Evaluating RNN performance across HW platforms (V0.96)

David Levinthal
Performance lead
Microsoft Cloud Services Infrastructure

Introduction

Recurrent neural networks are extremely effective at parametrizing and manipulating the patterns inherent in text. Human communications are most commonly through language and thus text, so the importance of RNNs for human interactions with computers is clear. As most of the data stored in data centers is text based the processing and targeted retrieval of information naturally uses these kinds of networks. These reasons and others have led to a heavy reliance on these recursive algorithms in machine learning, but their complex structures and the associated infrastructure used (word embedding, attention, beam searches and a plethora of transcendental functions) make performance evaluation complex.

In addition to the inherent complexity of RNNs there is the larger issue that the performance is a combination of the hardware performance and the effectiveness of the integration of the low-level hardware specific math libraries with the machine learning frameworks (Tensorflow, MXNet, PyTorch, CNTK etc). Unless this second step is well done, the hardware performance may not actually be available to a particular framework or even any of the frameworks.

In case these issues were insufficient, the question of numerical representation must also be addressed. While hardware might support various lowered precision modes one needs to know if the network can be either trained at these precisions or if inference can be accurately run at lowered precision through model quantization. Lowering the precision can pack a larger model into less memory while keeping the latency down. It can also increase the execution throughput with SIMD operations

The combination of these factors means that a truly useful RNN benchmark must be able to be roughly formulated at an equivalent level across frameworks, at least close enough to be sure the framework support can be evaluated. Further, the problem being evaluated must be able to be run in varying size to test capacity issues. Ideally the tests would be able to be run in various precisions and have a well understood methodology for evaluating accuracy.

The final critical component is that the tests must be able to run off a large public data set that can be used across frameworks.

Except for the issue of variable accuracy, multi-layer (4) sequence to sequence translators using large vocabularies and hidden/embedding size and some reasonable attention flow mechanism (Luong, Bahdanau, bilinear etc) address all the other concerns. In what follows, an illustration of how such a benchmark can be constructed from public domain source bases and data sets will be discussed and how the measurements and data interpretation can be done.

All measurements were performed on an Intel® Haswell based platform from ZT systems capable of hosting 4 high power GPUs. For the tests performed here, the system was equipped with one

each of the following Nvidia GPUs: A P4, a P40, a P100 and a V100. The system was loaded with Ubuntu 16.04. Cuda9, CUDNN7, NCCL2 were installed with dpkg. Care was taken to ensure that /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor was set to “performance”.

We found that we could install Tensorflow R1.4, MXNet 12.01 and PyTorch 0.4.0 side by side. The three frameworks were all built from source and run natively, without docker or Anaconda. They coexisted happily on our system. Installation was straight forward per the directions on the github sites (see references).

The evaluation was restricted to these three frameworks as there were well supported language translators that could be configured in reasonably consistent manners. Theano support appears to have been terminated by MILA and so none of the translators developed for that framework were considered. We were unable to find any serious RNN tests on caffe/caffe2, let alone language translators. Similarly, we could not find purely python based language translators for CNTK in the public domain. The three chosen frameworks have excellent language translators invoking the frameworks from python and require no additional builds. This ensures that only the python-framework is exercised by the tests without additional compiled components.

The point is to be able to evaluate the framework plus HW performance for multiple frameworks across the spectrum of machine learning execution HW solutions that will become available in the next few years. It is NOT to compare the translators. The translators are the vehicle of choice due to the wide variety of RNN features they employ. If some future or current solution does not support the full set across all frameworks it should be easily seen.

Translator descriptions

The basic baseline translator we use is a 4-layer seq2seq LSTM structure with a hidden size equal to the embedding size. There are differences in the details however. I am not sure that all the models use a bidirectional LSTM layer for the first encoder layer. I can find code for bidirectional encoder layers in TF/NMT and MXNet/Sockeye, but that cannot always be required. Further forward feeding of hidden states across the layers may only be a property of TF/NMT. Thus, the models considered could not easily be made to be exactly equivalent, so direct performance comparisons based on this analysis across the models and frameworks should not be taken too seriously. That said, the models are reasonably close and thus ensure that an important baseline functionality of the frameworks can be evaluated across the spectrum of hardware using these applications.

An attention mechanism along the lines of that introduced by Bahdanau (https://arxiv.org/pdf/1409.0473.pdf) and updated by Luong (https://arxiv.org/pdf/1508.04025.pdf) is also part of the specification but the exact details can be varied when one considers performance as the attention mechanism is not a bottleneck. The maximum sentence length was set to 50 tokens.

A beam search, applied to the final output word choice, is also required but 2 of the translators set the width to 5 and one to 10 and I did not standardize this. The beam scan is only executed during inference and in the analysis of inference performance the beam width was mostly run at 1 for reasons that will become clear.

During training the embedded representation of the target words produced by the attention + decoder is never directly compared to the vocabulary table generated during startup (code inspection of

the TF/NMT). The log likelihood probability/word (ie Perplexity) which gets minimized during training, is computed only from the context value for each time step and the tagged translation token so no accesses to the vocabulary are required during training.

During inference access to the vocabulary table is required. This introduces a substantial load that does not occur during training. As each time steps' decoder output is produced the embedded value must be shifted to the location of the closest word in the vocabulary. This requires a substantial access to the vocabulary table of the embedded encodings. Further with a beam width greater than 1 the pass through the decoder and attention mechanism must occur multiple times. Then at the end the string of translated tokens in embedded representation must be converted to the character strings and then printed.

The hidden size/embedding size focused on a value of 1024 for all cases, but this is varied to gain greater insight into the performance. The previous paper (https://github.com/David-Levinthal/machine-learning/blob/master/Introduction%20to%20machine%20learning%20algorithms.pdf) showed that the FP operation count was dominated by the FMAs of the dense matrix math in the LSTM cells with a value of $16*hiden\_size^2$. This quadratic behavior will also exist in the memory usage by the weights so that memory access performance costs will also grow quadratically This quadratic dependence means that for large hidden sizes the LSTM math and memory access costs will become the dominant performance limitations.

