Performance analysis of deep neural networks
making the world safe for Skynet

David Levinthal

Microsoft

Azure Cloud Services Infrastructure

V0.2

Performance of Deep Neural Networks

- Agenda

1. Introduction
2. Components and algebra of DNNs
3. Characterization of DNNs
4. Performance analysis of DNNs

Machine learning and Deep Neural Networks

- Machine learning works by building a network of simple computation nodes executing a “output = F(weight*input + bias)” calculation and using known data to find the optimal weights and biases to identify the patterns in the inputs that correlate to the outputs
- The model is (usually) trained on tagged data sets (training)
- The trained model can be used to predict the output for untagged input data (inference)

Deep neural networks (DNNs) for image, text and speech processing

- Identifying contents of images
  - Standard approach today are convolutional neural networks (CNNs)
  - Critical building block is sweeping a 2D filter over a surface
  - Stack many filters in a stage and then stack many stages
  - Filters are a set of weights and a bias which are optimized to reproduce tagged data (training)
- Processing text: Natural language processing (NLP)
  - A sentence is a sequence of words with the meaning contained in the choice of words and their positions in the sentence
    - Interpretation is accomplished (RNN method) through a recursive loop
      Or Stacked attention mechanism (Transformer)
  - Translation is done with a sequence to sequence/Encoder->Decoder network
    - Direction matters: German->English is easier than English->German
- Processing speech to text (STT) combines both
- See https://github.com/David-Levinthal/machine-learning/blob/master/Introduction%20to%20machine%20learning%20algorithms.pdf

Deep Neural Networks and their application

- Variety of algorithm designs:
- Fully connected networks (FCN)
  - Integrating disparate data to identify pattern
  - Identify social network, targeted advertising, targeted misinformation campaign
- Convolution Neural Network (CNN)
  - Image classification, sound file analysis
- Recursive Neural Network (RNN)
  - Time sequence data, language translation, question-answer
- Transformer Networks
  - Translation, hybrid usage
- Reinforced Learning
  - Games, simulations, things that can be replayed over and over
  - a different training technique, not a network style

Deep Neural Networks

- Deep Neural Networks (DNN) can be represented as fabrics of nodes, where the nodes represent numerical operations on (multi dimensional) arrays of data

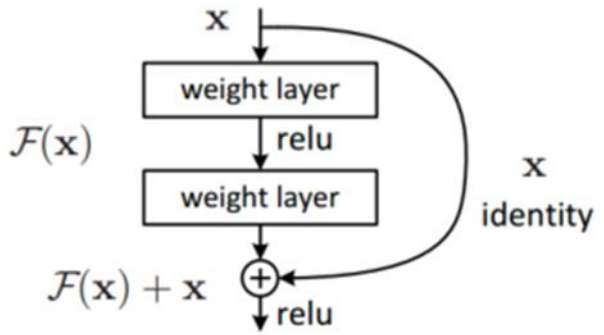
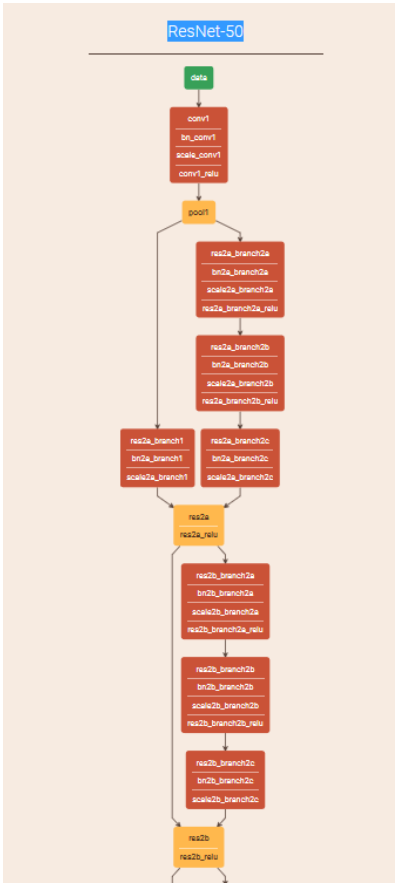
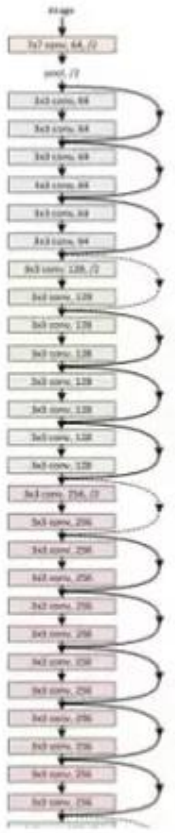


Figure 2. Residual learning: a building block.





Simple linear algebra at the lowest level

- Output = weight*input + bias is the basic expression used throughout
- In CNNs
  - this expression is summed over the area of the filter with different weights and inputs at each position
  - And then summed over the layers of inputs with different weights for each input layer
  - This process is repeated for each output layer

$Output_{l,m,n} = \sum \sum weight_{l,d,i,j} X\ input_{d,i+m,j+n} + bias_d$ (sums over I, j and d)

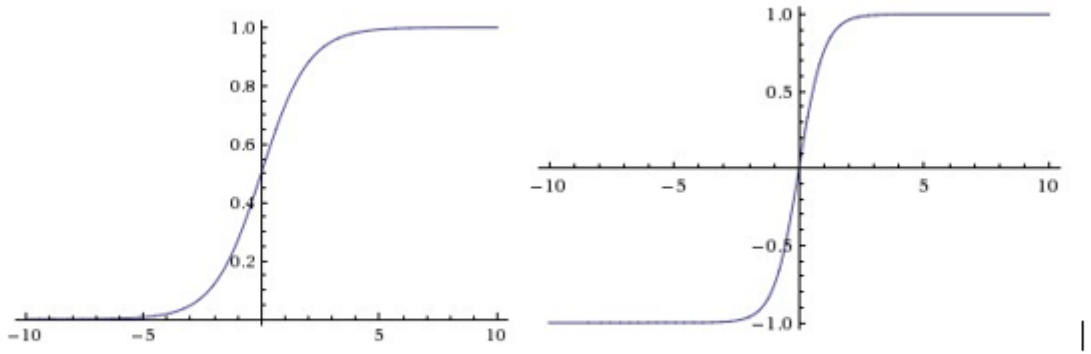
The number of FP operations can be calculated as

FP_ops = (2*Wfilt*Wfilt+1)*input_depth*filter_count*Wout*Wout

Where Wfilt is the filter width and Wout is the output size

Activation

- The output of the linear algebra operations are run through “activation” functions
  - Common ones are Relu(x) = max(0,x), tanh(x) and sigmoid(x)



Left: Sigmoid non-linearity squashes real numbers to range between [0,1] **Right:** The tanh non-linearity squashes real numbers to range between [-1,1].

- Pooling:
  - In CNNs, the outputs of activations are run through “pooling” to insulate calculations from lateral displacement
  - MaxPooling, the most common, is taking the maximum of the activated filter output

Optimizing the weights: Backpropagation I

- The objective is to calculate the dependency of the error in the result on an individual weight (or bias)
  - At the optimum, the derivative of the error between predicted and tagged results for each particular weight (and bias) should be 0

If $x_i = f_i(W_i x_{i-1} + b_i)$ is the output vector of the activations of layer i and X_L is the output of the last layer (prediction) and t is the tagged result and f' is the derivative of the activation wrt the weight

The error is $E=1/2\|x_L - t\|^2$

Chain rule yields:

$$\partial E/\partial W_L = (x_L - t) \circ f'_L(W_L x_{L-1})\, x^T_{L-1}$$

Optimizing the weights: Backpropagation II Chain rule..applied recursively

- Starting with $E=1/2\|x_L-t\|^2$ and $\partial E/\partial W_L = (x_L-t)\circ f'_L(W_L x_{L-1})\, x^T_{L-1}$

And differentiating the error wrt W_{L-1} (remember $x_{L-1} = f_{L-1}(W_{L-1}x_{L-2}+ b_{L-1})$)

$\partial E/\partial W_{L-1} = (\partial E/\partial x_L)(\partial x_L/\partial W_{L-1}) = (x_L-t)\circ f'_L(W_L x_{L-1})\, W^T_L\, f'_{L-1}(W_{L-1}x_{L-2})\, x^T_{L-2}$

This can be worked backwards creating a recursive calculation for each previous layer

Defining $\delta_L=(x_L-t)\circ f'_L(W_L x_{L-1})$

and $\delta_i=W^T_{i+1}\delta_{i+1}\circ f'_i(W_i x_{i-1})$

yields: $\partial E/\partial W_i=\delta_i x^T_{i-1}$

Giving us an expression for updating the weights

$$W_{i\ new}=W_i-\alpha_{Wi}\circ\partial E/\partial Wi$$

Where α_{Wi} is the learning rate

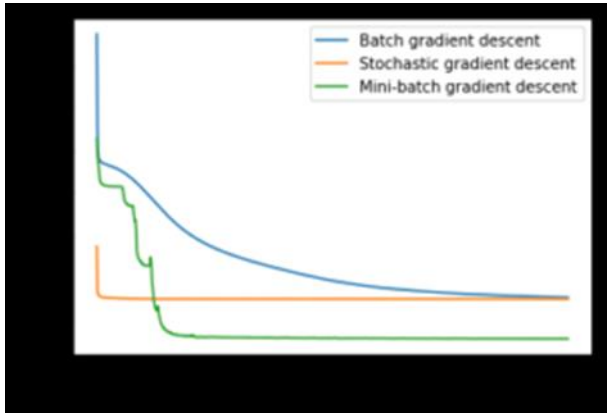
- usually set to a constant for all Wi
- Periodically lowered by some algorithm defined in the framework

Training vs Inference

- Training consists of running a tagged data set in the forward direction, calculating the errors in the results and backpropagating the errors to determine the changes in the weight values that the derivatives suggest. (Stochastic gradient descent/SGD, Adam optimization)
  - Different individual data (images, sentences) will produce different changes in the weights
  - An individual weight can be used in many activations, yielding many derivatives and suggested changes
- Inference is using an optimized model (optimal weights and bias') to make a prediction. A single forward pass through the network
  - Requires only a single forward pass/data element (image, sentence etc)

Training with striding backpropagation: Mini-Batches

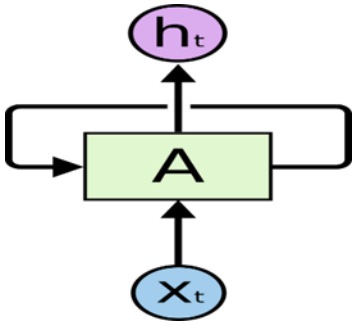
- A tagged data set (ex: 4.5 million sentences in German and English) can be
  - processed through training: forward calculation to produce predicted result
  - Followed by back propagation to evaluate indicated changes in weights
- One sentence at a time
- Or in batches of sentences (minibatch)
- Minibatches can stabilize the gradient descent of back propagation



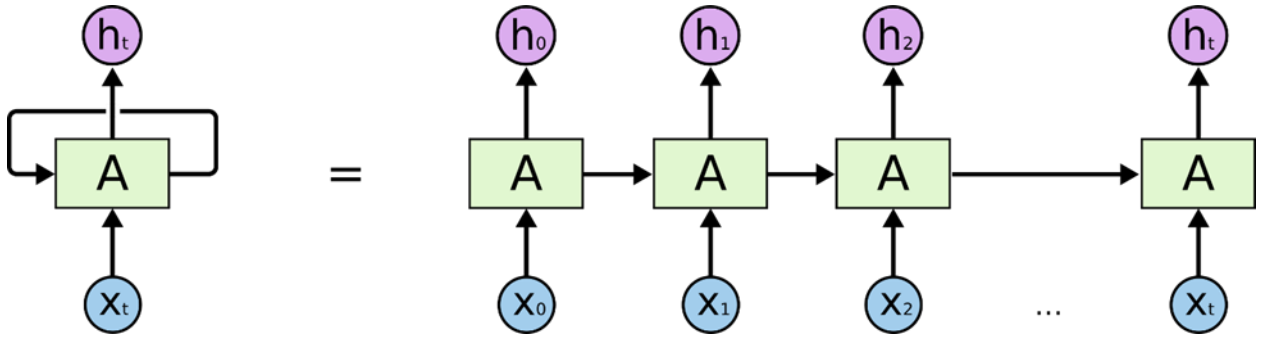
Processing Text and Language Representation

- Words must be represented as numbers to be processed
  - Standard technique is to represent them as a vector
  - A position on a multi dimensional unit sphere (personal interpretation)
  - Vocabulary[vocabulary_size, embedding_dimension]
- Relationships between words are expressed as relations between the vectors
  Vector(France) – Vector(Paris) + Vector(Berlin) ~ Vector(Germany)
- Correlated words should be near each other while randomly selected words should not
  - negative sampling or Noise Contrasted Estimation (NCE)
- Word2Vec and Glove are standard algorithms for creating/inititializing vocabulary

Recursive Neural Networks I



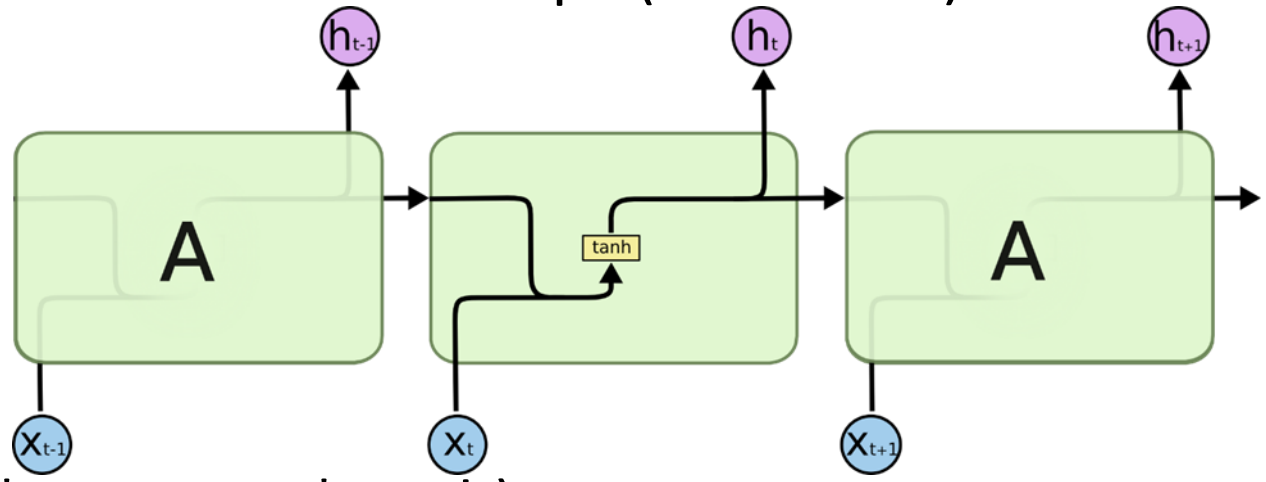
- Human processing of language is a recursive effort
  - The meaning of words depend on the syntax of usage
  - This creates a recursive loop over time
- A computer with finite resources unrolls the loop to a finite depth



  - layers can be stacked
  - Sentences can be processed in both directions of time to gain greater resolution (bi-directional)

Recursive Neural Networks II

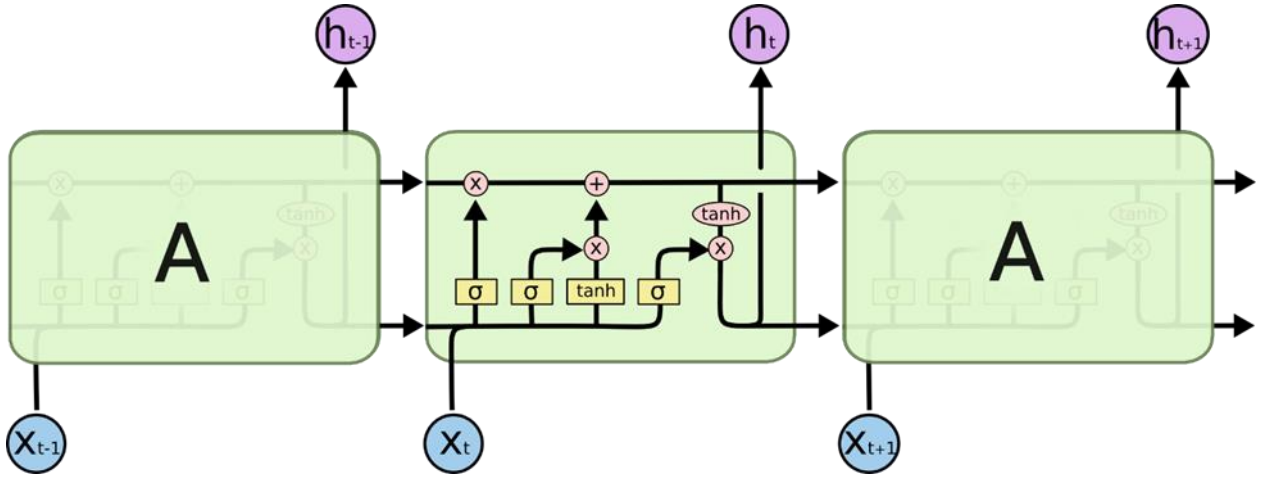
- What is evaluated at each time step? (Basic RNN)



