Fast Inference for Neural Vocoders

Welcome, everyone!

I’m excited to be here today and get the opportunity to tell you a little bit about a few incredibly interesting systems challenges my team has encountered recently.

I’d like to preface this by saying that this is the work of many people — the speech synthesis team at SVAIL, the systems team, and the applied machine learning team. Mohammad Shoeybi, one of my colleagues, contributed enormously to the CPU kernels, and Shubho Sengupta from Facebook AI Research spent a lot of time on the GPU persistent kernels.

So, let’s get started!

Agenda

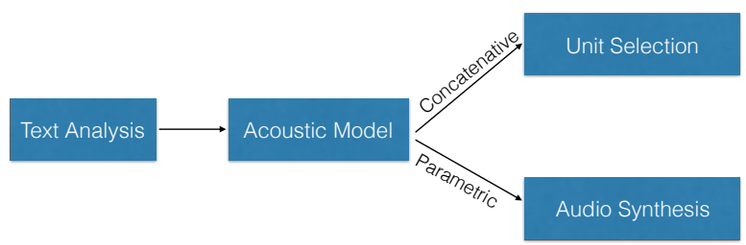
1. **Overview of neural speech synthesis**
2. Theoretical Neural Vocoder Peak Speed
3. Efficient WaveNet Inference (CPU)
4. Efficient WaveNet Inference (GPU)
5. Future Work
6. Questions

At SVAIL, we currently focus on speech — speech recognition and speech synthesis. This talk is primarily about speech synthesis, and doing it quickly and efficiently for inference.

Before jumping into implementation details, I'd like to take a brief detour into neural speech synthesis as a whole.

Speech Synthesis Pipeline

- **Concatenative:** Combine short audio clips into larger clip (“unit selection”) with “target cost” and “join cost”.
- **Parametric:** Synthesize audio directly with a vocoder.



Current speech synthesis systems come in roughly two flavors: concatenative and parametric systems.

These two aren’t really distinct things and share many parts of the pipeline, but in general, concatenative systems combine short audio clips from a large database, whereas parametric systems directly synthesize the audio with a deterministic process, such as a vocoder.

Before the actual audio synthesis, there is usually some sort of processing pipeline, which starts with text analysis. Text analysis can include normalization (turning numbers into words, for instance) and then conversion from text into phonemes.

After the text analysis, there’s traditionally a variety of acoustic models. All these models ultimately predict some sort of statistics that can then be used to synthesize the audio.

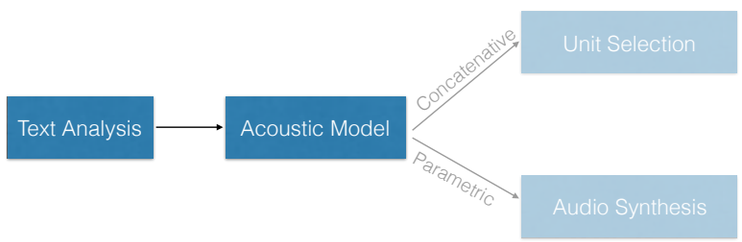
For example, in Deep Voice, a system published recently by my group at SVAIL, this component starts by doing duration prediction (assigning a duration in milliseconds to each phoneme) and then doing F0 prediction (predicting the dominant frequency and voicedness of each phoneme throughout the duration of the phoneme). Other systems could output things such as spectrograms or line spectral pairs or band aperiodic parameters or other statistics.

With a parametric system, those outputs are then used to synthesize the audio directly using a vocoder or a similar system.

With a concatenative system, a search procedure “unit selection” is used to find the best units that fit the desired audio properties.

Speech Synthesis Pipeline

- Recurrent neural networks for classification and F0 or spectrogram prediction
- Sequence-to-sequence networks for grapheme-to-phoneme conversion



First, let's focus on the shared stages.

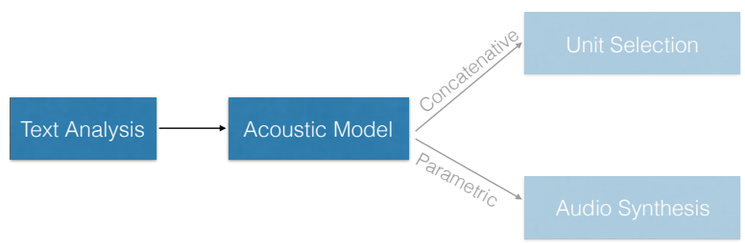
When it comes to deep learning approaches, recurrent neural networks are the bread and butter here.

