Abstract—Incremental text-to-speech, also known as streaming TTS, has been increasingly applied to online speech applications that require ultra-low response latency to provide an optimal user experience. However, most of the existing speech synthesis pipelines deployed on GPU are still non-incremental, which uncovers limitations in high-concurrency scenarios, especially when the pipeline is built with end-to-end neural network models. To address this issue, we present a highly efficient approach to perform real-time incremental TTS on GPUs with Instant Request Pooling and Module-wise Dynamic Batching. Experimental results demonstrate that the proposed method is capable of producing high-quality speech with a first-chunk latency lower than 80ms under 100 QPS on a single NVIDIA A10 GPU and significantly outperforms the non-incremental twin in both concurrency and latency. Our work reveals the effectiveness of high-performance incremental TTS on GPUs.

I. INTRODUCTION

With the recent blossoming in deep learning, speech synthesis methods have switched from traditional concatenation-based[1] and HMM-based[2], [3] statistical parametric methods to neural network based methods and have been widely used in various application scenarios. Compared with traditional methods, neural networks can produce more natural and high-fidelity speech at the cost of more computing power and larger latency. Therefore, reducing the latency of the speech synthesis is vital to improve the user experience of applications that require instant response, such as the virtual agents of call centers, especially in the case of highly concurrent requests during peak periods. With such needs, incremental synthesis shows its benefits. Instead of synthesizing the entire speech audio before responding, incremental synthesis can produce speech chunk-by-chunk to provide a lower response latency. Once the first audio chunk is generated, the synthesis time of the subsequent audio chunks is hidden within the playback time of the preceding chunks.

Speech synthesis pipelines deployed in production environments typically comprises three major components: a frontend for extracting linguistic features from the text, an acoustic model for synthesizing acoustic features such as the Mel spectrogram, and a vocoder for converting acoustic features into waveform samples. The frontend typically performs text normalization, grapheme-to-phoneme[4] conversion, prosodic structure prediction, and named entity recognition. BERT[5], a transformer-based pre-training language model for natural language processing, has been applied in several unified TTS frontend research efforts[6] and has been proven capable of effectively extracting a wide range of linguistic features for realistic speech synthesis. Acoustic models can be grouped into two major types: autoregressive and parallel. Among them, the autoregressive model Tacotron2[7], [8] and its variants have dominated the acoustic models used in industry for many years due to their outstanding and stable synthesis quality. In recent years, parallel models, such as FastSpeech[9], [10] and FastPitch[11], have dramatically improved the controllability of speech, and are widely in personalized synthesis and singing synthesis. Parallel speech synthesis models often have architecture based on Transformer[12] and Convolution. They have higher throughput compared with autoregressive models for non-incremental synthesis. However, the synthetic speech quality of parallel models is highly dependent on the visibility of the entire feature sequence. In the case of incremental synthesis, the perceptual field is usually limited to the size of the chunk resulting in quality degradation of parallel models. In comparison, the hidden state of autoregressive models passed over time carrying contextual information ensures speech quality unharmed. Meanwhile, Samsung’s study[13] also pointed out that incremental synthesis using an autoregressive model has a more stable latency when the text length increases, which is evident as the latency is positively correlated with the computational overhead.

Similar to acoustic models, there are autoregressive and parallel vocoders. Autoregressive vocoder models such as WaveNet[14], WaveRNN[15], and LPCNet[16] synthesize audio sample-by-sample based on the acoustic features and previously generated samples. Parallel vocoders such as Flow-based WaveGlow[17], GAN-based MelGAN[18], and HiFiGAN[19] directly map the entire acoustic feature sequence to the waveform. Unlike acoustic models, autoregressive vocoders need to perform an enormous number of regressions to generate a chunk of audio. Although the number of regressions can be reduced using the multi-band mechanism[20], further model compression may still be required to obtain real-time. In contrast, parallel vocoders can perform chunk-by-chunk generation for incremental TTS, which is friendly to GPUs.

