White Paper

FPGA
Artificial Intelligence

Implementing WaveNet Using Intel® Stratix® 10 NX FPGA for Real-Time Speech Synthesis

Authors

Jonny Shipton
Software Lead
Myrtle.ai

Jon Fowler
FPGA Lead
Myrtle.ai

Chris Chalmers
Senior Developer
Myrtle.ai

Sam Davis
Head of Machine Learning
Myrtle.ai

Sam Gooch
Machine Learning Engineer
Myrtle.ai

Giuseppe Coccia
Machine Learning Engineer
Myrtle.ai

Table of Contents

Abstract

Real-time text-to-speech models are widely deployed in interactive voice services such as voice assistants and smart speakers. However, deploying these models is often challenging due to the strict latency requirements that they face. We present an implementation of WaveNet, a state-of-the-art vocoder, that can generate 256 16 kHz audio streams at near-human level quality in real time: 8 times higher throughput than a hand optimized GPU solution. Our implementation is enabled by the Intel® Stratix® 10 NX FPGA, demonstrating the capability of FPGAs to deliver high-throughput, low-latency inference for realworld applications powered by neural networks.

I. Introduction

The increasing use of voice assistants has been powered, in large part, by advances in the field of Text-to-Speech (TTS). State-of-the-art speech synthesis systems consist of two neural networks that are run sequentially to generate audio. The first model takes text as input and generates acoustic features, such as spectrograms, while the second model, referred to as a vocoder, takes these intermediate features and produces speech. Tacotron 2 is often used as the first model. In this paper, we focus on the second model in the speech synthesis system. WaveNet [1] is a state-of-the art vocoder that is capable of producing speech with near-human-level naturalness [2]. The key to the model's quality is its autoregressive loop but this property also makes the network exceptionally challenging to implement for real-time applications.

As a result, TTS research has focused on finding alternative vocoder architectures such as Parallel-WaveNet [3], WaveRNN [4], ClariNet [5] and WaveGlow [6] that are better suited to efficient inference on existing hardware. There is a degree of ambiguity as to the highest quality vocoder, as audio quality evaluation is subjective, but all authors agree the original WaveNet architecture produces at least as good, if not higher, quality audio than more recent approaches [3], [6]–[9].

There have been some efforts made to accelerate the WaveNet model directly rather than using an alternative architecture, including some other FPGA solutions [10]. However, these are not known to achieve real-time audio synthesis. The implementation with the highest performance that we have been able to identify is the NVIDIA* nv-wavenet[1] that can achieve real-time audio synthesis using highly-optimised hand-written CUDA kernels.

In this paper, we implement a WaveNet model, with approximately the same number of parameters in the repeated part of the network, as the largest configuration of nv-wavenet available in the example repository. The repeated part of the network is the most latency sensitive section. By using Block Floating Point (BFP16) quantization and the Intel Stratix 10 NX FPGA, we are able to deploy this model and produce 256 16 kHz streams in real time. We show that the FPGA implementation remains efficient at higher frequencies, demonstrating 160 24 kHz streams and 128 32 kHz streams in real time.

The rest of this paper is structured as follows: the WaveNet model architecture is presented in more detail in Section II. The concepts of model compression by the use of quantization and the BFP16 data format are discussed in Section III. Section IV describes the implementation of the WaveNet model on Intel Stratix 10 NX FPGA, including the use of AI Tensor Blocks, BFP16 computation and high-bandwidth memory (HBM). In Section V we present our results, including the performance of the implementation, as well as the quality of the generated audio. Finally, in Section VI, we summarize our results and interpret their importance for future deployments of neural networks in the data center with FPGAs.

[1] https://github.com/NVIDIA/nv-wavenet

LAYER	TYPE	# PARAMETERS		GOP/SECOND AUDIO	
		Per-layer	Total	Per-layer	Total
Pre-processing Layers					
Embedding	Embedding		30,720		-
Feature Upsample	ConvTranspose1d		5,120,080		0.82
Repeated Layers					
Dilation	Dilated Conv1d	57,840	925,440	1.84	29.49
Conditional	Conv1d	19,440	311,040	0.61	9.83
Residual	Conv1d	14,520	217,800	0.46	6.91
Skip	Conv1d	29,040	464,640	0.92	14.75
Post-processing Layers					
Out	Conv1d		61,440		1.96
End	Conv1d		65,536		2.09
Total			7,196,696		65.85

Table 1. A Breakdown of the Model's Parameters and Giga-Operations Required per Second of Synthesized Audio Output.

