AGENDA

1) Brief introduction to speech processing
2) What we have done?
3) How can I use it?
INTRODUCTION TO ASR
Translating Speech into Text

Speech Recognition: the process of taking a raw audio signal and transcribing to text

Use of Automatic Speech Recognition has exploded in the last ten years:

Personal assistants, Medical transcription, Call center analytics, Video search, etc
SPEECH RECOGNITION

State of the Art

• Kaldi fuses known state-of-the-art techniques from speech recognition with deep learning

• Hybrid DL/ML approach continues to perform better than deep learning alone

• "Classical" ML Components:
  • Mel-Frequency Cepstral Coefficients (MFCC) features - represent audio as spectrum of spectrum
  • I-vectors - Uses factor analysis, Gaussian Mixture Models to learn speaker embedding - helps acoustic model adapt to variability in speakers
  • Predict phone states - HMM - Unlike "end-to-end" DL models, Kaldi Acoustic Models predict context-dependent phone substates as Hidden Markov Model (HMM) states

• Result is system that, to date, is more robust than DL-only approaches and typically requires less data to train
KALDI
Speech Processing Framework

Kaldi is a speech processing framework out of Johns Hopkins University

Uses a combination of DL and ML algorithms for speech processing

Started in 2009 with the intent to reduce the time and cost needed to build ASR systems

http://kaldi-asr.org/

Maintained by Dan Povey

Considered state-of-the-art
KALDI SPEECH PROCESSING PIPELINE

Raw Audio → Feature Extraction → Acoustic Model → Language Model → Output

Kaldi Components: MFCC & I-vectors, NNET3, Decoder, Lattice

NVIDIA is cool
FURTHER READING

“Speech Recognition with Kaldi Lectures.” Dan Povey, www.danielpovey.com/kaldi-lectures.html

WHAT HAVE WE DONE?
PREVIOUS WORK

Partnership between Johns Hopkins University and NVIDIA in October 2017

Goal: Accelerate Inference processing using GPUs

Used CPU for entire pipeline

NVIDIA Progress reports:

GTC On Demand: DC8189, S81034

INITIAL WORK

First Step: Move Acoustic Model to GPU

Was already implemented but not enabled, batch NNET3 added by Dan Povey

Enabled Tensor-Cores for NNET3 processing
Early on it was clear that we needed to target language model decoding.
LANGUAGE MODEL CHALLENGES

Dynamic Problem:
- Amount of parallelism changes significantly throughout decode
- Can have few or many candidates moving from frame to frame

Limited Parallelism:
- Even when there are lots of candidates the amount of parallelism is orders of magnitude smaller than required to saturate a large GPU

Solution:
1) Use graph processing techniques and a GPU-friendly data layout to maximize parallelism while load balancing across threads (See previous talks)
2) Process batches of decodes at a time in a single pipeline
3) Use multiple threads for multiple batched-pipelines
CHALLENGES

Kaldi APIs are single threaded, single instance, and synchronous

Makes batching and multi-threading challenging

Solution:

Create a CUDA-enabled Decoder with asynchronous APIs

Master threads submit work and later wait for that work

Batching/Multi-threading occur transparently to the user
EXAMPLE DECODER USAGE

for ( ... ) {
    ...
    //Enqueue decode for unique “key”
    CudaDecoder.OpenDecodeHandle(key, wave_data);
    ...
}

for ( ... ) {
    ...
    //Query results for “key”
    CudaDecoder.GetLattice(key, &lattice);
    ...
}

More Details: kaldi-src/cudadecoder/README
1. Master threads open decode handles and add waveforms to work pool.