Two of the models, Tensorflow/NMT (https://github.com/tensorflow/nmt)and MXNet/Sockeye (https://github.com/awslabs/sockeye), bucket the sentence training set into mini batches based on restricted sentence length. This can throw out short sentences that produce little constraint on translation and removes sentences longer than the maximum length which can be problematic for neural machine translation but this is not done in all cases. The minibatch size for these two was set to 128 sentences. The PyTorch/Fairseq-py (https://github.com/facebookresearch/fairseq-py) lstm based model defined the mini batch size by limiting the number of tokens and varying the number of sentences in the minibatch.

In all three cases the application has to deal with sentences within a minibatch of unequal size. How this is handled complicates the execution relative to the usual simple RNN benchmarks.

The output of Sockeye very kindly prints out the details of the bucketizing for the training and validation sets, their sizes and the relative word counts in the two languages making understanding of these processes very clear. Sentences of length less than 10 are put in the (10,10) bucket. Sentences longer than 50, per the limit set by flags are excluded. The average number of tokens per sentence was 30.5. The output to input word count ratio was 1.07. This difference illustrates the need for input and output wps rates if one wants to calculate achieved flop rates for a given piece of hardware.

Data set

A consistent data set is critical. For this purpose, the WMT set for German to English translation was chosen. The data set consists of the Europarl/v7, WMT13 and WMT16 files (http://www.statmt.org/europarl/v7/de-en.tgz, http://www.statmt.org/wmt13/training-parallel-commoncrawl.tgz, http://data.statmt.org/wmt16/translation-task/training-parallel-nc-v11.tgz,

http://data.statmt.org/wmt16/translation-task/dev.tgz, http://data.statmt.org/wmt16/translation-task/test.tgz)
downloaded by the tensorflow/NMT installation script (https://github.com/tensorflow/nmt/blob/master/nmt/scripts/wmt16_en_de.sh). The files were preprocessed for tensorflow/NMT and MXNet/Sockeye with the NMT script mentioned above. This results in a slightly different tokenizing than would be created by the Sockeye installation instructions in the WMT tutorial, however both preprocessing steps invoke the utilities in https://github.com/rsennrich/subword-nmt.git in very similar ways. The consistency was checked by inspecting the differences in the tokenized test files and Sockeye appeared to have no difficulties processing the NMT files.

Preparing the above data sets for PyTorch/Fairseq-py did require applying the preprocessing, explained in the Fairseq-py Readme.md file, to the data files from the above distributions. This merely required copying the appropriate files into the data subdirectory of the fairseq-py cloned tree and renaming them to be consistent with the structure outlined in the instructions for using the preprocess.py script (https://github.com/facebookresearch/fairseq-py/blob/master/preprocess.py). The resulting formatted files are rather difficult to read so a comparison of the tokenizing and thus the average sentence length is tricky to understand.

Training

Training a language translation model is a time-consuming process if one wants a model that can be used for accurate translations when running inference. Here we focus on performance measurements. It turns out the speeds from the translators rapidly stabilizes so that runs of a few hours can be used for the performance measurement. On the fastest GPUs used here (Nvidia V100) a model can converge in 1 to 2 days on a single card. All measurements were performed on single cards.

Invocations

As the platform had four different GPUs a bit of care was required to run the translators in the desired manner using the desired training, validation and test files consistently. Tensorflow/NMT was run with pyton2.7 as we had issues with TF/NMT on python 3.5 at one point. MXNet/Sockeye and PyTorch/Fairseq-py were run with python 3.5. The WMT data files were in a home directory which had to be explicitly typed out for the scripts to handle the paths correctly.

The GPU selection was made through the CUDA_VISIBLE_DEVICES environment variable for NMT and Fairseq-py. For Sockeye the –device-ids flag was used and nvidia-smi was used to be sure which GPU was activated. The logs for stdout and stderr were kept for post processing. The GPU activity was also logged with a command like the following identifying the translator and GPU

```
nvidia-smi dmon -i 1 -o DT -o T -d 10 -s pucvmet >fairseq-py_p100_dmon.log
```

The following are example invocations used for the “standard translator” runs

NMT

```
python -m nmt.nmt --src=de --tgt=en \
--hparams_path=nmt/standard_hparams/wmt16_gnmt_4_layer.json \
--out_dir=/home/levinth/nmt_out_v100/deen_gnmt \
```

--vocab_prefix=/home/levinth/WMT/vocab.bpe.32000 \
--train_prefix=/home/levinth/WMT/train.tok.clean.bpe.32000 \
--dev_prefix=/home/levinth/WMT/newstest2013.tok.bpe.32000 \
--test_prefix=/home/levinth/WMT/newstest2015.tok.bpe.32000 > deen_gnmt_log_v100_py2.txt 2>&1

Sockeye

python3 -m sockeye.train -s /home/levinth/WMT/train.tok.clean.bpe.32000.de \
-t /home/levinth/WMT/train.tok.clean.bpe.32000.en \
-vs /home/levinth/WMT/newstest2016.tok.bpe.32000.de \
-vt /home/levinth/WMT/newstest2016.tok.bpe.32000.en --num-embed 1024 --rnn-num-hidden 1024 \
--num-layers 4 --rnn-attention-type bilinear --max-seq-len 50 --monitor-bleu 500 --device-ids 0 \
--batch-size 128 -o wmt_model_gpu0 > wmt_4layer_32k_bilinear_len50_gpu0.log 2>&1

For many of the Sockeye runs the flag –checkpoint-frequency 20000 was used to lower the frequency of checkpoint files being created as the root disk was a 1TB SSD.

