Training for Speech Recognition on Co-processors

Sebastian Baunsgaard[*]
Graz University of Technology
baunsgaard@tugraz.at

Sebastian Benjamin Wrede
Know-Center
swrede@know-center.at

Pınar Tözün
IT University of Copenhagen
pito@itu.dk

ABSTRACT

Automatic Speech Recognition (ASR) has increased in popularity in recent years. The evolution of processor and storage technologies has enabled more advanced ASR mechanisms, fueling the development of virtual assistants such as Amazon Alexa, Apple Siri, Microsoft Cortana, and Google Home. The interest in such assistants, in turn, has amplified the novel developments in ASR research.

However, despite this popularity, there has not been a detailed training efficiency analysis of modern ASR systems. This mainly stems from: the proprietary nature of many modern applications that depend on ASR; the relatively expensive co-processor hardware that is used to accelerate ASR by big vendors to enable such applications; and the absence of well-established benchmarks. The goal of this paper is to address the latter two of these challenges.

The paper first describes an ASR model, based on a deep neural network inspired by recent work, and our experiences building it. Then we evaluate this model on three CPU-GPU co-processor platforms that represent different budget categories. Our results demonstrate that utilizing hardware acceleration yields good results even without high-end equipment. While the most expensive platform (10X price of the least expensive one) converges to the initial accuracy target 10-30% and 60-70% faster than the other two, the differences among the platforms almost disappear at slightly higher accuracy targets. In addition, our results further highlight both the difficulty of evaluating ASR systems due to the complex, long, and resource-intensive nature of the model training in this domain, and the importance of establishing benchmarks for ASR.

1. INTRODUCTION

Automatic Speech Recognition (ASR) has been an active research area for decades [51, 18, 40, 17]. Its popularity and complexity keep increasing as a result of the popularity of various virtual assistants [45, 7, 34, 22]. Earlier approaches to ASR were based on statistical models such as the Gaussian Mixture Model - Hidden Markov Model (GMM-HMM) [51, p. 19]. However, in recent years, the emergence of neural networks has also influenced ASR [26, 6, 11].

The computational requirements for the training and inference of such neural networks are immense [19, 43]. Most calculations in neural network models are independent matrix operations. Therefore, they are a natural fit for hardware acceleration on specialized multithreaded hardware such as GPUs [46, 39]. Any performance improvement through such hardware acceleration is significant for data scientists trying to enhance their machine learning models as they have to experiment with different hyperparameters before deciding on the final set of parameters.

Our goal in this paper is to have a more in-depth understanding of ASR by focusing on the ASR task of converting speech to text and utilization of CPU-GPU co-processors while training a neural network model for this task. Our contributions are as follows:

- We discuss our experience with building a well-established acoustic model that converts speech to text based on a deep neural network inspired by recent work [26, 6, 11]. We use the training of this model as a benchmarking tool while analyzing acoustic model training behavior on different types of hardware platforms.
- We evaluate three CPU-GPU co-processor platforms that represent three budget categories for training this model: a low-budget platform built by us via repurposing a crypto mining rig, and two more expensive platforms that represent different generations of commodity co-processor server hardware (a recent previous generation and a modern high-end one).
- We take Time-to-Accuracy as the primary metric in this evaluation to emphasize both training efficiency and model accuracy. The results demonstrate that utilizing hardware acceleration yields good results even without high-end co-processors. The most expensive platform (10X price of the least expensive one) converges to the initial accuracy target 10-30% faster than the platform that is 2X the price of the least expensive one and 60-70% faster than the least expensive one. However, the differences among the platforms almost disappear at slightly higher accuracy targets, which take roughly a couple of days to reach.

Our experience highlights the difficulty of establishing ASR models that both achieve good accuracy and are hardware-conscious while training. ASR differs from other established

[*]First two authors did this work while at IT University of Copenhagen. They equally contributed to the paper.

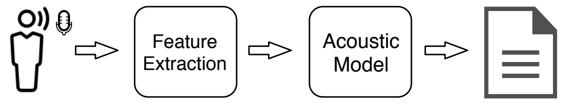


Figure 1: Process of converting speech to text.

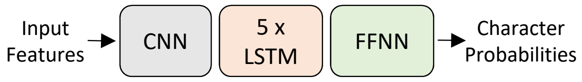


Figure 2: Overview of the layers of our AM.

machine learning benchmark domains due to its requirement of a more complex model training and larger space consumption of the dataset and the model. In addition, our results emphasize that price/performance ratio for neural network training on high-end co-processors can indeed be very poor in practice, especially for ASR models. Therefore, it is highly important to have well-established benchmarks in this domain and other similarly complex domains to characterize the performance of both high-end and low-budget platforms.

The rest of the paper is organized as follows. Section 2 introduces terminology related to ASR and gives an overview of our end-to-end system that converts speech to text. Section 3 surveys related work that inspired this system and recent work on benchmarking machine learning models. Section 4 presents the experimental setup, including model parameters and methodology, as well as the implementation details for efficient model training using the TensorFlow framework. Section 5 discusses the results of the experimental evaluation on three co-processors. Finally, Section 6 concludes the paper with a discussion of results.

2. SPEECH-TO-TEXT

The process of converting speech to text is composed of two broad components as Figure 1 illustrates [51, p. 4]: *Feature Extraction* (FE) and *Acoustic Model* (AM). Extracting features from audio in FE before training the acoustic model enables performance optimizations in AM. One can also add a *Language Model* (LM) trained separately on text data to improve the output of AM [26, 6, 11, 13, 25, 32]. Among these components, AM takes the longest time in an end-to-end system since it contains the training of the machine learning model. On the hardware platforms used in this paper (Section 4.2), FE takes a few hours, whereas training the AM takes several days. Therefore, the rest of this section provides details about the AM component in our system, even though we have the whole end-to-end system built.[1]

The ***Acoustic Model (AM)*** takes the features from FE as input and generates a probability distribution over the dictionary [51, p. 4]. In recent years, AM has been based on neural networks and several combinations of neural network layers have been explored [51, 26, 6, 13, 11]. The commonly used layers are Feed-Forward Neural Network (FFNN), Convolutional Neural Network (CNN), and Long Short-Term Memory (LSTM). The model used in this paper is inspired by Baidu Research's Deep Speech 2 system [6], which uses 1-3 CNNs as the first layers followed by 1-7 LSTM layers, and ending with a single FFNN. Figure 2 gives an overview of the model this paper uses, where the number of CNN, LSTM, and FFNN layers are 1, 5, and 1, respectively.

