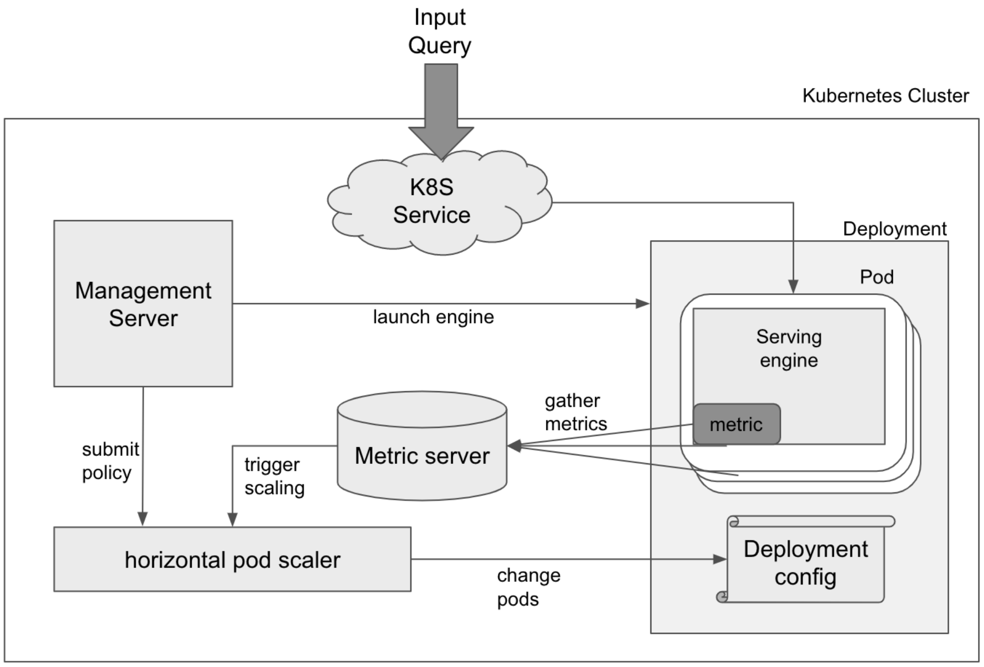

Figure 6.1 System architecture.

ter. This server contains kubernetes control plain for creating, reading and deleting various kubernetes objects, such as service, deployment and scaling policy in pod scaler. The management server receives a request from the user to create inference serving engine with submitted policy, and delivers pod scaling policy to scaler.

Each inference serving engine, which is launched by the server and executed in a pod, has metric exporter to the metric server. Metric server gather all metrics from each engine. Horizontal pod scaler polls the metric corresponding to its own scaling policy from the metric server every time interval. If the number of pods has to be changed by policy, scaler adjusts the number of engine pods by changing the configuration of the corresponding target deployment.

6.2 Management Server API Design

Based on the previous section, we provide the client RESTful API for management server. In this section, we describes API endpoints and some examples on usage. Query request and response is a JSON object, which User can create, read and delete deep learning serving engine to send request to management server. Table 6 describes RESTful APIs and Listing 6.1 shows body of creating deployment of inference serving engine.

6.3 Implementation

Our system implementation is based on Python 3.9 and packaged by helm chart. We use Kubernetes Python SDK for control kubernetes objects, and make RESTful API using FastAPI [29]. For exposing engine metric such as request latency and per-token latency, we implement prometheus [30] exporter which expose these metrics to the endpoint of engine. For horizontal pod scaler, we use KEDA [31], one of the custom resource definition in kubernetes cluster, to perform scaling using metric exposed by prometheus.

```
{
  // Cloud instance type.
  "instance_type": <string>,

  // Serving model id.
  "model_id": <string>,

  // Which serving engine to use.
  "engine_type": <string>

  // Define caling policy.
  "scaler": {
    // minimum number of engine pod.
    "min_deployment_count": <int>,
    // maximum number of engine pod.
    "max_deployment_count": <int>,
    // type of scaling policy.
    "type": "request" | "pertoken",
    // time window to gather metrics.
    "time_window": <int>,
    // threshold to scale out/in.
    "threshold": <int>
  }
}
```

Listing 6.1 Request body of CREATE API

Chapter 7

Evaluation

7.1 Evaluation Setup

7.1.1 Environment

We evaluated our system on a single node AWS instance, `g4dn.12xlarge`, with 4 NVIDIA T4 GPUs, 48 AWS custom Intel cascade lake virtual CPUs, 192 GiB of memory in us-west-2 region. We use client program, sending the request in the experiment, using the `c4.xlarge` instance in the same region to reduce the networking time. We use Python 3.9, Ubuntu 22.04 and CUDA 11.4 for all our experiments. .

