Interspeech 2024
1-5 September 2024, Kos, Greece

Faster Vocoder: A Multi Threading approach to achieve low latency during TTS Inference

Vishal Gourav, Ankit Tyagi, Phanindra Mankale

Oracle Corporation, India

vishal.gourav@oracle.com*, ankit.t.tyagi.@oracle.com, phanindra.mankale@oracle.com

Abstract

The first question by any customer who wants to buy or rent any service/product is always, "How fast is it?". So how fast is our Text-To-Speech(TTS) we get asked, how fast can it turn my text into speech, what is the Customer Perceived Latency (CPL) i.e. total time taken from the time a customer submits text to the time they get the audio file for that text? Like any other service, TTS also comprises of a number of moving parts contributing to CPL, for example, SSML parser, Text Normaliser, G2P, Models(Acoustic Model and Vocoder) and post processing. Each layer contributes in some way to the overall processing time. The goal is to reduce this processing time without any deterioration in the audio quality.

Index Terms: Text to Speech, Customer Perceived Latency

1. Introduction

Text to speech is one of the most sought after technology in today's fast moving AI world. The two parameters that make a service, that supports TTS, worthy of being called industry standard are the naturalness of the audio generated and low latency. Achieving both at the same time is a tough task. Our TTS service has already achieved the naturalness part and in this paper we would be discussing approaches we took to achieve our second goal of reducing the CPL. On average our TTS service takes 1.36 seconds to process 100 characters. On checking with different input sizes, we discovered that the total inference time is linearly proportional to the number of characters in the input. Similar work was tried in Parallel WaveGAN[1], but the quality of audio produced with Fastspeech 2 acoustic model was noisy and inferior. Thus, to improve the overall user experience i.e., faster response time without any compromise in the output audio quality, we need a solution to bring down the inference time in our ESPNeT[2] based TTS pipeline.

2. Experiments

We carried out experiments at every granular level so that we could reduce the TTS inference without compromising in the audio quality. Lets talk about each of the experiments.

2.1. Input Text Level

In this first approach, the entire text was divided into multiple chunks broken on punctuations(period, question mark, exclamation mark, etc.) and fed separately to multiple instances of the model for inference as independent threads, see Figure 1.

The results were promising as there was a reduction in overall processing time by 10%(when number of threads was chosen as 4). But the pauses between the joined chunks sounded monotonous and robotic, i.e, it had lost the human like voice.

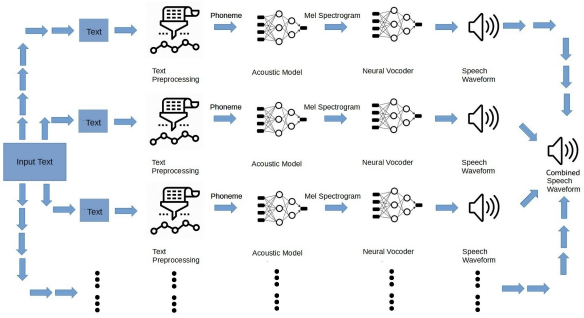

Figure 1: *Multi Threading at Input Text Level*

On further analysis, we realised that a case where the input contained a very large sentence, that particular chunk will take longer to process than the other smaller sentences, leading to reduced performance gain. The following equation demonstrates the situation.

$$\text{Total Inference Time} = Max(\text{Time for Chunk 1, Time for Chunk 2, ...Time for Chunk n}) \quad (1)$$

2.2. Phoneme(Acoustic + Vocoder) Level

In this experiment, as shown in Figure 2, we let the preprocessing step execute in serial manner and took the output of this step and broke it into equal sized pieces(chunked on phoneme count) and passed to each instance of the model as separate threads.

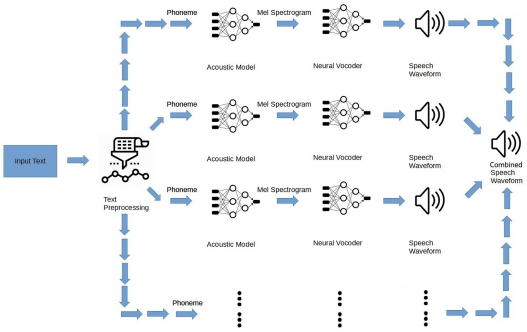

Figure 2: *Multi Threading at Phoneme Level*

We were able to save on time as each thread had equal number of phonemes. But the problem here is that when I divide the phonemes equally based on the number of threads available, it

Table 1: *Comparison between Inference Time without and with Threading using different approaches*