2. Features are placed in the GPU work queue.

3. A batch of worked processed by the GPU pipeline thread.


**BatchedThreadedCudaDecoder**

- **CUDA control threads**
  - Acoustic Model (NNET3)
  - Language Model

**Threaded CPU Work Pool**

- Feature Extraction
- Compute Lattice
KALDI SPEECH PROCESSING PIPELINE
GPU Accelerated

Raw Audio → Feature Extraction → Acoustic Model → Language Model → Output

NVIDIA is cool
BENCHMARK DETAILS

LibriSpeech

Model:
LibriSpeech - TDNN: https://github.com/kaldi-asr/kaldi/tree/master/egs/librispeech

Data:
LibriSpeech - Clean/Other: http://www.openslr.org/12/

Hardware:
CPU: 2x Intel Xeon Platinum 8168
NVIDIA GPUs: V100, T4, or Xavier AGX

Benchmarks:
CPU: online2-wav-nnet3-latgen-faster.cc (modified for multi-threading)
   Online decoding disabled
GPU: batched-wav-nnet3-cuda.cc
   2 GPU control threads, batch=100
TESLA V100
World’s Most Advanced
Data Center GPU

5,120 CUDA cores
640 Tensor cores
7.8 FP64 TFLOPS
15.7 FP32 TFLOPS
125 Tensor TFLOPS
20MB SM RF
16MB Cache
32 GB HBM2 @ 900GB/s
300GB/s NVLink
TESLA T4
World’s most advanced scale-out GPU

- 2,560 CUDA Cores
- 320 Turing Tensor Cores
- 65 FP16 TFLOPS
- 130 INT8 TOPS
- 260 INT4 TOPS
- 16GB  |  320GB/s
- 70 W
JETSON AGX XAVIER
World’s first AI computer for Autonomous Machines

AI Server Performance in
30W • 15W • 10W

512 Volta CUDA Cores • 2x NVDLA
8 core CPU
32 DL TOPS • 750 Gbps SerDes
2x Intel Xeon Platinum 8168, 410W, ~$13000
AGX Xavier: Devkit, 30W, $1299
V100*: SXM, (300W+410), ~$(9000+13000)

## KALDI PERFORMANCE

1 GPU, LibriSpeech

**Determinized Lattice Output**
- beam=10
- lattice-beam=7
- Uses all available HW threads

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Perf (RTFx)</th>
<th>WER</th>
<th>Perf</th>
<th>Perf/$</th>
<th>Perf/watt</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LibriSpeech Model, Libri Clean Data</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2x Intel Xeon</td>
<td>381</td>
<td>5.5</td>
<td>1.0x</td>
<td>1.0x</td>
<td>1.0x</td>
</tr>
<tr>
<td>AGX Xavier</td>
<td>500</td>
<td>5.5</td>
<td>1.3x</td>
<td>13.1x</td>
<td>17.9x</td>
</tr>
<tr>
<td>Tesla T4</td>
<td>1635</td>
<td>5.5</td>
<td>4.3x</td>
<td>3.7x</td>
<td>3.7x</td>
</tr>
<tr>
<td>Tesla V100</td>
<td>3524</td>
<td>5.5</td>
<td>9.2x</td>
<td>5.5x</td>
<td>5.3x</td>
</tr>
<tr>
<td><strong>LibriSpeech Model, Libri Other Data</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2x Intel Xeon</td>
<td>377</td>
<td>14.0</td>
<td>1.0x</td>
<td>1.0x</td>
<td>1.0x</td>
</tr>
<tr>
<td>AGX Xavier</td>
<td>450</td>
<td>14.0</td>
<td>1.2x</td>
<td>11.9x</td>
<td>16.3x</td>
</tr>
<tr>
<td>Tesla T4</td>
<td>1439</td>
<td>14.0</td>
<td>3.8x</td>
<td>3.3x</td>
<td>3.3x</td>
</tr>
<tr>
<td>Tesla V100</td>
<td>2854</td>
<td>14.0</td>
<td>7.6x</td>
<td>4.5x</td>
<td>4.4x</td>
</tr>
</tbody>
</table>

*Price/Power, not including, system, memory, storage, etc, price is an estimate
INCREASING VALUE

Amortizing System Cost

Adding more GPUs to a single system increases value

- Less system cost overhead
- Less system power overhead

Dense systems are the new norm:

- **DGX1V**: 8 V100s in a single node
- **DGX-2**: 16 V100s in a single node
- **SuperMicro 4U SuperServer 6049GP-TRT**: 20 T4s in a single node
Kaldi Inferencing Speedup Relative to 2x Intel 8168

