FastWave: Accelerating Autoregressive Convolutional Neural Networks on FPGA

Shehzeen Hussain*, Mojan Javaheripi*, Paarth Neekhara†, Ryan Kastner† and Farinaz Koushanfar*
*UC San Diego Department of Electrical and Computer Engineering
†UC San Diego Department of Computer Science
Email: ssh028@eng.ucsd.edu, mojavahe@eng.ucsd.edu

***Abstract*—Autoregressive convolutional neural networks (CNNs) have been widely exploited for sequence generation tasks such as audio synthesis, language modeling and neural machine translation. WaveNet is a deep autoregressive CNN composed of several stacked layers of dilated convolution that is used for sequence generation. While WaveNet produces state-of-the art audio generation results, the naive inference implementation is quite slow; it takes a few minutes to generate just one second of audio on a high-end GPU. In this work, we develop the first accelerator platform *FastWave* for autoregressive convolutional neural networks, and address the associated design challenges. We design the Fast-Wavenet inference model in Vivado HLS and perform a wide range of optimizations including fixed-point implementation, array partitioning and pipelining. Our model uses a fully parameterized parallel architecture for fast matrix-vector multiplication that enables per-layer customized latency fine-tuning for further throughput improvement. Our experiments comparatively assess the trade-off between throughput and resource utilization for various optimizations. Our best WaveNet design on the Xilinx XCVU13P FPGA that uses only on-chip memory, achieves 66× faster generation speed compared to CPU implementation and 11× faster generation speed than GPU implementation.**

I. Introduction

Autoregressive convolutional models achieve state-of-the-art results in audio [1]–[4] and language domains [5], [6] with respect to both estimating the data distribution and generating high-quality samples. Wavenet [1] is an example of autoregressive convolutional network, used for modelling audio for applications such as text-to-speech (TTS) synthesis and music generation. WaveNet has been rated by human listeners to provide substantially more natural sounding audio when compared to the best existing parametric and concatenative systems in TTS applications for both English and Mandarin [1]. Popular cloud based TTS synthesis systems such as Google Now and Google Assistant, that produce natural sounding speech, are built on WaveNet architecture [4], [7]. Alongside achieving state-of-the art results in the audio domain, convolutional models are prominent for natural language modeling tasks like text generation and machine translation [5].

Generally, both autoregressive convolutional neural networks (CNNs) and Recurrent Neural Networks (RNNs) [8] are widely popular for sequence modelling tasks. The main advantage of CNN based models is that they can achieve higher parallelism during training and can capture longer time-dependencies as compared to RNN based models [9], [10]. However, this comes at a cost of slower inference, since the inference still remains sequential and CNN based models are usually very deep. However, to overcome this problem, Fast-Wavenet [11] exploits the temporal dependencies by caching redundant computations using fixed-length convolutional queues and thus makes the generation time linear in length of the sequence. Such efforts have made it feasible to use autoregressive CNNs for practical sequence generation applications, as an alternative to RNN-based models.

While the Fast-Wavenet algorithm provides a speed-up in audio generation over naïve Wavenet implementation, the generation time is still high, even on a high-end GPU. It takes 120 seconds to generate 1 second of audio using Fast-Wavenet implementation on a NVIDIA TITAN Xp GPU. Prior works have shown FPGAs to be successful in accelerating the inference of pre-trained neural networks by providing custom data paths to achieve high parallelism. A vast amount of such research focuses on accelerating neural networks in the image domain [12], [13], speech recognition [14], [15] and language modelling [16]. To the best of our knowledge, similar efforts have not been made for accelerating neural networks for speech/audio synthesis.

We aim to accelerate audio synthesis using the autoregressive CNN model - WaveNet on FPGA. The primary challenges in deploying auto-regressive CNN inference on FPGA are designing modules for dilated convolutional layers, buffers for storing redundant computations using convolutional queues, and dealing with the large depth of these networks which is necessary to maintain high audio quality. In this work, we address these challenges of deploying large autoregressive convolutional models on FPGAs and perform a wide range of application and architectural optimizations. Furthermore, we comprehensively analyze and compare the performance of Fast-Wavenet implementation on FPGA with the CPU and GPU counterparts.

Summary of Contributions:

- Creation of the first accelerator platform for autoregressive convolutional neural networks. We deploy the fast inference model Fast-Wavenet [11] on Xilinx XCVU13P FPGA which achieves **11** times faster generation speed than a high-end GPU and **66** times faster generation speed than a high-end CPU.
- Development of reconfigurable basic blocks pertinent to autoregressive convolutional networks i.e., dilated causal convolutional layers, convolutional queues, and fully con-

978-1-7281-2350-9/19/$31.00 ©2019 IEEE

Authorized licensed use limited to: Univ of Calif San Diego. Downloaded on August 30,2024 at 07:55:35 UTC from IEEE Xplore. Restrictions apply.

nected layer. Our operations are powered by a fully-customizable matrix-multiplication engine that uses two levels of parallelism controlled by tunable parameters.

- Creation of an end-to-end framework that accesses only on-chip memory thereby ensuring high throughput and avoiding any latency caused by communication with off-chip memory.
- Exploration of the design space using different architectural and application optimizations, as well as comparing the performance and resource usage. We present extensive evaluation of throughput and power efficiency for our fully optimized and baseline designs.