II. Model Description

The task of audio generation is that of producing a sequence of values $[x_1,...,x_n]$. Just like in a regular audio file, each value x_i is an integer representing a discrete sample from a waveform. There are two critical parameters of any audio stream: the sample rate, which is the number of these samples that make up a single second of audio, measured in Hertz; and the bit depth, which is the range of discrete values that each x_i can take. Increasing these parameters improves the quality of the audio stream but requires the production of more samples per second or an increase in the number of computations required for each sample. In this paper, we use a sample rate of 16 kHz and a bit depth of 16, giving each sample a range of 65,536 possible values, as we find these to achieve both near-human-level quality and high performance.

WaveNet is a convolutional neural network that models the conditional probability of producing a sample given all previous samples: $p(x_t|x_1,..., x_{i-1})$. The input to the first layer of the model at a given step is an embedding of the model output from the previous time step. The core of the network consists of a series of repeated layers that each contain four convolutions; see Figure 1. The first convolution in each repeated layer is a dilated convolution where the input is the concatenation of the output of the previous layer at the current step, t, and a past step. The past step for the k^{th} repeated layer depends on the dilation cycle parameter D and is set to step $t - 2^{k \bmod D}$.

Each repeated layer also produces a skip output. The skip outputs are summed and used to produce the final model output after post-processing with two more convolutional layers. In addition to the outputs from the previous layer, each repeated layer also receives a conditional input that allows control over what audio is generated. The conditional inputs are generated by applying an upsampling transposed convolution to conditional features. In early versions of WaveNet these conditional features were simple linguistic features, however, current approaches typically use spectrograms generated by a separate model such as Tacotron 2 [2] or Deep Voice 3 [11].

After the two post-processing convolutional layers are applied, the model samples from the probability distribution $p(x_t|x_1,..., x_{i-1})$; we find sampling to produce higher fidelity speech than other methods such as selecting the value with highest probability. The model produces audio output that has a bit depth of 16. However, to reduce the computational overhead of producing 2^{16} = 65,536 values the model instead generates 8-bit μ-law encoded values that are then decoded to produce the final 16-bit audio sample.

At inference time, calculating the output for one step requires the value of the previous step, so it is necessary to calculate each step individually and sequentially. For 16 kHz audio it is necessary to run the model 16,000 times per second to achieve real-time audio generation. This means that a single forward pass through the model must take at most 62.5µs.

WaveNet is parameterized by the number of channels in its convolutions: the number of skip, residual and audio channels, as well as by the number of repeated layers and dilation cycle parameter D. In this paper we use a model with **s** = 240 skip channels, **r** = 120 residual channels, **a** = 256 audio channels, **L** = 16 repeated layers and D = 8. This model has 7.2 million parameters, as shown in Table 1, and requires approximately 65.85 GFLOPs of computation to generate one second of audio. We map the ConvTranspose1d layer to CPU and all other layers to the FPGA. This significantly reduces model parameters that must be held on the FPGA, requiring approximately 2.1 million parameters to be stored on the FPGA overall. The layers running on the FPGA still account for the majority of the computation, requiring 98.8% of the total operations of the model.

III. Model Compression

Quantization is one technique for model compression. Quantizing a model consists of using a different numerical format for the parameters and/or activations. This lowers the power consumption and latency, as both the memory requirements are reduced and the operations can be more efficiently executed in hardware.