Sequence-to-sequence models can be used for grapheme-to-phoneme conversion.

Recurrent neural networks can be used for classification. For example, in Chinese, we need to predict the tone for each part of the sentence, which can be done with a recurrent classifier; in English, we need to predict whether or not each phoneme is voiced, and what duration to ascribe to it.

Speech Synthesis Pipeline

- Small networks: 2-3 layer GRU, <512 wide
- Easy to deploy, both online and offline (mobile)



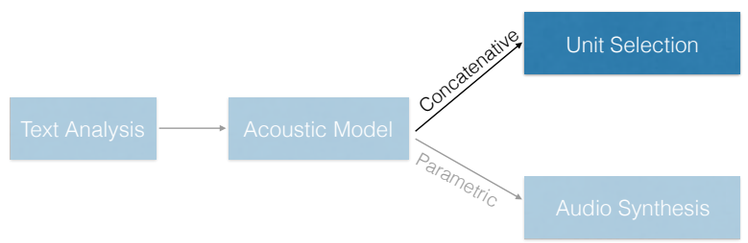
These networks tend to be fairly small: they are two or three layers of recurrent nets, or GRUs or LSTMs, with a fairly small width, less than 512, or even less than 256.

As a result, although these networks require a fair amount of engineering, they’re quite easy to deploy. You can deploy them on CPUs or GPUs, and you can even deploy them easily on mobile devices such as phones.

If you run the experiment, a modern iPhone can run networks that are something like 10-15 times larger than the ones I’m describing without too much difficulty.

Speech Synthesis Pipeline

- Unit selection: Larger database, better quality
- Database can grow to many gigabytes, 100+ hours of audio; impossible to deploy on mobile with high quality



Unit selection or concatenative pipelines will then take the output of the acoustic model and feed it to the unit selection search. The search usually consists of a “target cost” and a “join cost”; the target cost measures how well different units (sound clips) match the target acoustic model output, and the join cost measures how good two clips sound when placed next to each other.

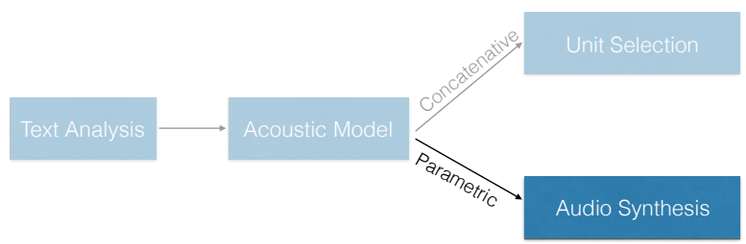
Because you need a variety of join points and phonemes, larger databases of clips lead to much better quality; the best quality databases can be one or two hundred hours of data, which ends up being gigabytes of data.

You cannot deploy such high quality systems to mobile devices, just because no one will download a ten gigabyte data file for their TTS on their phone, and you can’t store that much in the RAM of most small devices anyways. But the cost for these tends to be pretty cheap in terms of compute, so deploying them on servers isn’t too difficult.

I’m not going to focus any further on concatenative systems; just know that the best concatenative systems currently lead to higher quality results than parametric systems, but they’re effectively impossible to use offline.

Speech Synthesis Pipeline

- **Focus of this talk:** waveform synthesis!
- Traditionally done with fast vocoders, but high-quality neural methods are have been developed recently



Instead, the focus of the talk will be on the audio synthesis component of this pipeline for parametric systems; specifically, on the waveform synthesis.

This is traditionally done with fast vocoders, but recently high quality “neural vocoders” have been developed

A vocoder is a piece of software that can take some low-dimensional representation of a speech signal and synthesize it into a waveform

Neural Vocoders

- Traditionally, waveform synthesis is done with vocoders; can be *very* complex, hand-engineered

$$s(n) = A_1(n)\cos(\phi_1(n)) + \sum_{k=2}^{K} A_k(n)\left[\gamma_0 - \gamma_1\alpha_k(n)\cos(\phi_1(n))\right]\cos(\phi_k(n)), \quad (1)$$

$$g_k(n) = \gamma_0 + \gamma_1\alpha_k(n)\cos(\phi_1(n)). \quad (2)$$

$$\hat{\phi}_k = \psi_k + U\left(-h(\alpha_k)\frac{\pi}{4}, +h(\alpha_k)\frac{\pi}{4}\right), \quad (3)$$

… and so on …

Vocaine Vocoder (Agiomyrgiannakis, 2015)

So, why do we care about neural vocoders?