II. Background

In this section, we provide a background on autoregressive convolutional neural networks. We choose WaveNet as an ideal representation of such models, and describe its overall generative architecture. We first elaborate on the 1D convolution operation as it is the core computation performed in the WaveNet model. Next, we explain WaveNet and its more efficient inference algorithm called Fast-Wavenet.

A. *1D Convolution*

The 1D convolution operation is performed by sliding a one dimensional kernel over a 1D input signal. Each output value at position i is produced by calculating the dot product of the kernel and the overlapping values of the input signal, starting from position i. More formally, for an input vector a of length n and a kernel k of length m, the 1D convolution is calculated as follows:

$$(a*k)_i = \sigma_{j=1}^{m} k_j \times a_{i-j+\frac{m}{2}} \tag{1}$$

where i is an arbitrary index in the output vector, which has a total length of $n-m+1$. The subscripts denote the indices of the kernel/input vectors.

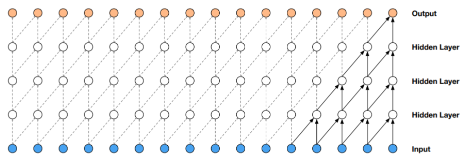
B. *WaveNet and Autoregressive CNNs*

Autoregressive Neural Networks are popularly used for sequence generation tasks which rely on ancestral sampling i.e. the predictive distribution for each sample in the sequence is conditioned on all previous ones. While RNNs are popular autoregressive models, they do not exhibit high receptive field making them unsuitable for modeling sequences with long-term dependencies like audio [9]. In contrast, autoregressive CNN based models use a stack of dilated convolutional layers to achieve higher receptive field necessary for modeling sequences with long-term dependencies.

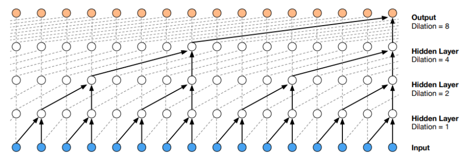
Wavenet [1] is an autoregressive convolutional neural network that produces raw audio waveforms by directly modeling the underlying probability distribution of audio samples. This has led to state-of-the-art performance in text-to-speech synthesis [2], [7], [17], [18], speech recognition [19], and other audio generation settings [1], [3], [4]. The Wavenet architecture aims to model the conditional probability among subsequent audio samples. The joint probability distribution of waveform sample points $x = x_0, x_1, ..., x_T$ can be written as: $P(x|\lambda) = \prod_{t=1}^{T} P(x_t|x_{t-1}, .., x_0, \lambda)$ where λ denotes the learnable parameters of Wavenet model. During inference, next-sample audio (x_t) generation is performed by sampling from the conditional probability distribution given all of the previous samples, $P(x_t|x_{t-1}, ..., x_1, x_0, \lambda)$.

One possible method for modeling the probability density is via a stack of causal convolutional layers as depicted in Figure 1(a). The input passes through this stack of convolutional layers and gated activation functions and finally through a softmax layer to get the posterior probability $P(x_t|x_{t-1}, ..., x_1, x_0)$. The downside of this approach is that in order to model long temporal dependencies from samples far in the past, the causal convolutional network requires either many layers or large filters to increase the receptive field. In general, the receptive field is calculated as $\#\ of\ layers + filter_{size} + 1$ which gives a receptive field of 5 in the architecture shown in Figure 1(a). To address this problem, WaveNet leverages dilated convolutions [20], [21] which deliver higher receptive fields without significant increase in computational cost of the model. Dilated convolution is equivalent to performing convolutions with dilated filters where the size of the filter is expanded by filling the empty positions with zeros. In practice, this is achieved by performing a convolution where the filter skips input values with a certain step.

Fig 1(b) illustrates a network with dilated causal convolutions for dilation values of 1, 2, 4, and 8. Here, the input nodes are shown with color blue and the output is shown with orange. Each edge in the graph correspond to a 1-dimensional convolution (See section II-A), more generally a matrix multiplication. Due to the binary tree structure of the network, the time complexity of computing output at each time-step is $\mathcal{O}(2^L)$ where L is the number of layers in the network, which gets highly undesirable as L increases. Similarly, the total memory required to store the inputs, output, and the intermediate layer features is $\mathcal{O}(2^L)$.



(a)



(b)

Fig. 1: **a.** (Left) Stacked causal convolution layers without any dilations. **b.** (Right) Stacked causal 1-d convolution layers with increasing dilation. (Figures from WaveNet paper [1]).

Authorized licensed use limited to: Univ of Calif San Diego. Downloaded on August 30,2024 at 07:55:35 UTC from IEEE Xplore. Restrictions apply.

C. *Fast-Wavenet*

The naïve implementation in Figure 1(b) has many redundant computations when generating a new sample, that is, it recomputes activations that have been already computed for generating previous samples. Fast-Wavenet [11] proposed an efficient algorithm that caches these recurrent activations in queues instead of recomputing them from scratch while generating a new sample. Fast-Wavenet uses a per-layer first-in-first-out queue to cache the states to be used in future timestamps.