An ***FFNN*** can be seen as a directed acyclic graph, where the nodes are functions and the output of each function is a directed edge. The nodes are organized in layers, where each layer represents a matrix multiplication of the input with a weight matrix for that layer. This weight matrix is also called a *kernel*. FFNN used in this paper is based on TensorFlow's `tf.contrib.slim.fully_connected`.

A ***CNN*** slides a kernel over the input doing matrix multiplications for each step. A CNN depends on the kernel *width*, which is the number of input elements included in each computation step, and *stride*, which is the number of inputs the kernel is slided over between each step. A CNN can be used to do computations on values from different time steps. The convolution used in this paper is `tf.nn.conv1d` from TensorFlow, which conducts a one-dimensional convolution on a three-dimensional input. This means that a convolution is done on the time-dimension of an input consisting of features, time, and batch.

An ***LSTM*** applies a function to all elements of an input sequence of arbitrary length while transferring information from one time-step to the next. This property makes it ideal for processing time series data such as audio. The size of an LSTM refers to the number of hidden states transferred between the time-steps. Increasing the size of an LSTM increases the capacity of the internal state, which improves the model's ability to fit to the training dataset. The LSTM in this paper is based on `tf.nn.rnn_cell.LSTMCell`, combined with `tf.nn.dynamic_rnn` from TensorFlow.

The weights of the layers are initialized using Xavier Initialization [20], `tf.contrib.layers.xavier_initializer` of TensorFlow, which scales all weights to a uniform distribution within a range.

The output of the FFNN is processed differently during training and evaluation. During training, a *loss* value has to be calculated for each element of the batch to determine how to fit the model to the training data. During evaluation, the model has already been adapted to the training data and the output of the FFNN should instead be converted to specific characters representing the output sentence of the model.

The loss function used depends on the problem domain. *Connectionist Temporal Classification* (CTC) is common for ASR [23, 21, 6, 11]. CTC converts the output of the FFNN to probabilities over the alphabet and aligns it to the label sequence. Given the input sequence of probabilities generated by the CTC denoted Y and the label sequence l, the ***CTC loss*** is the sum of the probability of different alignments of l in the probability sequence Y.

When evaluating a model, output probabilities should be calculated efficiently to produce the characters of the output sentence. This is done with ***Beam Search***, which is a modified Breadth First Search (BFS) with a specified search width that limits the search through the tree. Beam Search is not guaranteed to find an optimal solution, but it is more efficient than BFS. The efficiency depends on the search width. Beam Search can be improved by combining it with a Prefix Tree and a Language Model [41, 42], which constrains its search to words in a dictionary while simultaneously allowing arbitrary non-word characters between words in the dictionary. We adopt this and build the dictionary using the words in our training dataset. This limits the variation of words in the dictionary compared to the alternative, which

[1]https://github.com/ITU-PITLab/s2asrCode

is to build the dictionary from a separate dataset with a greater variation of words.

An ***optimization algorithm*** defines how to adapt the model weights to reduce the loss of the model for the given training set. We use `tf.train.AdadeltaOptimizer` from TensorFlow for this. *AdaDelta* [52] is able to both increase and decrease the learning rate during training. This property also implicitly removes the need of specifying initial learning rates.

A model needs to not only fit to the training dataset, but also generalize to previously unseen data. The error calculated on the training set is *training error* and the error calculated on the test set is *test error* or *generalization error*. ***Regularization*** aims to reduce the gap between the training and test error [21, pp. 107-108]. One of the most important regularization strategies is dropout, where some of the units in the neural network are randomly dropped during training [44]. We use this strategy in all of the layers of our model since it reduces the test error, which improves accuracy.

Finally, ***Batch Normalization*** is applied between each layer of the model [28] [21, pp. 313-317]. This technique helps the layers train more efficiently and independently of each other.

3. RELATED WORK

We mention related work throughout the paper wherever it is necessary. This section, in particular, details the work that inspired the system described in Section 2, and recent efforts on establishing benchmarks and analyzing machine learning models.

Speech to Text Models Based on Neural Networks. Baidu Research's Silicon Valley AI Lab is a large research group that among other fields also does research on building end-to-end deep learning models for speech recognition [26, 6, 11]. Deep Speech 1 [26], is one of the preliminary works that uses neural networks for the whole speech to text learning pipeline in contrast to prior work that used neural networks in a limited part of the whole pipeline. Deep Speech 1 is also the foundation of Mozilla's open-source TensorFlow speech recognition implementation [35].

Deep Speech 2 [6] is a follow-up to Deep Speech 1. It experiments with several convolution layers and up to 7 layers of LSTM and bidirectional LSTM. Furthermore, it introduces GPU-optimized implementations of CTC loss and efficient gradient sharing among GPUs, which improves latency and throughput. However, the impact of these optimizations on the training time are not based on time-to-accuracy, but on the time to go through an epoch and the time spent on individual operations such as their implementation of the all-reduce algorithm. This approach to evaluating performance has been criticized in DawnBench [16], because it does not make a strong link across end-to-end training efficiency, hardware utilization, and statistical performance.

Another follow-up work from the same research group [11] investigates the impact of different transducers on the ASR systems. The models presented in Deep Speech 1 & 2 are based on CTC, and this work compares these approaches to an RNN-Transducer and an attention model. The conclusion is that both RNN-Transducers and attention models outperform the CTC-based model if the models are allowed to look at the entire input (meaning both forward and backward model parts). The paper also shows that the CTC forward-only models have better results than their forward-only RNN-transducer and attention models. We choose to work with a CTC-based model instead of RNN-Transducers and attention models to avoid the need for the entire input for one classification.

While the main inspiration for our acoustic model comes from the work of Baidu Research mentioned above, there have been other proposals for neural network models for speech recognition, which we also take influence from, such as LAS from Google [13, 25].

Benchmarking Machine Learning Models. There have been several benchmarking studies in the recent years focusing on machine learning and deep learning. Shi et al. [43] analyze popular deep learning frameworks, such as TensorFlow and Torch. They take training time per mini-batch as the main metric. As mentioned earlier, this is problematic as it does not account for the total training time of the model in different frameworks. DawnBench [16] treats training time per mini-batch as a *proxy metric* rather than a main one. Instead, DawnBench measures end-to-end performance of training and inference. It focuses on the time-to-accuracy and throughput of models as mini-batch size, optimization algorithm, number of GPUs, etc. varies. These metrics are also adopted by MLPerf [3], which is the most popular benchmarking framework for machine learning today. Liu et al. [31] also adopts the metrics from DawnBench and experiment with image recognition models with the default configurations of the different deep learning frameworks. However, none of these works have focused on ASR.