Fairseq-py

python3 train.py data-bin/wmt_de_en --lr 0.25 --clip-norm 0.1 --dropout 0.2 --max-tokens 4000 \
--arch lstm_luong_wmt_en_de --save-dir checkpoints/wmt_luong_cuda2 \
> fairseq_luong_1024_cuda2.log 2>&1

For Fairseq-py the embedding and hidden sizes were adjusted from 1000 to 1024 by editing the 3 values in lstm.py script in the cloned directory (https://github.com/facebookresearch/fairseq-py/blob/master/fairseq/models/lstm.py)

Relative performance of P40, P100 and V100

Interpreting the outputs of the three programs in a consistent way is not entirely straightforward as the outputs reflect metrics with different definitions. Further, the frequency of output was not consistent with the above invocations of the applications. This introduced some difficulties in evaluating good average values. Typical output text is shown below:

Fairseq-py

epoch 002: 15000 / 37483 loss=2.62 (3.64), wps=8556, wpb=3537, bsz=120, lr=0.25, clip=100%, gnorm=0.402919

Sockeye

[INFO:root] Epoch[3] Batch [105700] Speed: 269.69 samples/sec perplexity=6.680722

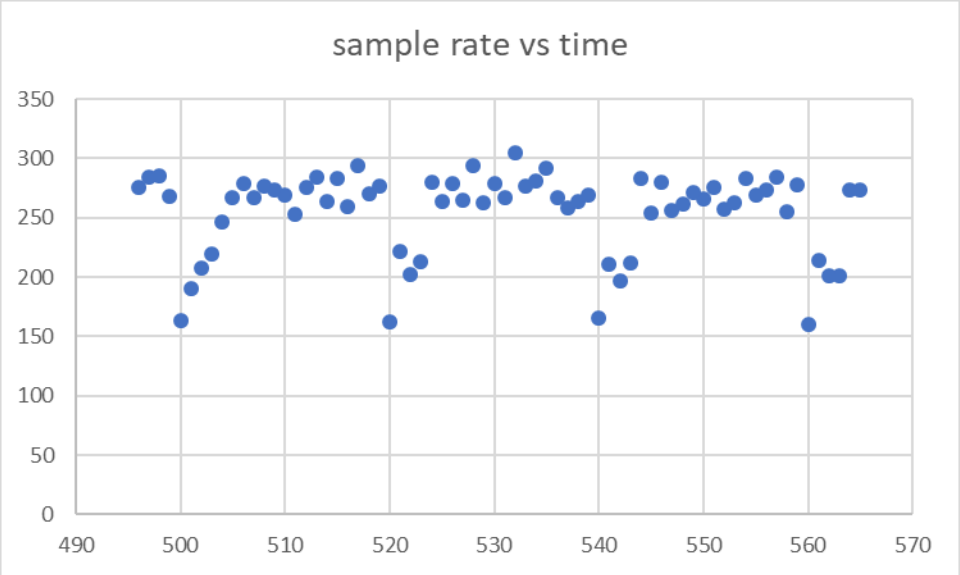
NMT

global step 339200 lr 0.00195312 step-time 0.57s wps 12.52K ppl 9.37 bleu 28.86

On communication with the developers I was informed the Fairseq-py wps value was for output words/sec. The NMT wps value represents the sum on input + output words/sec, while the step-time refers to the time/batch of (128) sentences. The Sockeye output was quite clear, in view of the other

output discussed earlier, and shows the speed in sentences/sec for the way it we invoked it. If sockeye had been invoked with word based batching then the samples/sec would refer to words.

Sockeye prints out values every 50 batches by default. Prior to lowering the checkpoint rate for Sockeye, the program showed a rather curious periodicity in the sample/sec rate as shown below



The effect vanished with the checkpoint-frequency flag. As the first run was done to convergence, it took a while and rerunning wasn't an attractive option. The speed averages used in the analysis represent values calculated avoiding the dips.

On communicating with the Sockeye authors, we learned that the speed measurement for the first 50 batches after the checkpointing includes the checkpointing time. Further as the monitor-bleu flag was set to 500 the bleu decoding subprocess is restarted after each checkpointing which effects the next several time bins.

The results for the relative performance of the three cards for the three frameworks is shown below

	P40	P100	V100
fairseq-py	4627.36	5797.84	8564.68
ratio	1.00	1.25	1.85
Sockeye	152.18	180.92	271.88
ratio	1.00	1.19	1.79
NMT	6.88	8.98	12.74
ratio	1.00	1.31	1.85

It should be noted that an initial test was done on theV100 with a cuda8/Cudnn6 based installation. The result for NMT was ~30% lower than what was seen with Cuda9/Cudnn7. The consistency of the V100/P40 ratios across the three frameworks and applications shows an impressive breadth of support by Nvidia and the framework developers.

Contributions to the performance

As shown in the previous paper, the number of FP operations executed in the forward pass through an LSTM cell is 16 times the hidden size squared. Also shown was that the FP operation count for training (forward pass + backpropagation) was 3 times that of the forward pass alone. Due to this dominance of the LSTM FP operation count, the FP operation per second rate can be evaluated for translators from the input and output wps rates. It simply requires multiplying those rates by the number of layers times 16 times the hidden size squared. As the encoders frequently have a bidirectional layer it would be good if the input and output word rates were printed separately, so the extra layer of encoder LSTM cells could be handled. This also enables handling the different sentence lengths that result between source and target languages. Thus, an evaluation of the computational efficiency of the GPU or alternate ML accelerator could be made with reasonable accuracy.

The dominance of the LSTM FP operation count allows us to explore the contributions to the performance of the translators on the assortment of GPUs. The hidden size and embedded size were set to the values of 256 and 512 in addition to the runs made at 1024. This results in the computation for the three runs being in the ratio of 1:4:16.