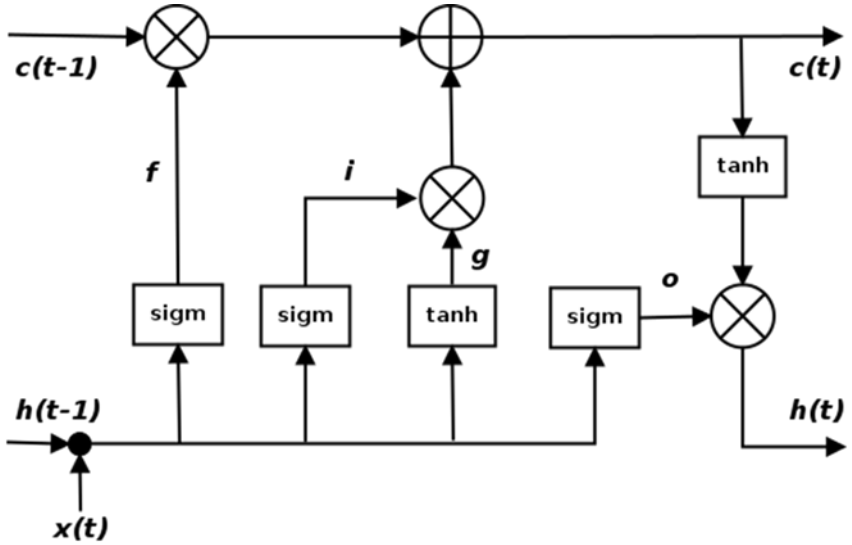
- $H_t = \tanh(W\_h * H_{t-1} + W\_x * X_t + b)$
  - H_t is the hidden state, a vector, processed recursively, passed to the next cell
  - X_t is the input for time step t
  - W_h, W_x are weight arrays and constant across time
  - b is a bias
  - W_h, W_x and b are optimized to process the time sequence correctly

Recursive Neural Networks III

- The simple RNN does not differentiate across the time range well
- Further there are divergences in the derivatives used for back propagation due to getting large numbers of products of derivatives (chain rule)
- Solution is the Long Short-Term Memory Cell (and variations)



Recursive Neural Networks IV LSTM

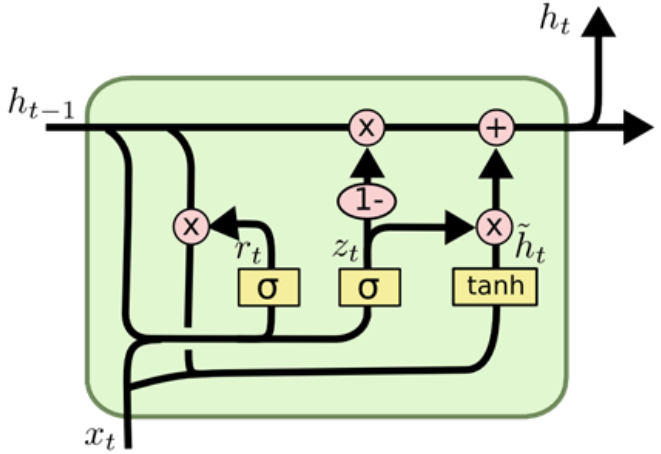


$$
\begin{aligned}
i &= \sigma(x_t U^i + s_{t-1} W^i) \\
f &= \sigma(x_t U^f + s_{t-1} W^f) \\
o &= \sigma(x_t U^o + s_{t-1} W^o) \\
g &= \ tanh(x_t U^g + s_{t-1} W^g) \\
c_t &= c_{t-1} \circ f + g \circ i \\
s_t &= \tanh(c_t) \circ o
\end{aligned}
$$

- There are two results passed over time (horizontally) C_t (the context/output) and h_t (hidden state, S_t in equations)
- h_t is referred to as the hidden state
- If the hidden state and inputs, (h,x) are vectors of the same size then there are 8 matrices of size h*h and a total of 16*h*h fp operations

Recursive Neural Networks V
GRU

- Gated Recurrent Unit (main variant of LSTM idea)



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

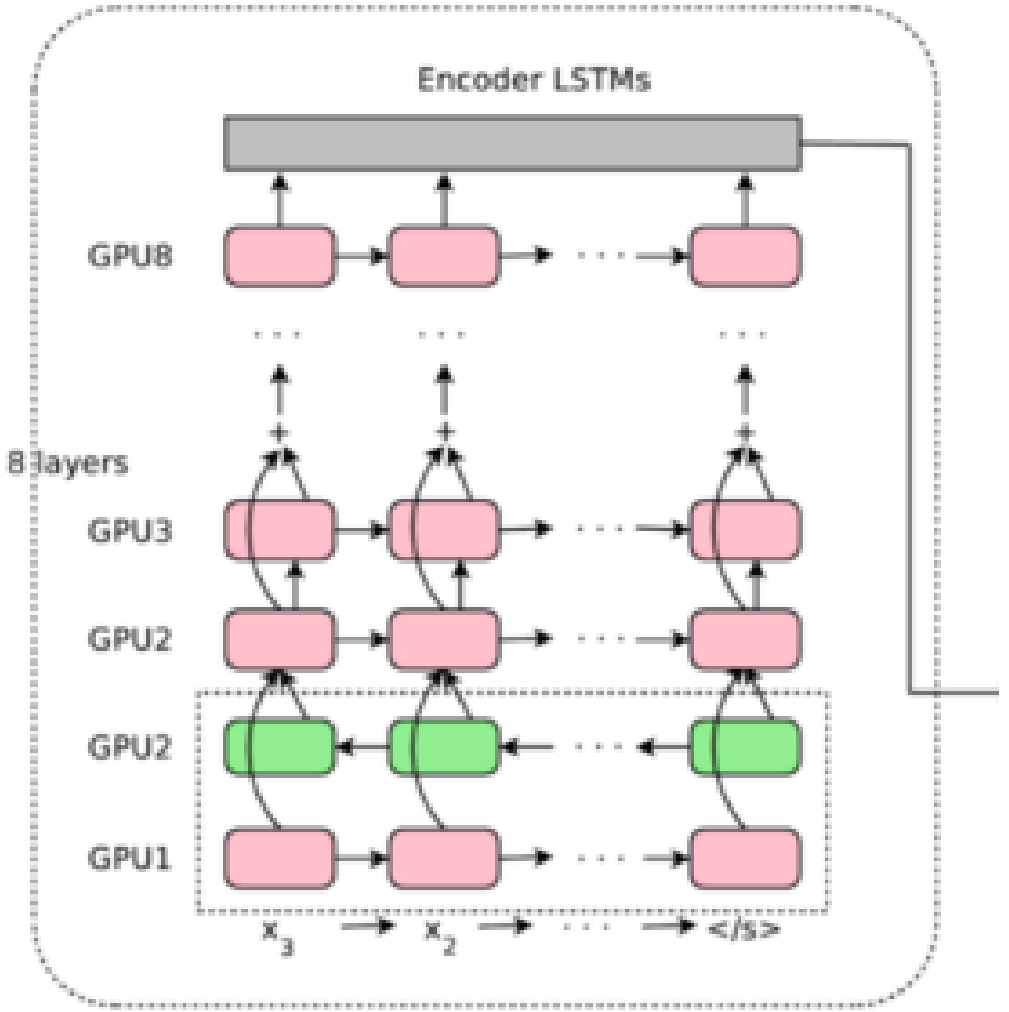
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

- 6 matrices (notation is a bit different)
- Note the pictures keep suggesting h_t can be passed up to another layer of processing

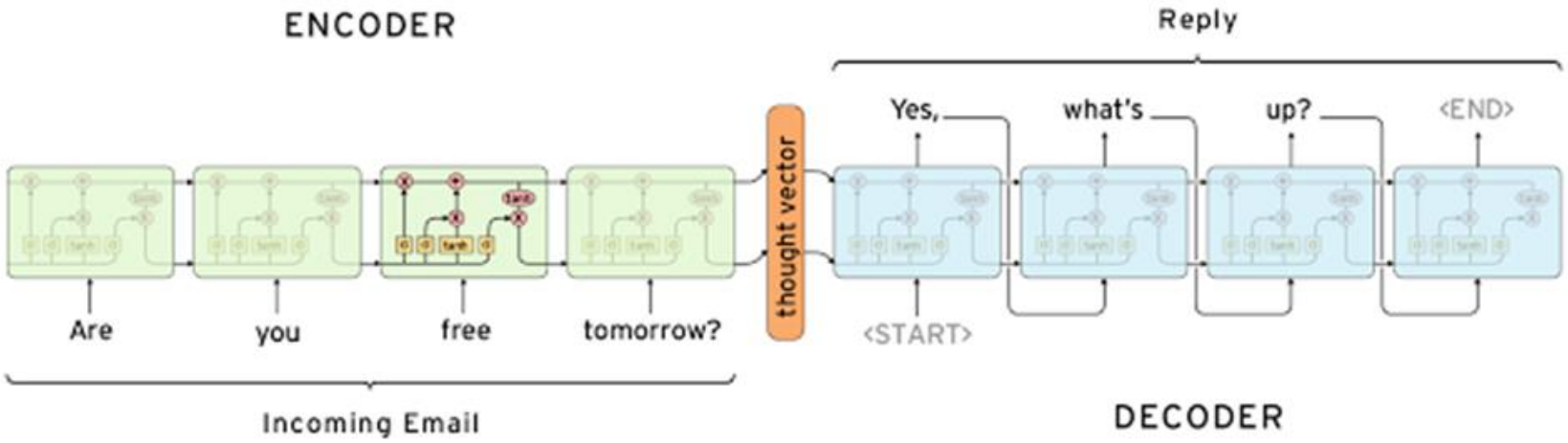
Recursive Neural Networks VI Stacking RNNs



Note residual feed forward to stabilize depth like resnet

Recursive Neural Networks VII Seq2Seq

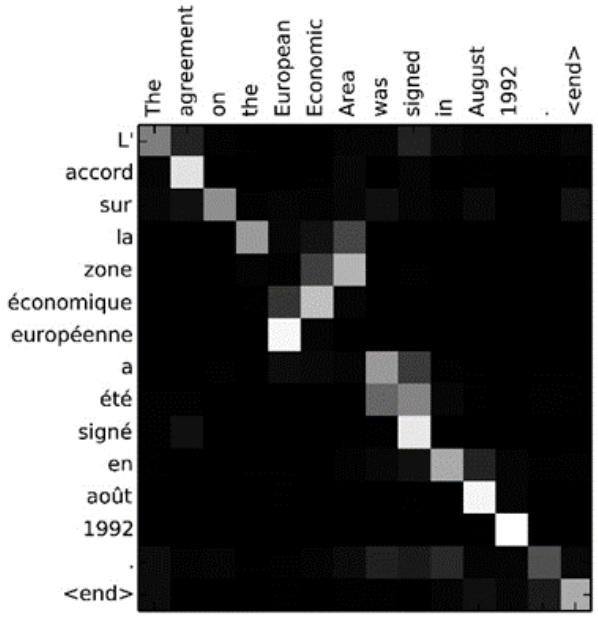
- If we can express the entire meaning of a sentence in the final output state vectors (also called the context), that can serve are the input to a second phase of text processing



- Encoder-Decoder/Sequence to Sequence RNN
- Use tagged input and output sentences (LOTS) and train all weights with backpropagation

Recursive Neural Networks VIII Attention

- When generating the next word during a translation of a sentence not all input sentence words should have equal importance
- Frequently (in related languages particularly) there can be strong correlations between words in two languages
  - Or else you couldn't write French-English dictionaries



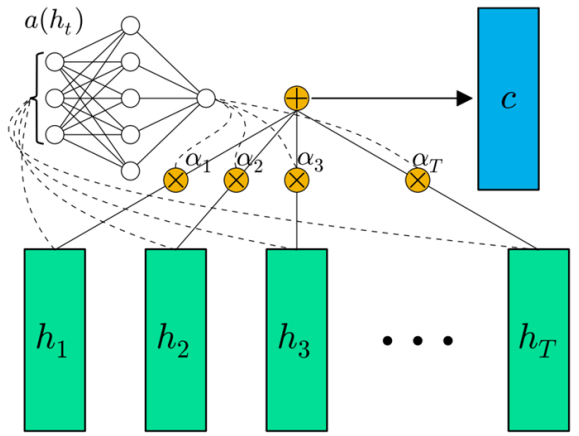
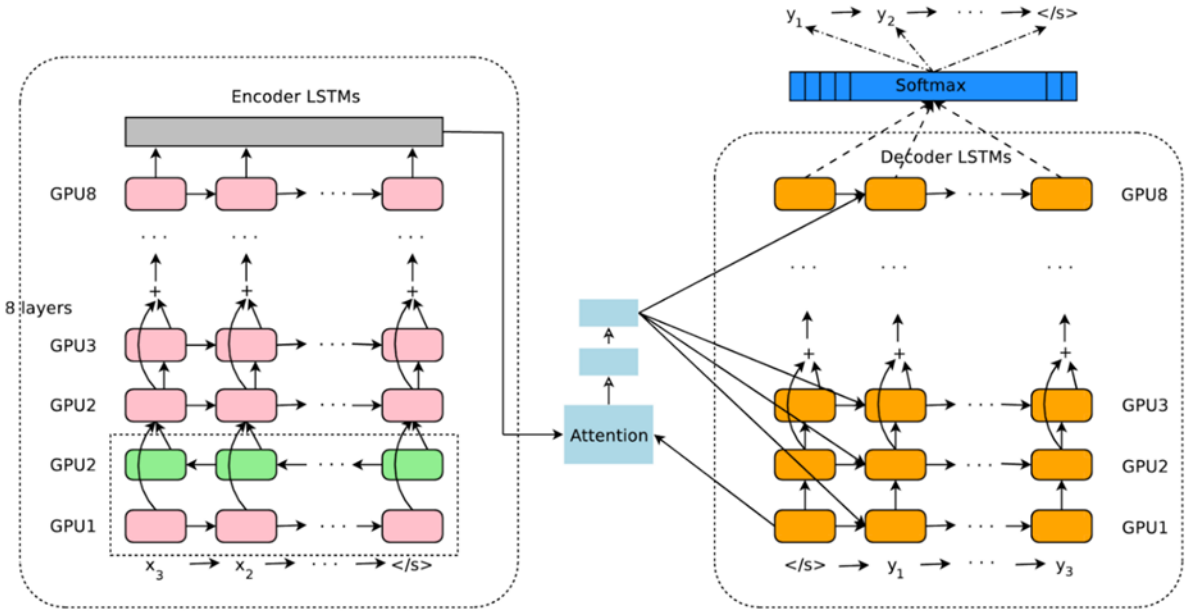


Figure 1: Schematic of our proposed "feed-forward" attention mechanism (cf. (Cho, 2015) Figure 1). Vectors in the hidden state sequence h_t are fed into the learnable function $a(h_t)$ to produce a probability vector α. The vector c is computed as a weighted average of h_t, with weighting given by α.

Recursive Neural Networks IX more Attention

- Neural Machine Translation
- Attention is an FCN with inputs of all encoder hidden states and decoder hidden states up to the word to be generated



$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \bar{\boldsymbol{h}}_s & \textit{dot} \\ \boldsymbol{h}_t^\top \boldsymbol{W_a} \bar{\boldsymbol{h}}_s & \textit{general} \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W_a}[\boldsymbol{h}_t; \bar{\boldsymbol{h}}_s]\right) & \textit{concat} \end{cases}$$

Recursive Neural Networks X inference Beam Search

- When we train the translator we know the correct output sentence
  - There is no ambiguity of the embedded representation of the output words
  - The error can be precisely calculated for backpropagation
- That is not the case when we use the translator for a purpose (inference)
  - The output embedded representations calculated will not exactly equal a value in the vocabulary
  - The objective is not to identify the highest probability word but to construct the highest probability sentence

Recursive Neural Networks XI
more Beam Search

- Solution is to keep a small collection of the highest probability sets of words and incrementally add the best words for each set/potential output sentence
- Finally choosing the highest probability sentence
- This means running the entire seq2seq + attention network multiple times for each version of the output sentence the beam search is tracking..ie the beam width

Transformer based Translation I

- Again a sequence to sequence algorithm of an encoder and decoder
- Built up from repeated layers (typically 6) of a basic structure
- A residual forward feed stabilizes the depth (as in resnet)

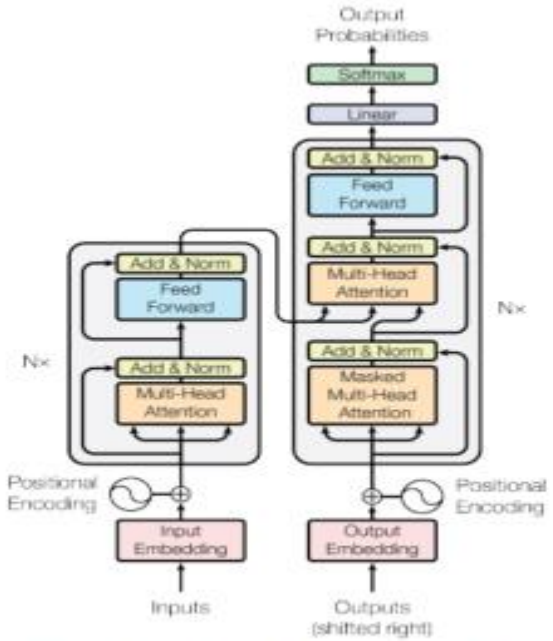


Figure 1: The Transformer - model architecture.