MLPerf [3] has very recently added ASR in its benchmark mix. The reference implementations are based on the work from Baidu Research surveyed above. QuTiBench [12] and DeepBench [1] describe benchmarks for different deep neural network training domains including ASR. DeepBench focuses on individual operations rather than end-to-end training, while QuTiBench focuses on end-to-end training. These are complementary to our work.

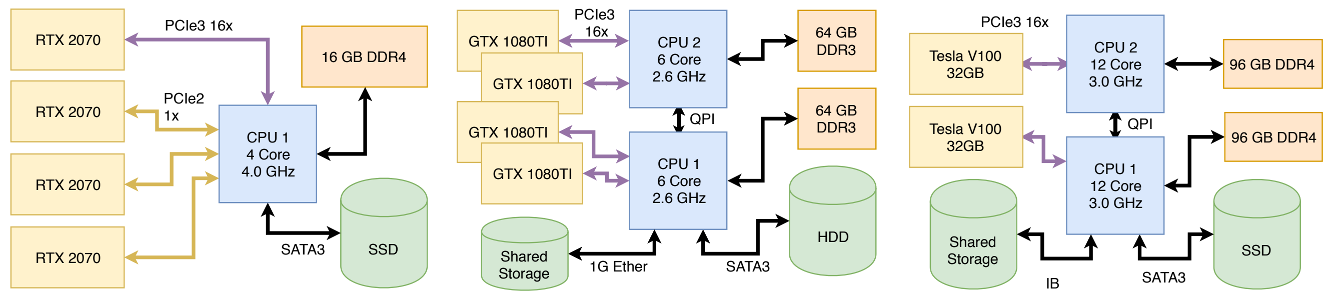
4. EXPERIMENTAL METHODOLOGY

Our goal in this paper is to analyze the behavior of training a state-of-the-art acoustic model for ASR on different types of co-processor hardware that reflect different price points. We focus on the acoustic model as its training is the major component in an end-to-end system for ASR (Section 2). We target CPU-GPU co-processors since training of neural network models is a natural fit for hardware acceleration on GPUs because of the independent matrix operations that are embarrassingly parallel, and such co-processors are heavily used for this purpose today [19, 46, 39].

To achieve our goal, we choose metrics to assess different aspects of the acoustic model and hardware platforms (Section 4.1), establish three CPU-GPU co-processors that represent different budget categories (Section 4.2), use a dataset with difficult characteristics (Section 4.3), and optimize the training setup to be fair to each hardware platform and avoid misleading conclusions (Section 4.4).

4.1 Metrics

Accuracy is measured in Word Error Rate (WER) and Character Error Rate (CER). While evaluating our trained acoustic model, the *input* is sound files with speech and the *output* is the text version of what is being said in those sound



(a) Sys1$ (b) Sys2$ (c) Sys10$

Figure 3: Overview of the co-processors used (left to right, from least expensive to most expensive).

files. The accuracy refers to the accuracy of this output.

$$WER = \frac{e(x,y)}{W} = \frac{I+D+S}{W} \tag{1}$$

As Equation (1) shows, WER is based on *edit* distance or, more specifically, *Levenshtein* distance $e(x, y)$. It is the sum of insertions I, deletions D, and substitutions S needed to convert the output sentence x to the target sentence y. An *insertion* is a word added to the output. A *deletion* is a word removed from the output. A *substitution* is a word replaced with another word. W refers to the number of words in the target sentence y. We use the Python package *Jiwer* [49] to calculate WER.

$$CER = \frac{e(x,y)}{\max(|x|,|y|)} \tag{2}$$

As Equation (2) shows, the definition of CER is similar to WER, where the denominator is the maximum number of characters in either the output sentence x or the target sentence y.

Time-to-Accuracy (TTA) is the primary metric in this evaluation since it covers both the accuracy and efficiency of training. (as also used by [15, 30]). TTA is the time it takes to train a model to a target median accuracy over the last t epochs. An *epoch* is an iteration of the entire *training* dataset during the training phase of the model. The accuracy of a single epoch is the accuracy of a single iteration over the entire *validation* dataset during the evaluation phase of the model after that particular epoch. The *median accuracy* is the middle value in the sorted list of accuracies from the last t epochs. The reason for considering the accuracy of several epochs in TTA is to make sure that the accuracy reached is persistent. To calculate $TTA(a, e)$, where a is the median accuracy target and e is the number of epochs, a and e needs to be determined. We set e to three and a to different values depending on the specific accuracy metric used (WER or CER). The $TTA(a, e)$ is measured in minutes from the creation of the model until the accuracy target is reached.

Throughput is defined as elements processed per second E_{ps}. If t is the time it takes for a batch to be processed in seconds, and b is the number of elements processed in that batch, then $E_{ps} = b/t$.

Utilization measures the active usage of CPUs and GPUs while training the model. The CPU utilization is measured based on the average percentage utilization over 0.03 seconds using the *top* command of linux. The GPU utilization is measured through *The NVIDIA System Management Interface (nvidia-smi)* [36].

4.2 Hardware

We use three hardware platforms for this study: Sys1$, Sys2$, Sys10$. They represent three budget categories in terms of co-processor hardware. Table 1 gives the cost breakdown for these co-processors, and Figure 3 illustrates their topology. We name them based on their relative total costs, i.e., Sys2$ and Sys10$ are roughly 2X and 10X the cost of Sys1$, respectively.

Sys1$ is a low-cost CPU-GPU co-processor designed to minimize the cost of a multi-GPU platform. It has 4 Nvidia RTX 2070 at 1.7GHz with 8GB memory each, an Intel i7 6700k desktop processor with 4 cores (8 logical cores with hyperthreading) at 4 GHz with 16GB memory, and a low-cost crypto mining rig motherboard ASRock H110 Pro BTC+ [8]. The storage device, which keeps the datasets, model checkpoints, and OS (Ubuntu 18.04 LTS), is a Micron M600 512GB SATA SSD connected through SATA3. The limitation of this platform is mainly twofold. (1) It has significantly smaller CPU and GPU memory compared to the other platforms. (2) Three of the GPUs are connected using PCIe 2.0 x1 to the CPU, while the other one is connected using PCIe 3.0 x16. This creates an asymmetry across the GPUs. This particular platform is built by us. Our goal was to build a low-cost platform inspired by cryptocurrency mining specifically repurposed for machine learning, and understand the relative effectiveness of such a hardware platform in comparison to more expensive hardware utilized by cloud providers.