Number of Characters in Input Text	Simple Inference Time (in seconds) without any threading	Number of Threads	Time Taken (in seconds)			
			Input Text	Phoneme Level	Acoustic Model	Vocoder Level
100	1.36	2	1.49	1.04	1.30	0.94
		4	1.19	1.11	1.24	0.66
		6	1.20	1.08	1.40	0.75
		8	1.13	1.30	1.48	0.82
250	3.80	2	3.17	3.66	4.23	3.42
		4	2.88	2.82	3.99	2.36
		6	3.21	3.55	4.55	3.11
		8	3.44	3.73	4.73	2.83

results in wrong pronunciation because for a particular word there might come a case where the first part of its phoneme set be sent to one thread and the remaining part to another. Both of the threads, being oblivious to the existence of the other will produce speech based on what phonemes they receive. Thus, we saved on processing time but at the cost of pronunciation which is not an acceptable trade.

2.3. Multi Threading just the Acoustic Model

The bad pronunciation output of the previous experiment led us to my next idea. What if we could just parallelise the acoustic model. So, we tried to pass the phonemes through the acoustic model in multi threaded format and through the vocoder serially. The Figure 3, below will help clarify our approach.

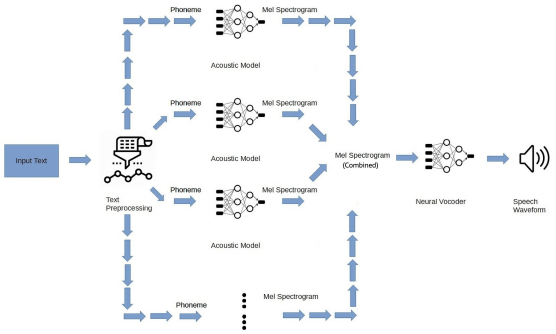

Figure 3: *Multi Threading at Acoustic Model Level Only*

As we expected, the pronunciation suffered the same fate as the previous experiment because the Mel Spectograms produced were incorrect due to the same situation as discussed in the end of the previous experiment.

2.4. Multi Threading just the Vocoder

The lack of improvement in latency of the previous experiment led us to now run the tests in phases to identify the bottleneck. To our surprise, the vocoder was taking most of the processing time which was, close to 70%. In this final experiment we tried to use the multi threading approach only on the vocoder, with the rest of the pipeline running serially, see Figure 4.

This experiment resulted in unparalleled success as we were able to reduce the latency while keeping the quality of the audio intact. This is because the Mel spectrogram being produced serially kept the pronunciations intact and the audio generation which was taking most of the time and is independent of the context was performed using multiple threads, thus reducing the latency, see Table 1.

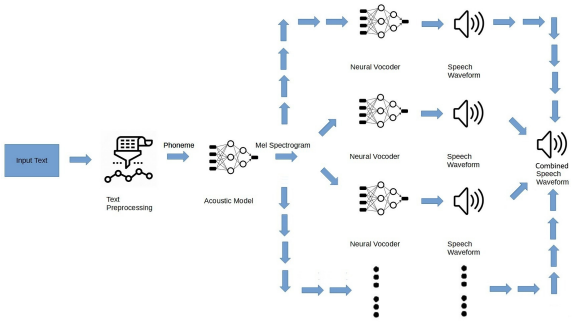

Figure 4: *Multi Threading at Vocoder Level Only*

3. Evaluation

On running evaluation, AMOS turned out to be 3.92 as compared to 3.93 for the model without multi threading while the WER saw a drop from 1.94 to 1.92 for the multi threading approach and the CER also saw a minor drop of 0.05. The load test gave improved results as well. The traditional serial inference, with an input of 300 characters, at 5 requests per second (RPS) for 10 minutes deployed on x86 returned 300 successful responses. Under the same conditions with multi threaded vocoder the successful responses increased to 425. This is an improvement of 42% over the traditional approach.

4. Conclusion

In this paper, we set out to find an approach that would help us reduce the latency of the TTS pipeline. We tried to find the optimum layer where we could apply multi threading to reduce the CPL. After multiple experiments we finally were able to identify the right location (the vocoder layer) to apply the right approach (multi thread logic) which not only brought down the latency but also did not impair the pronunciation accuracy of the model. All the evaluation scores and load test results are a testament to the efficacy of our approach which brought down the CPL, thus improving the overall user experience.

5. References

[1] R. Yamamoto, E. Song, and J.-M. Kim, "Parallel wavegan: A fast waveform generation model based on generative adversarial networks with multi-resolution spectrogram," Oct. 2019.

[2] S. Watanabe, T. Hori, S. Karita, T. Hayashi, J. Nishitoba, Y. Unno, N. E. Y. Soplin, J. Heymann, M. Wiesner, N. Chen, A. Renduchintala, and T. Ochiai, "Espnet: End-to-end speech processing toolkit," Mar. 2018.