Well, traditionally, waveform synthesis is done with pretty complex hand-engineered vocoders

Here is the intro to the vocaine vocoder, a pretty recent paper

It's the summation of a large number of sinusoids, but each sinusoid is then modulated by a phase and amplitude, and both the phase and the amplitude are modulated by a separate model

These sinusoids shift in frames, and the join points between frames end up being a pain point, so you have to solve a system of equations for each frame point to ensure continuity of phase and amplitude

You end up with a *ton* of complexity based on a fairly complex set of assumptions about speech signals, and even after you do that the audio cannot match a concatenative system

Neural Vocoders

- Waveform synthesis must model:
  - Periodicity
  - Aperiodicity
  - Voiced / unvoiced
  - Fundamental frequency
  - F0 and aperiodicity estimation algorithms
  - Noise distribution

Here's a few of the other features that you have to care about – periodicity, aperiodicity (because large parts of speech are inherently aperiodic, such as s's and z's), voicedness, how you estimate the aperiodicity and F0, the noise distribution, and more.

Neural Vocoders

- Insight: If possible, replace complex system, many specialized features, with deep neural network
- Not that insightful, but has been harder than it sounds!

Well, we’re deep learning researchers. What do we do when we see a problem with a ton of hand-engineered features we don’t understand?

Replace the system with a neural network, and instead do architecture engineering!

Agenda

1. Overview of neural speech synthesis
2. **Theoretical Neural Vocoder Peak Speed**
3. Efficient WaveNet Inference (CPU)
4. Efficient WaveNet Inference (GPU)
5. Future Work
6. Questions

Next, I’d like to present to you two different neural vocoders, and why inference speed becomes an issue with these systems

Neural Vocoders

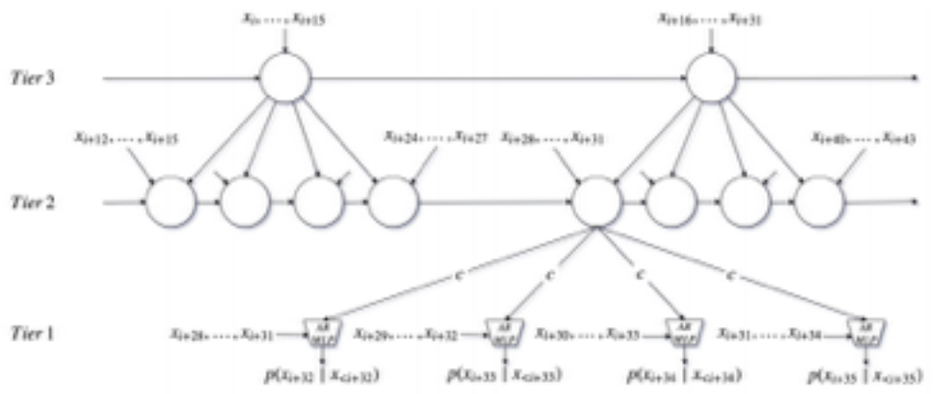
- Two available neural vocoders:
  - SampleRNN (used in Char2Wav)
  - WaveNet (used in Deep Voice)
- Predict next sample given previous samples
- Output Waveform: 16 - 48 kHz (samples per second)

two currently available neural vocoders are SampleRNN and WaveNet, and both of them model the probability of the next sample given the history of the audio

Audio is quantized into samples — a 16 kHz sample will have 16 thousand values for every second of audio, and we can represent these values by buckets

So the prediction problem can be viewed as a classification problem where the classes are 0 through 255, and we just have to make 16000 predictions for every second

SampleRNN



Mehri et al, 2016.

(Best model: 3X 1024-dimensional GRU with upsampling 2, 2, 8)

Sample RNN is a multi-layer RNN, with different layers running at different frequencies

Here, in the diagram Tier 3 RNN runs every 16 audio samples, Tier 2 runs for every 4 samples, and Tier 1 runs for every sample

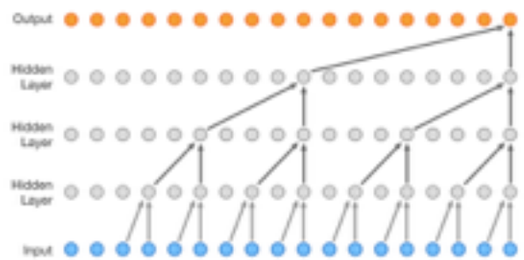
The outputs of tier 1 are then fed back into tier 2, and then also into tier 3

The best published SampleRNN model is three layers with 1000-dimensional GRUs, which is fairly hefty if you want to run it at 16000 time steps per second

WaveNet

van den Oord et al, 2016.