With more applications relying on low latency speech synthesis in recent years, several studies[21]–[23] have been proposed to optimize the quality and naturalness of incremental TTS. However, previous studies have disregarded the performance of incremental TTS on GPUs, which should be worthy of attention, especially under high concurrency scenarios. To the best of our knowledge, many incremental TTS pipelines deployed in production are CPU-based, which typically require reducing the computational cost by compressing[24], sparsi-
fying[25], and distilling[26] the model to obtain real-time. During inference, each TTS request is processed by one or more CPU cores, and batching is not involved. The additional optimization efforts for real-time on the CPU are usually time-consuming and may negatively impact the synthetic speech quality. Furthermore, the concurrency that each CPU server can handle is strictly limited by the number of CPU cores. In comparison, GPUs are better suited to computationally-intensive tasks and easier to scale. Despite the fact that GPUs have been widely used in TTS training and non-incremental synthesis as they are naturally friendly to batching, more needs to be investigated about efficient incremental TTS on GPUs.

To fill the research void, we propose an efficient approach for incremental text-to-speech synthesis on GPUs. The proposal contains a BERT-based frontend, a Tacotron 2-based acoustic model, and a HiFi-GAN-based vocoder. Furthermore, we propose the use of Instant Request Pooling and Module-wise Dynamic Batching strategies, which are vital for efficient incremental synthesis on GPUs. In the experiments, we make pressure and listening tests on the pipeline running on NVIDIA A10 GPU. Experiments prove that our approach can produce high-fidelity speech with ultra-low latency under high QPS.

II. METHODS

A. Modules

1) Frontend: The frontend consists of a grapheme-to-phone (G2P) conversion unit and a BERT-based prosodic structure prediction model. G2P first converts the text into a Chinese pinyin sequence using the forward maximum matching algorithm based on a pronunciation dict of Chinese phrases, then converts the pinyin of each Chinese character into its phone tokens based on a mapping table to get the final phoneme sequence pho. During the G2P conversion, the number of phone tokens for each Chinese character chari is recorded as counti. The prosodic structure of text can be divided into three levels: pw (prosodic word), pph (phonological phrase), and iph (intonational phrase). In order to predict the prosodic structure token sequence of each level, the text is first passed through a shared BERT backbone to extract hidden prosodic features hi of each character using multi-head attention blocks. Then three separate linear layers are used to predict the prosody token (beginning B or inside I) of each level for chari based on hi. Finally, the prosodic structure token sequences are regulated to have the same length as the phoneme token sequence using counti.

2) Acoustic Encoder: The acoustic encoder is based on the Tacotron 2 encoder, which contains a token embedding layer, three stacked convolutional layers with batch regularization and ReLU activation, and a bi-directional LSTM layer. We refer to it as the CBRL network. In addition, three embedding layers are introduced to encode three prosodic structure sequences. The prosodic structure embedding sequence Epw, Epph, Eiph are first summed with the phoneme token embedding sequence Epho, then passed to the CBRL network to generate the encoded feature sequence £enc.

$$\begin{align*}
E_{\text{all}} &= E_{\text{pho}} + E_{\text{pw}} + E_{\text{pph}} + E_{\text{iph}} \\
F_{\text{enc}} &= CBRL(E_{\text{all}})
\end{align*}$$

3) Acoustic Decoder: The acoustic decoder is based on the auto-regressive Tacotron 2 decoder. It contains an information bottleneck PreNet, a location-sensitive attention module LSA, two stacked unidirectional LSTM layers, two linear layers for predicting Mel spectrogram and stop token, and a PostNet for improving the Mel reconstruction. Unlike the decoder used in non-incremental TTS that generates the entire Mel before vocoding, the decoder used in incremental synthesis auto-regressively generates only a chunk of Mel frames at each time, then the Mel chunk is passed directly to the vocoder for waveform chunk generation. In order to maintain the transfer of contextual information across chunks, the final decoder LSTM hidden states of the last chunk need to be explicitly cached to be used as the initial decoder states of the current chunk. These states are the last Mel frame Mlast, the attention context A, the attention weight of the last Mel frame Wlast, the accumulated attention weights Wacc, the hidden states Hatt, Hdec and the cell states Catt, Cdec of the two LSTM layers.