7.1.2 Workloads

We evaluate our scaling policy using simulated input trace shown in Table 4.2. We use trace of complex input query with 10 requests/sec input rate(lambda of Poisson distribution). For natural language generation model, we use GPT2-XL [2] by OpenAI. We serve this model using our custom inference serving engine, which is similar to NVIDIA Triton [16]. For serving GPT2-XL model with our custom engine, we use 1 GPU resource for each engine pod. Specifically, we compare two main policies, query request latency based policy and per-token latency based policy. We fix 2000ms request latency as a service level objectives (SLO) for every experiment, and measure total violation response, query latency, GPU resource usage, scaling speed for each policy. The threshold of query request latency based policy to trigger scaler is 2000ms, same as SLO latency, and 20ms, 25ms, 30ms, 35ms for per-token latency based policy. In order to reduce the overhead of scaling pods, GPT2-XL pre-trained model for inference serving engine was previously downloaded to the node for every experiment.

7.2 First Scaling Time

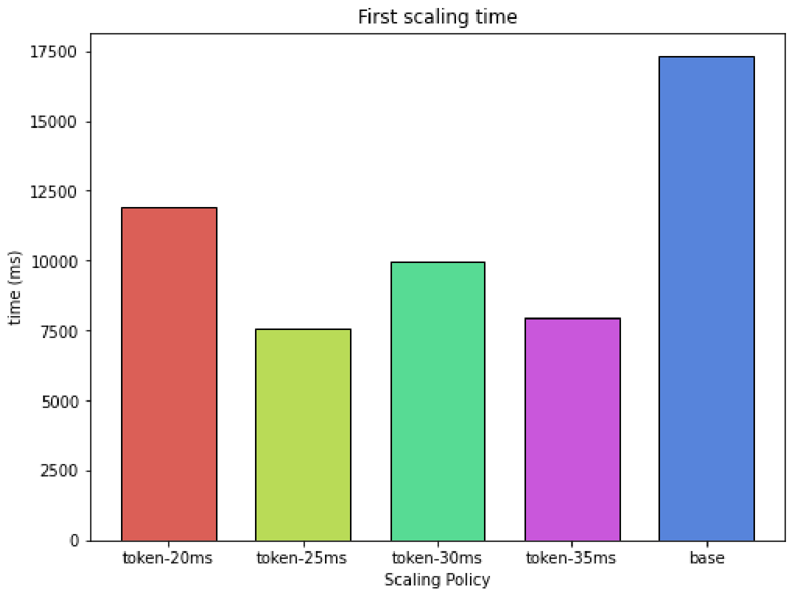

Figure 7.1 First Scaling Time of different policy.

Figure 7.2 shows the scaling latency of GPT-2XL model measured in a single node. In the figure, "token" stands for our per-token latency based policy, and our baseline, "base" refers to request latency based policy. The first scaling time is a measurement of scaling out latency from using 1 GPU to multiple GPU for each policies. This measurement shows how quickly policy triggers pod scaler to scale out inference serving engine. The smaller the first scaling time is, the more sensitive and flexible the corresponding policy is to the input query. All per-token latency based policies scale at most 2.29 times faster than baseline.

7.3 SLO Violations and Total Resource Usage

Figure 7.2 represents the number of total SLO violations of inference serving engine. The result of measuring violations for SLO latency 2000 ms, and there is 3840 queries as input in this experiment. And Figure 7.2 shows total GPU resource usage (GPUsec) while inference engine serves the same number of queries. We discover per-token latency based policy is much better than the baseline. In general, it can be thought that the number of viloations and total GPU resource usage are trad-off. However in per-token latency with 20 ms, the number of violations is 2.1 times less and total resource usage is 17% less than the baseline. Per-token latency based policy with 25 ms also defeat baseline in both violations and usage. This result shows that such policy efficiently saves resources while scaling.

7.4 Appropriate Resource Usage

Figure 7.4 explains that the GPU resource usage over time. For the first 120 seconds, though per-token latency based policy moves faster than the baseline, changes in the resource usage are similar between both policies. After 120 seconds, per-token latency based policy finds the optimal usage by reducing GPU resource. However, baseline policy cannot find appropriate optimal resource for inference serving. As we mentioned in Chapter 5, this is because the number of pods that are scaled at once is smaller and scaling occurs more often using per-token latency based policy.