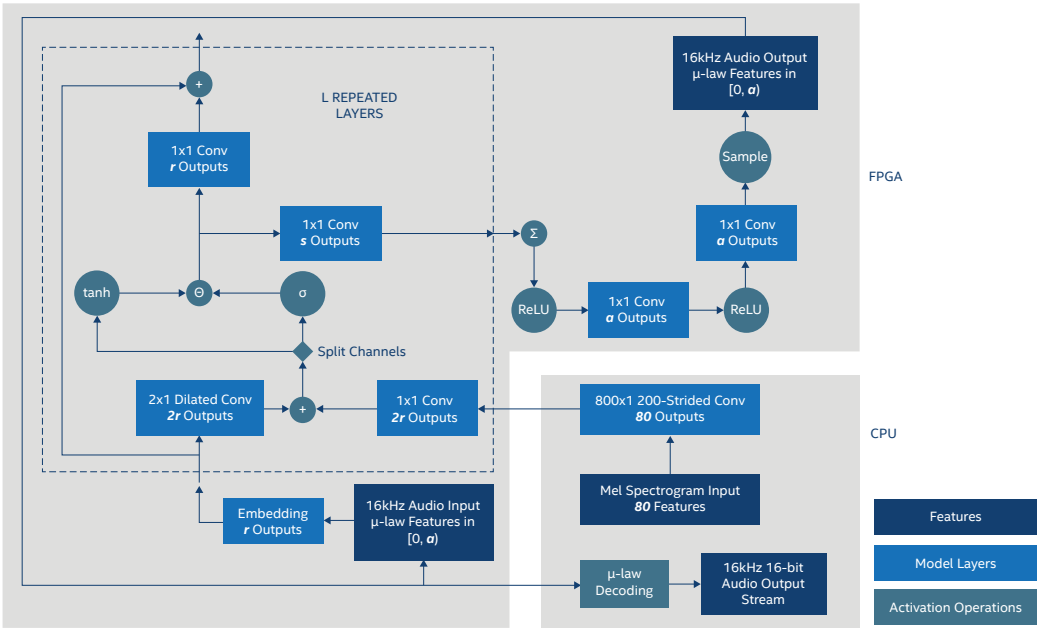

Figure 1. The WaveNet architecture where the number of skip, residual and audio channels is parameterised by **s, r, a** and the number of layers by **L**.

Converting parameters and activations to a different number format may require not only a reduction in the number of bits used to represent each value (e.g. 16 bits instead of 32 bits), but also a change to the arithmetic used when executing operations with these values (e.g. integer arithmetic instead of floating point).

There are two approaches that can be followed to quantize a model. The first approach is post-training quantization (PTQ) that converts the parameters to the desired numerical format after the model has been trained. This approach is simple but, in some cases, can negatively impact the quality or accuracy of the model. The second approach, Quantization Aware Training (QAT) [12], mitigates the quality degradation by simulating the target numerical precision to the weights during training. In this way, the model is better able to remove the noise generated by the quantization process.

BFP16 Quantization

The Intel Stratix 10 NX FPGA incorporates AI Tensor Blocks that compute with Block Floating Point 16 (BFP16). BFP16 is a number format that aims to reduce the quantization error compared to IEEE standard 16-bit floating point number format (FP16), by keeping the same number of bits but changing the process by which the values are quantized. It is a number format that is used especially in FPGA and ASIC processors because it allows the use of floating point arithmetic on their fixed-point arithmetic units, which are much more efficient than the floating point units.

To quantize a cluster of numbers to BFP16, it is necessary to define the desired block size N. The target cluster is divided into several blocks with a size determined by the parameter previously set, so that each block X becomes

$$X = (x_1,...,x_i,...,x_N)$$

where x_i is the i^{th} value in the cluster.

The following step is to find the maximum exponent ϵ_X for each block

$$\epsilon_X = \max_{i \in [1,N]} e_i$$

where e_i is the exponent of the i^{th} number in the cluster.

This exponent is the one that is shared by all numbers inside the target cluster after quantization, whereas the mantissa values are shifted such that they match the maximum exponent

$$m'_i = m_i > (\epsilon_X - e_i)$$

where: m'_i is the mantissa of the i^{th} number after quantization; m_i is the mantissa of the i^{th} number before quantization; > represents the right-shift bitwise operation.

The block floating point representation contributes to saving both the memory and interconnect bandwidth requirements because, if L_e is the number of bits for the exponents, L_m the number of bits for the mantissas and 1 bit is used to represent the sign, then the average length of n numbers is $1 + L_m + L_e/n$. Conversely, by using a floating point representation, the average length of n numbers is $1+L_m+L_e$, because all numbers have different exponent values.

BFP16 uses 7 bits for the mantissa, 8 bits for the exponent and 1 bit for the sign. The block size is an important parameter because it controls the trade-off between the error generated during the quantization process and the memory/bandwidth required for each operation. The bigger the block size, the bigger the quantization error, but the less space and bandwidth required for the representation.