$$\begin{align*}
H_{\text{att}}, C_{\text{att}} &= LSTM_{\text{att}}([\text{Pre}(M_{\text{last}}), A], H_{\text{att}}, C_{\text{att}}) \\
A, W_{\text{last}} &= \text{LSA}(H_{\text{att}}, F_{\text{enc}}, W_{\text{acc}}) \\
H_{\text{dec}}, C_{\text{dec}} &= LSTM_{\text{dec}}([H_{\text{att}}, A], H_{\text{dec}}, C_{\text{dec}}) \\
W_{\text{acc}} &= W_{\text{acc}} + W_{\text{last}} \\
M_{\text{last}} &= \text{Linear}(H_{\text{dec}}, A)
\end{align*}$$

4) Vocoder Model: The vocoder is a HiFi-GAN-based generator G that contains multiple transposed convolution and multi-receptive field fusion (MRF) blocks. In incremental TTS,
the vocoder synthesizes one waveform chunk at a time based on a chunk of Mel frames. In order to provide a smooth articulation between two adjacent waveform chunks, we splice multiple frames from the tail of the previous Mel spectrogram chunk $M_{\text{pre}}$ to the head of the current Mel spectrogram chunk $M_{\text{cur}}$ and then pass it through the vocoder. Finally, the overlap areas between the current waveform chunk $S_{\text{cur}}$ and the previous waveform chunk $S_{\text{pre}}$ are fused by applying fade-in and fade-out coefficients $\alpha$, $\beta$ from equal power cross-fade, as illustrated in Equation 3.

$$S_{\text{cur}} = G([\text{tail}(M_{\text{pre}}), M_{\text{cur}}])$$

$$\text{head}(S_{\text{cur}}) = \alpha \odot \text{head}(S_{\text{cur}}) + \beta \odot \text{tail}(S_{\text{pre}})$$

3. Incremental Synthesis

1) Incremental Synthesis of a Single Text: In order to achieve ultra-low latency, the proposed approach generates one short audio chunk at a time. In practice, each request contains a single text (sentence) corresponding to multiple responses. Each response carries an audio chunk of several hundred milliseconds in duration, depending on the configurable size of the chunk. The synthesis of each request involves four modules, the frontend ($F$), the acoustic encoder ($E$), the acoustic decoder ($D$), and the vocoder ($V$). The frontend and acoustic encoder modules are non-incremental and run only once per text to generate the encoded feature. On the other hand, the acoustic decoder and vocoder modules are incremental, running multiple times for each text. Each iteration produces a Mel chunk and a waveform chunk, respectively. The stop token output by the acoustic decoder exceeding threshold 0.5 flags the end of the synthesis.

2) Incremental Synthesis under High Concurrency: Incremental synthesis under high concurrency faces two challenges. Firstly, scheduling numerous requests while ensuring that each request is processed as soon as it arrives to the server. Secondly, synthesizing multiple requests simultaneously while keeping low latency. As shown in Figure 1, we introduce two strategies, Instant Request Pooling and Module-wise Dynamic Batching, to guarantee new requests are synthesized instantly, together with all the other incomplete requests in the server.

Instant Request Pooling. Conventional approach to handle the incoming TTS requests while the server is processing the ongoing requests is to keep the new requests waiting inside a queue to be processed when the ongoing requests are complete. However, this approach can hardly meet the requirement of low-latency incremental TTS because the waiting time can be long under high concurrency. Therefore, we introduce an additional request pool to hold all the ongoing and new requests. We use an infinite loop to handle the incremental synthesis. Each loop iteration consistently generates one audio chunk for each ongoing request in the pool. Once a new request arrives, it is immediately added to the pool as a new pool item and available for synthesis. New requests received during the current iteration only need to wait for the next iteration to be processed, regardless of the number of ongoing requests in the pool. For each request, its pool item caches the following variables, including the model states and a module indicator:

1. Model states, including the phoneme and prosody token sequence extracted by the frontend, encoder output $F_{\text{enc}}$, decoder states $M_{\text{last}}, W_{\text{last}}, A, W_{\text{acc}}, H_{\text{att}}, H_{\text{dec}}, C_{\text{att}}, C_{\text{dec}}$, and vocoder states $M_{\text{pre}}, M_{\text{cur}}, S_{\text{pre}}, S_{\text{cur}}$.