- Dilated convolutions (width two)
- Discrete output distribution with sampling
- Autoregressive sample-level generation
- Depth (40+ layers) with residual connections



Another vocoder is called WaveNet, and instead of being an RNN, it is a deep stack of convolutions

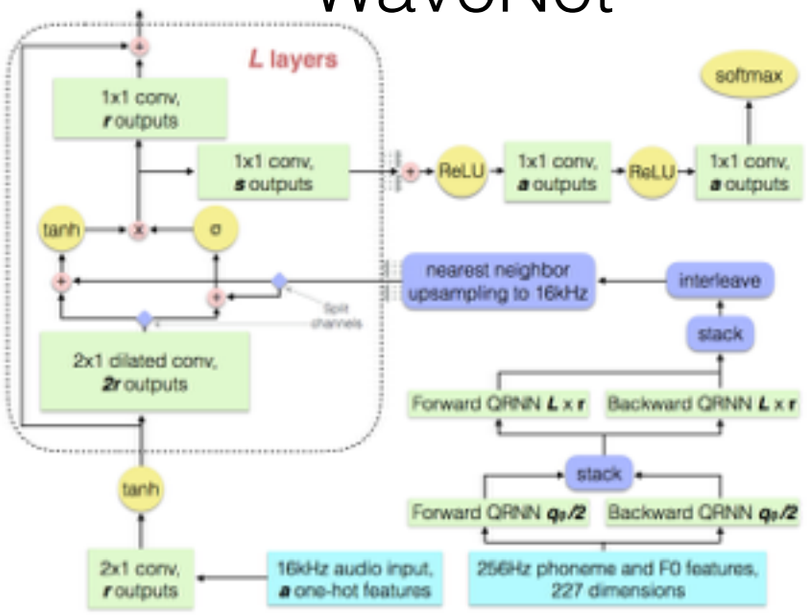
The convolutions are all width two and dilated, so that you end up convolving with the current sample and some sample far away in the “past” of your signal

The layers are all connected with residual connections

The key thing to remember is: You have 40 layers of width two convolutions

These two models produce comparable results

WaveNet



Since we'll be focusing on WaveNet later in this talk, I'd like to give you the complete schematic for it.

The key portion is in the dotted rounded rectangle, which is repeated 40 times — we do a width 2 dilated convolution, then apply a nonlinearity that is a sigmoid times a tanh, and then do two different width 1 convolutions, then apply the next layer

A width 1 convolution is just the same thing as an affine transform

Neural Vocoder Challenges

- Neural vocoder inference must be *very fast*
  - 16,000 - 48,000 timesteps/second
  - Speech recognition: 50 - 100 time steps/second
- Neural networks require a *lot* of compute
- Thus: requires good hardware and good software
- (Why bother? Is it possible?)

So, why do these post a challenge for inference?

Well, these must be *incredibly* fast — they're running 16 to 48 thousand times per second

Speech recognition networks have to run 50 to 100 times per second, a delta of 100X to 500X — so these networks need to be five HUNDRED times faster than current speech recognition networks

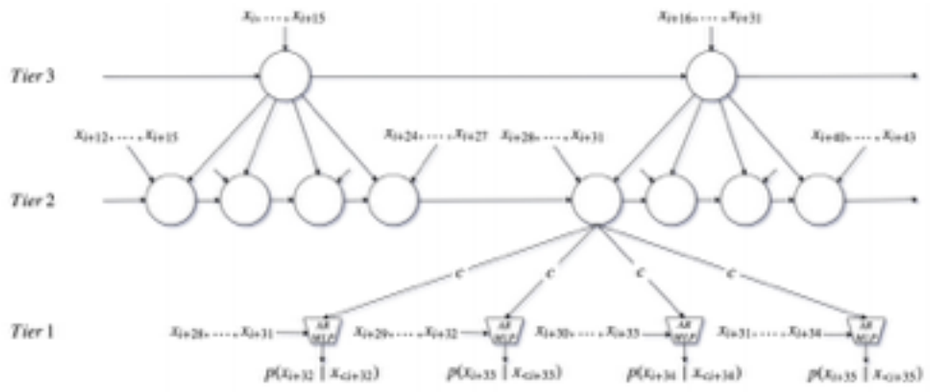
In order to do this, we need both high quality hardware and software — we really are pushing the boundaries of what's possible with the current grade of hardware

I'll note here that some people at this point will proceed to ask, why bother with this? Are we sure that there isn't a way to do this in a cheaper way, with different models?