Transformer based Translation II

- Built of modules consisting of a multi head attention (8 or 16 heads) and a residual feed forward (next slide)

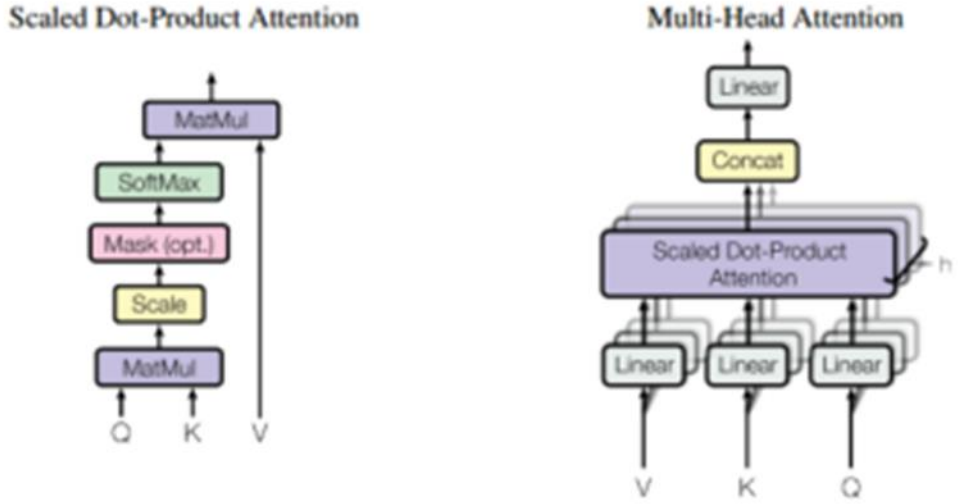


Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$$
$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Transformer based Translation III

- What are Query, key and value?
- Self attention (Encoder and first layer of decoder)
- All tokens/words in the sentence processed in parallel
- Self attention
  - Query is the matrix of input tokens
  - The Keys are matrices of all the other tokens
  - The Value is derived from the output of the previous layer
- Decoder attention
  - Query is the matrix of input tokens
  - The Keys and Values come from the last stage of the encoder (as in RNN)

Transformer based Translation IV

- Next component: Feed forward network (FFN)
- Fully connected network
- Input size d_model (512 or 1024)
- Hidden layers (usually 2048 or 4096)
- Output size d_model
- Thus d_model X Hidden + Hidden X d_model FP ops
- And if Hidden size scales with d_model, number of weights scales with d_model^2

Transformer based Translation V

- **Positional encoding**
- Unlike RNNs, a transformer analyzes all the tokens in the input sentence in parallel
- A position dependent value is added to the token embedding to solve this
- The value is made of a superposition of d_model, sines and cosines of the position
- 0 <= i < d_model/2 $PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

Transformer based Translation VI

- Encoder
- typically stages 2+3 are repeated as Nx (6) layers

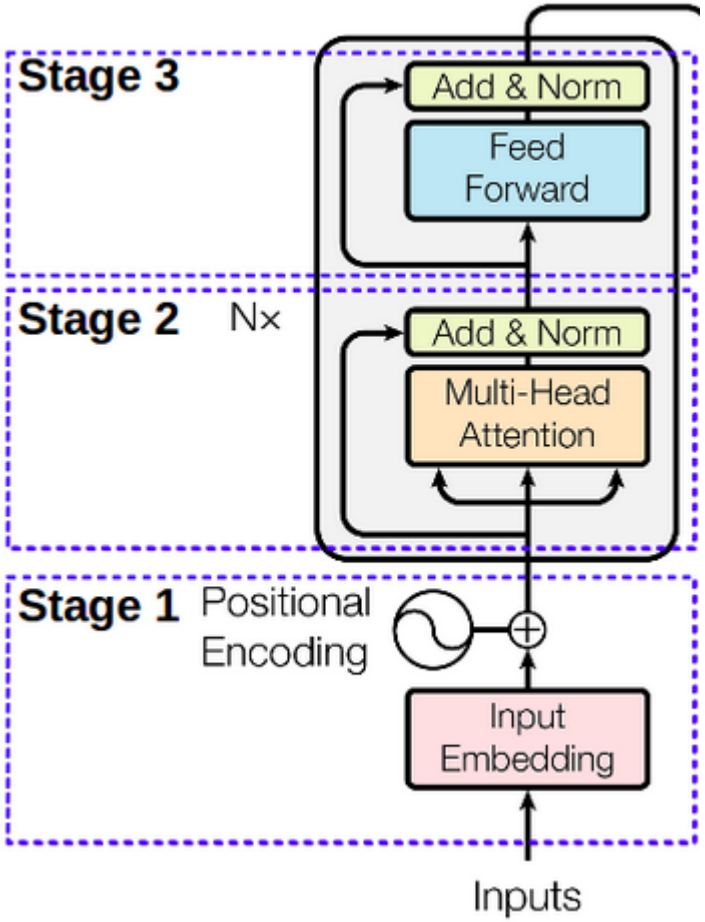
Stage1_out = Embedding512 + TokenPositionEncoding512

Stage2_out = layer_normalization(

multihead_attention(Stage1_out) + Stage1_out)

Stage3_out = layer_normalization(FFN(Stage2_out) + Stage2_out)

out_enc = Stage3_out



Transformer based Translation VII

- Decoder
- typically stages 2+3+4 are repeated as Nx (6) layers

Stage1_out = OutputEmbedding512 + TokenPositionEncoding512

Stage2_Mask = masked_multihead_attention(Stage1_out)
+ Stage1_out

Stage2_Norm = layer_normalization(Stage2_Mask)

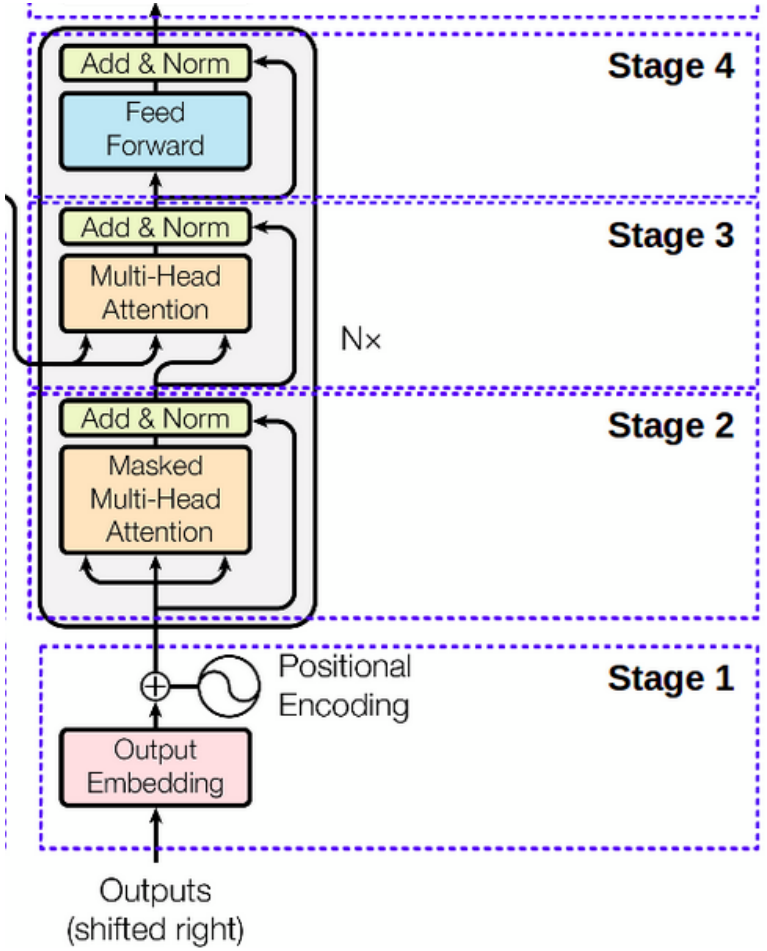
Stage3_Multi = multihead_attention(Stage2_Norm + out_enc)
+ Stage2_Norm

Stage3_Norm = layer_normalization(Stage3_Multi)

Stage4_FNN = FNN(Stage3_Norm)

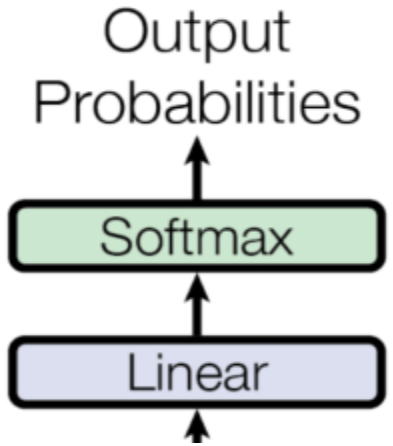
Stage4_Norm = layer_normalization(Stage3_FNN + Stage3_Norm)

out_dec = Stage4_Norm



Transformer based Translation VIII
Final Classifier

- Output is a final matrix vector multiply of the context with the vocabulary, then run through a softmax



- The loss is computed with a vector of the probabilities for each word in the vocabulary.
- For inference these are then fed as inputs to the decoder recursively

Transformer based Translation VII

- During training the output sentence is known
- This can be fed as input to the decoder
- Thus the decoder weights can be trained with a single pass once the encoder output is known.

Transformer based Translation VIII

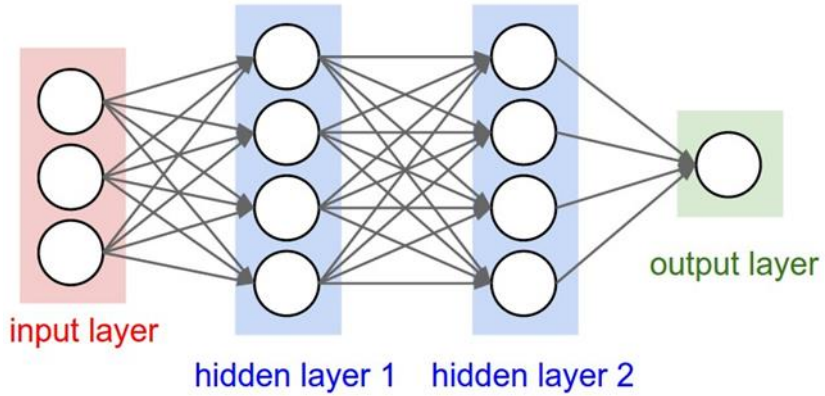
- References for Transformer based translation
- https://arxiv.org/abs/1706.03762
- http://nlp.seas.harvard.edu/2018/04/03/attention.html
- https://arxiv.org/abs/1804.00247
- https://mchromiak.github.io/articles/2017/Sep/12/Transformer-Attention-is-all-you-need/

Performance and resource usage Characterization of DNNs

- Understanding how these algorithms use the compute resources is critical to optimizing future hardware designs and deployments
- FCNs, CNNs and RNNs are very different in their FP compute intensity and memory reuse, capacity and bandwidth utilization.
- From the graphical representation we can ~compute the FP operations and number of weights used in a forward pass (inference)

Estimating FCN properties

- FCNs are exactly what they are called fully connected
- Each input and output combination requires its own weight
- The FP operation count is twice the number of weights (multiply + add)



- FP ops and weight counts can be very high
- Vector matrix multiply, input vector becomes a matrix with batch > 1
- Activations = weight count times batch size, driving memory consumption during training

Estimating CNN properties

- FP operation and weight counts are dominated by convolution layers and FCN layers

Consider square inputs and filters

- The 2-D filter (size K) is swept over the input image area (size W)
  - The output dimension = 1 + (W – K +2P)/S
    - P is padding added to make things work, S is the stride
  - For each input layer there is a different set of weights
  - Each filter has k*k*Input_depth weights
- A full convolution layer will have some number of filters = number of output layers
- Total number of weights = k*k*Input_depth*Output_depth

Estimating CNN properties II

- Estimating the FP operations for a convolution can be figured out as follows:
  - Each output cell is the sum of k^2 weight*input terms
  - Thus each output layer requires $2*k*k*O_w*O_w$ *input_layers FP operations
  - For a total of $2*k*K*O_w*O_w*$input_layers*output_layers FP operations
- This result is not always correct as there are optimizations (Ex: Winograd) that can be applied

Estimating CNN properties III

- Alexnet coded for Tensorflow

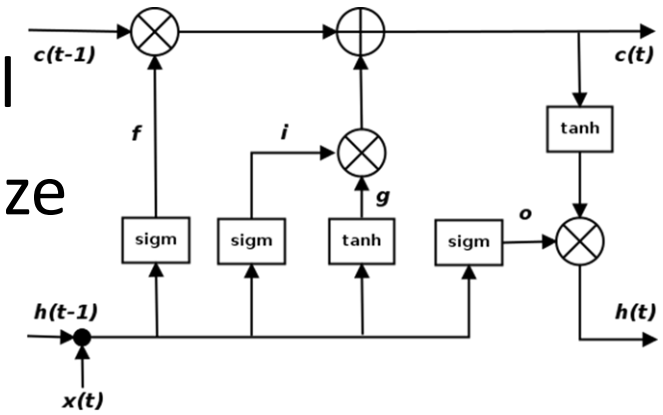
```
def __init__(self):
    super(AlexnetModel, self).__init__('alexnet', 224 + 3, 512, 0.005)
 def add_inference(self, cnn):
    # Note: VALID requires padding the images by 3 in width and height
    cnn.conv(64, 11, 11, 4, 4, 'VALID')
    cnn.mpool(3, 3, 2, 2)
    cnn.conv(192, 5, 5)
    cnn.mpool(3, 3, 2, 2)
    cnn.conv(384, 3, 3)
    cnn.conv(384, 3, 3)
    cnn.conv(256, 3, 3)
    cnn.mpool(3, 3, 2, 2)
    cnn.reshape([-1, 256 * 6 * 6])
    cnn.affine(4096)
    cnn.dropout()
    cnn.affine(4096)
    cnn.dropout()
```

Alexnet layer (Soumith/convnet)	expected	measured	ratio
conv (images, 3, 64, 11, 11, 4, 4, 'VALID')	140553600	143717440	1.02251
+ mpool(conv1, 3, 3, 2, 2)	small	46656	
+ conv (pool1, 64, 192, 5, 5, 1, 1, 'SAME')	447897600	472294080	1.054469
+ mpool(conv2, 3, 3, 2, 2)	small	32448	
+ conv (pool2, 192, 384, 3, 3, 1, 1, 'SAME')	224280576	162168191	0.723059
+ conv (conv3, 384, 256, 3, 3, 1, 1, 'SAME')	299040768	215558401	0.720833
+ conv (conv4, 256, 256, 3, 3, 1, 1, 'SAME')	199360512	143927547	0.721946
+ mpool(conv5, 3, 3, 2, 2)	small	9216	
+ tf.reshape(pool5, [-1, 256 * 6 * 6])	small	0	
+ affine(resh1, 256 * 6 * 6, 4096)	75497472	76988416	1.019748
+ affine(affn1, 4096, 4096)	33554432	34226176	1.02002
+ affine(affn2, 4096, 1000)	8192000	8522048	1.040289

- The 3X3 convolutions here were done with Winograd optimized functions
- Measurements were done with NVProf which uses binary instrumentation (165X slow down)

Estimating RNN properties I

- Consider an LSTM cell
- Assume the hidden size equals the input or embedded size



$$i = \sigma(x_t U^i + s_{t-1} W^i)$$
$$f = \sigma(x_t U^f + s_{t-1} W^f)$$
$$o = \sigma(x_t U^o + s_{t-1} W^o)$$
$$g = tanh(x_t U^g + s_{t-1} W^g)$$
$$c_t = c_{t-1} \circ f + g \circ i$$
$$s_t = \tanh(c_t) \circ o$$

- Then the U and W matrices are square
- FP operations/cell ~ 2*8*hidden_size*hidden_size
- As the weights in all the cells of a layer are the same, the number of weights ~ 8*hidden_size*hidden_size/layer

Estimating RNN properties II

- Single layer LSTM with variable length
- Batch size > 1 changes vector*matrix to matrix* matrix

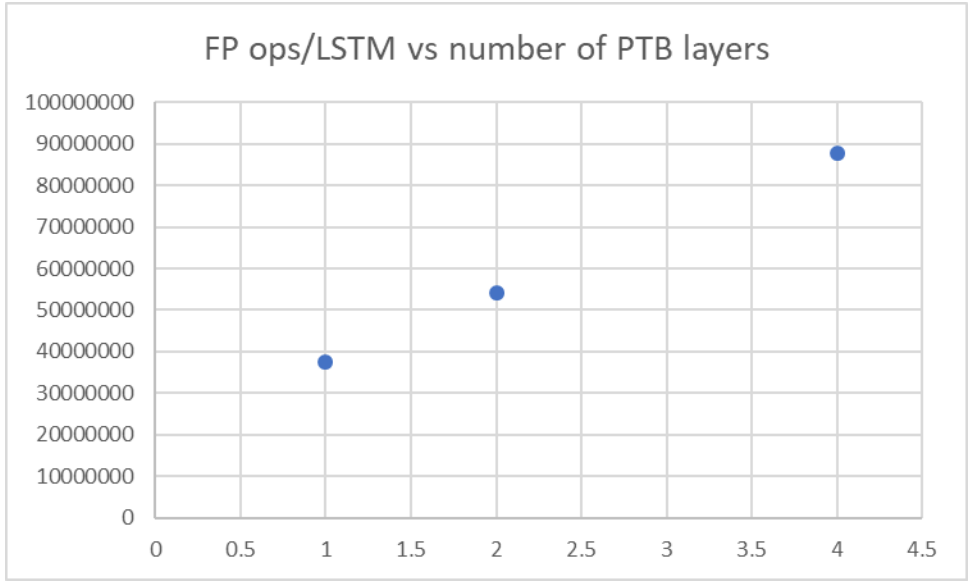
Num Iters	Batch Size	Hidden Size	Seq Len	FP Ops	FP_ops/LSTM	expected FP Ops	Ratio
10	1	64	8	7.59E+06	94912	65536	1.45
10	1	64	16	1.52E+07	94912	65536	1.45
10	1	512	16	6.92E+08	4322810	4194304	1.03
10	1	512	32	1.38E+09	4322810	4194304	1.03
10	1	1024	8	1.38E+09	17198080	16777216	1.03
10	1	1024	16	2.75E+09	17198074	16777216	1.03
10	1	1024	32	5.50E+09	17198080	16777216	1.03
10	128	512	32	1.74E+11	4236719	4194304	1.01
10	128	1024	8	1.73E+11	16861522	16777216	1.01
10	128	1024	16	3.45E+11	16860854	16777216	1
10	128	1024	32	6.91E+11	16860504	16777216	1
10	256	512	16	1.74E+11	4236014	4194304	1.01
10	256	512	32	3.47E+11	4236077	4194304	1.01
10	256	1024	8	3.45E+11	16858888	16777216	1
10	256	1024	16	6.91E+11	16858061	16777216	1
10	256	1024	32	1.38E+12	16858089	16777216	1

Estimating RNN properties II

- Penn TreeBank (PTB) test is a simple benchmark predicting next word
- It can have a variable number of layers, hidden size and time steps
  - Set hidden size=1024, time steps=32, batch size=128 and vary layer count

	1 Layer	2 Layers	4 Layers
total fp_ops	2.64E+12	3.82E+12	6.16E+12
Sgemm fp_ops	2.61E+12	3.78E+12	6.12E+12

	sgemm fp ops	sgemm fp opps/LSTM	2 point slope	ratio slope/expected value
1 layer	2.61E+12	3.75E+07		
2 layers	3.78E+12	5.43E+07	16781312	1.000244
4 layers	6.12E+12	8.78E+07	16781312	1.000244