2. Module indicator, indicating which module should process this item in the current loop iteration. Initialized to 0 for $F, E, V$, then changed to 1 for $D$ when $E$ completes.

Module-wise Dynamic Batching. Batch is an essential strategy to fully utilize GPU’s parallel computing capability for both training and inference. For incremental TTS, we propose to use a dynamic batch size for each module. In each loop iteration, all the modules are executed sequentially, and the batch size of each module is determined by the number of items in the pool that each module should process. Specifically, each module first retrieves the pool items it should process based on the module indicator saved in each pool item. Take the vocoder $V$ as an example ($F, E, D$ modules follows the same process), suppose $N$ pool items are retrieved for the vocoder in the current iteration. Their states $[M_{\text{pre}}^1, M_{\text{pre}}^2, \ldots, M_{\text{pre}}^N], [M_{\text{cur}}^1, M_{\text{cur}}^2, \ldots, M_{\text{cur}}^N]$ and $[S_{\text{pre}}^1, S_{\text{pre}}^2, \ldots, S_{\text{pre}}^N]$ are constructed to input batch $M_{\text{pre}}^B, M_{\text{cur}}^B$, and $S_{\text{pre}}^B$. After inference, the output batch $S_{\text{cur}}^B$ is split

---

Fig. 2. Incremental synthesis timeline with the proposed instant request pooling and module-wise dynamic batching strategies.
into \([S_1^{cur}, S_2^{cur}, ..., S_N^{cur}]\) and we update each retrieved item \(i\):

\[
M_{pre}^i = M_{cur}^i, \quad S_{pre}^i = S_{cur}^i
\]  

(4)

For each pool item, its \(F, E\) states keep unchanged after its first inference iteration, while its \(D, V\) states are constantly updated in the following iterations. Finally, if a request completes, its item is removed from the pool immediately.

3) Timeline and Latency Analysis: Figure 2 elaborates on the timeline of incremental synthesis with the proposed pooling and batching strategies. Each time step corresponds to one incremental synthesis iteration. In short, a new request can be processed in step \(t\) if it arrives before the start of the frontend module in step \(t\). Request 1 arrives right before step 0 and is processed in step 0. Requests 2, 3 arrive during the execution of step 1 and are processed in step 2 in dynamic batches along with request 1. Request 1 is completed and removed immediately from the pool in step 3. Requests 2, 3 are processed together in dynamic batches in steps 4, 5. Request 2 is then completed and removed in step 5. Request 4 arrives at the end of step 5 and is processed in dynamic batches in step 6 with request 3, which is completed and removed in step 6.

Define the execution time of time step \(t\) as \(T_t\). If a request arrives at step \(t\), the maximum first audio chunk latency is \(T_t + T_{t+1}\) (arrives right after the start of the frontend in step \(t\)), the minimum latency is \(\min(T_t, T_{t+1})\) (arrives right before the start of the frontend in step \(t\), or right before the end of step \(t\)), and the expectation of the latency is \(E = T_t/2 + T_{t+1}\).

III. EXPERIMENTS

A. Settings

All the models in our experiments are trained with the Chinese Standard Mandarin Speech Corpus (CSMSC)[27] open-sourced by Databaker. The corpus has 10,000 audio clips and a total duration of about 12 hours, each audio clip contains a short sentence of 4.27 seconds on average. We kept 100 clips for validation, 100 clips for test, and the rest for training. The Bert used in the frontend contains 12 Transformer layers with the same model parameters as the original proposal with three lengths of text (TL) and their mixture. For validation, 100 clips for test, and the rest for training. The Tacotron 2 has the same model parameters as the original proposal with three additional 512-dim prosody embedding table. The HiFi-GAN vocoder with 128 upsampl sample convolution channels and upsampling factors of [8,8,2,2] is used to synthesize 22.05kHz 16bit PCM audio. All the models are trained on NVIDIA Tesla V100 GPU, and the inference pipelines are tested on NVIDIA Ampere A10 GPU. The proposed incremental pipeline is implemented in C++ as a NVIDIA Triton custom backend and the models are accelerated with TensorRT.