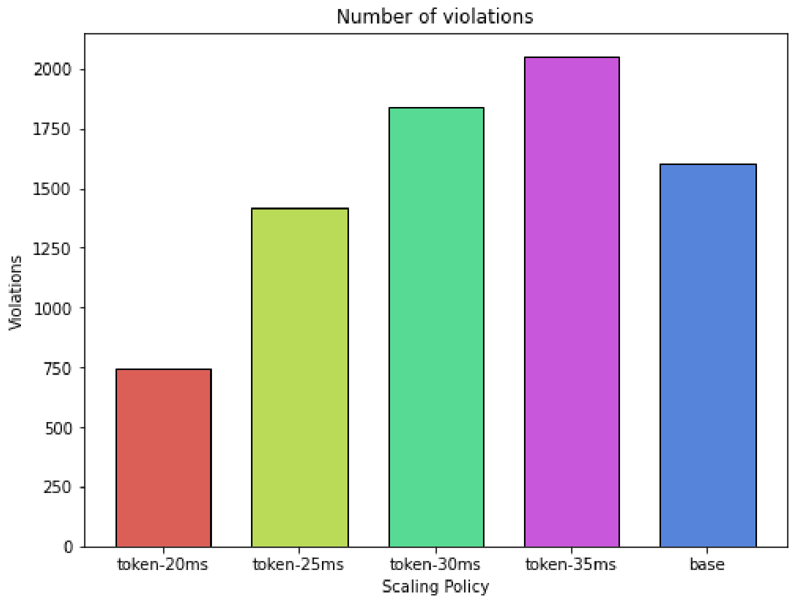

Figure 7.2 Number of SLO violations.

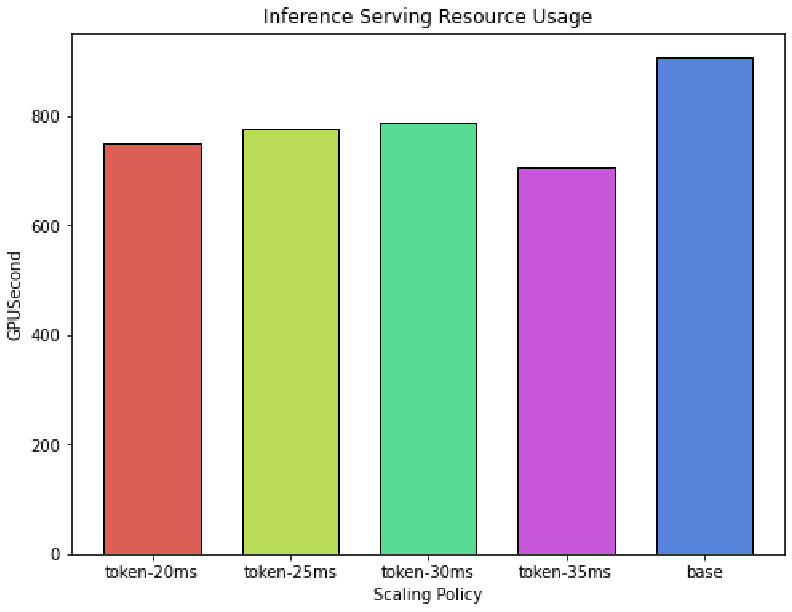

Figure 7.3 Total resource usage.