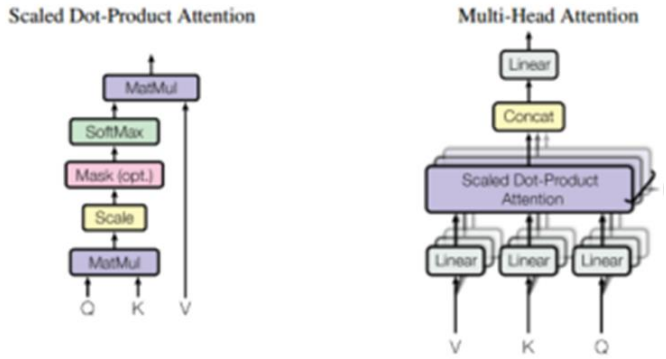
- There is a large non zero baseline in this Tensorflow version

Estimating Transformer Properties

- Built of modules consisting of a multi head attention (8 or 16 heads)

and a residual feed forward (next slide)

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$



$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$$
$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

Dim_Queries = Dim_values =(Dim_model = Dim_embed)/Num_head
Dim_keys is a matrix on size sentence length (L) vs embedding size/Num_head
The Weight matrices are embedding size/Num_head X embedding size/Num_head
Thus Multi head attention has 2 *L^2 * Dim_model/Num_head FP ops
The weighting requires L*3*2*(Dim_model/Num_head)^2 FP ops

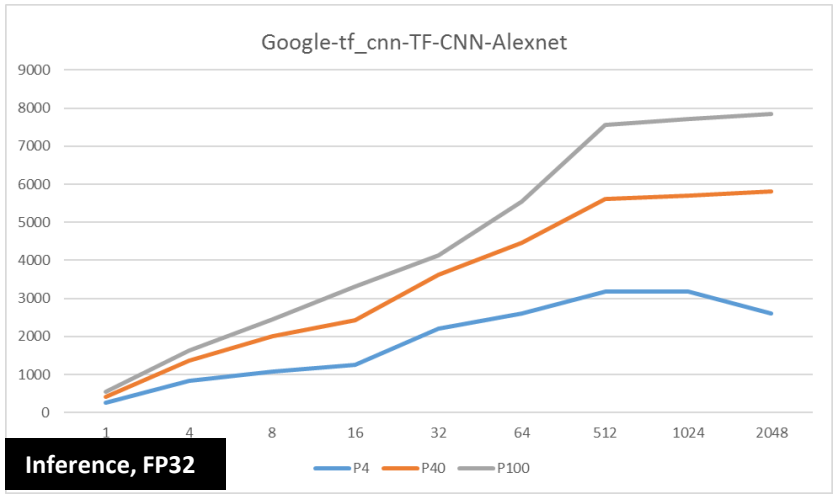
Estimating Transformer Properties II

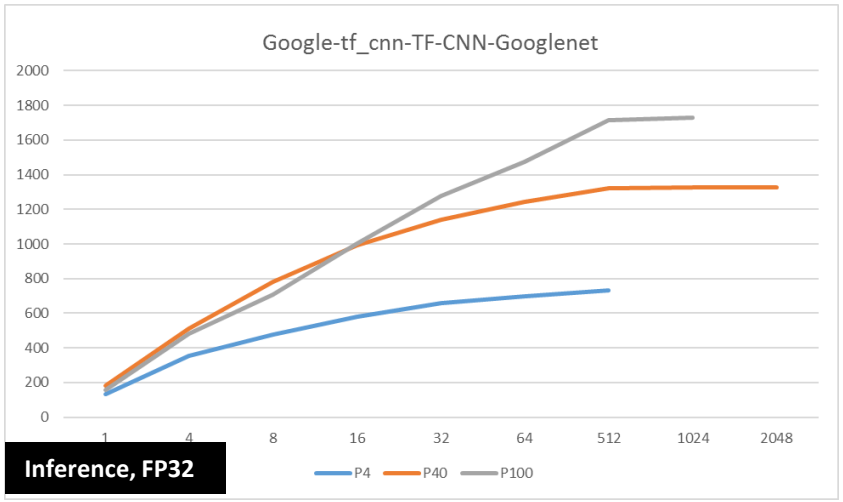
- Built of modules consisting of a multi head attention (8 or 16 heads)
- and a residual feed forward
- The feed forward block is a FCN with 2 hidden layers usually 4X the embedding size and a final output size of the embedding size
- FP Ops = $L*2 * 4 * dim\_model^2$
- With Nx = 6:
  Total FP ops ~ $6 * ($
  $3*2* (L^2*+L*dim\_model/num\_head)*dim\_model$
  $+ L* 8\ dim\_model^2)$
- More later

Viewing DNN performance from a hardware perspective

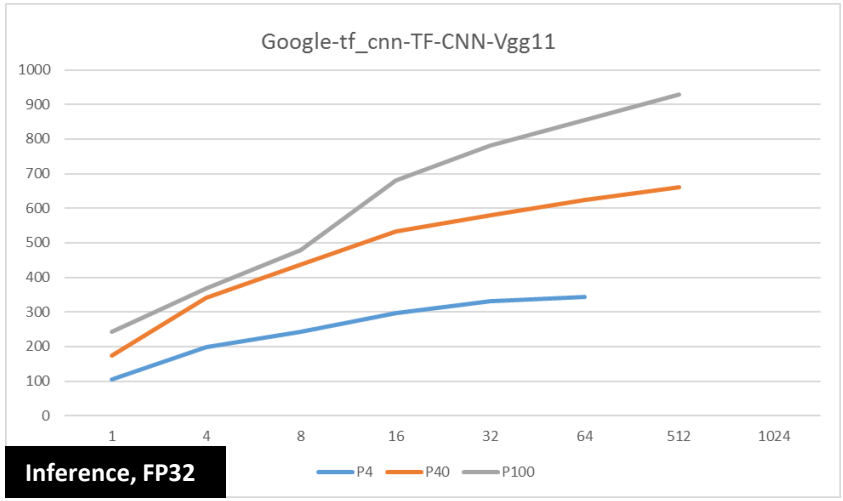
- DNN performance must be separated by training and inference
- In each case there are many run configuration options (hyperparameters)
- Both are effected by minbatch size: number of images or sentences processed together
  - Creates larger matrices for GEMM functions
  - Greatly effects speed
- Training has a many additional options (learning rate, dropout, gradient evaluation, synchronization strategy, etc)

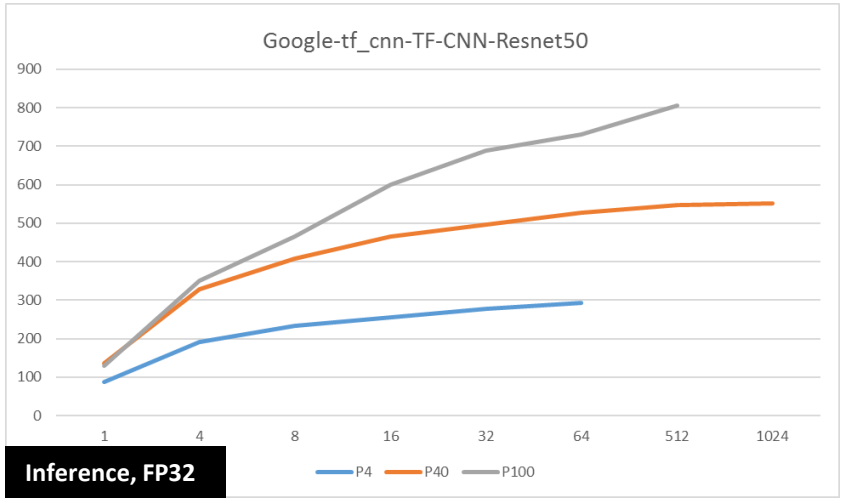
Google TF CNNs – Inference (Images/sec vs. Batch Size)

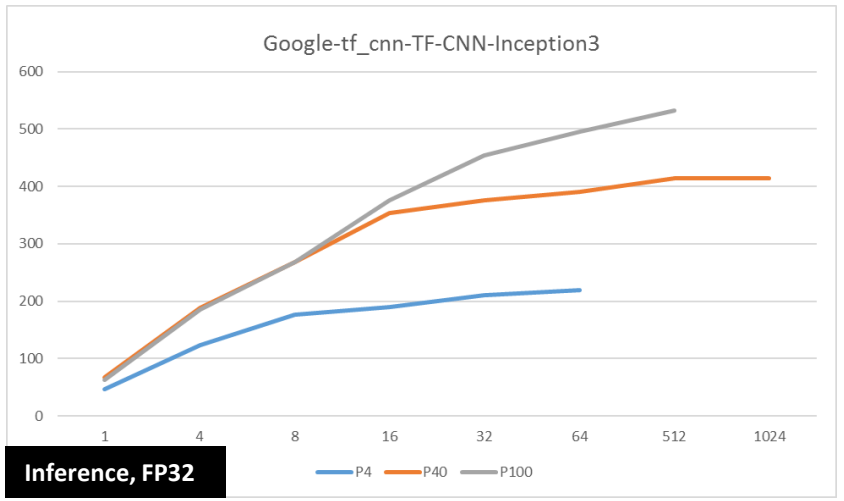




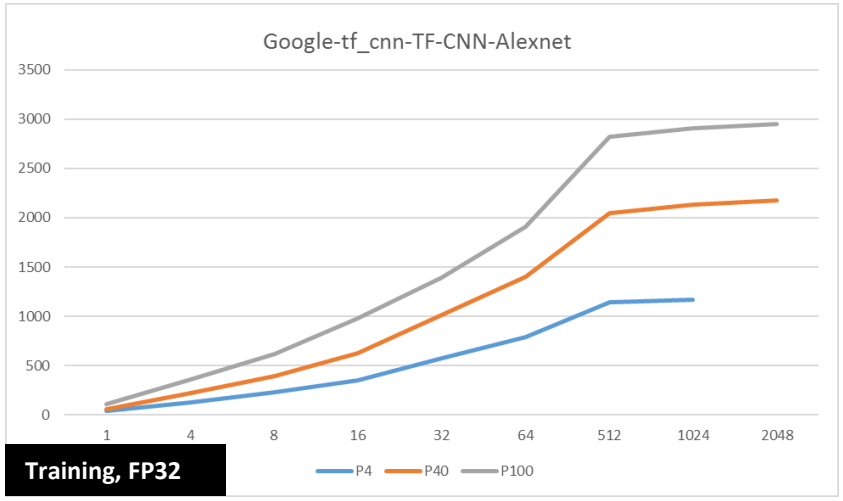
Perf flattens out > batch size 512
P40 very competitive with P100 at smaller batch sizes
P100 does a lot better than P40 at larger batch sizes
P4 is hampered by lower memory capacity (8 GB vs 16 and 24 GB for P40 and P100)
P4 performance pretty good for it's price at low batch sizes (it was designed for inference where small batches are desirable)

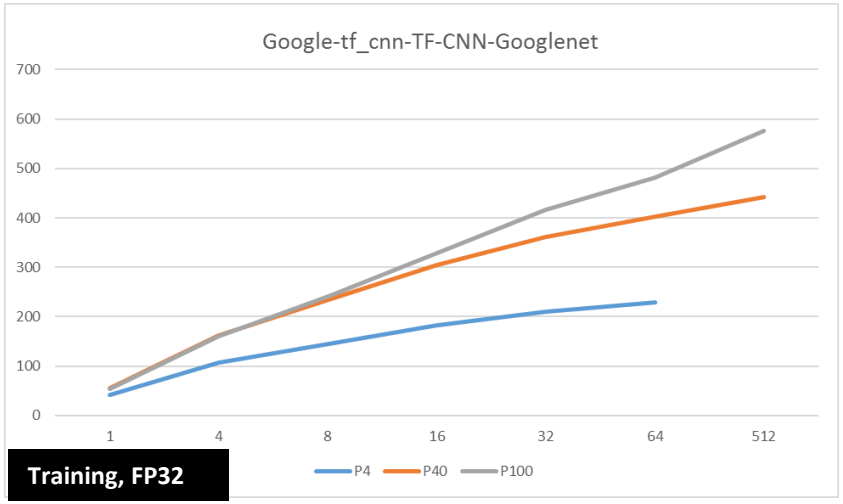






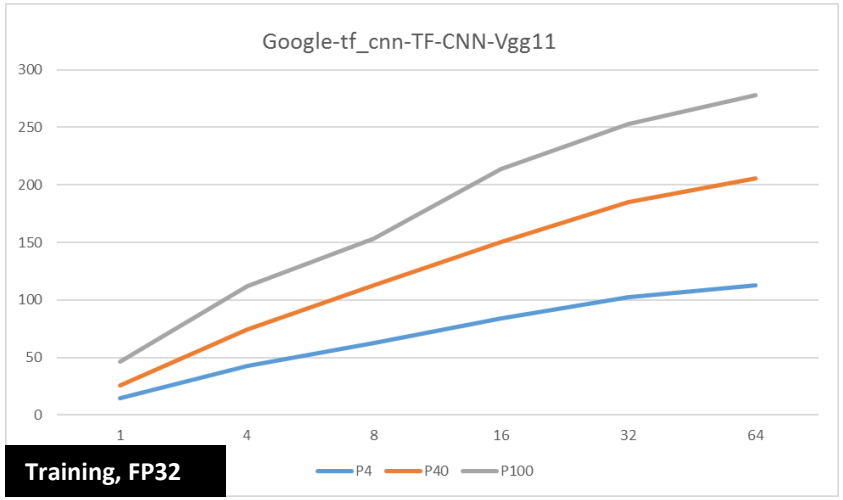
Google TF CNNs – Training (Images/sec vs. Batch Size)

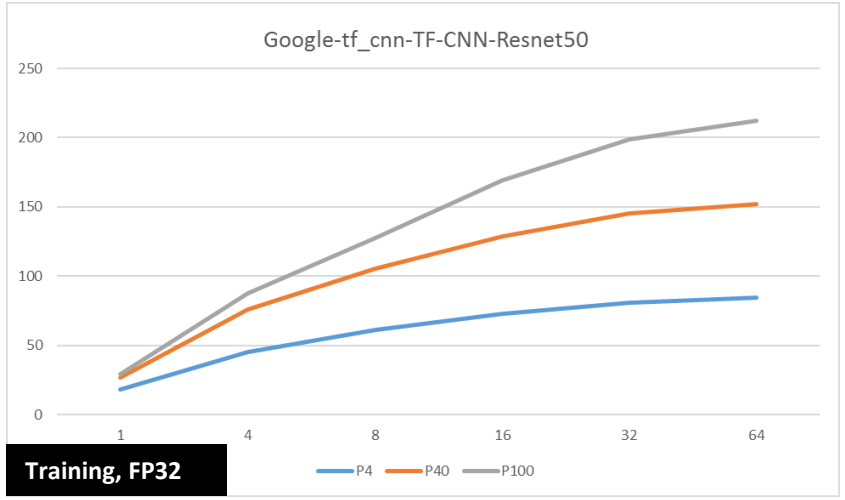


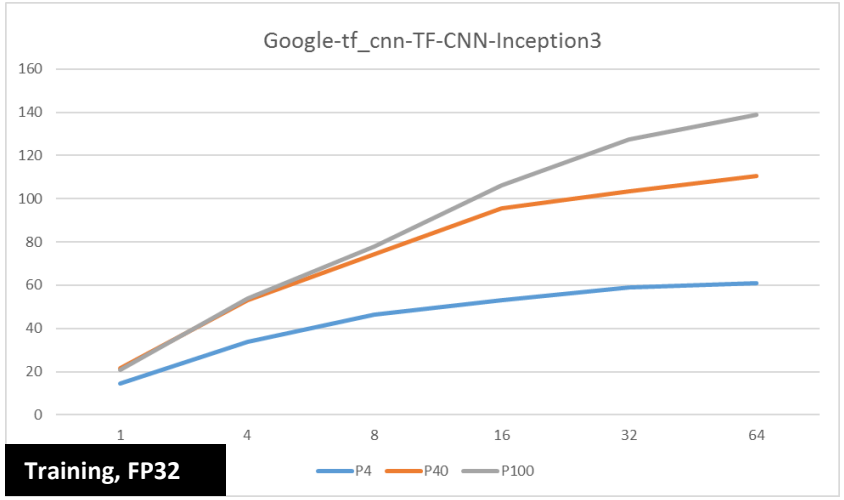


Training requires best throughput without latency SLAs (batch sizes can be large)
P100 is offers significantly higher max throughput than P40