Since there is no publicly available performance benchmark for incremental TTS on GPU under high concurrency, we compare the proposed approach with a self-developed non-incremental pipeline using the exact same models. For the non-incremental twin, the TTS requests received over a short period are constructed into a batch for inference. The size of the batch keeps unchanged for all the models within each inference round. New requests received during the current round need to wait in the queue for the next round to be processed. Moreover, TensorRT and Triton are also applied for a fair comparison.

In the experiments, a fixed chunk size of 32 (~372ms) Mel frames is used with an overlap (OL) of 4 and 8. We test both the incremental and non-incremental pipelines with three lengths of text (TL) and their mixture. The average audio duration (DUR) for each text type are Short (2.64s), Medium (4.08s), Long (7.00s), and Mixed (4.54s). For incremental synthesis (INCR), we measure the first-chunk latency (FCL) (between the request being sent and the first chunk being received), the last-chunk latency (LCL) (between the request being sent and the last chunk being received), and real-time factor RTF (LCL/DUR). For non-incremental synthesis (Non-INCR), we measure latency and RTF. Finally, we measure the mean opinion score (MOS) of the INCR, compared with the Non-INCR and the ground truth recordings. The MOS test is carried out by native Mandarin speakers hired from Amazon Mechanical Turk. A total of 50 fixed samples are selected from the test set for evaluating each experimental configuration. Each sample was listened by 10 evaluators and each evaluator was asked to give an opinion score from 1 to 5.

B. Discussion

| TL    | OL | DUR(s) | FCL(ms) | LCL(ms) | RTF |
|-------|----|--------|---------|---------|-----|
| Short | 4  | 3.89   | 227.66  | 168.60  | 0.086 |
| Medium|  8 | 4.08   | 32.26   | 305.88  | 0.075 |
| Long  |  8 | 4.08   | 32.04   | 168.60  | 0.064 |
| Mixed |  8 | 4.08   | 32.04   | 168.60  | 0.064 |

1) Response Latency and Real-Time Factor: We pressure-test the proposed approach using QPS (Queries-per-Second) from 10 to 100. Each test lasts 200 seconds. Requests sent to the server are evenly distributed within a second. Full performance metrics and samples can be found on our Github page. Figure 3 shows the latency of mixed-length texts at different QPS and overlaps, which is more comparable to the real-world scenario. In the figure, we omit the data with a latency of more than 1s. For INCR, the first-chunk latency can be kept within 40 ms, and the last-chunk latency within 400 ms if the QPS does not exceed 60. In contrast, Non-INCR has a latency of more than 200 ms at 10 QPS and more than 800 ms at 60 QPS. The response latency of INCR has a 95.4%, 91.2% and 89.3% deduction compare with Non-INCR at 60, 30, and 10 QPS, respectively. At 100 QPS, INCR can synthesize in 89.3% deduction compare with Non-INCR at 60, 30, and 10 QPS, respectively. At 100 QPS, INCR can synthesize in 1425.

https://muyangdu.github.io/Efficient-Incremental-TTS-on-GPUs
real-time and provide a response latency of less than 100ms, while Non-INCR is not real-time and its latency already far exceeds 1s. In regard to the overlap, using 8 frames for overlap has slightly higher latency than 4 under high QPS.

Figure 4 shows the variation of RTF with QPS up to 70, below which all the configurations can synthesize in real-time. INCR of short, medium, and mixed text maintains a smooth RTF below 0.1 within 70 QPS. In comparison, the RTF of Non-INCR on the mixed text has a dramatic increase after 50 QPS. This proves the effectiveness and stability of the proposed strategies. Table 1 shows the performance details at 70 QPS. We observe that the mixed LCL is comparable to the medium LCL for INCR. In contrast, the mixed latency is only comparable to the long latency for Non-INCR. Similar observation is also shown in the RTF figure, in which the mixed RTF is even greater than the long RTF for Non-INCR. We attribute this observation to the redundant padding introduced for batched inference of all the modules in Non-INCR. Although padding to the maximum sequence length within a batch is still required for the $F$ and $E$ in INCR, the redundant padding of the $D$ and $V$ has been greatly reduced, leading to more efficient synthesis on GPUs.