T4 Performance

- 1635 RTFx
- 3371 RTFx
- 6368 RTFx
- 7906 RTFx

V100 Performance

- 3524 RTFx
- 7082 RTFx
- 10011 RTFx
- 9399 RTFx

Legend:
- 1 GPU
- 2 GPUs
- 4 GPUs
- 8 GPUs
Kaldi Inferencing Performance Relative to 2x Intel 8168

**Performance Per Dollar**

**Performance Per Watt**

<table>
<thead>
<tr>
<th>GPU Config</th>
<th>T4 !/$</th>
<th>V100 !/$</th>
<th>T4 !/W</th>
<th>V100 !/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 GPU</td>
<td>4x</td>
<td>6x</td>
<td>4x</td>
<td>6x</td>
</tr>
<tr>
<td>2 GPUs</td>
<td>8x</td>
<td>8x</td>
<td>8x</td>
<td>8x</td>
</tr>
<tr>
<td>4 GPUs</td>
<td>12x</td>
<td>12x</td>
<td>12x</td>
<td>12x</td>
</tr>
<tr>
<td>8 GPUs</td>
<td>16x</td>
<td>16x</td>
<td>16x</td>
<td>16x</td>
</tr>
</tbody>
</table>
PERFORMANCE LIMITERS

Cannot feed the beast

  Feature Extraction and Determinization become bottlenecks
  CPU has a hard time keeping up with GPU performance

Small kernel launch overhead

  Kernels typically only run for a few microseconds
  Launch latency can become dominant
  Avoid this by using larger batch sizes (larger memory GPUs are crucial)
Feature Extraction on GPU is a natural next step: algorithms map well to GPUs

Allows us to increase density and therefore value
FUTURE WORK
Native Multi-GPU Support

Native multi-GPU will naturally load balance work pools

Master 1
...
Master N

GPU Work Queue

CUDA control threads

Acoustic Model (NNET3) → Language Model

Threaded CPU Work Pool

Feature Extraction

Compute Lattice

Master i
FUTURE WORK
Where We Want To Be

Master 1
...
Master N

GPU Work Queue

GPU Accelerated Feature Extraction

CUDA control threads

Feature Extraction → Acoustic Model (NNET3) → Language Model

Threaded CPU Work Pool

Compute Lattice

Multi-GPU Backend

Master i
HOW CAN I USE IT?
HOW TO GET STARTED

2 Methods

1) Download Kaldi, Pull in PR, Build yourself

https://github.com/kaldi-asr/kaldi/pull/3114

2) Run NVIDIA GPU Cloud Container

Get up and running in less than 10 minutes!
THE NGC CONTAINER REGISTRY
Simple Access to GPU-Accelerated Software

Discover over 40 GPU-Accelerated Containers
Spanning deep learning, machine learning, HPC applications, HPC visualization, and more

Innovate in Minutes, Not Weeks
Pre-configured, ready-to-run

Run Anywhere
The top cloud providers, NVIDIA DGX Systems, PCs and workstations with select NVIDIA GPUs, and NGC-Ready systems
Get an NGC account: https://ngc.nvidia.com/signup

#login in to NGC, pull container, and run it
%> docker login nvcr.io
%> docker pull nvcr.io/nvidia/kaldi:19.03-py3
%> docker run --rm -it nvcr.io/nvidia/kaldi:19.03-py3

#prepare models and data
%> cd /workspace/nvidia-examples/librispeech
%> ./prepare_data.sh

#run benchmarks
%> ./run_benchmark.sh
%> ./run_multigpu_benchmark.sh 4
BENCHMARK OUTPUT

NGC Container

BENCHMARK SUMMARY:

test_set: test_clean
   Overall: Aggregate Total Time: 55.1701 Total Audio: 194525 RealTimeX: 3525.91
   %WER 5.53 [ 2905 / 52576, 386 ins, 230 del, 2289 sub ]
   %SER 51.30 [ 1344 / 2620 ]
   Scored 2620 sentences, 0 not present in hyp.

test_set: test_other
   Overall: Aggregate Total Time: 64.7724 Total Audio: 192296 RealTimeX: 2968.79
   %WER 13.97 [ 7314 / 52343, 850 ins, 730 del, 5734 sub ]
   %SER 73.94 [ 2173 / 2939 ]
   Scored 2939 sentences, 0 not present in hyp.