Note: inference ~ 3X faster throughput



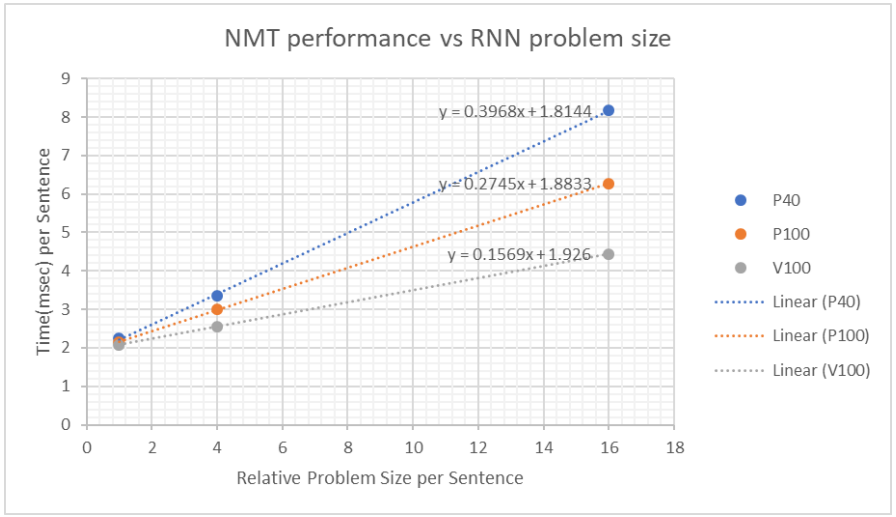


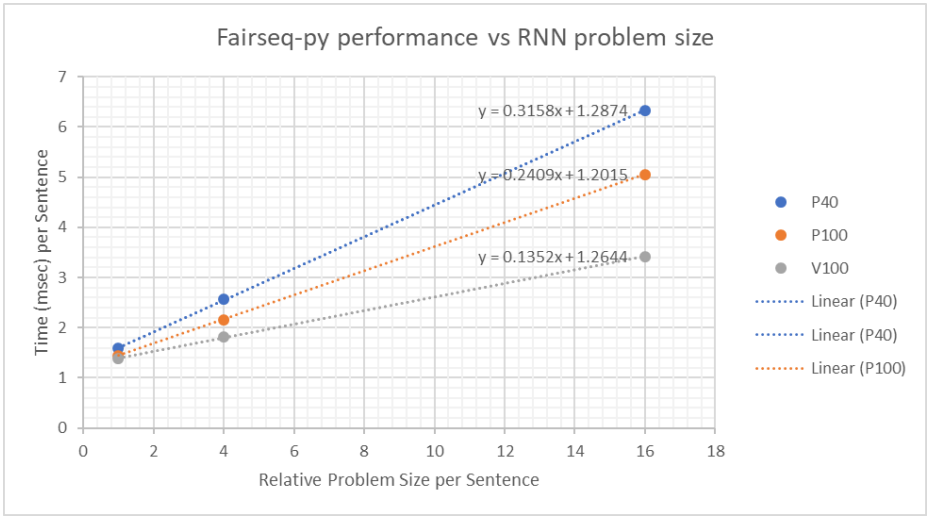


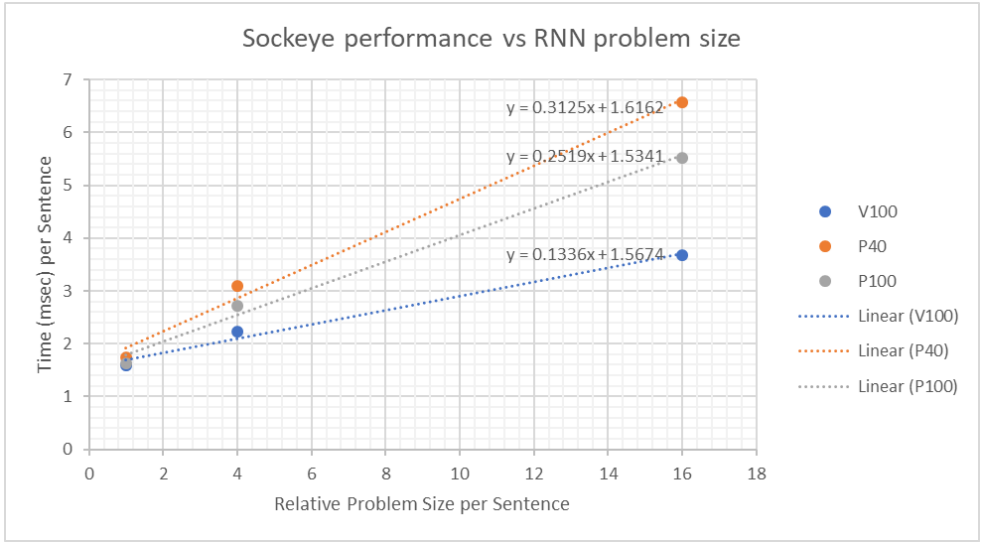
DNN timing vs model size

- One can scale the model size in many cases
  - In RNNs increasing equal values for hidden size and embedding size causes a quadratic increase in FP operations
  - Increasing embedding, model_size, head count and FF hidden size causes a quadratic increase in FP operations in Transformers
  - Increasing the input image size in CNNs causes a quadratic increase in FP operations in image processing
- Thus one tends to see quadratic, linear and constant terms in timing vs model size
- The Quadratic terms do not always dominate
  - Limiting the effectiveness of designs like systolic arrays.

Training speed vs hidden_size^2 (relative units) large baseline independent of GPU capacity







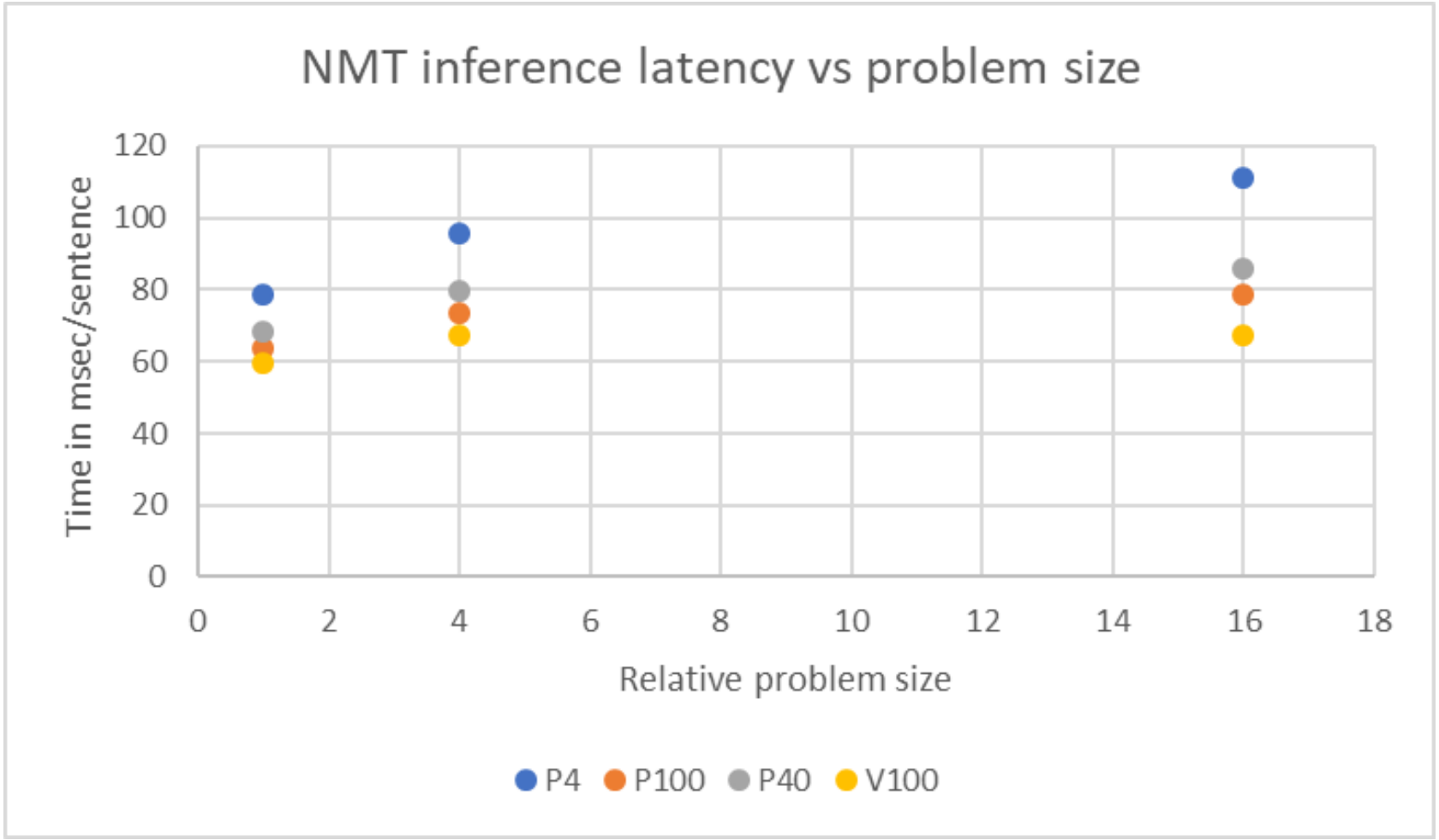
HW Performance gain/GPU as a function of framework

- Relative HW performance, for each framework, can be measured using approximately equivalent code bases
  - The applications do not need to be exactly the same
  - Just reasonably similar
    - Meaning ~equivalent linear algebraic coverage

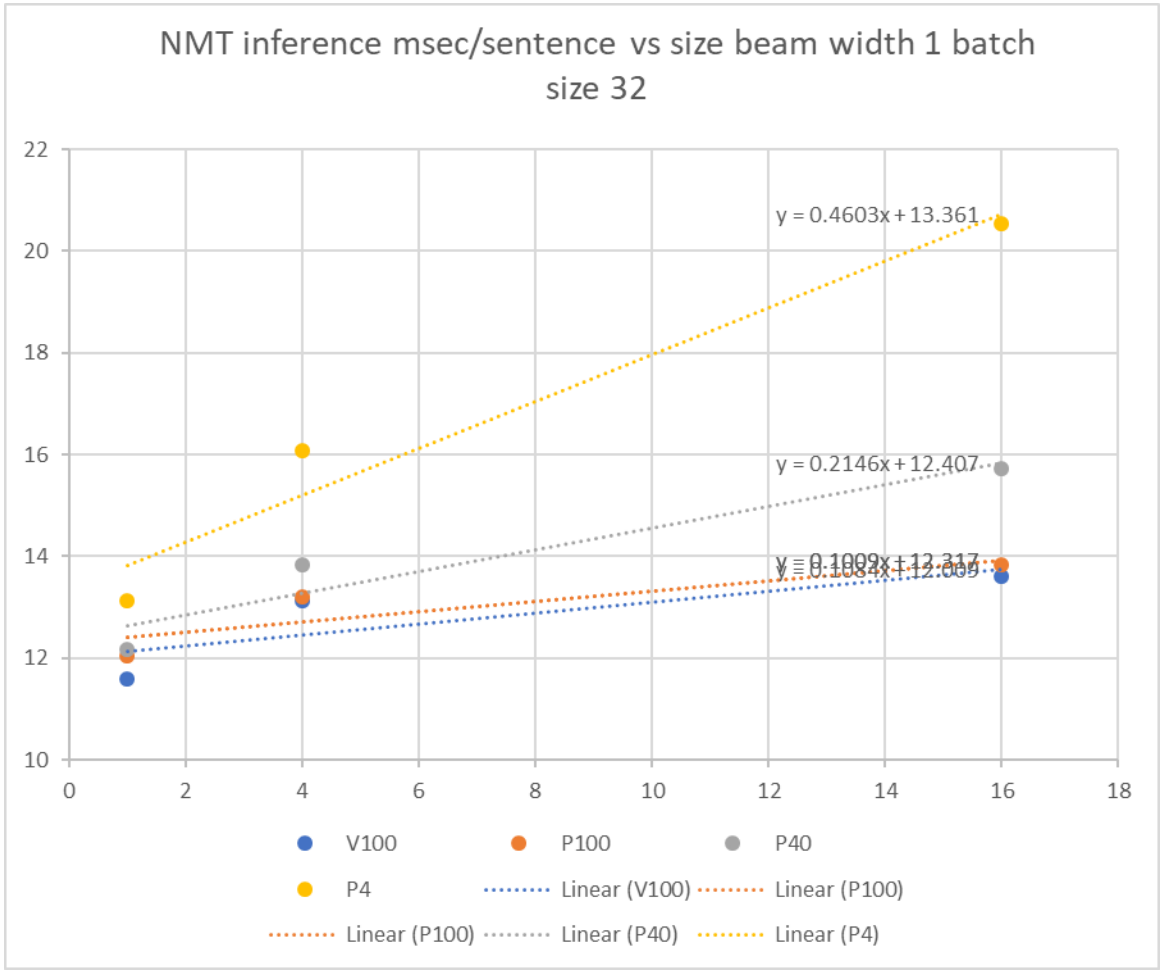
	P40	P100	V100
fairseq-py	4627.36	5797.84	8564.68
ratio	1.00	1.25	1.85
Sockeye	152.18	180.92	271.88
ratio	1.00	1.19	1.79
NMT	6.88	8.98	12.74
ratio	1.00	1.31	1.85

Perf metric units vary with the code base. Only the perf ratios are relevant

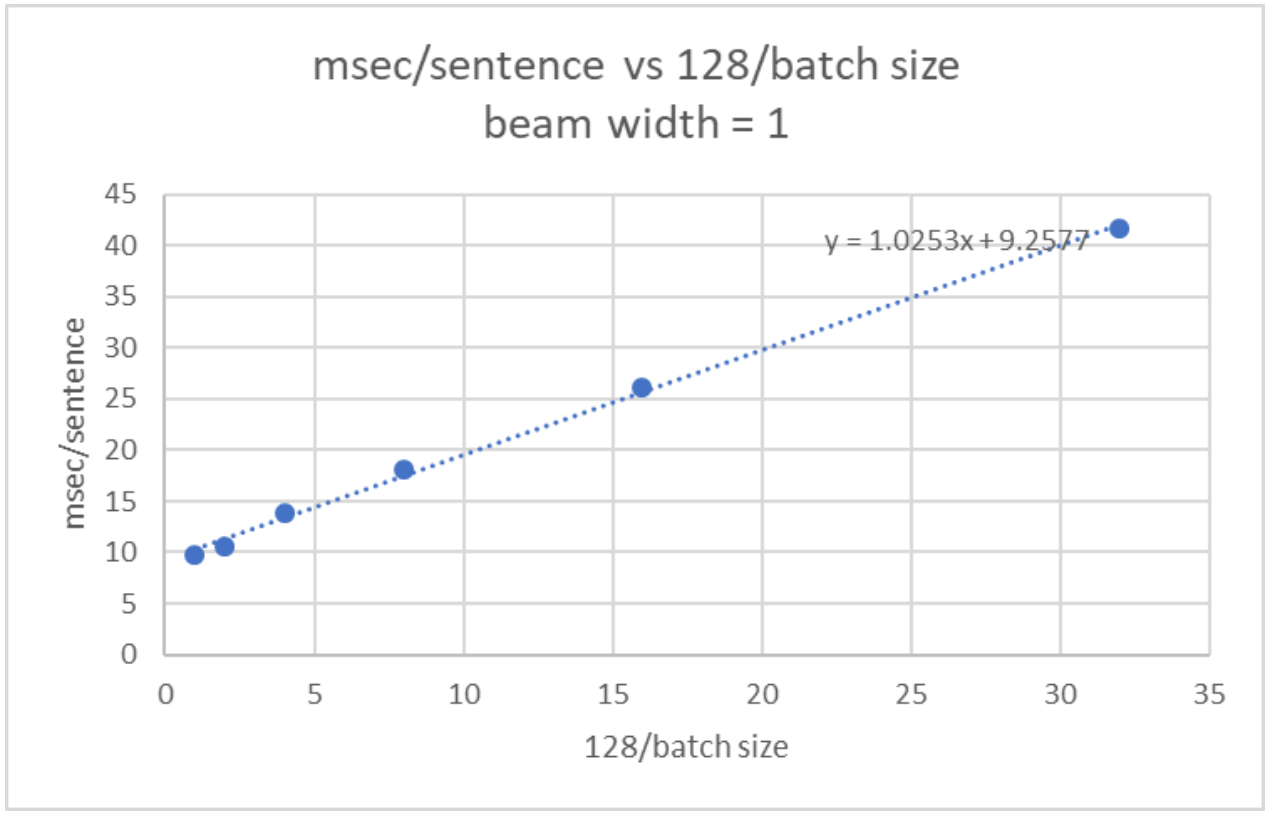
Inference is NOT 3X faster
It is 35X slower!



Reduce Beam width to 1 and lose accuracy gain 5-6X in speed



With Beam width 1 performance is limited by batch size



GPU training utilization of large translators is high
Smaller problems have lower utilization

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

Transformer Parameter Analysis

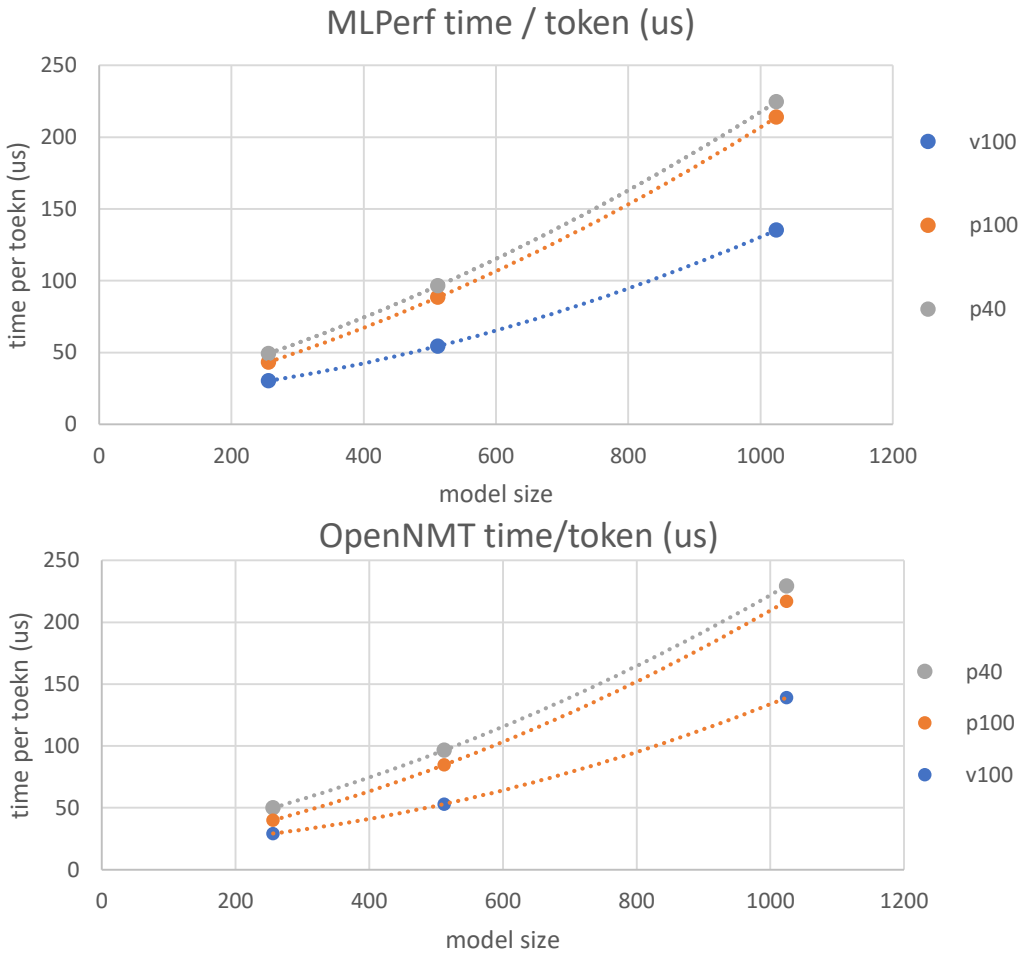
- Inclusion/exclusion of bias terms with certain operations varies between implementations
- Depending on translation task, weight-tying might not be appropriate
  - For German-English tasks, tied-embeddings+classifier actually improves BLEU, but depending on compatibility of languages this will not always be the case
- An example: base-size transformer for German-English translation (WMT, newstest2016)
- Similar vocabulary sizes (32000-36000), but variations depending on pre-processing and implementation-specific choices
  - Force shared vocab at pre-process or at training
  - Sorting + Forming minibatches
- Implications for inter-framework comparisons