2) Evaluation of Speech Quality: Table 2 indicates the MOS of INCR compared with Non-INCR and ground truth. The proposed approach can synthesize comparable speech quality with the non-incremental synthesis$^1$. Moreover, a weak trade-off can be observed between the incremental speech quality and overlap length. Combined with the performance analysis that larger overlap only introduces slightly higher latency under high QPS on GPU, we believe it’s worthwhile to use a larger overlap to achieve a better speech quality.

IV. LIMITATIONS AND FUTURE WORKS

One limitation we have identified is the use of an autoregressive decoder, which requires multiple regressions to generate a chunk of Mel frames. Replacing it with a transformer-like parallel decoder could further speedup chunk generation. However, transformer lacks time-domain hidden states for transferring contextual information if it is used for chunk-by-chunk generation. Simply limiting the input of parallel decoder to an encoded feature chunk can result in discontinuity artifact between adjacent Mel chunks. Therefore, a potential future direction is exploring methods to use a parallel decoder for Mel chunk generation while maintaining Mel quality unharmed.

Another limitation we have identified is the quality degradation caused by the overlap areas of the vocoder. While employing a larger overlap can alleviate the impact, it also reduces the ratio of valid payload of each chunk, leading to a marginally increased latency. Therefore, a potential future direction is to investigate a better streaming strategy for the parallel vocoder.

V. CONCLUSIONS

In this paper, we introduce a highly efficient approach for performing incremental text-to-speech synthesis on GPUs. Our approach allows for ultra-low latency TTS even in application scenarios with high concurrency by utilizing two important strategies: Instant Request Pooling and Module-wise Dynamic Batching. We show the effectiveness of our proposal through extensive experiments and timeline analysis. Furthermore, we offer a comprehensive performance benchmark of a complete incremental TTS system on GPUs. This benchmark can serve as a valuable reference and starting point for researchers and developers in this field. Future work will be focused on further enhancing efficiency while upholding synthetic quality.
REFERENCES

[1] L.-S. Lee, C.-Y. Tseng, and C.-J. Hsieh, “Improved tone concatenation rules in a formant-based chinese text-to-speech system,” IEEE Transactions on Speech and Audio Processing, vol. 1, no. 3, pp. 287–294, 1993.

[2] J. Yamagishi, T. Nose, H. Zen, et al., “Robust speaker-adaptive hmm-based text-to-speech synthesis,” IEEE Transactions on Audio, Speech, and Language Processing, vol. 17, no. 6, pp. 1208–1230, 2009.

[3] M. Pouget, T. Hueber, G. Bailly, and T. Baumann, “Hnm training strategy for incremental speech synthesis,” in Interspeech 2015-16th Annual Conference of the International Speech Communication Association, 2015, pp. 1201–1205.

[4] S. Hunnicutt, “Grapheme-to-phoneme rules: A review,” Speech Transmission Laboratory, Royal Institute of Technology, Stockholm, Sweden, QPSR, no. 2-3, pp. 38–60, 1980.

[5] J. Devlin, M.-W. 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, Volume 1 (Long and Short Papers), Minneapolis, Minnesota: Association for Computational Linguistics, pp. 4171–4186.

[6] Z. Bai and B. Hu, “A universal bert-based front-end model for mandarin text-to-speech synthesis,” in ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), IEEE, 2021, pp. 6074–6078.

[7] Y. Wang, R. J. Skerry-Ryan, D. Stanton, et al., “Tacotron: Towards end-to-end speech synthesis,” in Interspeech 2017, 18th Annual Conference of the International Speech Communication Association, Stockholm, Sweden, August 20-24, 2017, F. Lacerda, Ed., ISCA, pp. 4006–4010.

[8] J. Shen, R. Pang, R. J. Weiss, et al., “Natural TTS synthesis by conditioning wavenet on MEL spectrogram predictions,” in 2018 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2018, Calgary, AB, Canada, April 15-20, 2018, IEEE, pp. 4779–4783.

[9] Y. Ren, Y. Ruan, X. Tan, et al., “Fastspeech: Fast, robust and controllable text to speech,” in Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada, H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alchê-Buc, E. B. Fox, and R. Garnett, Eds., pp. 3165–3174.

[10] Y. Ren, C. Hu, X. Tan, et al., “Fastspeech 2: Fast and high-quality end-to-end text to speech,” in 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021, OpenReview.net.

