Improving the Bandwidth of GAN-Based Speech Synthesis

Max Morrison

Northwestern University, Evanston, IL, USA

maxrmorrison@gmail.com

Abstract

Text-to-speech (TTS) technology has become a crucial building block for user-facing systems such as screen readers, voice assistants, and podcast editing software. These systems are typically deployed either on-device using CPU compute, or via queries to a remote server, which can make use of high-bandwidth GPU compute to serve a large number of queries in a short period. In this paper, we improve the GPU bandwidth of one component of a TTS system via direct implementation in CUDA. We describe the optimizations performed relative to a baseline PyTorch implementation and evaluate the GPU bandwidth of the baseline and proposed implementations. We show that our implementation achieves a 3.78x improvement in inference speed over the PyTorch baseline.

1. Introduction

Text-to-speech (TTS) systems have become a mainstay technology in user interfaces, text-based speech editors, and accessible applications for the blind or dyslexic. Recent advances in machine learning, and specifically deep learning [1], have significantly advanced the perceptual quality of TTS systems, as evaluated by human listeners. These deep-learning-based TTS systems are most often comprised of two neural network components: (1) an acoustic feature predictor that converts text to an intermediate representation (usually a log-scaled mel-spectrogram) and (2) a neural vocoder that converts the acoustic features to a speech waveform. Of these two components, the neural vocoder is typically the compute bottleneck, as it must model the long sequence dependencies within a high-resolution signal.

Prior work on neural vocoders includes models such as WaveNet [2], WaveGlow [3], LPCNet [4], and MelGAN [5]. WaveNet is largely considered the first deep learning architecture to be successful at generating high-quality speech. However, its large parameter count and autoregressive structure make it significantly slower than real-time and unusable for most applications. WaveGlow and MelGAN omit the autoregressive structure, instead generating all waveform samples simultaneously—and thus more efficiently utilizing the GPU during inference. LPCNet retains the autoregressive dependency that makes GPU computation inefficient, but still provides a low latency on CPU devices by significantly reducing the model parameter count. In this paper we focus on improving the bandwidth of MelGAN. We choose to work with MelGAN as it has the highest GPU bandwidth among existing neural vocoders.

Our work is also related to prior efforts to utilize optimized GPU algorithms for training and inference of deep neural networks. Prior works in this area include frameworks for exposing low-level GPU operations via high-level APIs such as PyTorch [6] and Tensorflow [7]. Our work is most similar to inference-time optimization frameworks such as TensorRT [8], which provides automatic selection and merging of commonly used GPU kernels to improve the performance of deep neural networks during deployment.

We propose a CUDA implementation of MelGAN that significantly increases the GPU bandwidth relative to a baseline PyTorch implementation[1]. We describe two additional techniques used to reduce the number of GPU kernels launched during inference. We separately evaluate each unique layer of our implementation as well as the full system bandwidth. From this evaluation, we provide insights regarding why our implementation achieves a higher bandwidth, and where there is still room for improvement.

2. Method

We implement all of the layers of the MelGAN architecture directly in CUDA. The MelGAN architecture consists of 1D convolution, 1D transpose convolution, leaky ReLU, tanh, and reflection padding layers. All of the convolution and transpose convolution layers use weight normalization [9]. We implement the 1D convolution and transpose convolution layers using CUDNN [10] to make use of the higher floating point operation bandwidth of tensor cores. Alongside direct implementation in CUDA, we propose two additional layer fusions to further improve performance: (1) precomputing the forward pass of the weight normalization operation and (2) fusing the 1d convolution layer with the channel-wise broadcast-add of the bias term. We next describe each of these proposed optimizations.

2.1. Precomputing normalized weights

Weight normalization is a technique used to expedite the training of neural networks by adapting the rate of training to the variance of the first-order approximations of the gradients. Empirically, this allows the network to be more adaptive to suboptimal learning rates: if the learning rate is too low, the norm of the weights increases to compensate and vice versa [9]. Weight normalization reparameterizes a neural network weight $w \in \mathcal{R}^{m \times n}$ as a magnitude vector $g \in \mathcal{R}^m$ and a direction vector $v \in \mathcal{R}^{m \times n}$. During the backward pass, gradients propagate independently through g and v. During the forward pass, w is recomputed as $w = \frac{g}{\|v\|}v$. Thus, during inference time it is straightforward to precompute w from g and v to avoid the row-wise norm, broadcast division, and broadcast multiply operations.

2.2. Fusing convolution and bias kernels

The convolution and transpose convolution layers in MelGAN include bias terms, meaning that a learned scalar is broadcast-added to each channel of the activation after the convolution operation. In PyTorch, this is performed by loading the input to the convolution layer from high-latency global GPU memory to

[1] https://github.com/descriptinc/melgan-neurips

Layer	PyTorch	CUDA	Speedup
Leaky ReLU	**11.84**	12.55	0.940
Tanh	11.44	**9.55**	1.200
Reflection padding	12.14	**11.29**	1.080
Convolution	**82.25**	316.08	0.260
Transpose convolution	**76.89**	1335.57	0.058
Full model	224287.24	**59335.9**	3.78

Table 1: *Microbenchmarked timing results between the original PyTorch implementation and our proposed CUDA implementation. All times are averages over 100 trials in microseconds.*

faster shared memory, performing the convolution within shared memory, writing the result back to global memory, and then reading and writing *again* from global memory when applying the bias term. We improve upon this by adding the bias term while the activation is loaded in shared memory, thus omitting the additional reading and writing of the slower global memory. More concretely, this involves using the CUDNN function `cudnnConvolutionBiasActivationForward` instead of functions `cudnnConvolutionForward` and `cudnnAddTensor`. At the time of writing, the lack of support for this kernel fusion in PyTorch is the subject of an open issue[2].