The queue size at each layer is determined by its corresponding dilation value. Figure 2 demonstrates an example 4-layer network and their corresponding queues. For the first layer, the dilation value is 1 and therefore the corresponding queue ($Q1$) only keeps one value. Similarly, the output layer has a dilation value of 8, which means that its queue ($Q4$) will store 8 recurrent values. By removal of redundant computations due to the queue storing mechanism, the computational complexity of Fast-Wavenet is $\mathcal{O}(L)$ where L is the number of layers. The overall memory requirement for queues as well as the intermediate values remains the same as the naÿe implementation, i.e., $\mathcal{O}(2^L)$.

The basic queue operations performed in the Fast-Wavenet are as follows (refer to Figure 2):

1) **Pop phase**: The oldest recurrent states are popped off the queues in each layer and fed as input to the generation model. These popped off states and the current input are operated with the convolutional kernel to compute the current output and the new recurrent states.
2) **Push Phase**: Newly calculated recurrent states (orange dots) are pushed to the back of their respective layer queues to be used in future time stamps.

Maintaining the convolutional queues in the above manner allows us to handle the sparse convolutional operation and avoid redundant computations and makes the generation algorithm linear in terms of length of the sequence.

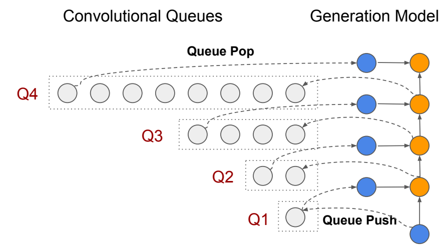


Fig. 2: Basic queue operations (Push and Pop) performed in Fast-Wavenet to achieve linear time in audio generation.

III. Methodology

Our primary objective is to accelerate the inference of autoregressive CNNs for sequence generation on FPGAs. As an ideal candidate for autoregressive models, we choose the WaveNet model for raw audio generation from random seed inputs. The computation and storage complexity of such state-of-the-art autoregressive CNNs is very high, particularly our FastWave architecture comprises of 28 convolutional layers with 128 channels each in order to maintain high audio quality. When designing an accelerator for such models, it is important to be aware of the system restrictions, particularly those of memory access bandwidth [12], [22]. Accessing off-chip memory is expensive and can limit the throughput of our network, making it important to compress DNNs into an optimal model for efficient inference.

Design Flow: We start with an open source TensorFlow implementation of the Fast-Wavenet algorithm. We save the weights of the convolutional and fully connected layers of our trained model which are used in the inference stage for generating audio. We implement the Fast-Wavenet inference in NumPy without using any high level deep learning libraries. This implementation serves as a bridge between the high level TensorFlow and the low level Vivado HLS implementation in C++. On the FPGA platform, we then accelerate the audio generation process from random seeds, and perform optimized operations using queue buffers and matrix-vector multiplications to generate raw audio.

A. *Model Architecture and Training on GPU*

Block No.	Layer No.	Filter Width	Queue Length	Input Channels	Output Channels	Queue Size
1	1	2	1	1	128	1
1	2	2	2	128	128	256
1	3	2	4	128	128	512
1	4	2	8	128	128	1024
1	5	2	16	128	128	2048
1	6	2	32	128	128	4096
1	7	2	64	128	128	8192
1	8	2	128	128	128	16384
1	9	2	256	128	128	32768
1	10	2	512	128	128	65536
1	11	2	1024	128	128	131072
1	12	2	2048	128	128	262144
1	13	2	4096	128	128	524288
1	14	2	8192	128	128	1048576
2	1	2	1	128	128	128
2	2	2	2	128	128	256
2	3	2	4	128	128	512
2	4	2	8	128	128	1024
2	5	2	16	128	128	2048
2	6	2	32	128	128	4096
2	7	2	64	128	128	8192
2	8	2	128	128	128	16384
2	9	2	256	128	128	32768
2	10	2	512	128	128	65536
2	11	2	1024	128	128	131072
2	12	2	2048	128	128	262144
2	13	2	4096	128	128	524288
2	14	2	8192	128	128	1048576

TABLE I: Details of Fast-Wavenet Architecture. The column *Queue Size* denotes the number of floating point numbers stored in each queue and is equal to $QueueLength \times InputChannels$.

We use an open-source TensorFlow implementation of Fast-Wavenet [11] to pre-train our network in Python. The network architecture we use is a stack of two dilated convolutional blocks. Each block consists of 14 convolutional layers with kernel size (filter width) = 2 and dilation increasing in powers of 2. Therefore each of the kernels is a 3-dimensional array of shape $2 \times inputchannels \times outputchannels$. The number of

Authorized licensed use limited to: Univ of Calif San Diego. Downloaded on August 30,2024 at 07:55:35 UTC from IEEE Xplore. Restrictions apply.

output channels in each layer is 128 in the baseline implementation. After each convolutional layer there is a $tanh$ activation function which serves as the non-linearity in our model as used in the original WaveNet paper [1]. A $tanh$ activation normalizes values between -1 and 1 and also allows us to better utilize fixed point data-types in the Vivado implementation without compromising on accuracy.

After the two convolutional blocks, we have a single fully connected layer which maps the activation of size 100 from the last convolutional layer to an output vector of size 256 followed by a softmax normalization layer. The output after the softmax layer is the generated distribution. The target audio is quantized linearly between -1 and 1 into 256 values. The one-hot representation of each sample of size 256 serves as the target distribution at each time-step. The cross entropy loss between the generated and target distribution is back-propagated to train the convolutional kernels and weights of the fully connected layer.