**Sys2$** has 4 Nvidia GTX 1080 Ti with 11GB memory each and two 6-core (12 logical cores with hyperthreading) Intel Xeon E5-2630 v2 processors clocked at 2.6 GHz and 128GB RAM in total. Each processor has two GPUs attached via PCIe 3.0 x16. The OS is Centos 7 and installed on a locally-attached HDD, whereas the rest of the storage needs (the model checkpoints and datasets) are handled via shared storage connected through 1G ethernet. This increases the startup time by a few seconds when a previous model has to be loaded, but has insignificant impact on the active training phase. One can view Sys2$ as representative of the previous generation's high-end commodity co-processor. The hardware components are modern but slightly older, i.e., Intel's Ivy Bridge and Nvidia's Pascal microarchitectures.

Sys10$ has 2 Nvidia Tesla V100 GPUs with 32GB memory each and two 12-core (24 logical cores with hyperthreading) Intel Xeon Gold 6136 processors at 3.0 GHz and 192GB RAM in total. The storage is split between a locally-attached SSD and shared storage connected via InfiniBand. The OS is

Platform	Total	GPUs only	CPU Only
Sys1$	2,605.29$	1,980.00$	354.25$
Sys2$	5,699.95$	3,599.97$	97.98$
Sys10$	25,999.00$	17,956.00$	5,422.20$

Table 1: Costs of hardware platforms estimated based on prices from amazon.com in October 2019. The total cost is composed of CPU, GPU, RAM, PCIe riser card, chassis, and motherboard costs (not including storage).

Platform	**Total Batch**	**Batch per GPU**	**Readers**	**Buffer**
Sys1$	96	24	8	40
Sys2$	240	60	16	100
Sys10$	300	150	16	100

Table 2: Default parameters for batch size (number of elements), number of readers, and buffer size (number of batches) for each hardware platform.

Centos 7 and is on locally-attached SSD, and the rest of the storage needs (the model checkpoints and datasets) are kept on shared storage. Unlike Sys2$, the remote storage does not cause a slow down during initialization thanks to InfiniBand. Sys10$ represents the modern high-end co-processor, i.e., Intel's Skylake and Nvidia's Volta microarchitectures.

4.3 Data

We use the LibriSpeech dataset [38, 37], which contains continuous speech that uses a large vocabulary with different styles of speech and a large number of speakers. It is created from the LibriVox project, which is a collection of audiobooks read by different speakers [2]. LibriSpeech contains 1000 hours of speech and is split into multiple parts. The *training* data is split into three parts: 100 and 360 hours of *clean* speech, and 500 hours of *other* speech that is more noisy. We train on the combination of these three parts. Both the *validation* and *test* sets are split into two parts, a *clean* and *other* set, with each combination containing around 5 hours speech. For all experiments, we remove all elements above 16700 ms, since the time duration of the files greatly impacts the time it takes for an inference. This results in usage of 95% of the validation and test data, and 99% of the training data. This is reasonable since our aim is not to compete in accuracy, but to have enough data to verify generalization of the models.

4.4 Training

Framework and Parameters. The training of the acoustic model is done using TensorFlow version 1.14 following TensorFlow's guides for efficient training [47]. First, we extract features from the original sound files and store them in compressed TFRecord files before training [48]. Extracting the features into intermediate files reduces the number of redundant computations while training, thereby shortening the training time.

The movement of data is based on TensorFlow's abstraction called `tf.data.Dataset`. This abstraction creates parallel readers that read records from local disk or shared storage. The readers shuffle, batch and pad elements to supply a buffer. The elements are padded to a fixed length of 1670 samples, which gives a maximum training sample sound length of 16.7 seconds. The loader fills a buffer containing batches ready to train on.

The parameters for batch, buffer sizes, and the number of readers are set to the identified optimal parameters based on preliminary experiments, which Table 2 displays. The buffer and batch sizes are different for the three platforms because of the varying memory capacities. The main memory consumption during model training increases proportionally to the batch and buffer sizes. Therefore, this limits the values for batch and buffer sizes on Sys1$, which has smaller memory both on CPU and GPUs. The number of parallel readers was increased until there was no throughput improvements.

The training was conducted using Synchronous Stochastic Gradient Descent (S-SGD) similar to the related work [26, 6, 13, 14, 10] using the AdaDelta optimizer [52].

Each element in the buffer is an entire batch comprised of multiple elements. The batch is split into subsets depending on the number of GPUs in use to exploit data parallelism, e.g. a batch of ten elements is split into two subsets of five elements when using two GPUs. The subsets are each an input to a distinct GPU that contains a copy of the acoustic model described in Section 2.

The model parameters are located on the first GPU, from which they are also updated and distributed. Due to the main memory constraints on Sys1$, it is better to allocate the model's parameters directly on a GPU rather than in main memory of the CPU. This is fair to all platforms since all of them have similar PCIe 3 16x connection to the first GPU. This also enables better utilization of the potential direct connections among the GPUs on the platforms instead of always communication with the CPU.

The model parameters are with floating-point 32-bit precision. It is possible for both Sys1$ and Sys10$, to take advantage of 16-bit precision because of their newer GPU microarchitecture. Using 16-bit or mixed precision [33] increase arithmetic operation efficiency and reduce memory footprint on the GPUs, thereby reducing the training time. This is not done because Sys2$ does not support this optimization, and would default to 32-bit precision. Further work would be to explore the gains of 16-bit precision, and potentially going even further with quantized neural networks [27] that utilize 8-bit values which have recently shown great promise without significant loss in accuracies [12].

States of Training. The training is conducted while monitoring loss, CER, WER, and output sentences. Calculating the loss, CER, and WER requires evaluation of the model using the validation set, which means simultaneous training and evaluation.

Figure 4 illustrates the training process in the form of a state machine. The transition between the states includes the iteration interval that triggers the state transition. The initial state is *training*, where the model is fitted to the training data by continuously iterating through different batches. Once the training iterates through 50 batches, we switch to a new state that *generates sentences* from the training dataset based on the current model with a beam width of one. Every 250th batch iteration, a *checkpoint* of the model is saved. The checkpoint is used when running evaluation. It can also be used to restart the training in case the training has been interrupted. The model is *evaluated* every epoch by running

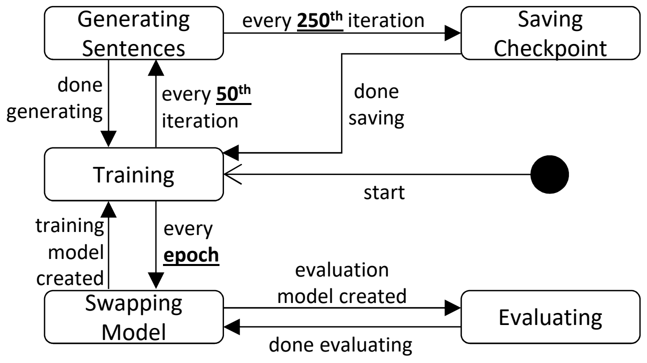


Figure 4: States during training.

the model on the validation set. To do this, the training model needs to be removed from the GPUs and the evaluation model needs to be loaded. *Swapping Model* represents this action, which has significant influence on throughput (Section 5.2).