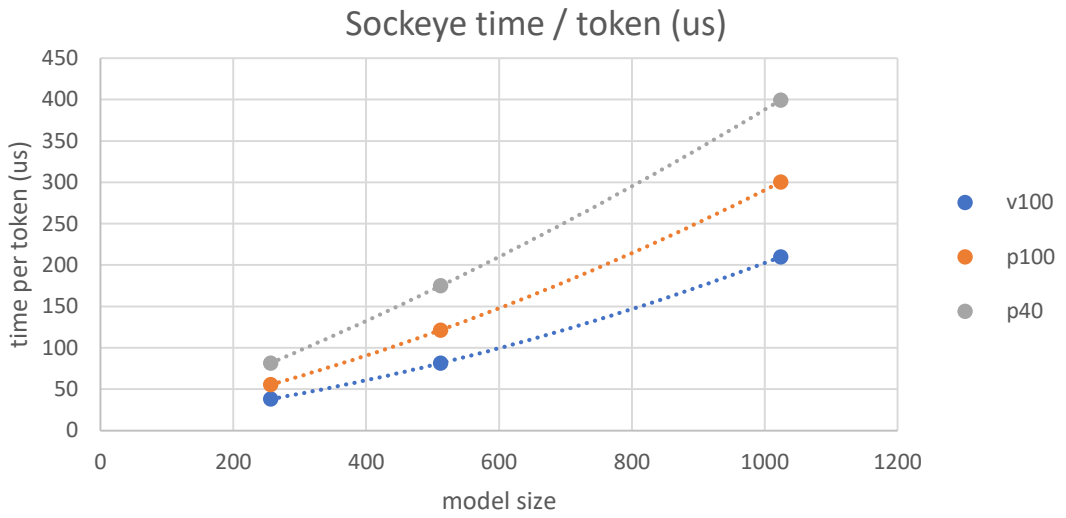
Model	Embedding Size d_m	Number of Heads N_H	FFN Size d_{ff}	Vocab Size d_V	Blocks in Stack N_B
small	256	4	1024	≈ 32000	6
base	512	8	2048		
big	1024	16	4096		

	Trained Params	No Weight-Tying	With Biases
Embedding/SoftMax	$d_M \times d_V$	$d_M \times d_{Vt}$ trained, $d_M \times d_{Vs}$ static	d_{Vt}
Self-Attention	$8 \times d_M^2 \times N_B$		$8 \times d_M \times N_B$
Encoder-Decoder Attention	$4 \times d_M^2 \times N_B$		$4 \times d_M \times N_B$
Feed-Forward Networks	$4 \times N_B \times d_M \times d_{ff}$		$2 \times N_B \times (d_{ff} + d_M)$
Layer Normalization	$(10 \times N_B + 4) \times d_M$		

	Sockeye (MXNet)	OpenNMT (PyTorch)	MLPerf (TensorFlow)
Embedding/Softmax	37,462,725	32,804,100	34,516,992
Self-Attention	50,331,648	50,380,800	50,331,648
Encoder-Decoder Attention	25,165,824	25,190,400	25,165,824
Feed-Forward Networks	100,724,736	100,724,736	100,724,736
Layer Normalization	65,536	65,536	65,536
Total Parameters	**213,750,469**	**209,165,572**	**210,804,736**
Total Parameters (base)	62,853,317	60,558,596	61,362,176

Baseline Performance Results





Time / Epoch	Small, v100
OpenNMT	1 hr, 12 min.
Sockeye	1 hr, 8 min.

Application printed training throughput/latency times don’t tell the whole story

Benchmarks for HW performance evaluation

- Public Domain

Usage	TF	MxNet	PyTorch-Caffe2 merge
CNN	TF_CNN_Benchmark	mxnet image-classification	pytorch/vision
seq2seq Translation	NMT	Sockeye	OpenNMT
Transformer Translation	tensor2tensor	Sockeye	OpenNMT
CNN Translation		Sockeye	OpenNMT
Deepspeech2		examples/speech-recognition	OpenNMT

MLPerf https://mlperf.org/

- Objective multi framework training benchmark run to convergence
- Compare performance/cost of cloud VMs
- Current belief is that equal versions across frameworks can be ported

Usage	implementation
image_classification	resnet50-tensorflow
object_detection	rcnn-caffe2
recommendation	neural filtering-pytorch
reinforcement	minigo-tensorflow
sentiment_analysis	cnn/rnn text categorization-paddle
speech_recognition	deepspeech2-pytorch
translation	transformer-tensorflow

Tools for performance evaluation

- Most machine learning codes are written in python
  - Which invoke compiled framework libraries
  - Which in turn invoke hardware specific math libraries (Cuda, MKL etc)
- As python is an interpreted language, dynamic tracing is required for most analysis.
- There are native python tracers, tracers built into the frameworks in some cases and HW based tools
- HW based tools for CPUs are tied to the performance counters
- HW based tools for GPUs are proprietary (NVProf for Nvidia) and may require binary instrumentation for some measurements
- Transformer – MXNet Profiler Output

Python profile tools ex:cProfile

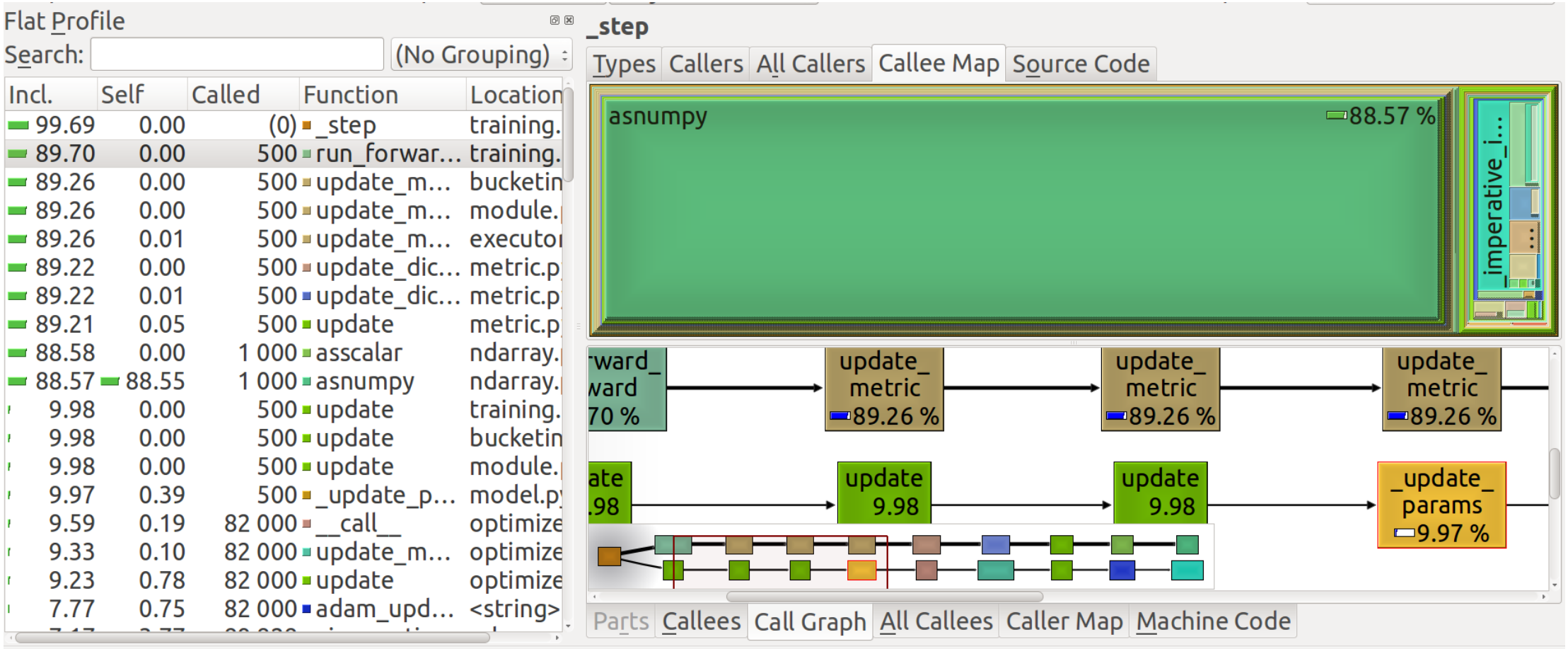
- Profile entire application
- python *-m cProfile -s time -o alex_inf_bs64.cprof* tf_cnn_benchmarks.py --local_parameter_device=gpu --num_gpus=1 --batch_size=64 --model=alexnet --variable_update=replicated --num_batches=200 --forward_only=True -tfprof_enable=False
- OR Insert code and profile within selected loop (ex: loop over minibatches)

  If(start_loop_count == loop_count) cProfile.Profile.enable

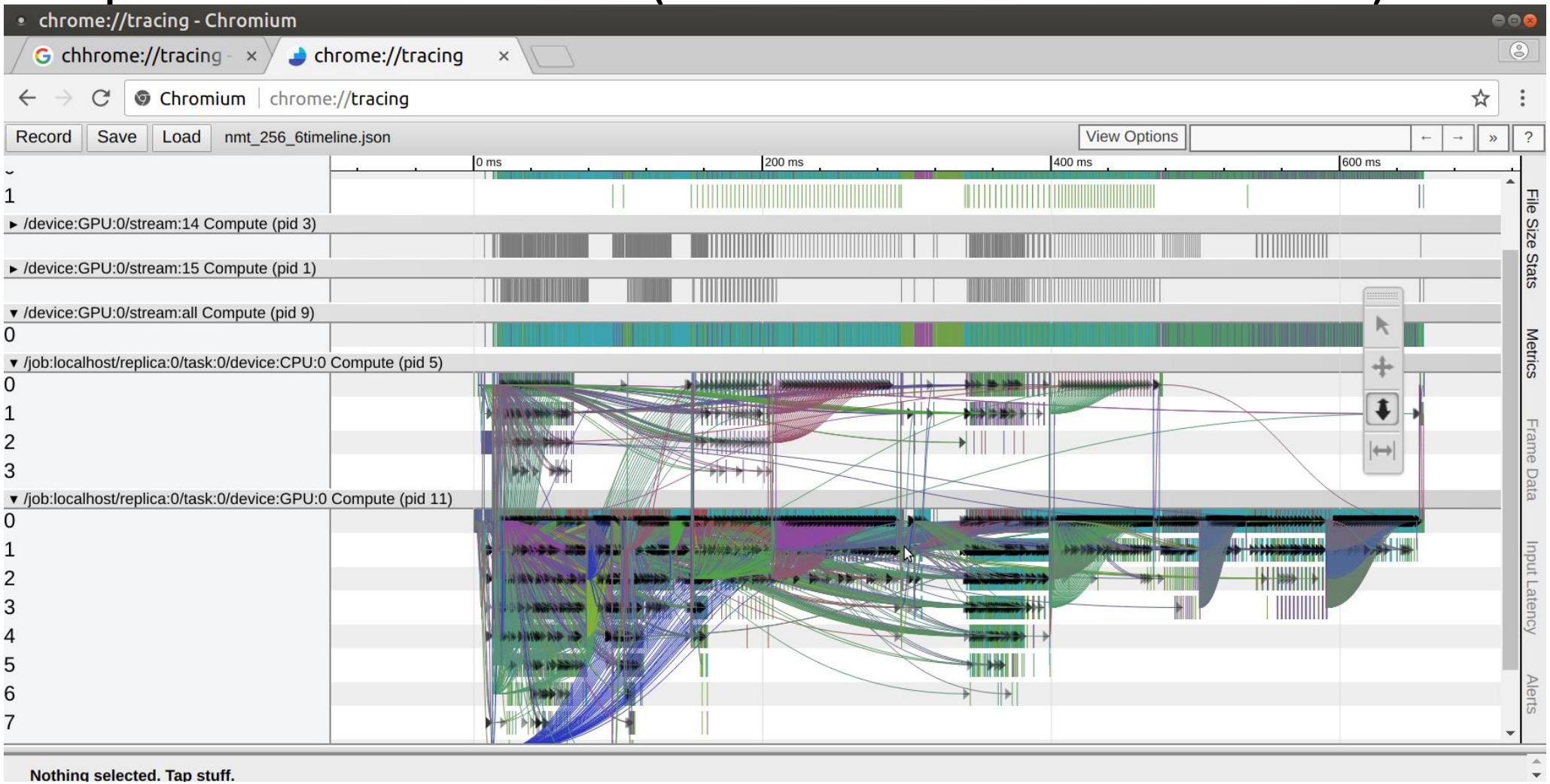
  #invoke “step” to process mini batch

  If(max_loop_count == loop_count) cProfile.Profile.disable()

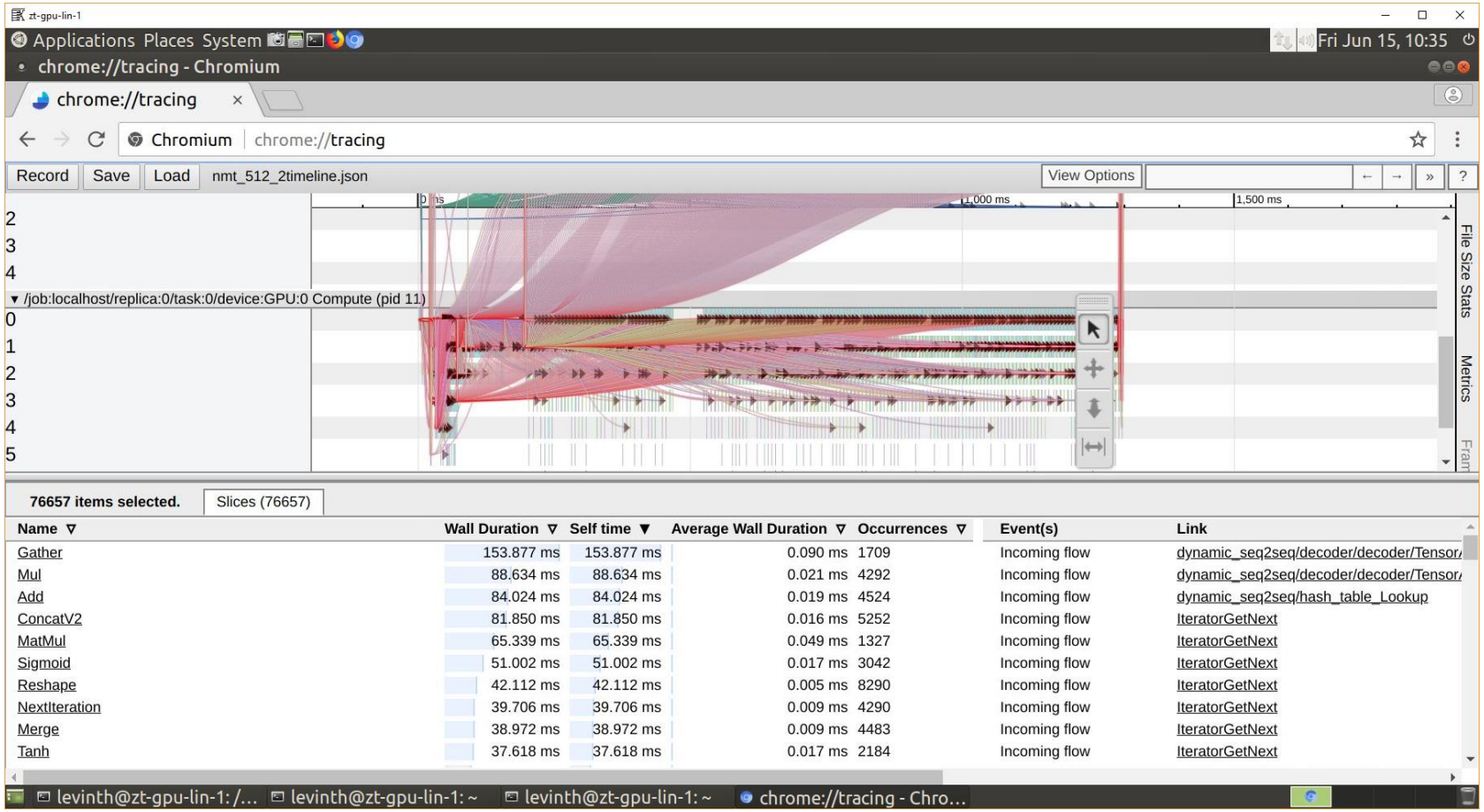
Cprofile only see Python execution



Tracing real ML networks yields a complicated result (Tensorflow timeline)