I think we'll see in the near future where exactly neural vocoders will go — these are a fairly recent technology

However, it's important to know that if we need to, we *can* actually deploy and run these seemingly insanely computational expensive vocoders — and we may end up finding something better in the research community, but the ideas we develop here will likely carry over there as well

SampleRNN



Mehri et al, 2016.

(Best model: 3X 1024-dimensional GRU with upsampling 2, 2, 8)

So, I’d like to start by estimating: how hard is it to run these things?

SampleRNN Peak

- GRU cost (batch size of one):

  Layers X Timesteps X (6 * Width ^ 2) X 2 FLOP

- 3-tier SampleRNN, for one second: ~0.6 TFLOP/s
- Titan X Maxwell Theoretical Peak: 6 TFLOP/s
- Getting 6 TFLOP/s with batch size one is *very* hard
- CPU Core Theoretical Peak: 0.077 TFLOP/s

Well, the cost of running a GRU network is the layers times time steps times six times the width squared times two floating point operations

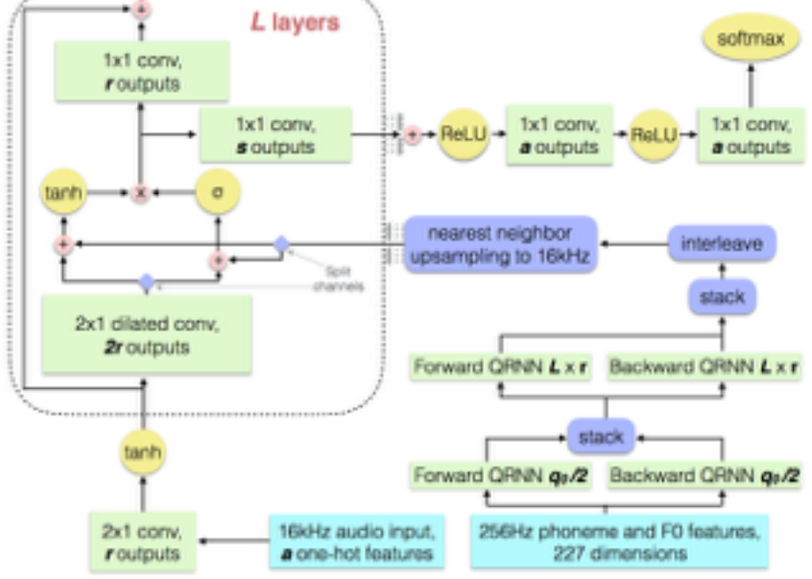
The six comes from the fact that the GRU matrix is twice as wide and three times as tall as the hidden state, due to the forget gates and the gating

So, for a 3-tier SampleRNN, when including the feedforward layers in the output layer, you get approximately 0.6 teraflops / second

The peak throughput of a Titan X Maxwell is about 6 teraflops — and getting anywhere near that is *very* hard, and doubly so with low batch size

In comparison, the theoretical peak of a single CPU core is 0.077 teraflops, which makes this even more challenging

WaveNet



How about WaveNet?

We’re going to focus on the autoregressive portion, in the dotted lines and up through the softmax

WaveNet Peak

- (Approximate) Cost Model:

  Single Layer = 2 X **r** X 2**r** + r X **s**
  Residual Stack = 40 X Single Layer
  Timestep Cost = Residual Stack + **s** X **a** + 2 **a** x **a**
  Total Cost = Timestep Cost X Timesteps X 2 FLOP

- Total Cost for Deep Voice WaveNet: 0.055 TFLOP/s

- *Much* cheaper – could fit on single CPU core?

If you do the same computation for WaveNet, you'll find that it requires about 0.055 teraflops/second

The full computation is in the Deep Voice paper

This is *much* cheaper than SampleRNN — and could plausibly even fit on a single CPU core?

Summary

- SampleRNN synthesis – requires **0.6** TFLOP/s
- WaveNet synthesis – requires **0.055** TFLOP/s
- CPU core – approximately **0.077** TFLOP/s
- Titan X Maxwell – approximately **6** TFLOP/s
- Big simplification: we are ignoring memory bandwidth!
- Focus on WaveNet (SampleRNN is harder)

So, to recap:

SampleRNN synthesis requires 0.6 teraflops, and WaveNet synthesis requires 0.055 teraflops

A CPU core is about 0.077 teraflops and a Titan X Maxwell is about 6 teraflops

But note: this is a huge simplification! We're completely ignoring memory bandwidth. In order to do this analysis fully, we'd need to repeat the process for memory