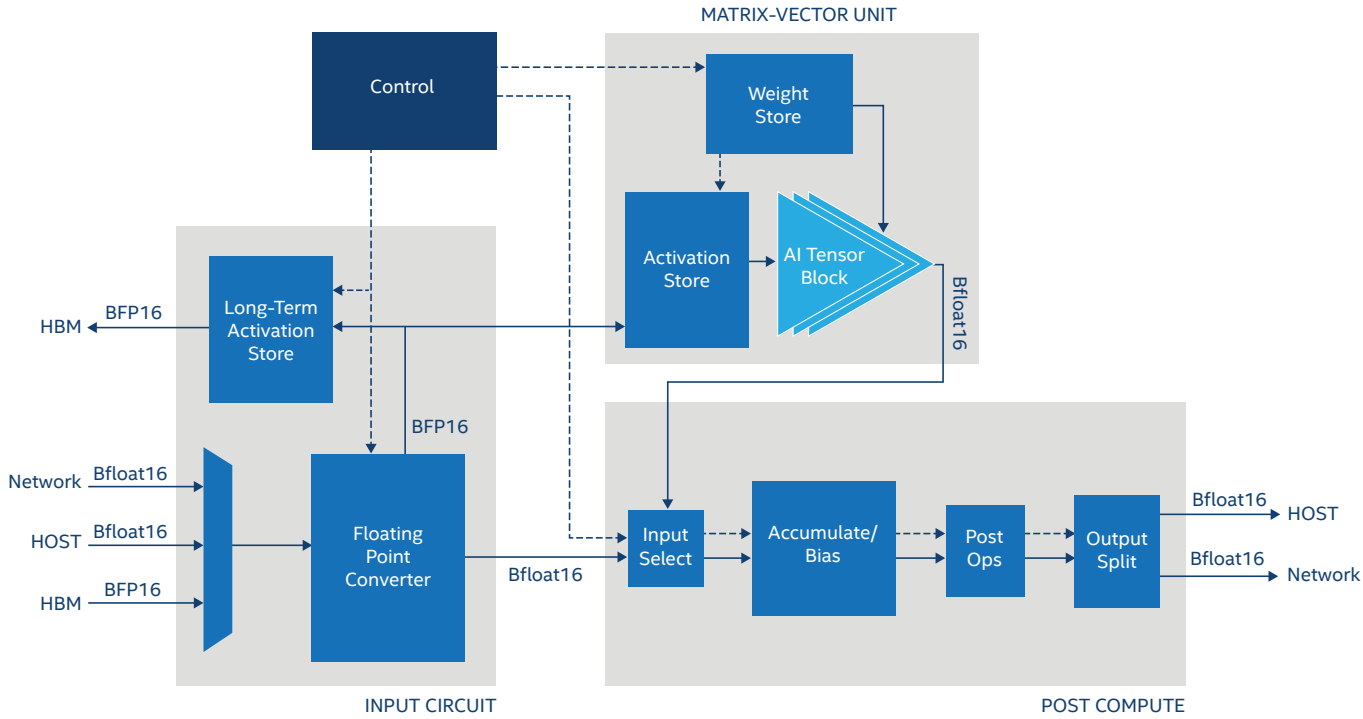

Figure 2. The MAU Core Architecture on Intel Stratix 10 NX FPGA.

IV. Hardware Implementation

We implement the WaveNet model on an Intel Stratix 10 NX FPGA. All layers are processed on the FPGA with the exception of the ConvTranspose1d layer at the input. This preprocessing layer is not dependent on any previously generated values and so it can be performed outside of the core autoregressive WaveNet loop. It is also processed less frequently than the autoregressive loop at a frequency of 12.5 ms, and without the latency requirement of the main autoregressive loop. We implement this layer on CPU so that the FPGA processing can be dedicated to the latency critical autoregressive part of the network. The remainder of the network runs on the FPGA, including the 1x1 conv layer that follows the conditional upsampling layer. While this layer is not strictly part of the core autoregressive loop, we place this layer on FPGA, to make use of the computation capability provided, for this layer, which contributes a significant number of operations to the overall computation in the network.