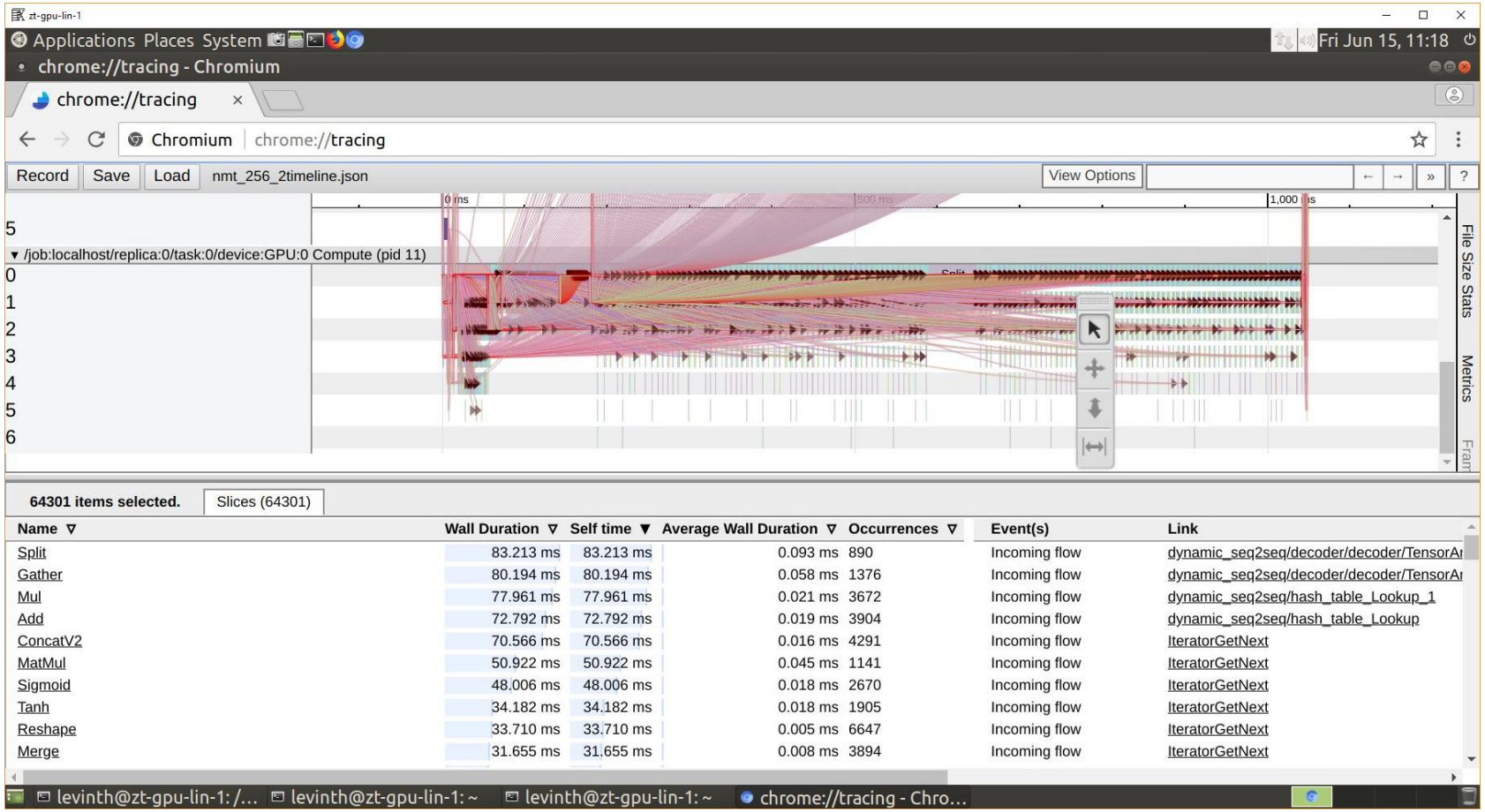


Tensorflow traced NMT, hidden_size=512 Total: 1,115.753 ms for 128 sentences

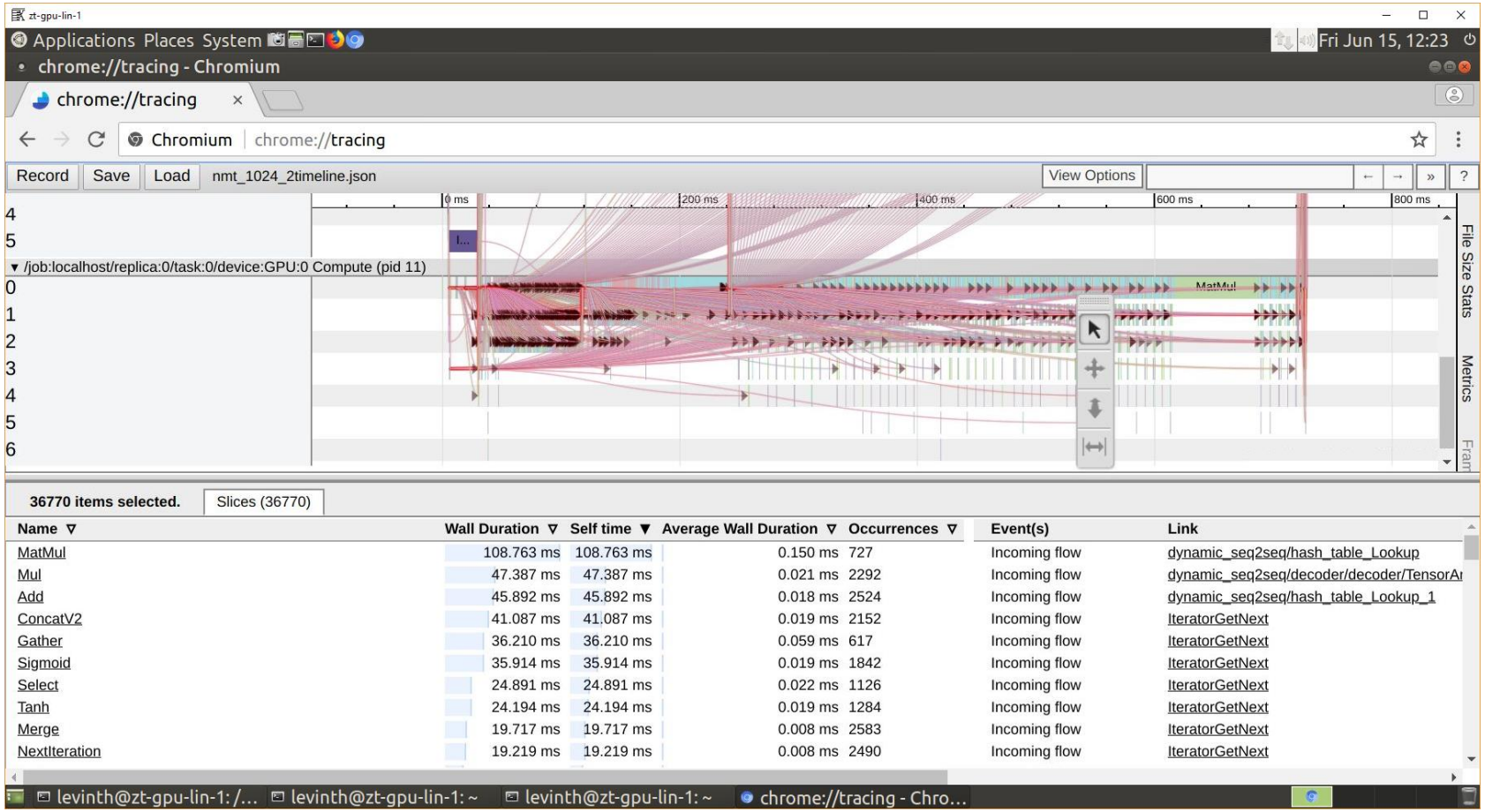


Tracing overhead is ~ 4X

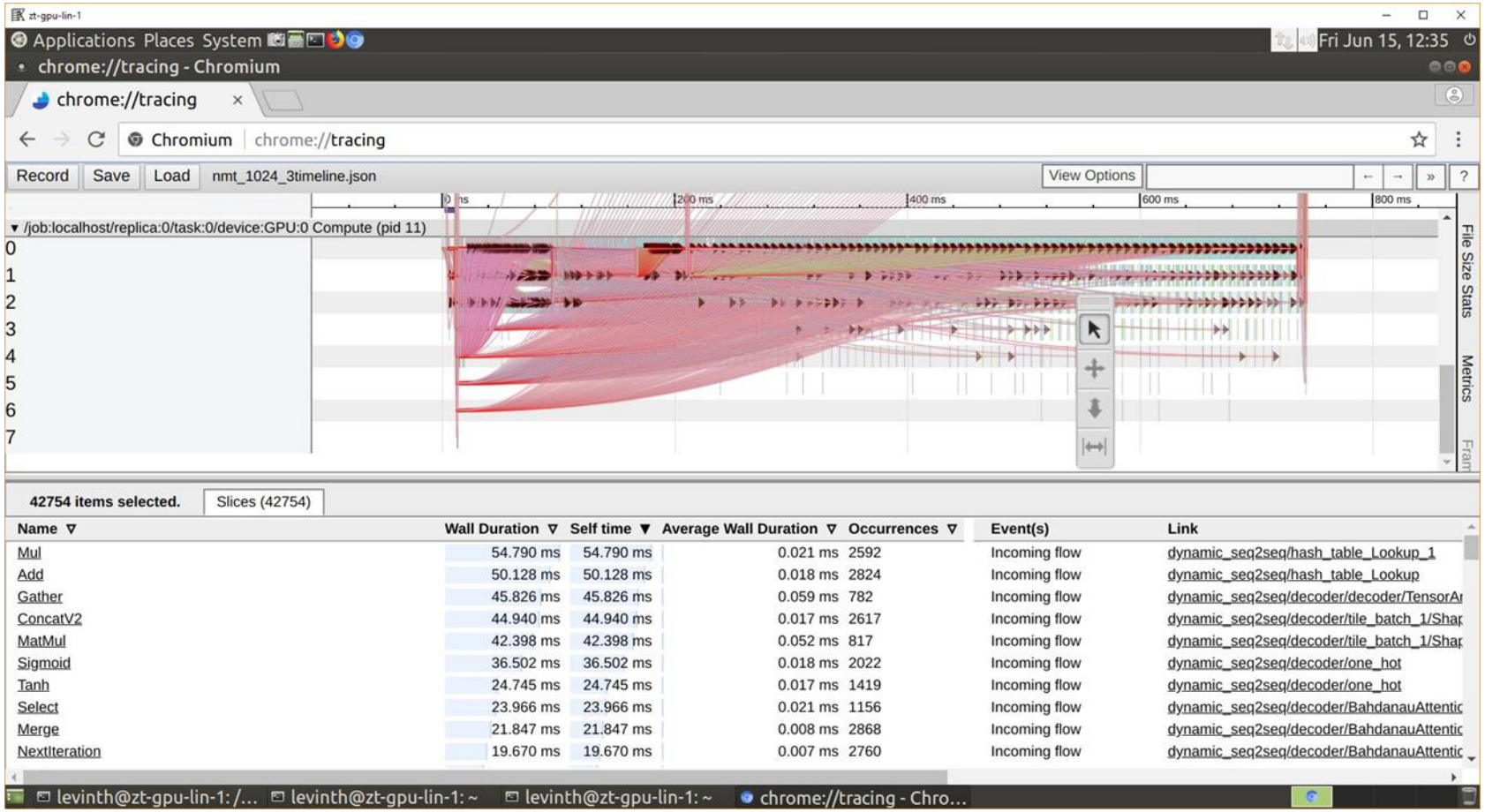
Hidden_size = 256



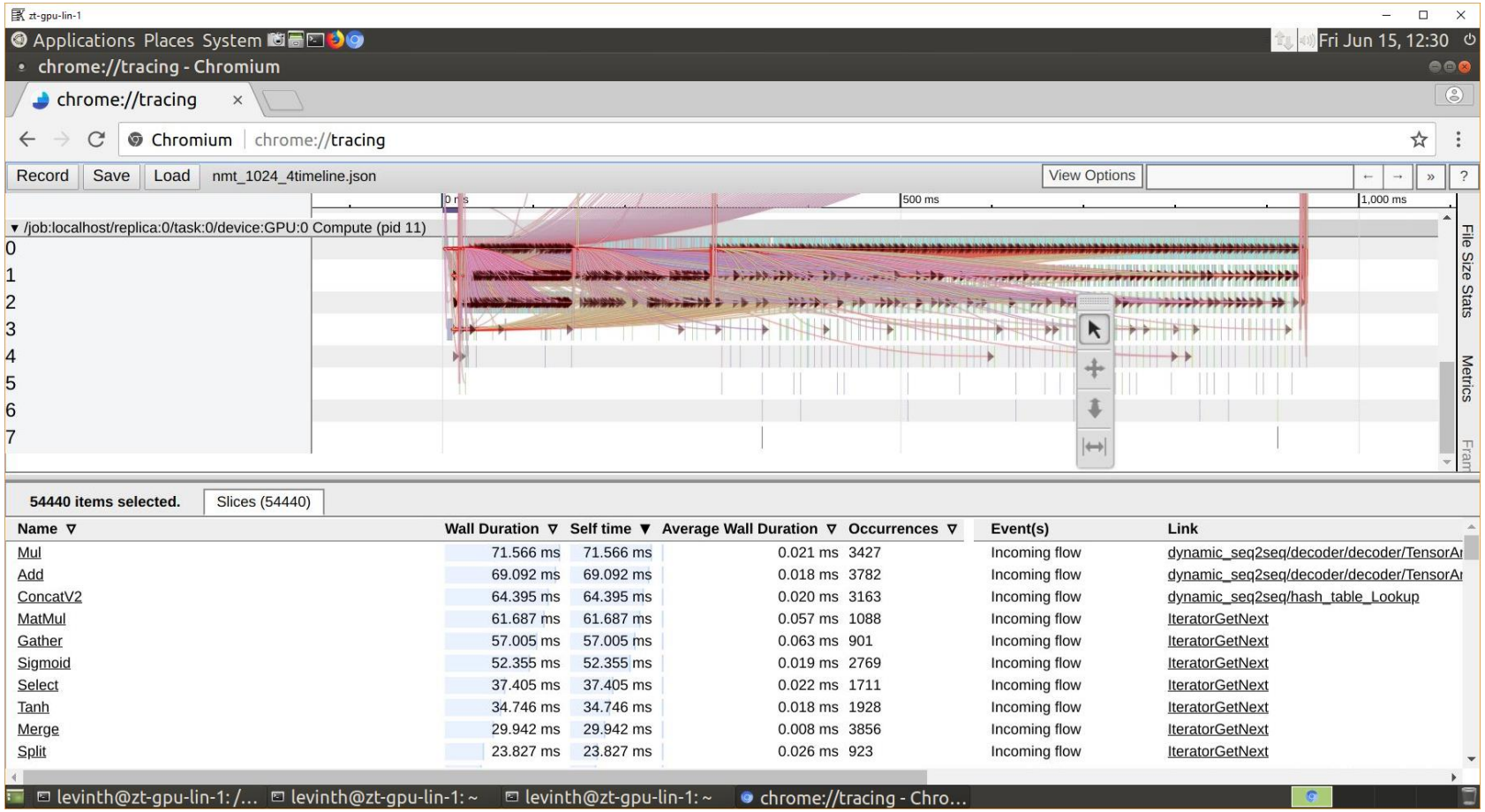
Hidden_size = 1024 minibatch 2



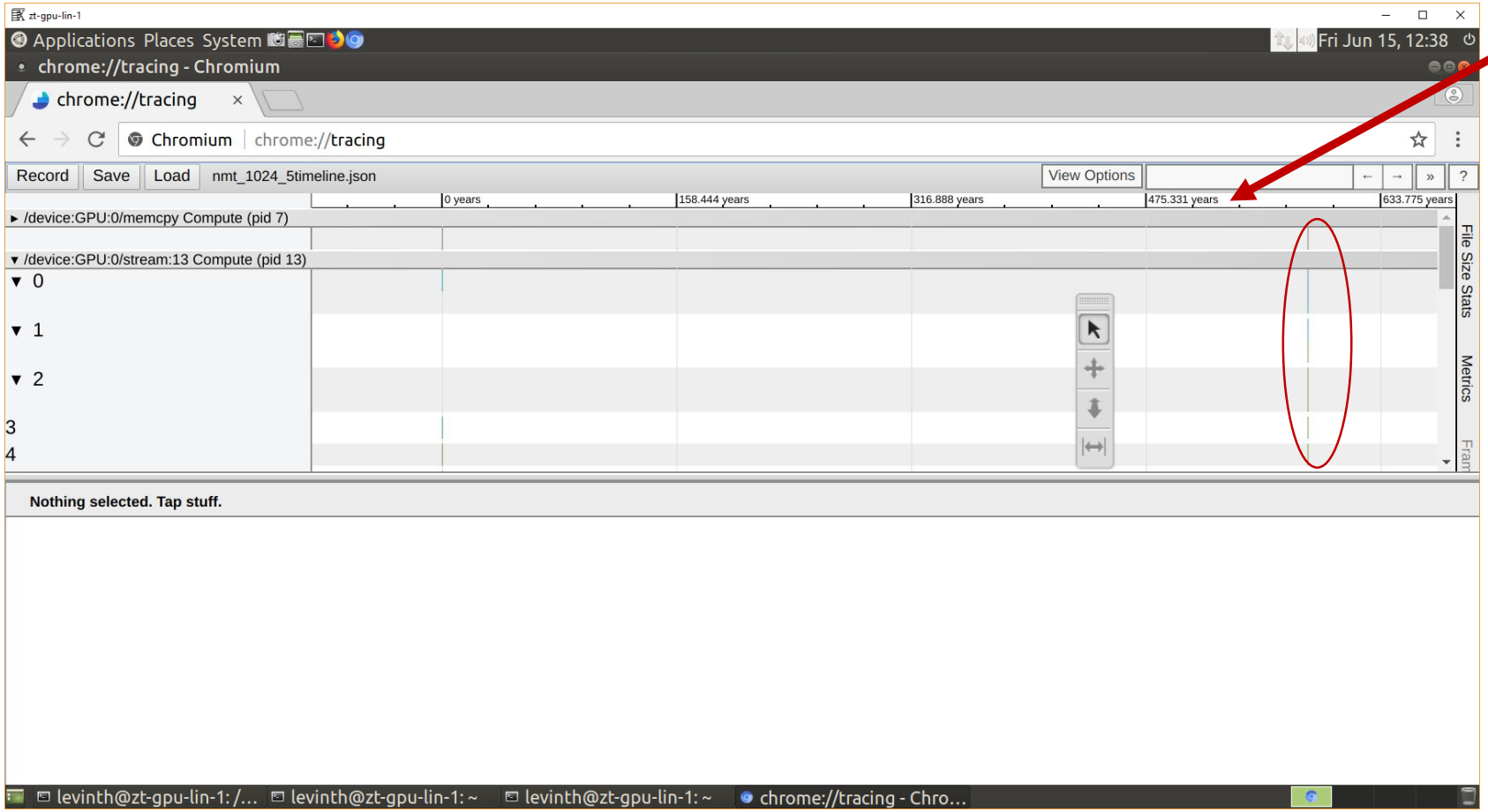
Hidden_size = 1024 minibatch 3



Hidden_size = 1024 minibatch 4

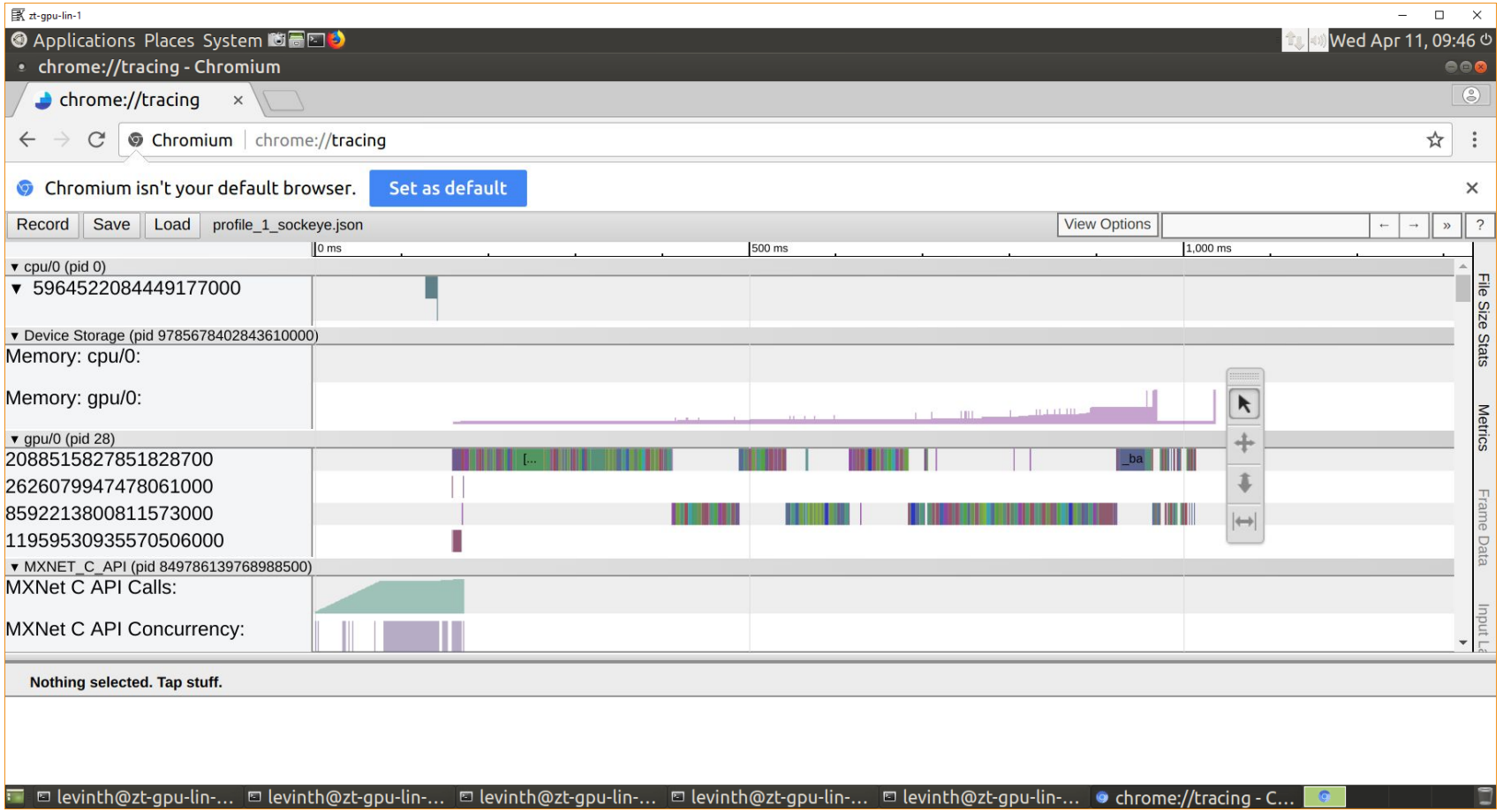


Hidden size = 1024 minibatch 5