For now, we'll focus on WaveNet, since SampleRNN is much more computationally intense and achieves similar results

Agenda

1. **Overview of neural speech synthesis**
2. **Theoretical Neural Vocoder Peak Speed**
3. **Efficient WaveNet Inference (CPU)**
4. Efficient WaveNet Inference (GPU)
5. Future Work
6. Questions

Our best results were obtained on CPUs, so we'll focus on the CPU inference kernels for WaveNet

We'll go into the reasons *why* CPUs ended up with better results in future sections

Fast CPU WaveNet Inference

- General approach:

    Start simple, iteratively optimize bottlenecks

- Optimize, *measure*, repeat!
- Keep in mind: FLOPS and memory bandwidth

Building these CPU kernels took a lot of iteration, and I'd like to take you through that process in the next few slides

The goal is not necessarily to have you understand each of the optimizations — these again, are documented in more detail in the paper — but rather to gain an intuitive sense for what needs to happen to hit a similar level of performance for a different application

Fast CPU WaveNet Inference

Step 1: Simplest possible implementation

- TensorFlow tf.while_loop
- Redo convolutions for every layer
- Advantage: very flexible! (useful for development)
- **Timing:** 600X – 1000X *slower* than realtime

First of all, we started with the simplest possible implementation

The simplest implementation is just a TensorFlow loop which uses the same code as our primary model, and ends up redoing the convolution over the entire history in order to generate a single sample

This is as crazy as it sounds — it's doing 1000X the necessary work

However, it's *really* flexible — you can generate from this model as soon as you make a change. This is really valuable for debugging

Sadly, it ends up being about 1000X *slower* than realtime

Fast CPU WaveNet Inference

Step 2: Don’t redo computation

- TensorFlow tf.while_loop
- Inference as series of gemvs (matrix-vector mult)
- Dominated by TensorFlow overhead
- **Timing:** 100X – 200X *slower* than realtime

So, we take the next obvious step. Don’t redo computation.

In this case, this means that you cache all the intermediate activations, and your inference becomes a series of matrix-vector multiplies

Each layer has two matrix-vector multiplies, since it’s a width-two convolution

This is faster than our previous implementation, but much less flexible — you now have a separate codebase for training and inference

And, it’s still incredibly slow — 200X slower than realtime, as its dominated by TF overhead

Fast CPU WaveNet Inference

Step 3: Get rid of TensorFlow overhead

- Re-implement in C (carefully!)
- Use OpenBLAS for GEMV
- Dominated by GEMV performance
- **Timing:** 3X – 5X *slower* than realtime

So the next step is, of course: ditch TensorFlow.

We re-implemented the entire inference process in C, avoiding memory allocation, avoiding stack usage and function call overhead (inlining), etc.

We used OpenBLAS for multiplies

This was *way* faster — only about 5X slower than realtime, at worst — but now it's dominated by time spend in OpenBLAS matrix-vector multiply

Fast CPU WaveNet Inference

Step 4: Improve GEMV performance

- Replace OpenBLAS with Intel MKL
- Still dominated by GEMV performance
- **Timing:** 2X – 3X *slower* than realtime

We embarked on a quest to improve performance of the matrix-vector multiplies

The first step was to switch to implementations from Intel, using Intel MKL

This helped significantly, but still not enough. We’re still dominated by the compute.

Fast CPU WaveNet Inference

Step 5: Improve GEMV performance

- Replace Intel MKL with AVX2 assembly kernels
- Written using PeachPy
- Dominated by nonlinearities (`std::tanh` or `vmsExp`)
- **Timing:** 1.5X – 2.5X *slower* than realtime

So, we took the next logical step: rewrite the Intel MKL matrix-vector multiplies with custom kernels written in assembly for the specific matrix and vector sizes used in our models and for the processors we were targetting.

At this point I want to emphasize — this is actually quite doable with tools like PeachPy, which is one of my favorite tools for this sort of low-level engineering

(Aside) PeachPy

Amazing library written
by Marat Dukhan

```
# Also request a virtual 128-bit SIMD register...
xmm_x = XMMRegister()

# ...and fill it with data
MOVAPS(xmm_x, [reg_x])

# It is fine to mix virtual and physical (xmm0-xmm15) registers in the same code
MOVAPS(xmm2, [reg_y])

# Execute dot product instruction, put result into xmm_x
DPPS(xmm_x, xmm2, 0xF1)

# This is a cross-platform way to return results. PeachPy will take care of ABI specifics.
RETURN(xmm_x)
```