There are alternatives to avoid model swapping during the training process. An alternative is to always allocate memory on the GPUs for a process that evaluates the model whenever a checkpoint is saved (as adopted by Mozilla Deep Speech implementation [35]). The drawback of this approach is that allocating memory for the evaluation limits the available memory for the model training. Another alternative is to avoid the interleaving of the training and evaluation on the same platform by training and evaluating on separate machines. This would require more complex and expensive hardware setup in addition to the more complicated management of the dataflow across machines.

For the evaluation phase, there is a tradeoff between the beam width selected for decoding the resulting sentence (Section 2) and accuracy. Based on our preliminary experiments, we set beam width to 256 since increasing it further leads to diminishing returns in accuracy while increasing the time for evaluation phase.

The Model. The model has the same parameters for the different platforms except the differences listed in Table 2. The model is smaller and simpler compared to related work (Section 3), but still contains the same building blocks and overall topology (Figure 2).

The input features are 93-dimensional per 10 ms time-step. The CNN has a kernel size of 11 x 93 x 600, takes all the input features in 11 consecutive time-steps, and produces a 600-dimensional output feature. The CNN is applied with a stride of 2, and is only applied to valid inputs. This means that the model cannot take inputs shorter than 11 time-steps.

The first LSTM layer takes the 600-dimensional input and all subsequent layers take the 800-dimensional output from the previous LSTM layer. The output of the fifth layer is passed to a feed-forward layer that transforms the output to character probability distributions of dimension 30, which represent {space, a, b, ... , z, ', separation character, blank character}. The separation character is used in cases where the word contains double letters, such as the word "cool", having the output "co_ol", where "_" is used to represent the separation character. Between each layer, batch normalization and a dropout of 5% are applied.

TTA / Sys	*CER* 10.0	8.0	7.0	*WER* 22.0	20.2	19.2
Sys1$	820	1641	2450	1908	2174	2725
Sys2$	672	1118	2429	1251	1980	2834
Sys10$	511	974	NA	1143	2840	NA

Table 3: TTA values expressed in minutes.

5. RESULTS

This section splits the results into three parts: (1) loss values, accuracy, and TTA, (2) hardware utilization and throughput, and (3) a comparison of the platforms with the same batch size.

5.1 Loss, Accuracy, & TTA

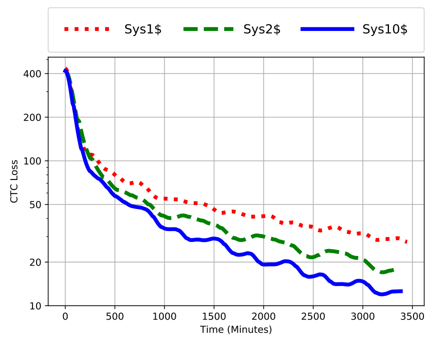
Figure 5a has the CTC **loss values** over time for the three hardware setups. The CTC loss axis is logarithmic to better represent the nuances over time. The loss values are smoothed by applying a 1-dimensional Gaussian filter from SciPy with a kernel standard deviation of 10 since the actual loss value oscillates throughout training. We can observe that the training loss is reduced over time, which means that the model is able to adapt to the training data. Sys10$ has the fastest and Sys1$ has the slowest convergence. The results correspond to the costs of the platforms with Sys10$ having the highest cost and Sys1$ the lowest one. All three executions stay above a loss value of 10, and they are stopped within approximately 3400 minutes, which is equivalent to 2 days and 8 hours.

The **accuracy** of the clean validation set over time is shown in Figure 5b and Figure 5c. The figures also have a logarithmic y-axis. The TTA levels chosen for CER are 10, 8, and 7, and for WER are 22, 20.2, and 19.2. All TTA levels are marked with horizontal lines to indicate when they are reached. Table 3 reports the specific TTA values, where NA represents the case, where the specified accuracy level is not reached.

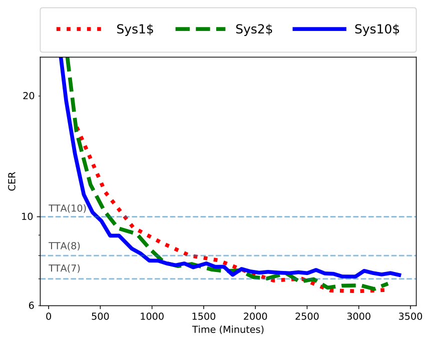
The initial 1000 minutes of the CER has a development similar to the training loss, where Sys10$ converges quicker than Sys2$, which in turn converges quicker than Sys1$. This situation is in effect at both TTA(10) and TTA(8). Afterward, the trends change. Sys2$ surpasses Sys10$, and it is later surpassed by Sys1$. Sys2$ and Sys1$ reach TTA(7) approximately at the same time step, while Sys10$ never reaches this CER, which is denoted by NA in Table 3. This demonstrates the effect of large batch sizes. The large batch size of Sys10$ improves the pace of the convergence initially, but ends up impeding the accuracy of the model. Similar results of reduced statistical efficiency with larger batch sizes were also identified in [30]. The reduced statistical efficiency stems from the higher generalization error of models trained with large batch sizes, which is a problem known for many years [21].

Figure 5c has similar trends to Figure 5b: Sys10$ reaching TTA(22) first, and the other two platforms reaching TTA(20.2) and TTA(19.2) first.

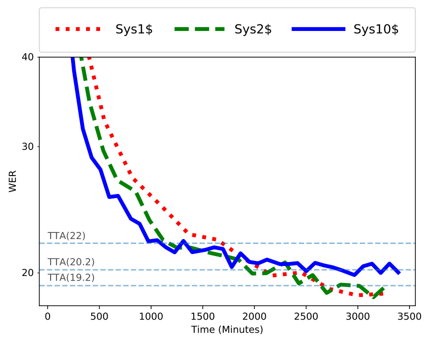
The accuracies described above are calculated on the validation dataset, the actual accuracy of the model has to be calculated on the separate test dataset. Table 4 reports the accuracies with the test dataset. The evaluation is done on both the *clean* and the *other* dataset and the table also in-



(a) Smoothed loss of training set over time with logarithmic y-axis.