Memory challenges: The primary memory bottle-neck in implementing the Fast-Wavenet inference is not the parameters of the convolutional kernels, but convolutional queues which cache the intermediate outputs of the convolutional layers to be used for future predictions. The size of these queues increases exponentially with the depth of the block in the neural network. As highlighted in Table I, the 14th convolutional queue in each of the blocks stores $1,048,576$ floating point numbers ($\approx 33Mb$). One way of addressing this challenge is to reduce the number of channels in the 13th and 14th convolutional layers via pruning. However in our experiments, we found pruning to degrade the quality of generated audio. To address this memory challenge without pruning the network, we utilize both BRAMs and URAMs available on our FPGA board. We store all convolutional queues on the BRAMs by default and off-load the 14th convolutional queue of each block onto the URAMs on our board. In this way, we are able to utilize only on-chip memory and achieve higher bandwidth without compromising on audio quality.

B. *Accelerator Design Overview*

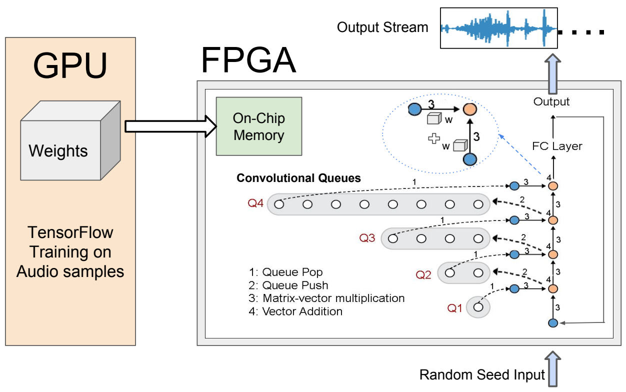


Fig. 3: Acceleration Methodology

The primary objective of our system is to generate an output stream given a seed input. Figure 3 shows the overview of our accelerator design. Given a seed input, our system generates an output stream in an autoregressive manner, one-sample at a time. The output sample produced at each time-step is fed back as input to generate the next output sample. During each cycle, as the input goes through all the convolutional layers, the corresponding convolutional queues are updated using push and pop operations as explained in section II. It is important to note that the entire model including the convolutional queues and the parameters does not use any off-chip memory and are stored in the BRAM and URAM available on the FPGA board. We describe the details of implementing the convolution operations, queue updates and output generation using the fully connected layers in the following section.

IV. Implementation Details

Our design is composed of 5 main elements: (i) The *dilated convolution* layers, (ii) *the queue control unit*, (iii) the *fully-connected* layer, (iv) the *matrix multiplication engine*, and (v) the *network description module*. We implement and accelerate the inference of WaveNet on the Xilinx XCVU13P FPGA.

A. *Dilated Convolution Layer*

As explained in the Section II-C, Fast-WaveNet leverages queues to implement the dilated convolutional layers. A convolution of size $= 2$ is used in the WaveNet architecture, and can be implemented as two matrix-vector multiplications followed by vector addition in the manner explained below. Notations used for our variables along with the shapes are listed below:

IC_n : Number of Input channels of layer n.
OC_n : Number of Output channels of layer n.
$O[n]_{(OC_n \times 1)}$: Output of convolutional layer n.
$K[n]_{(2 \times OC_n \times IC_n)}$: Convolutional kernel of layer n.
$Q[n]_{(queueLength \times IC_n)}$: Convolutional queue of layer n.

$$O_1[n] = K[n][0]_{(OC_n \times IC_n)} \times Q[n][0]_{(IC_n \times 1)}$$

$$O_2[n] = K[n][1]_{(OC_n \times IC_n)} \times O[n-1]_{(IC_n \times 1)}$$

$$O[n]_{OC_n \times 1} = O_1[n]_{(OC_n \times 1)} + O_2[n]_{(OC_n \times 1)}$$

In other words, we matrix-multiply the first component of the convolutional kernel with the first element of the queue, and the 2nd component of the kernel with the previous layer's output and then add the two products to obtain the output of any layer. The details of the matrix-vector multiplication engine have been provided in Section IV-D.

The output of the convolution layer is then passed to $tanh$ activation function. We use the CORDIC implementation available in Vivado HLS math library for applying $tanh$ allowing us to optimize our design and memory usage. The output of the dilated convolution module is a vector of length equal to the number of layer output channels.

Authorized licensed use limited to: Univ of Calif San Diego. Downloaded on August 30,2024 at 07:55:35 UTC from IEEE Xplore. Restrictions apply.

B. Cyclic Queue Buffer Unit

In order to reduce the number of operations, Fast-Wavenet aims to remove redundant convolution operations by caching previous calculations in a Queue, thereby reducing the complexity of synthesis to O(L) time. This means that after performing a convolutional operation, we push the compute into the end of the queue and pop the out the first element. These push and pop operations are shown in figure 3. As described above the queue in each layer $Q[n]$ is a 2-d array of shape $QueueLength \times InputChannels$. The $QueueLength$ depends on the $dilationFactor$ of the layer and is equal to $2^{dilationFactor}$. We aim to fit our queue computations in the on-chip memory BRAMs and URAMs. Our baseline queue implementation in Vivado HLS used shift operations to perform pop and push functionalities of a queue. The longest queue in our model is of size 8192×24. The shifting of a large number of elements in the queue resulted in very high latency.