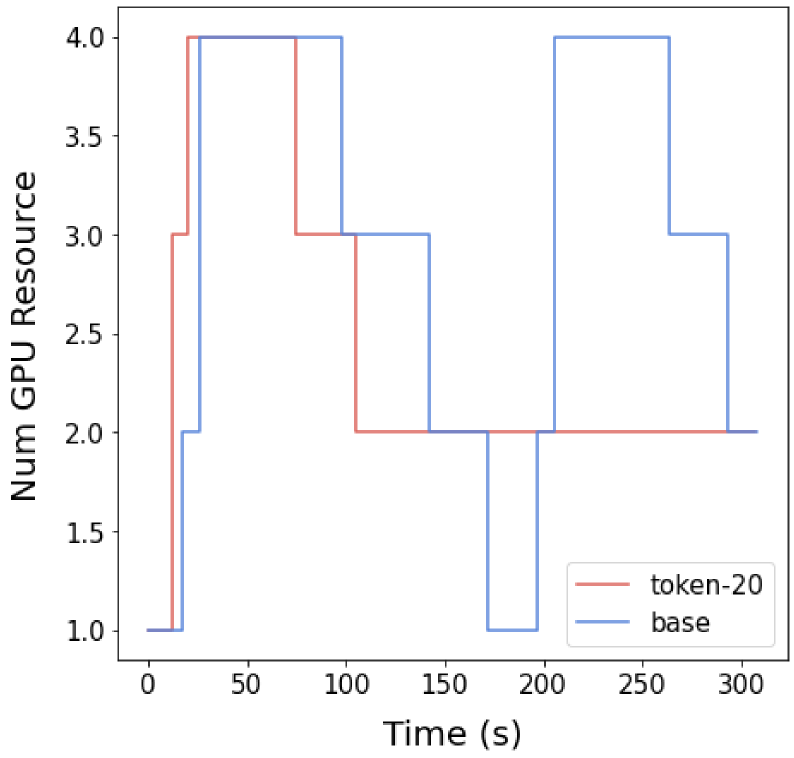

Figure 7.4 GPU resource usage over time.

Chapter 8

Conclusion

In this paper, we have discussed the problem of using SLO request latency in scaling inference serving for natural language generation model. And we proposed a per-token latency policy which is a fast and correct resource scaling policy. We implemented a inference serving deployment system to compare scaling policy.

As a result of comparing the performance, the per-token latency based policy of all thresholds had a smaller scaling latency than the baseline request latency based policy. Furthermore, the Per-token latency based policy minimizes both violation of service level objective (SLO) and GPU resource usage.

In conclusion, for serving various inference models, it is good to use efficient resource scaling policy which is suitable for the characteristics of each application.

Bibliography

[1] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)* (J. Burstein, C. Doran, and T. Solorio, eds.), pp. 4171–4186, Association for Computational Linguistics, 2019.

[2] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[3] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," 2020.

[4] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and

D. Amodei, “Language models are few-shot learners,” in *Advances in Neural Information Processing Systems* (H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, eds.), vol. 33, pp. 1877–1901, Curran Associates, Inc., 2020.

[5] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, “Exploring the limits of transfer learning with a unified text-to-text transformer,” *Journal of Machine Learning Research*, vol. 21, no. 140, pp. 1–67, 2020.

[6] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou, “Gandiva: Introspective cluster scheduling for deep learning,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, (Carlsbad, CA), pp. 595–610, USENIX Association, Oct. 2018.

[7] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, “Optimus: an efficient dynamic resource scheduler for deep learning clusters,” in *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018* (R. Oliveira, P. Felber, and Y. C. Hu, eds.), pp. 3:1–3:14, ACM, 2018.

[8] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla, “Themis: Fair and efficient GPU cluster scheduling,” in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, (Santa Clara, CA), pp. 289–304, USENIX Association, Feb. 2020.

[9] “Google vertex ai..” `https://cloud.google.com/vertex-ai`.

[10] "Amazon machine learning - predictive analytics with aws.." `https://aws.amazon.com/machine-learning/`.

[11] "Azure machine learning.." `https://azure.microsoft.com/en-us/services/machine-learning`.

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

[13] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, 2016.

[14] "Torchserve." `https://github.com/pytorch/serve`.

[15] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), vol. 32, Curran Associates, Inc., 2019.

[16] "Nvidia triton." `https://github.com/triton-inference-server/server`.

[17] "Nvidia tensorrt." `https://github.com/NVIDIA/TensorRT`.

[18] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. d. L. Casas, L. A. Hendricks, J. Welbl, A. Clark, T. Hennigan, E. Noland, K. Millican, G. v. d. Driessche, B. Damoc, A. Guy, S. Osindero, K. Simonyan, E. Elsen, J. W. Rae, O. Vinyals, and L. Sifre, "Training compute-optimal large language models," 2022.

[19] R. Thoppilan, D. De Freitas, J. Hall, N. Shazeer, A. Kulshreshtha, H.-T. Cheng, A. Jin, T. Bos, L. Baker, Y. Du, Y. Li, H. Lee, H. S. Zheng, A. Ghafouri, M. Menegali, Y. Huang, M. Krikun, D. Lepikhin, J. Qin, D. Chen, Y. Xu, Z. Chen, A. Roberts, M. Bosma, V. Zhao, Y. Zhou, C.-C. Chang, I. Krivokon, W. Rusch, M. Pickett, P. Srinivasan, L. Man, K. Meier-Hellstern, M. R. Morris, T. Doshi, R. D. Santos, T. Duke, J. Soraker, B. Zevenbergen, V. Prabhakaran, M. Diaz, B. Hutchinson, K. Olson, A. Molina, E. Hoffman-John, J. Lee, L. Aroyo, R. Rajakumar, A. Butryna, M. Lamm, V. Kuzmina, J. Fenton, A. Cohen, R. Bernstein, R. Kurzweil, B. Aguera-Arcas, C. Cui, M. Croak, E. Chi, and Q. Le, "Lamda: Language models for dialog applications," 2022.