FPGA Processing Architecture

To realise the implementation of the network on the FPGA, we use the Myrtle.ai programmable MAU* Core architecture. The MAU Core is a programmable processing engine for deep neural networks, that overlays the FPGA fabric to provide a flexible and run time configurable inference engine. We place 4 MAU Cores, optimized for the Intel Stratix 10 NX FPGA architecture, onto the FPGA. We floorplan the MAU Cores in the Intel® Quartus® Prime software to ensure that a high level of logic utilization can be achieved, whilst retaining a high clock frequency. The FPGA uses a core processing frequency of 240 MHz in this design.

Each MAU Core connects to the host CPU via PCIe, a dedicated high-bandwidth memory (HBM) interface and to the other MAU Cores via a routing network. The routing network between each pair of neighbouring MAU Cores carries 120 bfloat16 values on each clock cycle, resulting in an internal bus bandwidth of 60 GBps.

The architecture of the Intel Stratix 10 NX FPGA optimized MAU Core is shown in Figure 2. This has all the functionality required to implement the convolutions within each layer. In addition to convolutional layers, the MAU Core also has functionality for tanh, sigmoid, ReLU, gated activations, softmax sampling and one-hot embedding operations, that form the post compute unit of the MAU Core.

Using Intel Stratix 10 NX FPGA AI Tensor Blocks

The Intel Stratix 10 NX FPGA features an innovative AI tensor block that enables 15 times greater INT8 Tera Operations Per Second (TOPS) compared to other FPGAs in the Intel Stratix 10 FPGA range. To achieve efficient implementation of the 1D convolutional layers using Intel Stratix 10 NX FPGA AI Tensor Blocks we configure the blocks to operate in Tensor Mode. Computation is performed in BFP16, using a shared exponent for each block of 10 matrix elements. Each Intel Stratix 10 NX FPGA AI Tensor Block can compute three dot products of size 10 per clock cycle. Four AI Tensor Blocks are cascaded to form a dot product width of 120. The architecture of the Intel Stratix 10 NX FPGA AI Tensor Block is shown in Figure 3.

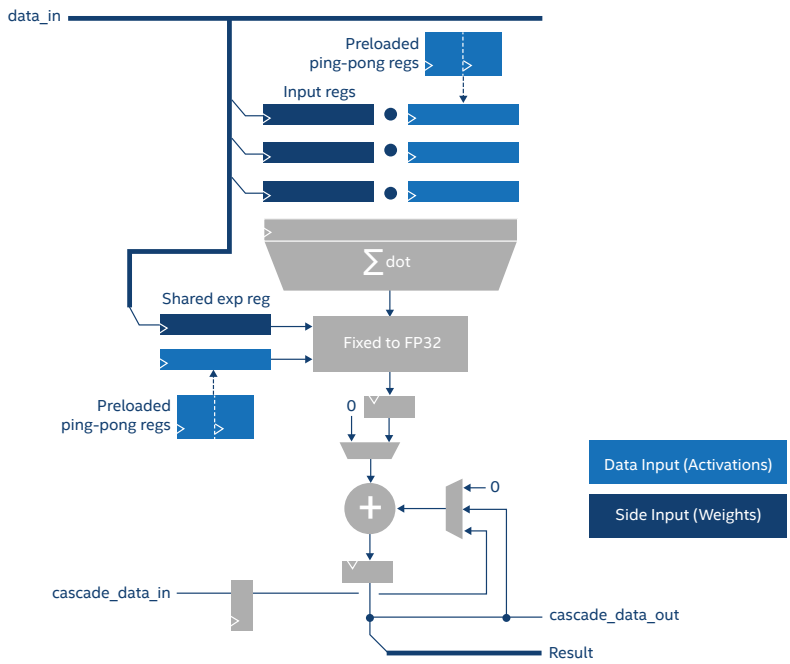

Figure 3. Intel Stratix 10 NX FPGA AI Tensor Block.[2]

The model weights are loaded by the side input. The AI Tensor Block requires 18 clock cycles to load new weights, and this update occurs in parallel with computation through the use of a ping pong register. We process different channels in a batch of 32 so that the weights can be updated every batch, utilising the Tensor Mode operation at maximum efficiency. We use 480 AI Tensor Blocks in each MAU Core. As a result each MAU Core can execute a single 120 x 120 matrix vector multiplication per clock cycle. This gives the accelerator design a total capacity of 27.6 TOPS across all four cores.