To make queue push and pop operations computationally efficient, we implemented our queues using fixed length circular arrays for each layer. This is a lot more efficient than shifting all the elements present in the queue. The push and pop operations are reduced to just overwriting one column of our circular array which is indexed using modulo $QueueLength$ index.

C. Fully-connected Layer

The fully connected layer in WaveNet is a linear layer after all the convolutional layers. This layer is characterized by a weight matrix $W_{channels \times OutputSize}$ and a bias vector $b_{1 \times OutputSize}$. The fully connected layer performs the following operation on $ConvOut$: the output of the last convolution layer:

$$FinalOutput = ConvOut \times W + b$$

In our design, the weight matrix W has shape 100×256 and bias b has shape 1×256. We use arg-max sampling on the final vector of length 256 to obtain the quantized output value between -1 and 1.

D. Matrix Multiplication Engine

The most computationally-intensive operation in DNN execution is matrix-vector multiplication. FPGAs are equipped with DSP units which offer a high computation capacity together with the reconfigurable logic. The basic function of a DSP unit is Multiplication Accumulation (MAC). Layers in a convolutional neural network take as input a vector $X_{N \times 1}$ and compute the output vector $Y_{M \times 1}$ as formulated below:

$$Y = f(WX + b) \tag{2}$$

where f(.) is a nonlinear function, $W_{M \times N}$ is the 2D matrix of the weights and $b_{M \times 1}$ is a vector of bias values. As can be seen, each layer is computing a vector-matrix multiplication and a vector-vector addition. In order to optimize the design and make efficient use of the DSP blocks, we proposed a parallelized approach to convert layer computations into multiple MAC operations. Figure 4 presents our method to parallelize the matrix-vector multiplication computations.

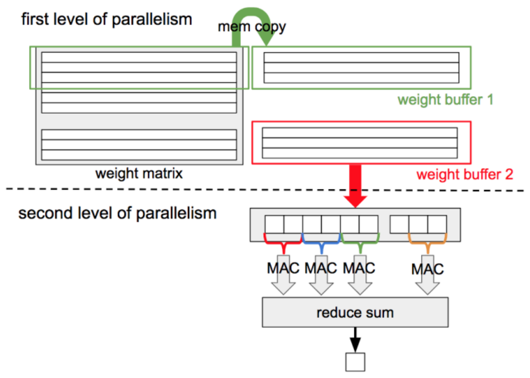


Fig. 4: Schematic representation of the matrix multiplication engine and the corresponding parallelization factors.

We define two levels of parallelism for our engine which control the parallel computations with parameters *num_parallel_in* and *num_parallel_out*, denoting the level of parallelism in the input and output, respectively. For the first level of parallelism, multiple rows of the weight matrix are processed simultaneously by dividing it into chunks, each having *num_parallel_out* rows. In each round, a chunk of the weights matrix is copied to one of the weight buffers while the other weight buffer is fed into the *dot product* modules together with a copy of the input vector. The iterations end when all rows of the weight matrix have been processed. For the second level of parallelism, each *dot-product* function partitions its input vectors into *num_parallel_in* chunks and concurrently executes MAC operations over the partitioned subsets. The accumulated results of the subsets are then added together within the *reduce_sum* function to compute the final output. The *reduce_sum* module performs a tree-base reduction algorithm as outlined in Figure 5. The reduction function takes an array of size $2M$ as its input (array a) and oscillates between 2 different modes. In mode 0, the function reduces a by using temp as a temporary array. In mode 1, temp is reduced using a. The result is returned based on the final mode.

The aforementioned parameters *num_parallel_in* and *num_parallel_out* are individually defined for each of the layers to enable fine-tuning according to the per-layer requirements. Due to the limited number of available resources on the FPGA platform, it is not possible to define high parallelization factors for all layers. As such, we give priority to layers with higher computational complexity, i.e., higher number of input and output channels, by instantiating their corresponding matrix multiplication engines with larger parallelization parameters.

E. Network Description Module

In this module, we implement the overall architecture of our network as a stack of dilated conventional layers and perform queue update operations followed by a fully connected layer.

Authorized licensed use limited to: Univ of Calif San Diego. Downloaded on August 30,2024 at 07:55:35 UTC from IEEE Xplore. Restrictions apply.

	Resource Utilization					Performance			Correctness	
	BRAM (Mb)	**URAM (Mb)**	**FF (K)**	**LUT (K)**	**DSP48E**	**Latency**	**Clock-Cycle Time (ns)**	**Throughput (Hz)**	**MSE**	**LSD**
Design / Available Resources	94.5	360	3456	1728	12288					
FloatingPointBaseline	93 (98%)	144 (40%)	35 (1%)	86 (5%)	288 (2%)	12110989	8.83	9.4	0	0
FloatingPointCQ	93 (98%)	144 (40%)	35 (1%)	83 (5%)	330 (3%)	6170104	8.83	18.4	0	0
FloatingPointPipeline	93 (99%)	144 (40%)	231 (7%)	231 (13%)	475 (4%)	612952	8.88	183.7	0	0
FixedPointUnrolling	79 (84%)	144 (40%)	22 (1%)	146 (8%)	660 (5%)	293914	8.75	388.8	0.006	0.104
FixedPointMME (Best)	90 (96%)	144 (40%)	425 (12%)	1669 (97%)	540 (4%)	78275	8.66	1475.2	0.006	0.104

TABLE II: Resource Utilization, Performance and Measured Error in generation for each design implementation. The error metrics namely, Mean Squared Error (MSE) and Log-Spectral Distance (LSD) is measured by comparing the generated audio from FPGA implementations against audio generated from corresponding GPU implementation. The percentages reported indicate percentage of resources utilized by the design.