[20] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer, "Opt: Open pre-trained transformer language models," 2022.

[21] J. W. Rae, S. Borgeaud, T. Cai, K. Millican, J. Hoffmann, F. Song, J. Aslanides, S. Henderson, R. Ring, S. Young, E. Rutherford, T. Hennigan, J. Menick, A. Cassirer, R. Powell, G. v. d. Driessche, L. A. Hendricks, M. Rauh, P.-S. Huang, A. Glaese, J. Welbl, S. Dathathri, S. Huang, J. Uesato, J. Mellor, I. Higgins, A. Creswell, N. McAleese, A. Wu, E. Elsen, S. Jayakumar, E. Buchatskaya, D. Budden, E. Suther-

land, K. Simonyan, M. Paganini, L. Sifre, L. Martens, X. L. Li, A. Kuncoro, A. Nematzadeh, E. Gribovskaya, D. Donato, A. Lazaridou, A. Mensch, J.-B. Lespiau, M. Tsimpoukelli, N. Grigorev, D. Fritz, T. Sottiaux, M. Pajarskas, T. Pohlen, Z. Gong, D. Toyama, C. d. M. d'Autume, Y. Li, T. Terzi, V. Mikulik, I. Babuschkin, A. Clark, D. d. L. Casas, A. Guy, C. Jones, J. Bradbury, M. Johnson, B. Hechtman, L. Weidinger, I. Gabriel, W. Isaac, E. Lockhart, S. Osindero, L. Rimell, C. Dyer, O. Vinyals, K. Ayoub, J. Stanway, L. Bennett, D. Hassabis, K. Kavukcuoglu, and G. Irving, "Scaling language models: Methods, analysis &; insights from training gopher," 2021.

[22] S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhumoye, G. Zerveas, V. Korthikanti, E. Zhang, R. Child, R. Y. Aminabadi, J. Bernauer, X. Song, M. Shoeybi, Y. He, M. Houston, S. Tiwary, and B. Catanzaro, "Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model," 2022.

[23] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, "Palm: Scaling language modeling with

pathways," 2022.

[24] B. Burns, J. Beda, and K. Hightower, *Kubernetes: up and running: dive into the future of infrastructure.* O'Reilly Media, 2019.

[25] B. Jennings and R. Stadler, "Resource management in clouds: Survey and research challenges," *Journal of Network and Systems Management*, vol. 23, no. 3, pp. 567–619, 2015.

[26] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of grid computing*, vol. 12, no. 4, pp. 559–592, 2014.

[27] A. Gujarati, S. Elnikety, Y. He, K. S. McKinley, and B. B. Brandenburg, "Swayam: distributed autoscaling to meet slas of machine learning inference services with resource efficiency," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pp. 109–120, 2017.

[28] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "Infaas: Automated model-less inference serving," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 397–411, 2021.

[29] "Fastapi." `https://fastapi.tiangolo.com`.

[30] "Prometheus monitoring system.." `https://prometheus.io/`.

[31] "Keda." `https://github.com/kedacore/keda`.

초록

다양한 유형의 심층 신경망 모델 (DNN)이 증가함에 따라 자연어 생성 모델에 대한 관심이 많아지고 있다. 또한 학습된 모델 이용한 추론 서비스에 대한 수요 또한 함께 증가하고 있다. 그러나 자연어 생성 모델 추론 서비스를 운용하는 데 있어서 컴퓨팅 자원을 효율적으로 사용하는 것은 단순한 문제가 아니다. 이는 추론 서비스에 들어오는 각 쿼리마다 추론 엔진에서 사용하는 컴퓨팅 자원이 다르기 때문이다. 그렇기에 추론 서비스에 대해 자원 스케일링 정책을 사용하는 것은 훨씬 더 어려운 일이다. 본 논문에서는 언어 생성 모델 추론 서비스에서 GPU 자원을 효율적으로 사용하는 문제에 대해 논의한다. 문제를 해결하기 위한 빠르고 정확한 자원 스케일링 정책을 제안하고, 요청 쿼리 워크로드에 대해서 해당 정책을 검증한다.

주요어: 딥러닝, 서빙, 자연어처리, 자연어 생성 모델, GPU, 자원관리자, 스케일링 정책

학번: 2018-23073