Sockeye:

There are a lot of useful metrics printed to stdout by sockeye. If the msec/sentence rate (average of 1000/"sample rate" value) for the Sockeye translator is plotted this way for the three GPUs one sees the following:

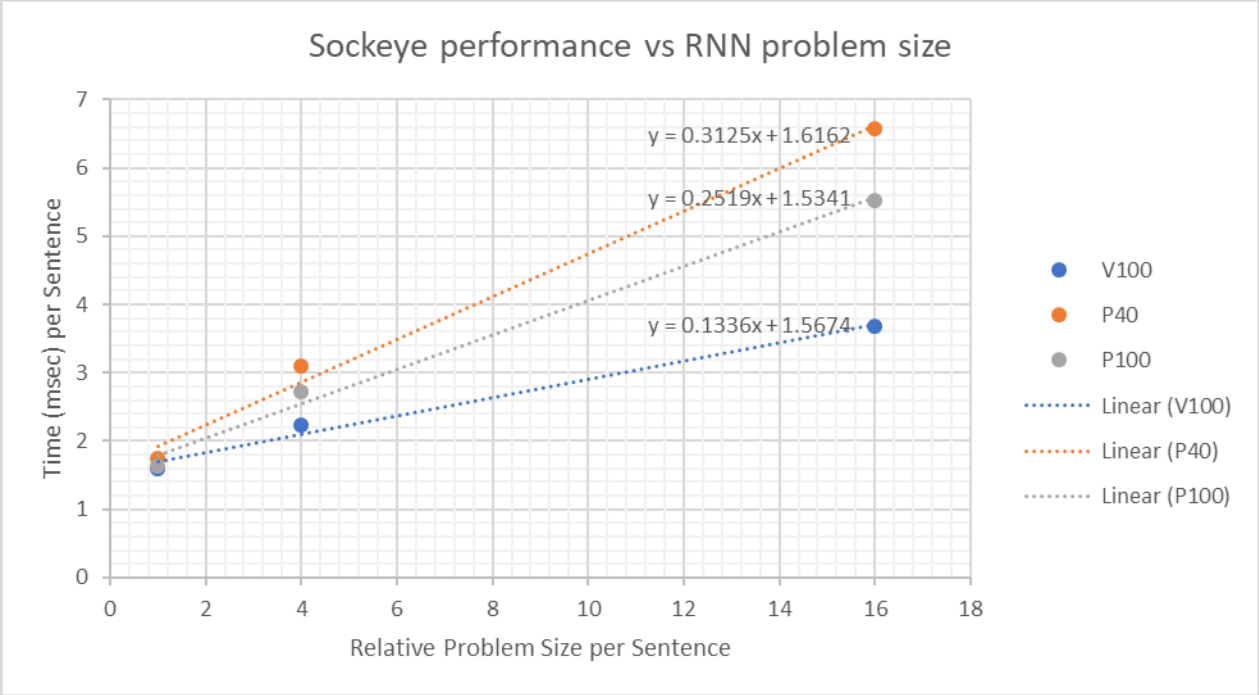


Figure 1

The consistency of the intercepts of the straight-line fits is worth noting. It shows that there is a large baseline of execution time that is independent of the capacity of the GPU!

NMT

NMT shows the same pattern with the msec/sentence value being 1000 times the average of the “step time/128 (the batch size). The intercepts are within ~2.5% of their average, which is again consistent with a constant.

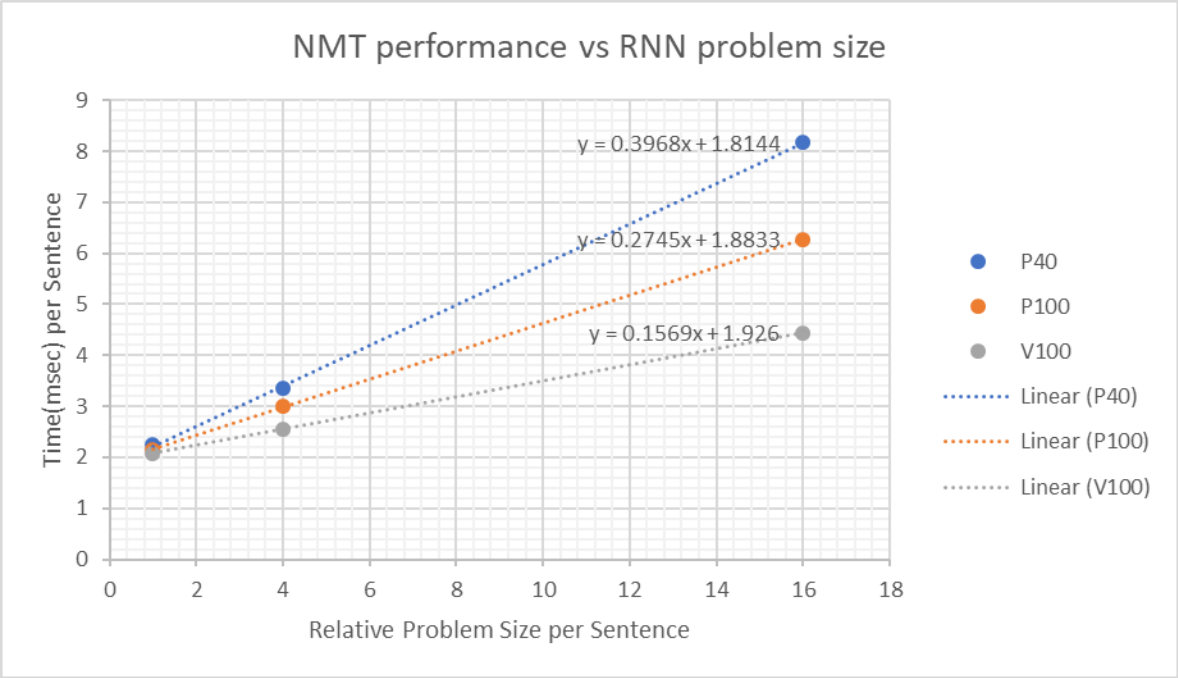


Figure 2

For Fairseq-py getting a time/sentence from the output required some assistance from the authors. In hindsight simply reading the code of train.py (https://github.com/facebookresearch/fairseq-py/blob/master/train.py) made everything quite clear

Using the periodic output

| epoch 001: 37000 / 37483 loss=3.21 (5.19), wps=8572, wpb=3543, bsz=120, lr=0.25, clip=100%, gnorm=0.575767

one gets a words/sentence value of wpb/bsz = 29.5. The msec/sentence rate would be 1000*words_per_sentence/wps = 3.44msec/sentence and then averaging the values