Model parameters are held locally in on-chip RAM. WaveNet is a relatively small model, so there is sufficient on-chip memory available to hold all network parameters locally to the AI Tensor Blocks. The design allocates 4.5 MB of local storage to network parameters with 1.3 TBps of bandwidth.

Implementing Dilated Convolutions on the FPGA

During inference, we make use of the Fast WaveNet Generation Algorithm [13] which replaces the dilated convolutions in the model architecture with regular convolutions and a buffer to store previously computed values. The correct values to simulate the dilated convolution can then be selected from the buffer. This avoids having to recompute values from previous samples when generating each new value as we can simply look up the value in the buffer, greatly improving the model performance.

Dilated convolutions in the network require activation data from up to 128 previous timesteps. This places a high throughput and low latency requirement on storage of these intermediate parameters. Internally to the WaveNet network the bandwidth requirement to retrieve activations to feed dilated convolutions is 3.1 GBps per MAU Core. With a need to store up to 128 activations from previous timesteps, each core requires a total storage of 10.3 MB. Since this size of data store is too large to be accommodated locally inside of the core we use the HBM memory to store this information.

The use of HBM is preferred over external DDR4 memory on the platform as the bandwidth available from the HBM on the platform is 256 GBps vs approximately 21 GBps available from external DDR4 memory, providing more overhead on bandwidth for data movement within the system.

Programming Model

The MAU Cores are programmed at run time for the WaveNet model. This enables the FPGA to be retargetted to different model architectures without requiring a recompile of the FPGA implementation.

We map WaveNet to the four MAU Cores by running multiple WaveNet layers on each core and time sharing the core logic. The MAU Core controller handles the execution of the core logic. Each core operates on a batch of 32 voice channels, operating the same instruction over each channel of the batch on consecutive clock cycles. Once all layer operations have completed for the batch, the activations advance to the next MAU Core for processing by later layers. The four cores are pipelined such that a batch of 32 voice channels are processed in each of the four cores simultaneously, giving 128 voice channels in flight in the FPGA at any one time.

To process greater than 128 concurrent voice channels in real time, the voice channels are processed as chunks, where a chunk is an integer number of Mel Spectogram inputs. The FPGA is able to generate samples at significantly faster than real-time, enabling the FPGA acceleration to be time shared. The FPGA is designed to enable chunks to be processed in a non-continuous mode of operation, with intermediate state for all active concurrent voice channels held within the HBM memory.

[2] Credit Graham McKenzie, Intel Programmable Solutions Group.

For WaveNet we map the network layers to MAU Cores as shown in Table 2.

# CORE	LAYER MAPPING
1	Input Embedding & WaveNet Layers 1-3
2	WaveNet Layers 4-8
3	WaveNet Layers 9-13
4	WaveNet Layers 14-16 & Output Convolutions

Table 2. Mapping WaveNet to MAU Cores

V. Results

Methodology

We use the LJSpeech dataset subsampled to 16 kHz [14]. Prior to starting this project, 100 samples were randomly selected from the dataset for use as a validation set and 100 samples were randomly selected from the dataset for use as a test set. All remaining samples are used as a training set. No data augmentation is applied.

The WaveNet architecture parameters are as defined in Section II. All models are trained for 140,000 steps with a batch size of 16 distributed across 1–8 GPUs using mixed precision training. One element in the batch consists of a randomly selected 1 second segment from an audio clip in the training set, padding with zeros where necessary. We use the Adam optimizer [15] with a fixed learning rate of 10^{-3}, $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$. We use PyTorch [16] to implement WaveNet and QPyTorch [17] to simulate BFP16 when using QAT.

We report both the teacher-forced cross-entropy validation and test loss as well as the Mean Opinion Score (MOS) for each model. For each model, we repeat the training process 3 times and select the model with median validation loss to compute and report the test loss and test MOS score. The MOS is computed by asking 30 independent Amazon Mechanical Turk workers to rate each of the generated samples on how natural the samples sound on a five point scale, see Table 3. The reported MOS is the mean of these scores and a 95% confidence interval is computed using the t-distribution.