3. Evaluation

Our goal is to show that our CUDA-based implementation of MelGAN can achieve a higher bandwidth than the original PyTorch implementation. We provide a micro-benchmark of the neural network layers for both the original implementation and our proposed CUDA implementation. We omit comparison with inference-time optimization tools for deep learning such as TensorRT for simplicity, but we note that a more complete evaluation of our work should include these direct comparisons. For input, we use one audio file consisting of 227,072 samples at a sampling rate of 22,050 Hz. We preprocess the audio as in the original implementation, yielding an input log-mel-spectrogram consisting of 80 channels and 887 frames. The per-layer timings and relative speed-ups are shown in Table 1. All values are the result of 100 trials on a NVIDIA RTX 2060 Super and are given in microseconds.

4. Discussion

The results in Table 1 indicate that our proposed CUDA implementation can significantly outperform the PyTorch implementation when performing full model inference, but lags behind the PyTorch implementation on key layers such as convolution and transposed convolution. Further evaluation is needed to completely resolve the discrepancy that PyTorch is faster during crucial layers but slower overall. We provide hypotheses for why our convolution and transposed convolution layers are slower, why PyTorch may exhibit slow inference despite its high layer bandwidth, and how we can further improve the overall network performance.

4.1. Towards faster convolutions

The CUDNN framework relies on the user to carefully manage allocated memory to achieve maximal performance. Specifically, the CUDNN Developer Guide [11] provides the following guidelines for deep learning applications:

- "Transform the inputs and filters to NHWC, pre-pad channel and batch size to be a multiple of 8"
- "Make sure that all user-provided tensors, workspace, and reserve space are aligned to 128-bit boundaries"

We note that these details were not addressed in our CUDA implementation. This is a likely cause for the discrepancy between our convolution layer timings and those reported by PyTorch.

4.2. Understanding the performance of PyTorch

The PyTorch microbenchmark demonstrates that individual layers are fast, but the full model is significantly slower. How can we justify this discrepancy? We hypothesize two possibilities: (1) running a single PyTorch layer many times in a loop induces caching behaviors, whereby the result is read directly from a node in a compute graph rather than recomputed or (2) slow reading and writing of memory occurs between layers. Further evaluation is necessary to determine which (if any) of these hypotheses hold.

4.3. Towards faster model inference

We include here two additional avenues for improving performance that were not attempted: (1) performing inference with 16-bit floating point values and (2) increasing the batch size via slicing. It is well known that using 16-bit floating point values can improve network speed and reduce memory consumption [12]. This improves speed because the GPU tensor cores used to perform the convolution operations expect 16-bit inputs. If the inputs are not 16-bit, casting is performed. We can avoid this casting by using only 16-bit floating point values for all activations, kernels, and biases.

When performing a CUDNN convolution with a batch size that is not a multiple of 8, CUDNN will automatically pad up to the next multiple of 8. For a batch size of 1, this incurs a significant penalty at every convolution. While pre-padding inputs to have a batch size that is a multiple of 8 can prevent the additional padding operation at every convolution layer, it would be far more efficient to make use of the sequential nature of our input by slicing the input into 8 subsequences and always running with a batch size of 8. Careful slicing and crossfading may be necessary to avoid perceptual artifacts at the slice points when reconstructing the final signal.

5. Conclusion

Improving the bandwidth neural vocoders directly directly benefits text-to-speech application domains such as voice assistants and screen readers. In this paper, we improve on the bandwidth of the MelGAN vocoder via direct implementation in CUDA and manual kernel fusion. We demonstrate a 3.78x speedup relative to a baseline PyTorch implementation, and discuss avenues for further improvement.

6. References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[2] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *arXiv preprint arXiv:1609.03499*, 2016.

[2]https://github.com/pytorch/pytorch/issues/3823

[3] R. Prenger, R. Valle, and B. Catanzaro, “Waveglow: A flow-based generative network for speech synthesis,” in *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2019.

[4] J.-M. Valin and J. S., “LPCNet: Improving neural speech synthesis through linear prediction,” in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2019.

[5] K. Kumar, R. Kumar, T. de Boissiere, L. Gestin, W. Z. Teoh, J. Sotelo, A. de Brébisson, Y. Bengio, and A. C. Courville, “MelGAN: Generative adversarial networks for conditional waveform synthesis,” in *Advances in Neural Information Processing Systems*, 2019.

[6] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in pytorch,” 2017.

[7] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, “Tensorflow: A system for large-scale machine learning,” in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[8] H. Vanholder, “Efficient inference with tensorrt,” 2016.

[9] T. Salimans and D. P. Kingma, “Weight normalization: A simple reparameterization to accelerate training of deep neural networks,” *Advances in neural information processing systems*, vol. 29, pp. 901–909, 2016.

[10] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, “cudnn: Efficient primitives for deep learning,” *arXiv preprint arXiv:1410.0759*, 2014.

[11] “Cudnn developer guide,” 2020, https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html.

[12] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh *et al.*, “Mixed precision training,” *arXiv preprint arXiv:1710.03740*, 2017.