(b) CER of validation set over time with logarithmic y-axis.



(c) WER of validation set over time with logarithmic y-axis.

Figure 5: Loss, Character Error Rate (CER), and Word Error Rate (WER) over time to measure Time-To-Accuracy (TTA).

	Clean		*Other*	
System	**Val**	**Test**	**Val**	**Test**
Sys1$	17.90	17.32	41.85	45.04
Sys2$	18.50	18.68	44.31	48.15
Sys10$	19.86	19.45	45.86	49.43
Related Work				
Deep Speech 2 [6]	NA	5.15	NA	12.73
Lüscher et al. [32]	1.90	2.30	4.50	5.00

Table 4: Final WER of trained model with beam width 512.

cludes the accuracy results of the validation set, which is referred to as *Val*. The accuracies reported in related work is also included in the table for comparison. We include the results from Deep Speech 2 [6] because the model is similar to our work, and the results from Lüscher et al. [32] because it represents the state-of-the-art accuracy for our dataset.

The model in our work is similar to Deep Speech 2, but it diverges in two ways, which results in shorter training time and lower accuracy: (1) CNN output size is 600 in our system and 1280 in Deep Speech 2, and (2) LSTM size is 800 in our system and 1510 in Deep Speech 2. By reducing the sizes of the model layers, accuracy is traded for faster computation. In addition, quoting their paper [6] ”Training a single model at these scales requires tens of exaFLOPs that would require 3-6 weeks to execute on a single GPU.” This was in late 2015 and their model was trained on 8-16 unspecified GPUs that are older GPU versions than the Nvidia Tesla V100 GPUs used by Sys10$ in this project.

Lüscher et al. [32] achieves better accuracy than both us and Deep Speech 2. The difference is their model is based on the attention-based architecture, which is also found in [13] and originates from [9]. The attention-based model clearly performs better, but it requires even longer training time, especially with the model parameters used for the specific solution in [32]. An analysis with a scaled down version of an attention-based model could be an alternative to the analysis done in this paper.

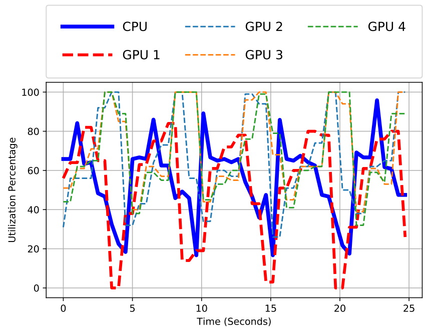
5.2 Utilization & Throughput

Figure 6 shows the **utilization** of the three platforms over a 25 second interval. The solid line represents the utilization of the CPU, while the dashed lines are the GPUs' utilization. The horizontal axis is time in seconds, and the vertical axis is the utilization percentage of the corresponding GPU and CPU. The CPU utilization is the average across all cores for the specific training process.

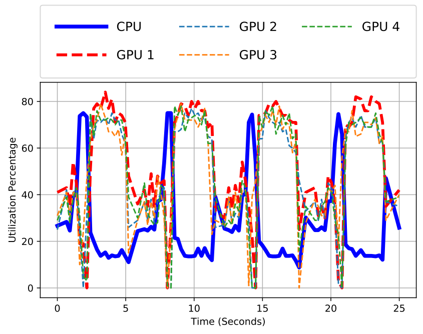
Figure 6c highlights three repetitive phases of utilization during a single batch processing for Sys10$. The first phase is represented by a spike on the CPU when the training data and model is transferred to the GPU. These spikes are most visible in Figure 6c at 4, 8, and 17 seconds, but are also present at 13 and 23 seconds. The second phase happens on the GPUs when they are using the acoustic model to infer results from the input data. This is seen in Figure 6c, where the GPUs are capped at 100% utilization just after the CPU spikes. The last phase is where the model is collected, gradients are summed, and the model is updated. This is seen as a small bump on the CPU utilization after the utilization of the GPUs are reduced.

Sys2$'s utilization is shown in Figure 6b. The spikes are wider indicating a longer startup for each iteration, as expected due to the slower CPU and main memory (Section 4.2). This gives a clearer view of the 0% utilization of the GPUs while transferring the data. This 0% utilization is also present in Figure 6c, but less visible thanks to the shorter duration. Figure 6b shows that the GPU utilization is capped at around 75%. The model and batch size use all of the available memory on the GPUs, but the cores are not fully utilized. This means that neither the model size nor the batch size can be further increased to fully utilize the cores because the allocated memory would exceed the memory limit.

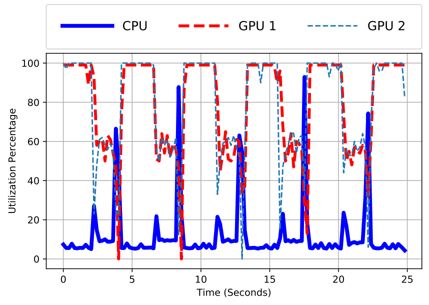
Figure 6a has a slightly different pattern than the utilization phases in Figure 6c and Figure 6b. GPU 1 is connected using PCIe 3 x16 and the rest of the GPUs are connected with PCIe 2 x1, which is a characteristic of the motherboard. GPU 1 is highly utilized in the beginning of each batch iteration, but then its utilization quickly drops. On the other hand, GPU 2, 3, and 4 have moderate utilization before being fully utilized around the time GPU 1 is done with its batch. This utilization difference is caused by the different connection types between the graphics cards. When GPU 1 finishes earlier, it waits for the three other GPUs to finish, where it goes down to almost 0% utilization. CPU utilization is also slightly different for Sys1$. The CPU has fewer logical cores compared to the CPUs in other platforms. This means that the CPU is generally highly utilized, which leads



(a) Sys1$



(b) Sys2$



(c) Sys10$

Figure 6: Hardware utilization over time on three co-processors. The figure focuses on a 25 second interval for ease of visualization. The utilization trends are stable over time.