Running test_clean on 4 GPUs with 24 threads per GPU
GPU: 0 RTF: 2469.55
GPU: 1 RTF: 2472.81
GPU: 2 RTF: 2519.33
GPU: 3 RTF: 2515.81
Total RTF: 9977.50 Average RTF: 2494.3750
NVIDIA GPUS ARE ON EVERY CLOUD

Over 30 Offerings Across USA and China

<table>
<thead>
<tr>
<th></th>
<th>K520</th>
<th>K80</th>
<th>P40</th>
<th>M60</th>
<th>P4</th>
<th>P100</th>
<th>T4</th>
<th>V100</th>
<th>NGC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alibaba Cloud</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>AWS</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Baidu Cloud</td>
<td></td>
<td></td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>Google Cloud</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>IBM Cloud</td>
<td>●</td>
<td></td>
<td>●</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Microsoft Azure</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Oracle Cloud</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Tencent Cloud</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
</tbody>
</table>
CONTAINER FUTURE WORK

Add more models

Add scripts to help users run quickly on their own models

NUMA pinning

Continue to update Kaldi source with latest updates
KALDI CHANGES
Source Layout

https://github.com/kaldi-asr/kaldi/pull/3114

Added two new directories to source tree

  cudadecoder/*:
    Implements framework/library classes for use in applications
  cudadecoder/README: Detailed documentation on how to use
  cudadecoderbin/*:
    Binary example using cuda-accelerated decoder
TUNING PERFORMANCE
Functional Parameters

determinize-lattice:

- determinize lattice in CPU pool or not
- If not determinized in CPU pool master thread will determinize if GetLattice is called

beam:

- width of beam during search
- Smaller beam = faster but possibly less accuracy

lattice_beam:

- width of lattice beam before determinization
- Smaller beam = smaller lattice, less I/O, less determinization time
TUNING PERFORMANCE

GPU Performance

cuda-control-threads:
number of concurrent CPU threads controlling a single GPU pipeline
Typically 2-4 is ideal (more = more GPU memory and less batch size)
cuda-worker-threads:
number of CPU threads in the CPU workpool, should use all CPU resources available
max-batch-size:
maximum batch size per pipeline (more = more GPU memory and less control threads)
Want as large as memory allows (<200 is currently possible)
batch-drain-size:
how far to drain a batch before refilling (batches NNET3)
typically 20% of max-batch-size works well
cuda-use-tensor-cores:
Turn on Tensor Cores (FP16)
TUNING PERFORMANCE
Memory Utilization

max-outstanding-queue-length:
   Length of GPU work queue, Consumes CPU memory only

ntokens-preallocated:
   Preallocated host memory to store output, CPU memory only
   Will grow dynamically if needed

max-tokens-per-frame:
   Maximum tokens in GPU memory per frame
   Cannot resize, will reduce accuracy if it fills up

max-active:
   maximum number of arcs retained in a given frame (keeping only the max-active best ones)
   Less = faster & less accurate
AUTHORS

Hugo Braun is a Senior AI Developer Technology Engineer at NVIDIA. With a background in mathematics and physics, he has been working on performance-oriented machine learning algorithms. His work at NVIDIA focuses on the design and implementation of high-performance GPU algorithms, specializing in deep learning and graph analytics. He holds a M.S. in Mathematics and Computer Science from Ecole Polytechnique, France.

Justin Luitjens is a Senior Developer Technology Engineer at NVIDIA. He has spent the last 16 years working on HPC applications with the last 8 focusing directly on CUDA acceleration at NVIDIA. He holds a Ph.D. in Scientific Computing from the University of Utah, a Bachelor of Science in Computer Science from Dakota State University and a Bachelor of Science in Mathematics for Information Systems from Dakota State University.

Ryan Leary is a Senior Applied Research Scientist specializing in speech recognition and natural language processing at NVIDIA. He has published research in peer-reviewed venues on machine learning techniques tailored for scalability and performance as well as natural language processing for health applications. He holds a M.S. in Electrical & Computer Engineering from Johns Hopkins University, and a Bachelor of Science in Computer Science from Rensselaer Polytechnic Institute.