RATING	LABEL	DESCRIPTION
1	Bad	Completely unnatural speech
2	Poor	Mostly unnatural speech
3	Fair	Equally natural and unnatural speech
4	Good	Mostly natural speech
5	Excellent	Completely natural speech

Table 3. Mean Opinion Score Scale

Model Quality

The loss on the validation and test set as well as the MOS for the baseline FP32 model and the BFP16 models using PTQ and QAT are presented in Table 4. We find that both the BFP16 (PTQ) model and the BFP16 (QAT) model are capable of synthesizing high fidelity audio that is near the baseline FP32 model.

MODEL	VALIDATION LOSS	TEST LOSS	TEST MOS
Human	-	-	4.056 ± 0.034
FP32	2.189	2.182	3.976 ± 0.027
BFP16 (PTQ)	2.203	2.197	3.711 ± 0.029
BFP16 (QAT)	2.195	2.188	3.823 ± 0.028

Table 4. Quality Results for the FP32 and BFP16 WaveNet Models.

Model Performance

We present key parameters and processing performance in Table 5 for the WaveNet model running on an Intel Stratix 10 NX FPGA. Comparisons are made against nv-wavenet running on V100 Graphics Processing Unit (GPU). At the time of writing, this is the fastest previous implementation of WaveNet that we have been able to reference.

We make power comparisons based on Thermal Design Power (TDP) of the two platforms. This is an overestimate of true power consumption for the application for both platforms. TDP for the Intel Stratix 10 NX FPGA is based on design analysis provided by Intel for a PCIe-based accelerator card deployment.

The number of concurrent voice channels is the maximum number of channels that can be generated in real time to meet a 16 kHz sample rate. The FPGA implementation uses slightly smaller layer sizes than the GPU implementation, but has a larger number of model parameters and operations, due to the inclusion of the conditional layer in the accelerated part of the model, which is not part of the nv-wavenet implementation.

The Intel Stratix 10 NX FPGA implementation demonstrates a 8 times improvement in number of concurrent voice streams when compared to nv-wavenet. The FPGA is able to achieve a 8.6 times improvement in achieved TOPS for the WaveNet model.

The FPGA provides a significantly more power efficient solution, providing a 9.3 times improvement in voice channels per watt and a 10 times improvement in GOPS/W.

We present the number of concurrent voice channels for audio frequencies at 16 kHz and above in Table 6. This shows that the FPGA solution remains efficient, even as the latency requirement to produce a single time step is reduced.

	MYRTLE.AI WAVENET	NV-WAVENET
Platform	Intel® Stratix® 10 NX FPGA	NVIDIA* V100 GPU
Frequency (MHz)	240	1530
Numerical Precision	BFP16 / bfloat16	fp16
WaveNet Configuration	r=120, s=240, L=16, a=256, D=8	r=128, s=256, L=16, a=256, D=8
Operations per 1 second audio (GOPS/second)	65.03	60.36
Model Parameters (Millions)	2.08	1.99
Concurrent Voice Channels	256	32
Application TOPS	16.6	1.93
TDP Power (W)	215	250
Performance per Watt (GOPS/W)	77.4	7.7
Voice Channels per Watt (1/W)	1.19	0.128

Table 5. Performance Results for WaveNet Implementation at 16 kHz.

AUDIO FREQUENCY	CONCURRENT VOICE CHANNELS	
	MYRTLE.AI WAVENET	NV-WAVENET
16 kHz	256	32
24 kHz	160	16
32 kHz	128	8

Table 6. Performance Results for WaveNet Implementation for Different Audio Frequencies.

We implement the ConvTranspose1d on an Intel® Xeon® processor dual socket 16 core CPU running at 2.8 GHz. Each input step corresponds to 200 output steps (stride) and hence one input step must be processed every 12.5 ms to generate output steps for 16 kHz audio. We implement our own ConvTranspose1d variant, which runs at an order of magnitude faster than the PyTorch native implementation. We use a batch size of 64 running independently on each socket. We measure the 99.999-ile latency at 3.85 ms for each batch, sufficiently fast to run 3 batches of 64 on each socket within the 12.5 ms processing requirement. This configuration enables the CPU to compute the ConvTranspose1d for up to 384 concurrent voice channels, sufficient to feed the FPGA implementation, and demonstrating a full system implementation of WaveNet capable of generating 256 concurrent voice channels.