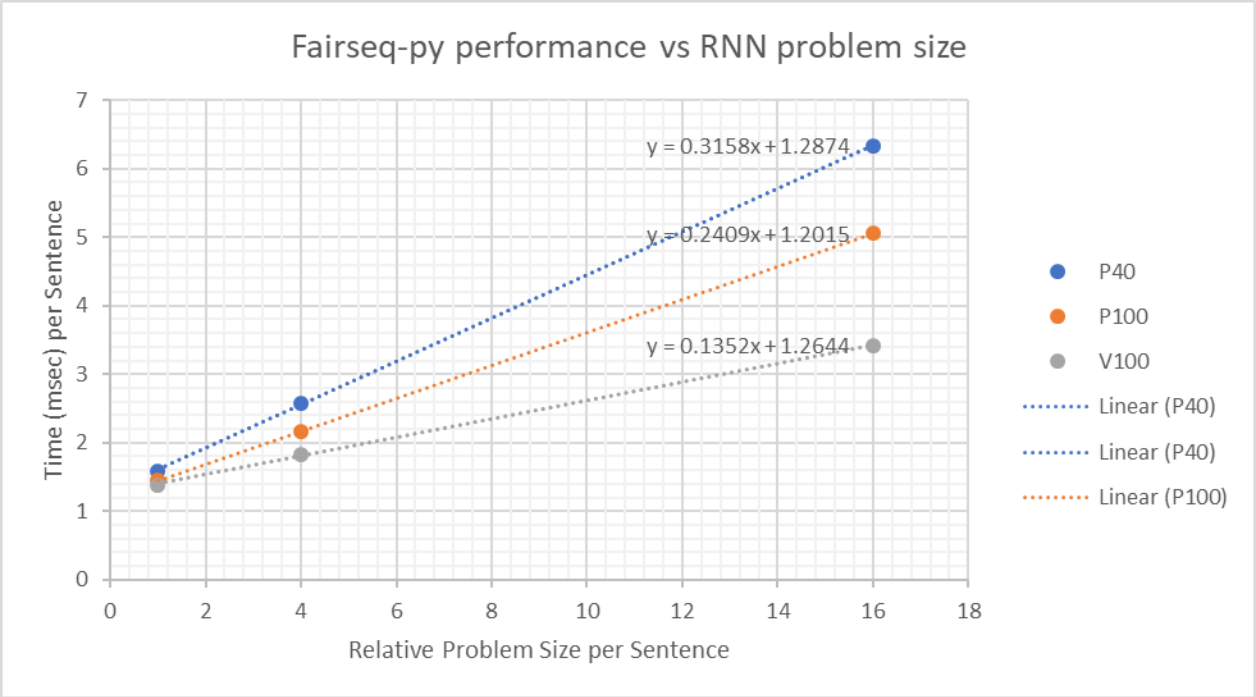


Figure 3

Interpretation

The data for the three GPUs is reasonably fit with 3 straight lines, for each of the translators, confirming that the quadratic behavior dominates. What is of particular interest is that the intercepts for all three GPUs are very close to the same values for each of the translators.

	P40	P100	V100	Average
Sockeye	1.62	1.53	1.57	1.57
NMT	1.81	1.88	1.93	1.87
Fairseq-py	1.29	1.20	1.26	1.25

This suggests that the large constant components have nothing to do with the capacity of the GPUs but rather is a function of the framework and platform. Further, as the GPUs become more capable this results in a lower slope. By the time one reaches the V100 only slightly more than half (57%) of the execution time is effected by the GPU capability and making faster accelerators, without addressing the large baseline, may not be a cost effective approach to improving training time.

The question of the root cause of the baseline is not obvious but a few numbers perhaps offer some enlightenment. The training data set is 4.5 million sentences. With the length limit this is brought down to 4 million roughly. At 30 tokens per sentence this means 120 million tokens. An embedded representation of a token in these tests varies between 1KB to 4KB. If the training set were stored as embedded data this would mean 120GB to 480 GB, which is clearly far too large for the available memory. This means that each mini batch buffer of embedded representation of the sentences (ie 256->1024 weights/token, 30 tokens/sentence, 128 sentences/mini batch) must be constructed on the CPU.

This suggests there may be a value in building this large collection of pre-embedded mini-batches once and storing the entire thing to a file, particularly on an NVMe drive.

If the above explanation is not the root cause then reducing the baseline might be addressed through improvements to the frameworks possibly, running the epoch across many machines in a parallel decomposition with some sort of message passing like MPI to synchronize the changing values for the fit parameters or by constructing an ML accelerator that can contain the entire problem so that no interaction with the CPUs, PCIe bus and operating system is required.

We can similarly look at the slopes, investigating how increasing matrix size effects performance:

	P40	P100	V100
Sockeye	0.3125	0.2519	0.1336
NMT	0.3968	0.2745	0.1569
Fairseq	0.3158	0.2409	0.1352

The slopes for MXNet/Sockeye and PyTorch/Fairseq are very similar while the TF/NMT slopes are uniformly steeper suggesting that MXNet and PyTorch are using the hardware a bit more efficiently.

The usage of the GPUs can also be investigated with the nvidia-smi tool. The utilization during training is high increasing with problem size but high even for the smaller problems.

GPU utilization	v100	P100	P40
NMT1024	91.3	94.3	95.2
Sockeye1024	94.5	95	96.1
Fairseq1024	96.7	99	99.1
Sockeye512	Not taken	88.6	92.7
Sockeye256	82.2	83.6	84.9

More translators and frameworks will be added to this as they become available or are brought to my attention.

The technique of varying the problem size and extrapolating to a zero work intercept can probably be done with CNNs by varying the size of the input images when doing synthetic data performance runs.

Inference

For most benchmarks inference is many times (typically around 3X) faster than training (ex: tf_cnn_benchmark, the RNN benchmark ptb as shown later). For these kinds of seq2seq based translators, this appears not to be the case. This is illustrated by the data in the graph below for TF/NMT run in a default mode for the predefined 4-layer model, plus two other hidden size versions, trained earlier. The input data was a concatenation of the newstest preprocessed files for 2009 through 2016. This yields a file with 22191 sentences (~a book) which takes about 30 minutes to process on the V100 (including setup).