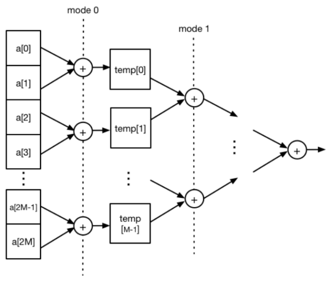


Fig. 5: Realization of the tree-based vector reduction algorithm.

This module instantiates the corresponding function for each network layer and manages the layer inter-connections. Since each layer is independently instantiated, we can use custom dilation, channels and parallelization parameters for each layer. After the last fully connected layer, to make audio generation deterministic we use arg-max sampling. This allows us to bypass the final softmax layer since we can directly apply the arg-max function on the output of our final fully connected layer.

V. Results and Experiments

In this section, we evaluate the effect of our optimizations, namely cyclic queues, pipelining, loop unrolling and customized matrix multiplication engine, by conducting extensive design space exploration. Our design experiments are synthesized for the Xilinx XCVU13P board using Xilinx Vivado HLS 2017.4. In particular, we discuss the experimental techniques applied to reduce resource utilization and latency of our baseline implementation. We further provide a comprehensive comparison of our best designs with CPU and GPU implemented baselines in terms of throughput and power efficiency.

A. *Evaluation Metrics*

To evaluate the accuracy of our implementation we compare the output generated from our FPGA implementation with the golden output generated by the TensorFlow GPU implementation for the same initial seed. We use the following metrics to compare any two audio signals x_1, x_2 of the same length:

- **Mean Squared Error (MSE)**: The mean squared error (MSE) between any two given signals x_1, x_2 is the mean squared error between their representations in time domain as a sequence of floating point numbers. That is, $MSE = mean((x_1 - x_2)^2)$. The MSE losses reported are from the comparison of the entire waveform i.e. the total mean squared error from all 32000 samples.
- **Log-Spectral Distance (LSD)**: The log-spectral distance [23] is a commonly utilized metric, obtained as the root mean square error between the normalized log-spectra of given signals. Given two signals x_1, x_2, we calculate log-spectral distance between them as follows:

$$\begin{aligned} ps_1 &= (abs(stft(x_1)))^2 \\ ps_2 &= (abs(stft(x_2)))^2 \\ ls_1 &= normalize(log(ps_1)) \\ ls_2 &= normalize(log(ps_2)) \\ LSD &= RMSE(ls_1, ls_2) \end{aligned} \tag{3}$$

Here ps_1, ps_2 are the power spectra and ls_1, ls_2 are the normalized log spectra of signals x_1, x_2 respectively. The normalization is performed across all frequencies in the log spectrograms.
- **Qualitative Evaluation**: Along with the quantitative results, we also provide log-spectrogram visualizations of the audio signal generated using our FPGA implementation and the golden-output audio signal generated from the TensorFlow implementation in Figure 6 (c).

B. *Design Space Exploration*

We implement the following designs to study the effect of various optimization techniques in isolation and in combination with other techniques. The resource utilization, performance (throughput) and error in the generated audio, for each of the following designs have been reported in Table II. Throughput measures the number of audio samples generated per second by our implementation of an autoregressive model. Note that one second of audio contains 16000 samples if audio is sampled at 16KHz.

1) Baseline Floating Point Implementation (FloatingPoint-Baseline): The baseline design of our network is comprised of modules to implement the basic functionality of each layer, queue, initialization of weights from stored data files and forward propagation. We use a array-shifting implementation of queue which results in a fairly high latency as shown in Table II because of the very long queues (length = 8192 and

Authorized licensed use limited to: Univ of Calif San Diego. Downloaded on August 30,2024 at 07:55:35 UTC from IEEE Xplore. Restrictions apply.

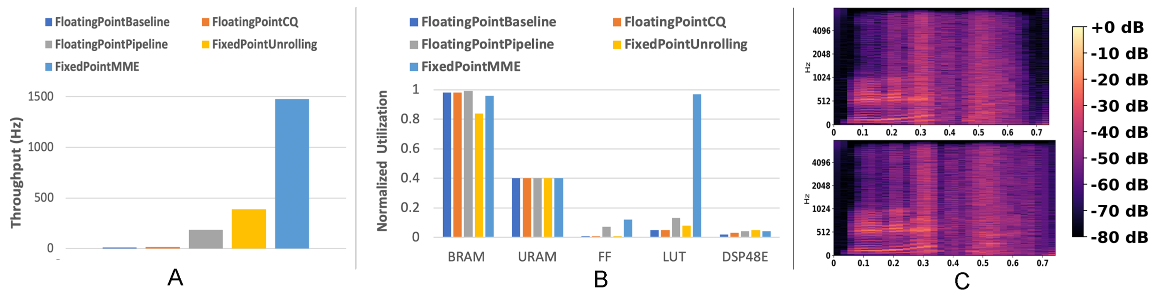


Fig. 6: **A:** Throughput (Number of Samples generated per second) of different designs. **B:** Normalized Resource Utilization of different designs. **C:** Log-Spectrograms of the 2-second audio generated from the TensorFlow implementation (top) and FPGA FixedPointMME design implementation (bottom).