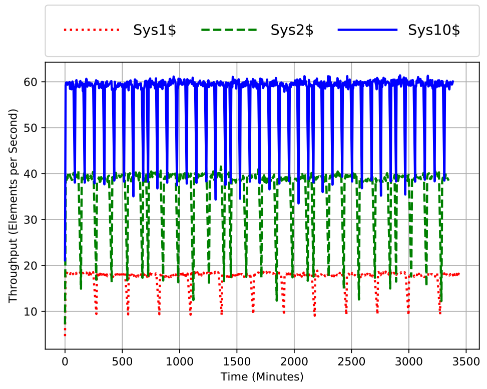


Figure 7: Processing throughput during training.

to less drastic spikes.

The **throughput** values in Figure 7 follows the same pattern as the utilization figures. Sys10$ has better utilization and throughput than Sys2$, which has better utilization and throughput than Sys1$. Throughput oscillates, but the minimum and maximum values are stable over time for the respective systems. Every time the training model is swapped by the evaluation model (Figure 4), throughput is reduced by 30-50% on all platforms.

Comparing the utilization and throughput trends to the accuracy in Figure 5, demonstrates that the highly utilized, high throughput systems may achieve lower accuracy eventually. The throughput and utilization analyses represent the hardware efficiency, whereas Figure 5b and Figure 5c represent the training efficiency. This means that Sys10$ has high hardware efficiency but low training efficiency, and Sys1$ has low hardware efficiency but high training efficiency. *This highlights that when training models, one cannot purely focus on one type of efficiency over the other as both are highly important for a sustainable machine learning evolution.*

5.3 Impact of Batch Size

To confirm that the accuracy Sys1$ achieves can be achieved on the other platforms, we apply the parameters used when training on Sys1$ to the training on Sys10$ and Sys2$. The total batch size is reduced to the same value. Since Sys10$ has two GPUs, the total batch is split into two,

TTA / Sys	*CER* 10.0	8.0	7.0	*WER* 22.0	20.0	19.0
Sys1$	771	1291	2322	1806	2322	2322
Sys2$	791	1571	2091	1571	2091	2352
Sys10$	479	879	1600	1197	1600	2078

Table 5: TTA values expressed in minutes with reduced batch sizes.

unlike Sys1$ and Sys2$ that have four. We also chose to reduce the beam width to 64 while evaluating thereby reducing overall training time. The change in beam width only reduces accuracy without effecting the convergence.

The reduction in batch size impedes throughput. Throughput is halved on both Sys10$ and Sys2$ compared to their throughput in Figure 7. The utilization of the GPUs on Sys10$ is also reduced, from 76% to 67% on average. Sys2$ and Sys1$ have similar throughput in this case, and therefore converge similarly, while Sys10$ converges faster than the other two systems due to its higher throughput.

Table 5 reports the TTA values achieved with the reduced batch size. Models trained on all platforms reach the same accuracy levels in this case, with a better TTA on Sys10$. However, the price/performance discrepancy across the platforms remains.

6. SUMMARY & CONCLUSION

This paper studied the behavior of the training of an acoustic model for ASR on different types of CPU-GPU co-processor hardware that fall into different price categories. The acoustic model was based on state-of-the-art proposals that use deep neural networks for this purpose. Our goal was to observe the impact of higher-end processors in comparison with lower-end ones in this problem domain. By focusing on time-to-accuracy as the main metric, we observed that utilizing hardware acceleration yields good results even without high-end equipment.

The latest generations of co-processor hardware, such as the one evaluated as Sys10$, offer a huge computation and acceleration power. Such co-processors are becoming more and more widely available to the end-users. The embarrassingly parallel nature of most neural network tasks make them great for exploiting these types of hardware. However, there is no free lunch, especially for the more complex application domains like ASR. One has to pay attention to design both a statistically accurate model and a hardware-conscious one in order to avoid wasting hardware resources and create a more sustainable machine learning ecosystem.

Going forward, we have various options to investigate. There are already well-established frameworks (TensorFlow, PyTorch, etc.) for crafting neural networks and hardware-conscious libraries for data scientists [4, 5]. We need to invest further in these frameworks and libraries, and understand their behavior on different types of processing units in more detail. It is unreasonable to expect every data scientist crafting models based on neural networks for a specific problem to be hardware gurus. However, it is also unreasonable to underutilize extremely powerful hardware. Furthermore, similarly to the research on co-locating different tasks on cloud, we should also investigate ways to co-locate different types of model training (such as [24, 29, 50]) to exploit idle-sitting hardware resources without breaking the performance of individual training processes.

7. REFERENCES

[1] DeepBench. `https://github.com/baidu-research/DeepBench`, 2018.

[2] LibriVox: Free Public Domain Audiobooks. `https://librivox.org`, 2019.

[3] MLPerf. `www.mlperf.org`, 2019.

[4] PYNQ: Python Productivity For ZYNQ. `http://www.pynq.io`, 2019.

[5] Rapids - Open GPU Data Science. `https://rapids.ai`, 2019.

[6] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen, J. Chen, J. Chen, Z. Chen, M. Chrzanowski, A. Coates, G. Diamos, K. Ding, N. Du, E. Elsen, J. Engel, W. Fang, L. Fan, C. Fougner, L. Gao, C. Gong, A. Hannun, T. Han, L. V. Johannes, B. Jiang, C. Ju, B. Jun, P. LeGresley, L. Lin, J. Liu, Y. Liu, W. Li, X. Li, D. Ma, S. Narang, A. Ng, S. Ozair, Y. Peng, R. Prenger, S. Qian, Z. Quan, J. Raiman, V. Rao, S. Satheesh, D. Seetapun, S. Sengupta, K. Srinet, A. Sriram, H. Tang, L. Tang, C. Wang, J. Wang, K. Wang, Y. Wang, Z. Wang, Z. Wang, S. Wu, L. Wei, B. Xiao, W. Xie, Y. Xie, D. Yogatama, B. Yuan, J. Zhan, and Z. Zhu. Deep Speech 2: End-to-end Speech Recognition in English and Mandarin. In *International Conference on Machine Learning (ICML)*, pages 173–182, 2016.

[7] Apple. Siri. `https://www.apple.com/siri/`, 2019.

[8] AsRock. AsRock H110 Pro BTC+ Motherboard. `https://www.asrock.com/mb/Intel/H110\%20Pro\%20BTC+/index.asp`, 2019.

[9] D. Bahdanau, K. Cho, and Y. Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. In *International Conference on Learning Representations (ICLR)*, 2015.

[10] D. Bahdanau, J. Chorowski, D. Serdyuk, P. Brakel, and Y. Bengio. End-to-end attention-based large vocabulary speech recognition. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4945–4949, 2016.

[11] E. Battenberg, J. Chen, R. Child, A. Coates, Y. G. Y. Li, H. Liu, S. Satheesh, A. Sriram, and Z. Zhu. Exploring neural transducers for end-to-end speech recognition. In *IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, pages 206–213, 2017.