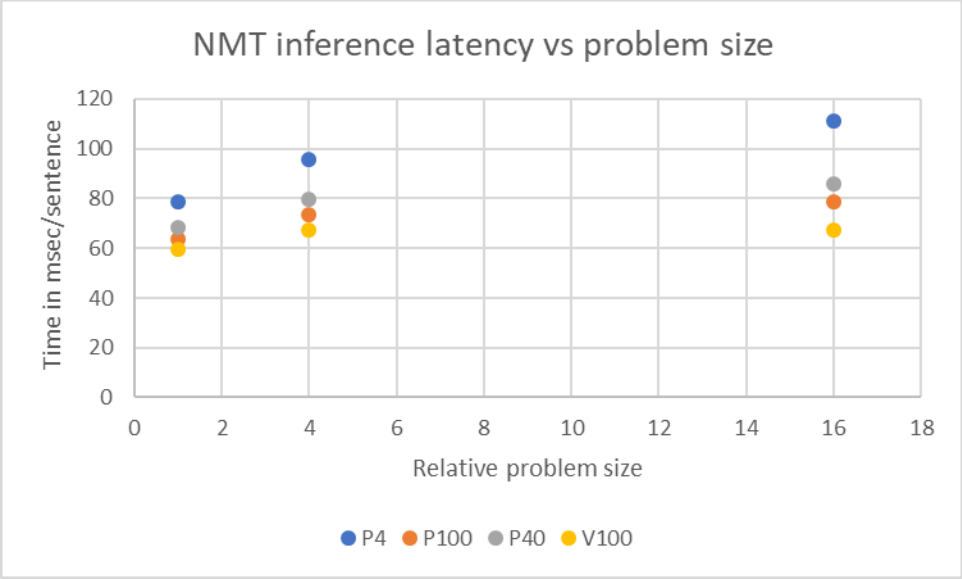


Figure 4

In fact, the time/sentence is 10X to 35X larger than it was for training (see fig 2) and this was true for the other translators as well. The graph clearly shows that the time per sentence saturates, particularly on the faster V100 and P100. This indicates that the work on the GPU is not the limiting factor.

On communicating with the authors of both Sockeye and NMT we were told that in translators, the beam search is only applied during inference causing a great increase in the amount of work. In such a case each target word must be found from the vocabulary file of embedded representations. I suspect this is done on the CPU creating an exchange of the decoder output embedded representation to the CPU for each word in the time sequence time the beam search width. Finally, a real output sentence is passed back to the CPU for conversion to character representation and logging or sending to the user of the service, which is not required during training. In addition, the sentences are likely translated in order and the variations in sentence length within the minibatch lowers the efficiency.

The impact of the beam search was investigated by lowering it through a flag for the NMT inference runs with the large 4-layer model. Lowering the beam width to 1 and raising the batch size resulted in lowering the time/sentence by 7X as shown below. The value of 0 was suggested as a way of disabling the beam search but using a value of 1 was a bit faster.

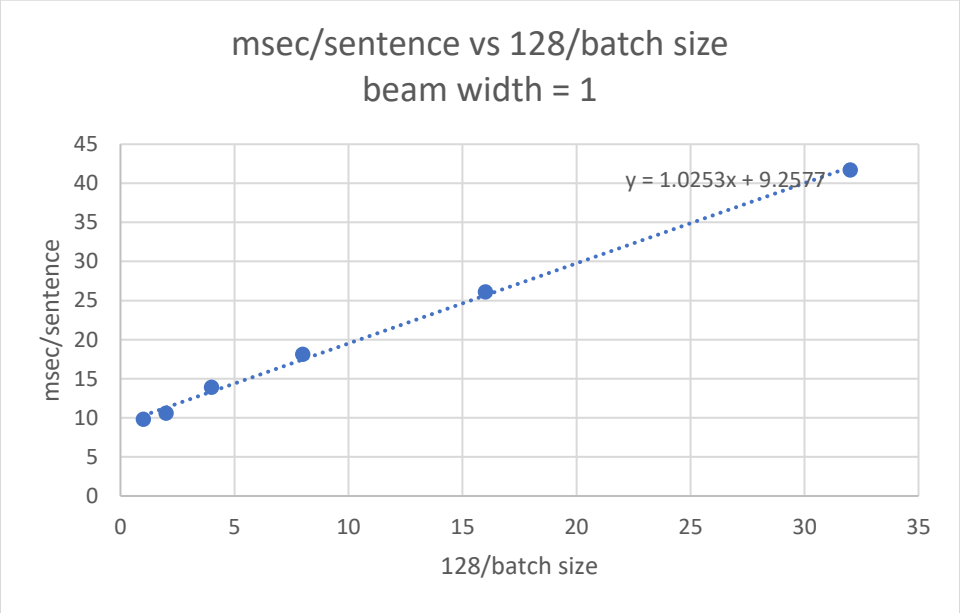
Of course, removing the beam search hurt the bleu score a fair bit, so it would be unlikely that this would be a good thing to do in a real usage case. BLEU (Bilingual evaluation understudy) is the standard technique for evaluating translation accuracy, incorporating word selection and positioning in the output sentence. A discussion of BLEU can be found in the BLEU references at the end.

batch size	beam width	msec/sentence	bleu
32	10	67.50	27.8
128	10	74.99	27.8
32	1	13.88	26.2
32	0	14.87	26.2
128	1	9.78	26.2
256	1	9.73	26.2

Clearly the time/sentence decreases with batch size. On might even guess that the time might go something like:

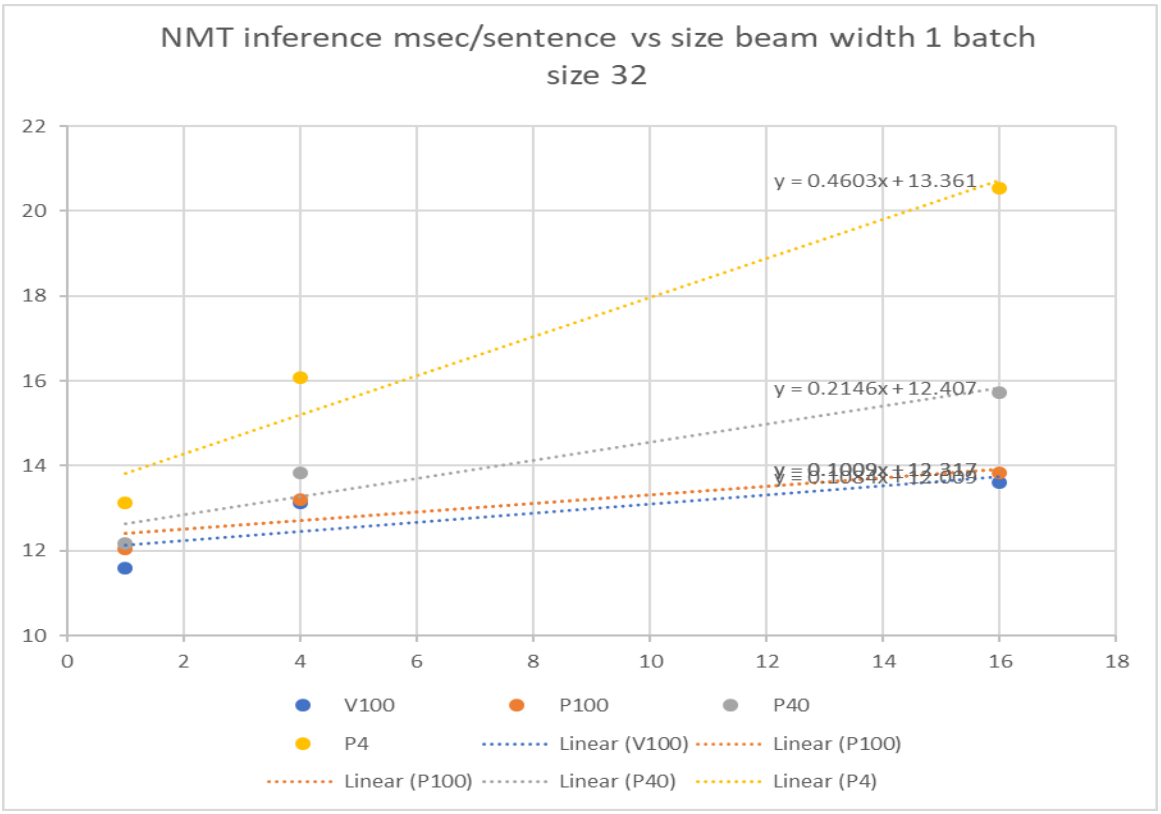
Time/sentence = Const_1/batch_size + const_2

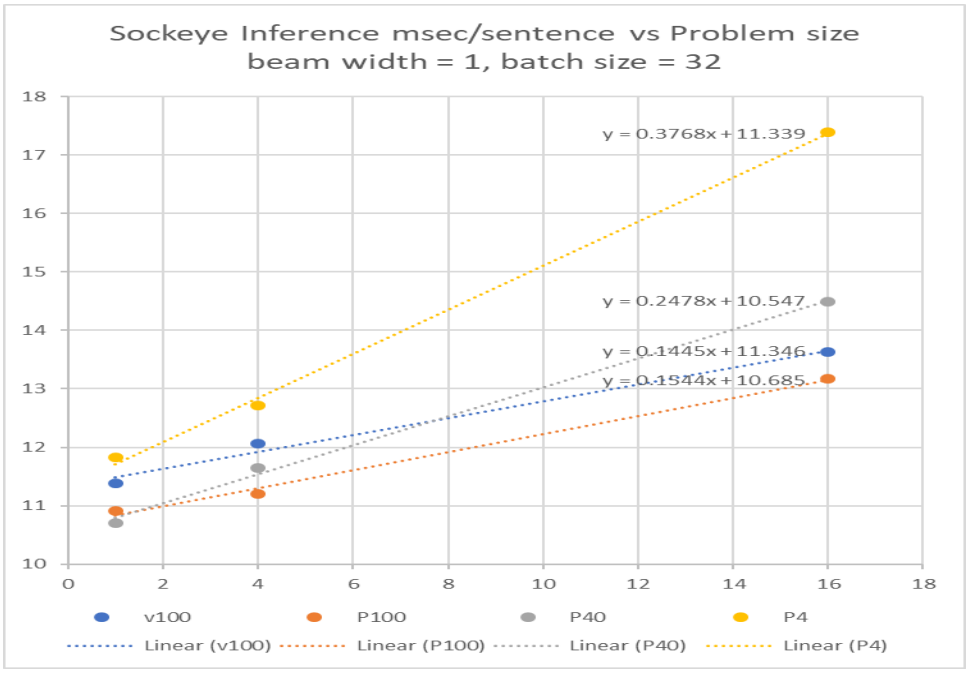
For a beam width of 1 such a suspicion appears to be well warranted:



This constant (with beam width 1) is ~7X larger than what we saw for training. I believe that in the case of inference as each time step's decoder output is generated it must be correlated with the best softmax relation with the words of the vocabulary and then that value is used in the decoder for the next time step. This complexity would not be required during training as the correctly embedded word value is already known.

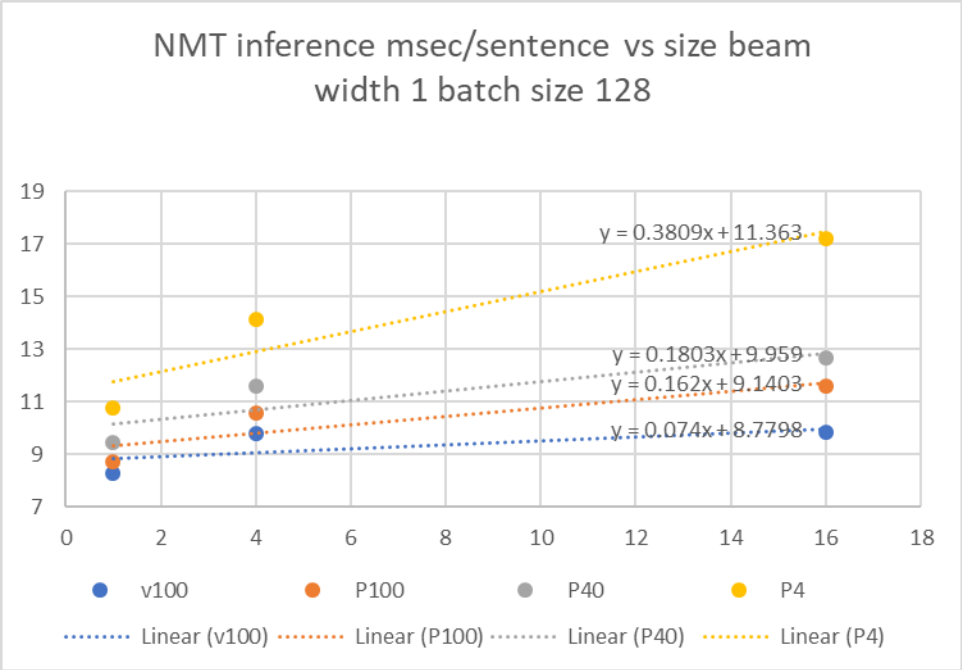
This leads us to redoing the problem size scans with a beam width of 1. The data for both Sockeye (upgraded to V1.15.1) and NMT are shown below for a mini bath size of 32.