4096) in the **13th** and **14th** layers of our design. For matrix vector multiplication we use simple for loops without any optimization.

2) Floating Point + Cyclic Queue (FloatingPointCQ) : In this design, we replace our shifting based queue implementation with a cyclic queue implementation that uses dynamic indexing to produce the same effect as push and pop operations. This helps reduce latency substantially since shifting operations in the longer queues was the bottleneck in our baseline design. The resource utilization however, stays almost the same as our baseline design.

3) Floating Point + Cyclic Queue + Pipelining (FloatingPointPipeline) : In this design, we modify the above design and add pipelining pragma in the dot product computation and queue update operations. Pipelining the above design helped increasing the throughput substantially at the cost of higher resource utilization.

4) Fixed Point + Cyclic Queue + Unrolling (FixedPointUnrolling) : Including both Cyclic Queue and Pipelining optimization, we switch to fixed point operations from floating point operations. Since the order of magnitude of our kernels, inputs, activations and outputs is nearly the same, we keep a common data-type across all of them. After some experimentation, we found that Loop Unrolling outperforms pipelining in terms of both resource utilization and throughput for fixed point data-types. We use *loop unrolling factor = 8* for the inner loop of our dot product and also the queue update operations. We observe a trade-off between precision and resource utilization for different fixed point bit width and chose `ap_fixed<27,8>` (8 bits for the integer and 19 bits for the fractional part)since it gives reasonable MSE under the constraints of resources.

5) Fixed Point + Matrix Multiplication Engine (FixedPointMME - Best) : For our best design, we use fixed-point implementation in a parallelized approach to convert layer computations into multiple MAC operations (refer to Section IV-D for details). For the first dilated convolution layer we set *num_parallel_out* and *num_parallel_in* as 1 since the number of input channels is just 1. For all other layers, including the fully connected layer we set *num_parallel_out* as 8 and *num_parallel_in* as 4 to get the best throughput under the constraint of available resources.

Implementation	**Time (in seconds) for 1-Second Audio Generation**	**Power (W)**
CPU (Numpy)	732	
GPU - NVIDIA Titan Xp (TensorFlow)	120	70
GPU - NVIDIA Tesla V100 (TensorFlow)	85	66
FPGA- FloatingPointPipeline	87	10.2
FPGA- FixedPointUnrolling	41	7.6
FPGA- FixedPointMME (Best)	11	23

TABLE III: Power Consumption and Wall-Clock time required when generating 1-second audio for different implementations.

C. *Performance and Power Analysis*

Table III illustrates the performance and power consumption for our implemented designs and a highly optimized CPU and GPU implementation. We benchmark the optimized Tensorflow implementation of Fast-Wavenet on two GPUs: NVIDIA TITAN Xp and Nvidia Tesla V100. The CPU implementation is the NumPy inference program written by us and optimized fully. We measure the power consumption for the GPU benchmarks using the NVIDIA power measurement tool (*nvidia-smi*) running on *Linux* operating system which is invoked during program execution. For our FPGA implementations, we synthesize our designs using Xilinx Vivado v2017.4. We then integrate the synthesized modules accompanied by the corresponding peripherals into a system-level schematic using Vivado IP Integrator. The frequency is set to 150 MHz and power consumption is estimated using the synthesis tool.

As shown, our best FPGA implementation achieves **11**$\times$ speed-up in audio generation while being **3**$\times$ more power efficient as compared to NVIDIA Titan Xp GPU. As compared to a NumPy based CPU implementation, our best design is **66**$\times$ faster.

VI. Prior Works on Accelerating DNNs for FPGAs

Prior works have made significant efforts in compressing Deep Neural Networks (DNNs) to support fast energy-efficient applications. However, recent research on DNNs is still increasing the depth of models and introducing new architectures, resulting in higher number of parameters per network and higher computational complexity. Other than

Authorized licensed use limited to: Univ of Calif San Diego. Downloaded on August 30,2024 at 07:55:35 UTC from IEEE Xplore. Restrictions apply.

CPUs and GPUs, FPGAs are becoming a platform candidate to achieve energy efficient neural network computation [12], [13], [22], [24]–[27]. Equipped with the necessary hardware for basic DNN operations, FPGAs are able to achieve high parallelism and utilize the properties of neural network computation to remove unnecessary logic. Algorithm explorations also show that neural networks can be simplified to become more hardware friendly without sacrificing the accuracy of the model. Therefore, it has become possible to achieve increased speedup and higher energy efficiency on FPGAs compared to CPU and GPU platforms [28], [29] while maintaining state-of-the-art accuracy. Prior efforts have been made in FPGA acceleration of speech recognition, classification and language modelling using Recurrent Neural Networks [14], [16], [27]; however the challenges in *generation* of sequences with *long-term dependencies*, particularly in audio domain have not been addressed.

VII. Conclusion

We present the first accelerator platform for deep autoregressive convolutional neural networks. While prior works have proposed algorithms for making the inference of such networks faster on GPUs and CPUs, they do not exploit the potential parallelism offered by FPGAs. We develop a systematic approach to accelerate the inference of WaveNet based neural networks by optimizing their fundamental computational blocks and utilizing only on-chip memory. We demonstrate the effectiveness of using FPGAs for fast audio generation by achieving a significant speed-up over prior efforts on CPU and GPU based implementations.