[12] M. Blott, L. Halder, M. Leeser, and L. Doyle. QuTiBench: Benchmarking Neural Networks on Heterogeneous Hardware. *ACM Journal on Emerging Technologies in Computing Systems*, 15(4), 2019.

[13] W. Chan, N. Jaitly, Q. Le, and O. Vinyals. Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4960–4964, 2016.

[14] C. Chiu, T. N. Sainath, Y. Wu, R. Prabhavalkar, P. Nguyen, Z. Chen, A. Kannan, R. J. Weiss, K. Rao, E. Gonina, N. Jaitly, B. Li, J. Chorowski, and M. Bacchiani. State-of-the-Art Speech Recognition with Sequence-to-Sequence Models. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4774–4778, 2018.

[15] C. Coleman, D. Kang, D. Narayanan, L. Nardi, T. Zhao, J. Zhang, P. Bailis, K. Olukotun, C. Ré, and M. Zaharia. Analysis of DAWNBench, a Time-to-Accuracy Machine Learning Performance Benchmark. *Operating Systems Review*, 53(1):14–25, 2019.

[16] C. A. Coleman, D. Narayanan, D. Kang, T. Zhao, J. Zhang, L. Nardi, P. Bailis, K. Olukotun, C. Ré, and M. Zaharia. DAWNBench: An End-to-End Deep Learning Benchmark and Competition. In *NIPS ML Systems Workshop*, 2017.

[17] K. Davis, R. Biddulph, and S. Balashek. Automatic Recognition of Spoken Digits. *Journal of the Acoustical Society of America*, 24(6):637–642, 1952.

[18] S. Davis and P. Mermelstein. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(4):357–366, 1980.

[19] J. Dean, D. Patterson, and C. Young. A New Golden Age in Computer Architecture: Empowering the Machine-Learning Revolution. *IEEE Micro*, 38(2):21–29, 2018.

[20] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 9, pages 249–256, 2010.

[21] I. J. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[22] Google. Google Assistant. `https://assistant.google.com/`, 2019.

[23] A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber. Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks. In *International Conference on Machine Learning (ICML)*, pages 369–376, 2006.

[24] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon,

J. Qian, H. Liu, and C. Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *NSDI*, pages 485–500, 2019.

[25] J. Guo, T. N. Sainath, and R. J. Weiss. A Spelling Correction Model for End-to-end Speech Recognition. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5651–5655, 2019.

[26] A. Y. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, and A. Y. Ng. Deep Speech: Scaling up end-to-end speech recognition. *CoRR*, 2014.

[27] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. *J. Mach. Learn. Res.*, 18(1):6869–6898, 2017.

[28] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR*, abs/1502.03167, 2015.

[29] M. Jeon, S. Venkataraman, A. Phanishayee, u. Qian, W. Xiao, and F. Yang. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *USENIX ATC*, page 947–960, 2019.

[30] A. Koliousis, P. Watcharapichat, M. Weidlich, L. Mai, P. Costa, and P. Pietzuch. Crossbow: Scaling Deep Learning with Small Batch Sizes on multi-GPU Servers. *Proceedings of the VLDB Endowment*, 12(11):1399–1412, 2019.

[31] L. Liu, Y. Wu, W. Wei, W. Cao, S. Sahin, and Q. Zhang. Benchmarking deep learning frameworks: Design considerations, metrics and beyond. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 1258–1269, 2018.

[32] C. Lüscher, E. Beck, K. Irie, M. Kitza, W. Michel, A. Zeyer, R. Schlüter, and H. Ney. RWTH ASR Systems for LibriSpeech: Hybrid vs Attention. In *Proc. Interspeech 2019*, pages 231–235, 2019.

[33] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu. Mixed Precision Training. In *International Conference on Learning Representations*, 2018.

[34] Microsoft. Cortana. `https://www.microsoft.com/en-us/cortana`, 2019.

[35] Mozilla. A TensorFlow implementation of Baidu's DeepSpeech architecture. `https://github.com/mozilla/DeepSpeech`, 2019.

[36] Nvidia. Nvidia System Management Interface. `https://developer.nvidia.com/nvidia-system-management-interface`, 2019.

[37] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur. Librispeech: An ASR corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5206–5210, April 2015.

[38] V. Panayotov and D. Povey. LibriSpeech ASR Corpus. `http://www.openslr.org/12`, 2019.

[39] R. Raina, A. Madhavan, and A. Y. Ng. Large-scale Deep Unsupervised Learning Using Graphics Processors. pages 873–880, 2009.

[40] T. Sakai and S. Doshita. The Automatic Speech Recognition System for Conversational Sound. *IEEE Transactions on Electronic Computers*, (6):835–846, 1963.

[41] H. Scheidl. CTC Word Beam Search Decoding Algorithm. `https://github.com/githubharald/CTCWordBeamSearch`, 2019.

[42] H. Scheidl, S. Fiel, and R. Sablatnig. Word Beam Search: A Connectionist Temporal Classification Decoding Algorithm. In *International Conference on Frontiers in Handwriting Recognition*, pages 253–258, 2018.

[43] S. Shi, Q. Wang, P. Xu, and X. Chu. Benchmarking State-of-the-Art Deep Learning Software Tools. In *International Conference on Cloud Computing and Big Data (CCBD)*, pages 99–104, 2016.

[44] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[45] D. O. Staff. New ways alexa makes life simpler and more convenient. `https://blog.aboutamazon.com/devices/new-ways-alexa-makes-life-simpler-and-more-convenient`, 2019.

[46] D. Steinkraus, I. Buck, and P. Y. Simard. Using GPUs for machine learning algorithms. In *International Conference on Document Analysis and Recognition (ICDAR)*, pages 1115–1120, 2005.

[47] TensorFlow. Data Input Pipeline Performance. `https://www.tensorflow.org/guide/performance/datasets`, 2019.

[48] TensorFlow. TensorFlow: TFRecord and tf.Example. `https://www.tensorflow.org/tutorials/load_data/tfrecord`, 2019.

[49] N. Vaessen. Word Error Rate for Automatic Speech Recognition. `https://pypi.org/project/jiwer/`, 2019.

[50] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *OSDI*, pages 595–610, 2018.

[51] D. Yu and L. Deng. *Automatic Speech Recognition: A Deep Learning Approach (Signals and Communication Technology)*. Springer, 2015.

[52] M. D. Zeiler. ADADELTA: An Adaptive Learning Rate Method. *CoRR*, 2012.