(from README on Github: https://github.com/Maratyszcza/PeachPy)

PeachPy is a tool developed by Marat Dukhan

It lets you write Python code which *generates* assembly kernels, and it handles a lot of the annoying assembly programming mental overhead for you, so you can focus on which instructions you want to use

It’s really an excellent project — I can’t recommend it enough

Fast CPU WaveNet Inference

Step 6: Replace nonlinearities with approximations

- Use rational approximations for exp, tanh
- Dominated by cache misses (finally!)
- **Timing:** 0.9X – 2X *slower* than realtime

Once we had switched to custom assembly kernels, surprisingly enough our performance was dominated by the cost of nonlinearities like tanh and exp

Many times, when optimizing models, researchers have a tendency to jump to nonlinearities as the performance issue, but they very rarely were — imagine my surprise when I found that the unary operators in our kernels were by far the slowest part

We replaced the approximations with rational approximations which take somewhere around 10-30 cycles instead of 1000 or so cycles

At this point, our smaller models were actually running *faster* than realtime, which was an exciting moment — but the faster our models, the bigger they could be, the better they sounded, so we didn’t stop there

Fast CPU WaveNet Inference

Step 7: Pin thread to a single core

- Reduces cache thrashing
- Still dominated by cache misses (finally!)
- **Timing:** 0.6X – 1.9X *slower* than realtime

At this point, measuring the cache miss rate, we realized that we could improve performance significantly by making sure that each thread stayed on one core, to keep the cache warm and avoid cache thrashing

We got a free 30% speedup out of this. It was nice.

Fast CPU WaveNet Inference

Step 8: Multiple threads

- Overlay computation on two types of threads
- **Timing:** 0.3X – 1.7X *slower* than realtime

Finally, we turned to something we'd been dreading — parallelizing this over multiple threads

We started by overlaying the computation onto two threads

Thread Work Overlaying

from Deep Voice (Arik et al., 2017)

- Begin computing output layer part-way through WaveNet convolution st
- Begin computing left half of convolution while remainder of output layer computed

The structure of the computation makes this fairly easy

The first matrix-vector multiply in the output layer can be computed in parallel, slightly lagging behind, the width two convolutions

And, while the output layer is being computed, the left half of the width two convolutions can be computed

So we can have two threads that neatly parallelize the work and only once or twice per timestep communicate with each other

Our smaller models at this point run about 3X faster than realtime, but our largest models are still about 1.5X slower than realtime, due to the larger matrix-vector multiplies

Fast CPU WaveNet Inference

Step 8: Multiple threads (for large matrices)

- Split matrix multiplies across multiple threads
- Thread barrier after each multiple (must be fast!)
- **Timing:** 0.3X – 1.1X *slower* than realtime

In order to deal with these larger matrix-vector multiplies, we can actually break up those multiplies across multiple threads as well

Each matrix-vector multiply is done in parallel on two or more threads, and there is a thread barrier afterwards such that all threads wait until all threads are done before continuing

There’s a lot more between-thread communication, but it does speed things up significantly

Fast CPU WaveNet Inference

Step 9: Switch to int16 arithmetic (fixed point)

- No perceptual difference, if avoiding overflow
- Requires new GEMV kernels (also PeachPy)
- **Timing:** 0.3X – 1X *slower* than realtime
  AKA 1X – 3X ***faster* than realtime**
- On two CPU cores; implementation reaches ~35% of *peak theoretical* throughput

Finally, we have our last trick: switch to fixed point arithmetic, using int16 to represent our weights and values instead of float32

If we carefully avoid overflow in int16, we notice no perceptual difference in the generation quality, but get a hefty speedup (with fewer cores)

And, finally: all our models are realtime. The largest are 1X realtime, while the smaller ones are 3X faster than realtime.

This runs on two cores, so our implementation gets about 35% of the maximum possible peak theoretical throughput on CPU, which is pretty good for batch size one!

Main Ideas

- Start simple, measure bottlenecks, optimize
- Biggest Gains:
  - Core implementation in C / C++ (careful!)
  - Tuned assembly GEMV kernels
  - Fixed-point arithmetic
  - Limited multithreading

So, in summary:

measure, optimize, repeat — that should be your mantra

What helped us was avoiding overhead and being careful in our memory allocation, tuning our floating point kernels, doing a limited amount of multithreading, and switching to fixed point integer arithmetic

These ideas and techniques are applicable to many problems, nothing specific to WaveNet!