References

[1] A. Van Den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. W. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio." in *SSW*, 2016, p. 125.

[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. Gibiansky, S. Arik, G. Diamos, J. Miller, K. Peng, W. Ping, J. Raiman, and Y. Zhou, "Deep voice 2: Multi-speaker neural text-to-speech," in *Advances in neural information processing systems*, 2017, pp. 2962–2970.

[4] K. Qian, Y. Zhang, S. Chang, X. Yang, D. A. F. Florêncio, and M. Hasegawa-Johnson, "Speech enhancement using bayesian wavenet," in *INTERSPEECH*, 2017.

[5] N. Kalchbrenner, L. Espeholt, K. Simonyan, A. van den Oord, A. Graves, and K. Kavukcuoglu, "Neural machine translation in linear time," *CoRR*, vol. abs/1610.10099, 2016.

[6] Z. Yang, Z. Hu, R. Salakhutdinov, and T. Berg-Kirkpatrick, "Improved variational autoencoders for text modeling using dilated convolutions," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 3881–3890.

[7] A. van den Oord, T. Walters, and T. Strohman, "Wavenet launches in the google assistant," 2017. [Online]. Available: https://deepmind.com/blog/wavenet-launches-google-assistant/

[8] R. J. Williams and D. Zipser, "Backpropagation," Y. Chauvin and D. E. Rumelhart, Eds. L. Erlbaum Associates Inc., 1995, ch. Gradient-based Learning Algorithms for Recurrent Networks and Their Computational Complexity, pp. 433–486.

[9] S. Bai, J. Z. Kolter, and V. Koltun, "An empirical evaluation of generic convolutional and recurrent networks for sequence modeling," *CoRR*, vol. abs/1803.01271, 2018.

[10] H. Tachibana, K. Uenoyama, and S. Aihara, "Efficiently trainable text-to-speech system based on deep convolutional networks with guided attention," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2018, pp. 4784–4788.

[11] 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.

[12] M. Samragh, M. Ghasemzadeh, and F. Koushanfar, "Customizing neural networks for efficient fpga implementation," in *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*. IEEE, 2017, pp. 85–92.

[13] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2016, p. 17.

[14] M. Lee, K. Hwang, J. Park, S. Choi, S. Shin, and W. Sung, "Fpga-based low-power speech recognition with recurrent neural networks," in *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*. IEEE, 2016, pp. 230–235.

[15] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. B. J. Dally, "Ese: Efficient speech recognition engine with sparse lstm on fpga," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. ACM, 2017, pp. 75–84.

[16] S. Li, C. Wu, H. Li, B. Li, Y. Wang, and Q. Qiu, "Fpga acceleration of recurrent neural network based language model," in *Proceedings of the 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '15. IEEE Computer Society, 2015, pp. 111–118.

[17] A. Tamamori, T. Hayashi, K. Kobayashi, K. Takeda, and T. Toda, "Speaker-dependent wavenet vocoder," in *INTERSPEECH*, 2017.

[18] L.-J. Liu, Z.-H. Ling, Y. Jiang, M. Zhou, and L.-R. Dai, "Wavenet vocoder with limited training data for voice conversion," in *Proc. Interspeech*, 2018, pp. 1983–1987.

[19] G. Saon, G. Kurata, T. Sercu, K. Audhkhasi, S. Thomas, D. Dimitriadis, X. Cui, B. Ramabhadran, M. Picheny, L.-L. Lim, B. Roomi, and P. Hall, "English conversational telephone speech recognition by humans and machines," in *Proc. Interspeech 2017*, 2017, pp. 132–136.

[20] F. Yu and V. Koltun, "Multi-scale context aggregation by dilated convolutions," *arXiv preprint arXiv:1511.07122*, 2015.

[21] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs," *IEEE transactions on pattern analysis and machine intelligence*, vol. 40, no. 4, pp. 834–848, 2018.

[22] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 161–170.

[23] L. Rabiner and B.-H. Juang, *Fundamentals of Speech Recognition*. Prentice-Hall, Inc., 1993.

[24] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," *Microsoft Research Whitepaper*, vol. 2, no. 11, 2015.

[25] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 16–25.

[26] C. Shea, A. Page, and T. Mohsenin, "Scalenet: A scalable low power accelerator for real-time embedded deep neural networks," in *Proceedings of the 2018 on Great Lakes Symposium on VLSI*. ACM, 2018, pp. 129–134.

[27] Y. Guan, Z. Yuan, G. Sun, and J. Cong, "Fpga-based accelerator for long short-term memory recurrent neural networks," 01 2017, pp. 629–634.

[28] J. Y. Y. W. Kaiyuan G., Shulin Z and H. Y., "A survery of fpga based neural network accelerator," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 9, no. 4, 2017.

[29] X. Zhang, J. Zou, X. Ming, K. He, and J. Sun, "Efficient and accurate approximations of nonlinear convolutional networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1984–1992.

Authorized licensed use limited to: Univ of Calif San Diego. Downloaded on August 30,2024 at 07:55:35 UTC from IEEE Xplore. Restrictions apply.