VI. Conclusion

This paper has shown that real-time performance can be achieved for 256 concurrent streams of the state-of-the-art WaveNet model to achieve near-human levels of synthesized speech by using a dedicated FPGA-based accelerator. This is an improvement of 8 times the best GPU solution currently available for this model.

We show that in higher frequency deployments, the FPGA advantage increases further, enabling a 10 times advantage at 24 kHz and 16 times advantage at 32 kHz, compared to the best GPU solution currently available. This enables a cost effective platform for deployment of WaveNet at higher audio frequencies, enabling 128 concurrent streams of 32 kHz audio on one accelerator.

We show that the FPGA is able to provide a significant energy reduction in the implementation ofWaveNet, a 10 times improvement compared to a GPU implementation, enabling significant energy savings to be made by those who deploy real-time speech synthesis at scale.

We demonstrate that the BFP16 format can be applied post training to enable a simple quantization flow from FP32, with minimal loss of accuracy. This provides a simple and effective quantization flow from machine learning frameworks, whilst enabling the benefit of a more efficient hardware implementation.

References

[1] 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] J. Shen, R. Pang, R. J. Weiss, M. Schuster, N. Jaitly, Z. Yang, Z. Chen, Y. Zhang, Y. Wang, R. Skerrv-Ryan et al., "Natural tts synthesis by conditioning wavenet on mel spectrogram predictions," in 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2018, pp. 4779–4783.

[3] A. Oord, Y. Li, I. Babuschkin, K. Simonyan, O. Vinyals, K. Kavukcuoglu, G. Driessche, E. Lockhart, L. Cobo, F. Stimberg et al., "Parallel wavenet: Fast high-fidelity speech synthesis," in International conference on machine learning, 2018, pp. 3918–3926.

[4] N. Kalchbrenner, E. Elsen, K. Simonyan, S. Noury, N. Casagrande, E. Lockhart, F. Stimberg, A. v. d. Oord, S. Dieleman, and K. Kavukcuoglu, "Efficient neural audio synthesis," arXiv preprint arXiv:1802.08435, 2018.

[5] W. Ping, K. Peng, and J. Chen, "Clarinet: Parallel wave generation in end-to-end text-to-speech," arXiv preprint arXiv:1807.07281, 2018.

[6] R. Prenger, R. Valle, and B. Catanzaro, "Waveglow: A flow-based generative network for speech synthesis," in ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2019, pp. 3617–3621.

[7] S. Kim, S.-g. Lee, J. Song, J. Kim, and S. Yoon, "Flowavenet: A generative flow for raw audio," arXiv preprint arXiv:1811.02155, 2018.

[8] Q. Tian, Z. Zhang, H. Lu, L.-H. Chen, and S. Liu, "Featherwave: An efficient high-fidelity neural vocoder with multi-band linear prediction," arXiv preprint arXiv:2005.05551, 2020.

[9] P.-c. Hsu and H.-y. Lee, "Wg-wavenet: Real-time high-fidelity speech synthesis without gpu," arXiv preprint arXiv:2005.07412, 2020.

[10] S. Hussain, M. Javaheripi, P. Neekhara, R. Kastner, and F. Koushanfar, "Fastwave: Accelerating autoregressive convolutional neural networks on fpga," arXiv preprint arXiv:2002.04971, 2020.

[11] W. Ping, K. Peng, A. Gibiansky, S. O. Arik, A. Kannan, S. Narang, J. Raiman, and J. Miller, "Deep voice 3: Scaling text-to-speech with convolutional sequence learning," arXiv preprint arXiv:1710.07654, 2017.

[12] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 2704–2713.

[13] T. L. Paine, P. Khorrami, S. Chang, Y. Zhang, P. Ramachandran, M. A. Hasegawa-Johnson, and T. S. Huang, "Fast wavenet generation algorithm," arXiv preprint arXiv:1611.09482, 2016.

[14] K. Ito and L. Johnson, "The lj speech dataset," https://keithito.com/ LJ-Speech-Dataset/, 2017.

[15] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," arXiv preprint arXiv:1412.6980, 2014.

[16] 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 32, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alch´e-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[17] T. Zhang, Z. Lin, G. Yang, and C. De Sa, "Qpytorch: A low-precision arithmetic simulation framework," arXiv preprint arXiv:1910.04540, 2019.

WP-01304-1.0