Even with the beam size of 1, the purely linear dependence on problem size with a constant baseline does not appear. The NMT time/sentence saturates for the faster V100 and P100 as the mid points are all above the line, suggesting that the GPUs are still not the performance limitation. For those cards the time/sentence for the 512 and 1024 embedded size models were approximately constant.

The scan on NMT was then performed with a batch size of 128. The data is shown below



The saturation becomes more severe with even the P40 runs becoming limited by something other than the GPU. This is clear from the GPU utilizations which do not approach the values of 80 to 95% seen during training.

GPU utilization	v100	p100	p40	p4
1	19.7	29.1	31.6	44.5
4	33	33.3	40.8	55.7
16	48.8	58.4	58	69.4

The complexity of translation inference with these substantial algorithms is driven home when one looks at the PTB example. This uses a fixed length “sentence” of 32 words so there are never LSTM cells in the unrolled loop that have no inputs. The data shown below had a hidden size of 1024 and a fixed sentence length of 32. The number of LSTM layers were varied. This was done with the TF R1.2 version of PTB on TF R1.3 built with cuda8/cudnn6. The values are in msec/sentence.

	layers	P4	P40	P100
training	1	1.25	0.65	0.65
inference	1	0.44	0.24	0.23
ratio	1	**2.86**	**2.65**	**2.82**
training	2	1.76	0.85	0.87
inference	2	0.56	0.29	0.29
ratio	2	**3.12**	**2.91**	**2.97**
training	4	2.81	1.34	1.49
inference	4	0.83	0.43	0.45
ratio	4	**3.37**	**3.15**	**3.26**

Thus, we see the training/inference time ratios of ~3 instead of 1/15 to 1/35 that we saw with the more complex translation applications. This highlights the problem of identifying realistic benchmarks for evaluating all the new HW options that we will be seeing and guiding the simulations the HW architects use in designing the future machine learning accelerators.

Conclusions

Establishing a good machine learning benchmark suite for performance evaluation is complicated. The plethora of frameworks must be addressed, and the applications cannot be just simple kernels. Mutli layer seq2seq language translators appear to offer a wealth of challenges for both training and inference. The technique of varying the problem size can be useful for isolating performance dependencies and seeing what can actually be improved. For the current frameworks and translation algorithms of the type discussed here, there may not be much room for improvement from faster PCIe gen3 based devices in cost effective manner.
Inference is actually harder to characterize than training. The use of a beam search has a very large impact on the performance of the translators investigated. With the beam search disabled the inference timing behaves in a more intuitive manner but clearly becomes limited when larger problems are run. For translation as a cloud service to be fully practical from mobile devices these issues probably need to be addressed.

Conjecture

I wonder if simply using the distance between the translation result and the tagged translation token would work during training. This might prove to be computationally easier than the log likelihood perplexity standardly used.

$$D = \sum\sum(decode_i - tagged\_translation_i)^2$$

The inner sum being over the embedding size components of the token representation, while the outer sum is over the words in the minibatch.
If the above vectors are normalized to unit length, so the tokens occupy the surface of a unit sphere in an embedded size dimensional space and the decoder output is a unit vector in that space, the expression can be greatly simplified.

$$D = 2*\sum\sum(1 - decode_i * tagged\_translation_i)$$

And since this is to be minimized one could simply use

$$D = -\sum\sum(decode_i * tagged\_translation_i)$$

which computes easily on an accelerator being a simple sum of dot products.

This might be particularly effective for the vocabulary searches during inference as this becomes a simple matrix * vector operation followed by a search for the large values for the beam search and final construction of the output translation.

Future work

It will be interesting to see if the technique of varying the problem size can be more broadly applied to include other kinds of problems like those addressed by CNNs, where the approach would be to vary the input image size.

This work will be continued to include Pytorch OpenNMT and Chainer KNMT. Further we will investigate the practicality of profiling these systems under training and inference loads. An investigation of the transformer and convolution based translators will also be conducted.

Acknowledgements

The author would like to thank the teams that developed the translators used and particularly Rui Zhao of Google, Tobias Domhan and Felix Hieber of AWS and Myle Ott of Facebook who made sincere efforts to try to get me to understand language translation. The mistakes in the text are purely mine. Without their considerable efforts there would be a great deal more. I would like to thank Adi Oltean of Microsoft for his insights and last but not least I would like to thank my colleague Rajiv Kapoor of Microsoft for discussions/debates and his efforts in preparing this.

References

https://www.tensorflow.org/install/install_sources

https://github.com/pytorch/pytorch

https://github.com/apache/incubator-mxnet/tree/0.12.1

https://github.com/tensorflow/nmt

https://github.com/awslabs/sockeye

https://github.com/facebookresearch/fairseq-py

https://arxiv.org/pdf/1409.0473.pdf

https://arxiv.org/pdf/1508.04025.pdf

Bleu:

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.19.9416&rep=rep1&type=pdf

https://nlp.stanford.edu/pubs/luong-manning-iwslt15.pdf

https://slator.com/technology/how-bleu-measures-translation-and-why-it-matters/