Agenda

1. **Overview of neural speech synthesis**
2. **Theoretical Neural Vocoder Peak Speed**
3. **Efficient WaveNet Inference (CPU)**
4. **Efficient WaveNet Inference (GPU)**
5. Future Work
6. Questions

And, with that, let’s briefly talk about WaveNet inference on GPU

We have not yet managed to do inference as quickly on GPU, for a variety of reasons — none of them having to do with the capabilities of the hardware itself!

Fast GPU WaveNet Inference

- Cannot use multiple CUDA kernel calls
  - Call is ~10μs overhead, we can spare 60μs total
- **Result**: **must** write single kernel for inference

So, first of all, it's important to note that this problem is unlike most CUDA kernels

In most cases, you can call kernels for things like matrix-vector multiplies, and have the CPU enqueue work onto the GPU

In this case, however, the overhead from a CUDA kernel call is too high — we have about 60 microseconds per iteration *total*, and the call from a single kernel is on the order of 10 microseconds, so there's no way we can just launch hundreds of thousands of individual kernels and expect good results

We *have* to write a single kernel that does the entire inference

Fast GPU WaveNet Inference

- GPU global memory is slow
- Reading matrices is slow
- Very low utilization of GPU due to small matrix sizes
- **Result**: Store matrices in registers!

Our next problem is GPU memory speed

Unlike CPUs, we don’t have a nice cache hierarchy with large amounts of cache per core

GPU global memory is slow, shared memory is small and also slow.

However, GPUs can easily store the entire models in *registers* across the different cores or SMs on the GPU

Fast GPU WaveNet Inference

- Persistent kernels eliminate memory bandwidth need
- Model fits in 1 million parameters
- Titan X Maxwell has 6 MB of register space!
- Lots of #pragma unroll, only allow specific sizes
- **Problem:** nvcc causes register spilling
- **Timing**: 2.5X – 5X *slower* than realtime

So, the approach we took was to store the model in registers across different SMs of the GPU, which eliminates the need for memory bandwidth

The model fits in 1 million parameters, and a Titan X Maxwell has about 6 MB of registers, which is more than enough

You can use #pragma unroll to unroll loops to allow you to do this, even though you have to specialize your kernels to specific sizes

Sadly: if you write this in CUDA, you have to use nvcc, and it just isn't built for this — it continually spills registers into memory, which results in a huge slowdown

The result is that even our smaller models are about 3X slower than realtime – simply because the programming interface does not give us low-level enough control of the hardware

If there were a PeachPy for GPUs, we would likely be able to do *much, much* better — at least one order of magnitude, possibly more, likely significantly faster than on CPUs

Agenda

1. **Overview of neural speech synthesis**
2. **Theoretical Neural Vocoder Peak Speed**
3. **Efficient WaveNet Inference (CPU)**
4. **Efficient WaveNet Inference (GPU)**
5. **Future Work**
6. Questions

So, before we get to questions, I just wanted to mention a few things we'd still like to do — most of which are just the next obvious step in these optimization loops

Future Work (CPU)

1. No more than 3X faster than current (on Haswell)
2. Fixed-point nonlinearities
3. Find remaining cache misses
4. Optimize L2 to L1 bandwidth
5. **Skylake**: Use new AVX-512 instructions

On CPU, we're close to maxing out the capabilities of current hardware, even though I'm fairly sure we could do a little bit better

The biggest gains can come from Skylake processors, which are just now starting to roll out from Intel

Skylake introduces a new instruction set called AVX 512, which increases the theoretical throughput by a factor of 2 and adds a lot of useful instructions

Future Work (GPU)

1. Write persistent kernels that don't spill to memory
2. Implement same persistent kernels with int8

   Theoretically, much faster than CPU.

   Practically, *much,* ***much*** harder to implement.

   (Need better tools for writing persistent kernels!)

On GPU, we really just need to write persistent kernels that don't spill to memory

This is possible, it's just a *lot* of work

And, on the new Pascal architectures, we can use int8 for inference to dramatically speed up inference as well

All in all, in theory, we should be able to be ten times faster or more on GPU than on CPU, but in practice, the toolchain around highly optimized assembly kernels is just much better on CPUs right now than on GPUs

Agenda

1. Overview of neural speech synthesis
2. Theoretical Neural Vocoder Peak Speed
3. Efficient WaveNet Inference (CPU)
4. Efficient WaveNet Inference (GPU)
5. Future Work
6. **Questions**

Thank you!

I hope you've found this interesting, and perhaps you can even take some of the lessons learned and apply it to your own work.

I'm happy to take questions now, and of course feel free to come up to me after the talk with additional questions or reach out via email if you'd like to talk more in depth.