[11] A. Łańcucki, “Fastpitch: Parallel text-to-speech with pitch prediction,” in ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), IEEE, 2021, pp. 6588–6592.

[12] A. Vaswani, N. Shazeer, N. Parmar, et al., “Attention is all you need,” in Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA, I. Guyon, U. von Luxburg, S. Bengio, et al., Eds., pp. 5998–6008.

[13] N. Ellinas, G. Vanvoukakis, K. Markopoulos, et al., “High quality streaming speech synthesis with low, sentence-length-independent latency,” in Interspeech 2020, 21st Annual Conference of the International Speech Communication Association, Virtual Event, Shanghai, China, 25-29 October 2020, H. Meng, B. Xu, and T. F. Zheng, Eds., ISCA, pp. 2022–2026.

[14] A. van den Oord, S. Dieleman, H. Zen, et al., Wavenet: A generative model for raw audio, corr, vol. abs/1609.03499, 2017.

[15] N. Kalchbrenner, E. Elsen, K. Simonyan, et al., “Efficient neural audio synthesis,” in Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018, J. G. Dy and A. Krause, Eds., ser. Proceedings of Machine Learning Research, vol. 80, PMLR, 2018, pp. 2415–2424.

[16] J. Valin and J. Skoglund, “LPCNET: improving neural speech synthesis through linear prediction,” in IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2019, Brighton, United Kingdom, May 12-17, 2019, IEEE, 2019, pp. 5891–5895.

[17] R. Prenger, R. Valle, and B. Catanzaro, “Waveglow: A flow-based generative network for speech synthesis,” in IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2019, Brighton, United Kingdom, May 12-17, 2019, IEEE, 2019, pp. 3617–3621.

[18] K. Kumar, R. Kumar, T. de Boissiere, et al., “Melgan: Generative adversarial networks for conditional waveform synthesis,” in Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada, H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alchê-Buc, E. B. Fox, and R. Garnett, Eds., pp. 14 892.

[19] J. Kong, J. Kim, and J. Bae, “Hifi-gan: Generative adversarial networks for efficient and high fidelity speech synthesis,” in Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-
[20] T. Q. Nguyen, “Near-perfect-reconstruction pseudo-qmf banks,” *IEEE Transactions on signal processing*, vol. 42, no. 1, pp. 65–76, 1994.

[21] B. Stephenson, T. Hueber, L. Girin, and L. Besacier, “Alternate endings: Improving prosody for incremental neural TTS with predicted future text input,” in *Interspeech 2021, 22nd Annual Conference of the International Speech Communication Association, Brno, Czechia, 30 August - 3 September 2021*, ISCA, pp. 3865–3869.

[22] D. S. R. Mohan, R. Lenain, L. Foglianti, et al., “Incremental text to speech for neural sequence-to-sequence models using reinforcement learning,” in *Interspeech 2020, 21st Annual Conference of the International Speech Communication Association, Virtual Event, Shanghai, China, 25-29 October 2020*, H. Meng, B. Xu, and T. F. Zheng, Eds., ISCA, pp. 3186–3190.

[23] B. Stephenson, L. Besacier, L. Girin, and T. Hueber, “What the future brings: Investigating the impact of lookahead for incremental neural TTS,” in *Interspeech 2020, 21st Annual Conference of the International Speech Communication Association, Virtual Event, Shanghai, China, 25-29 October 2020*, H. Meng, B. Xu, and T. F. Zheng, Eds., ISCA, pp. 215–219.

[24] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, “Model compression and acceleration for deep neural networks: The principles, progress, and challenges,” *IEEE Signal Processing Magazine*, vol. 35, no. 1, pp. 126–136, 2018.

[25] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, “Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks,” *J. Mach. Learn. Res.*, vol. 22, 241:1–241:124, 2021.

[26] G. Hinton, O. Vinyals, and J. Dean, “Distilling the knowledge in a neural network,” *stat*, vol. 1050, p. 9, 2015.

[27] D. Baker, *Chinese Standard Mandarin Speech Corpus*, https://www.data-baker.com/open_source.html, [Online; accessed 7-March-2008], 2018.