Time stamps of some records went crazy

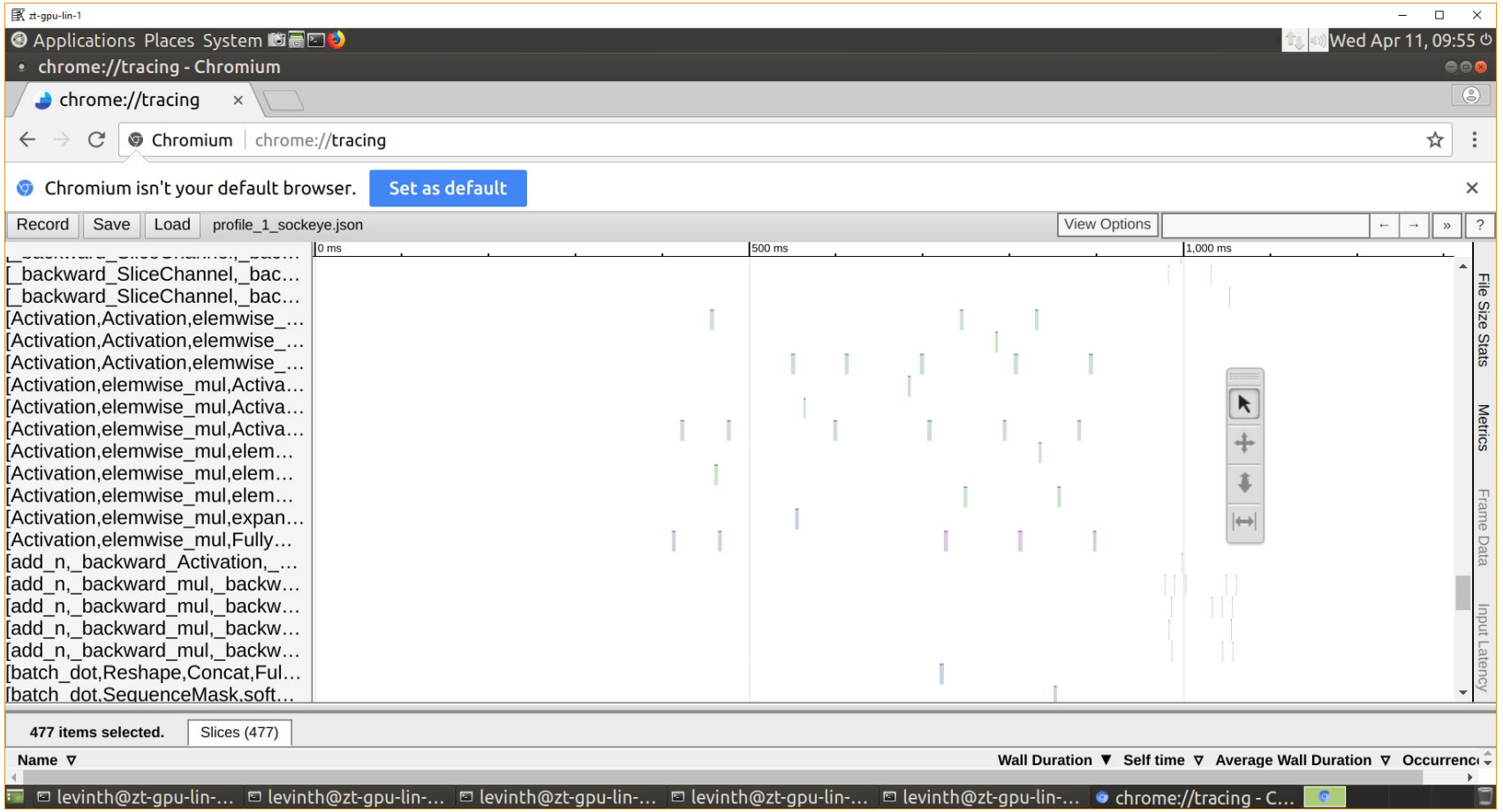


475 years??

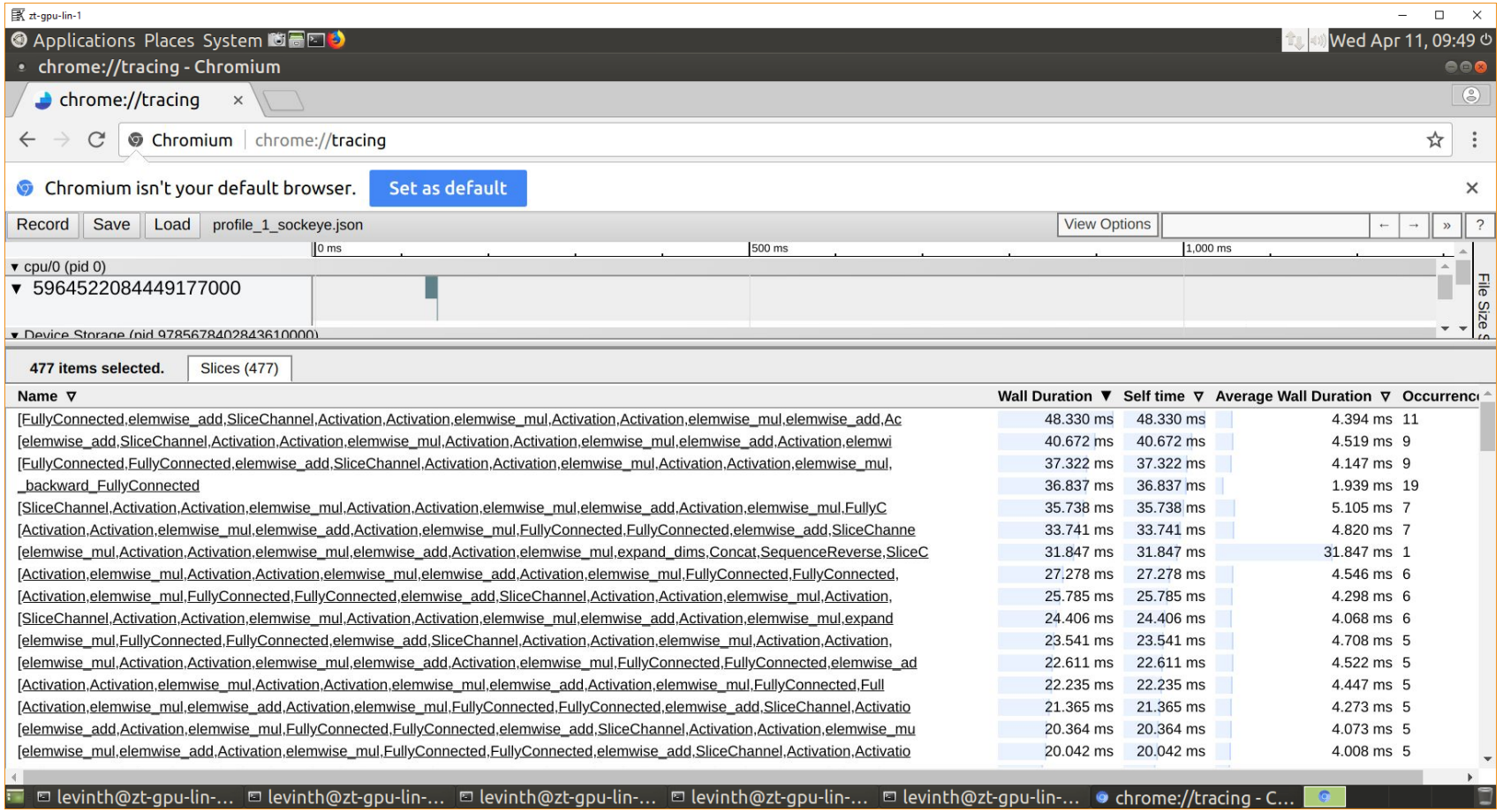
Mxnet profiler1



Mxnet profiler concurrency by function

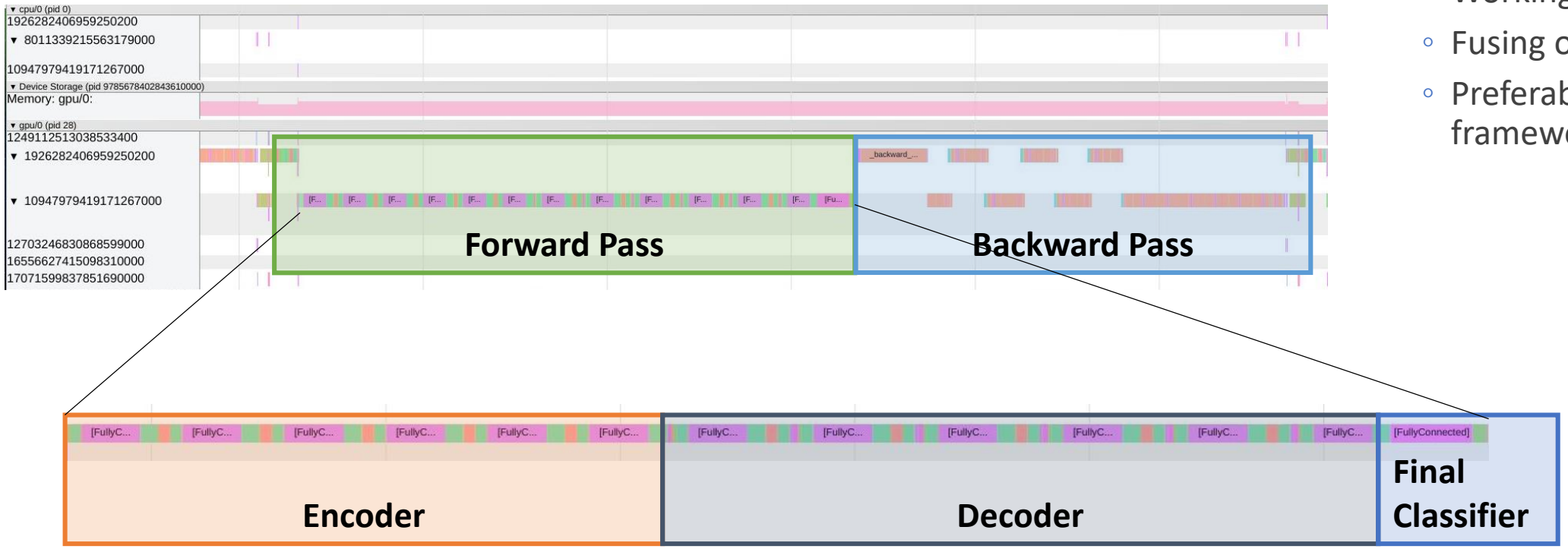


Mxnet profiler aggregated by functions



Transformer – MXNet Profiler Output

Training, big model, v100

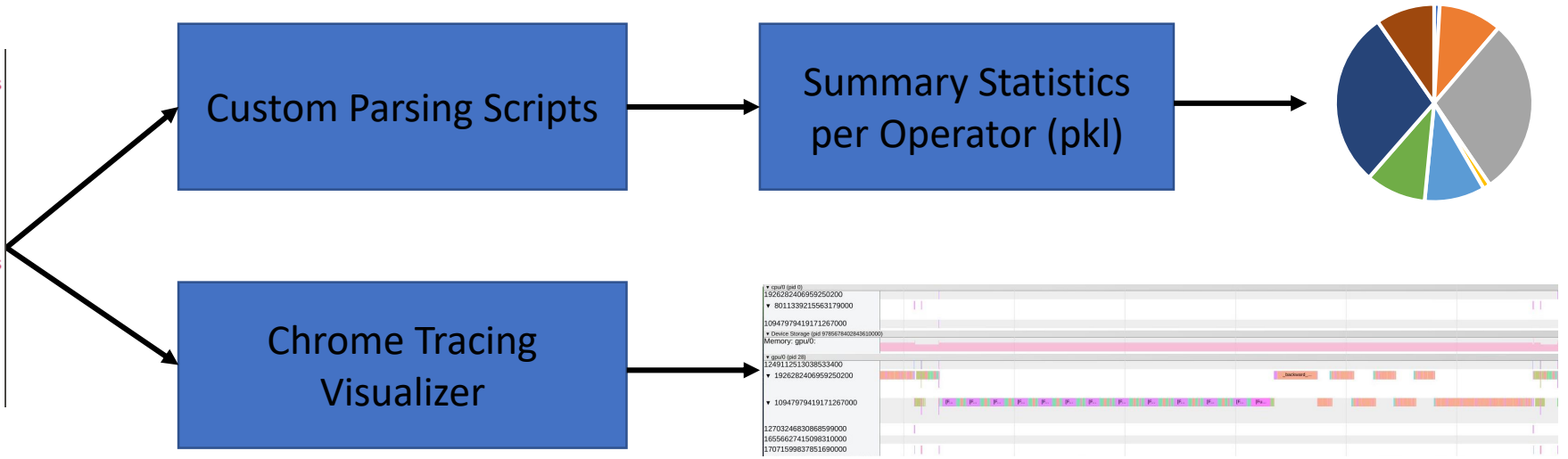


- Working set size
- Fusing operators
- Preferable to other framework profilers

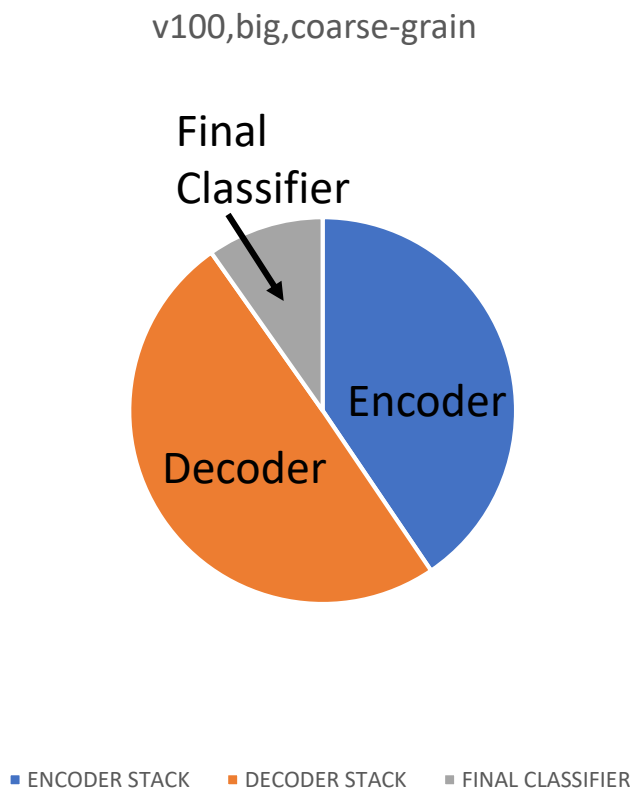
Process of Generating Results

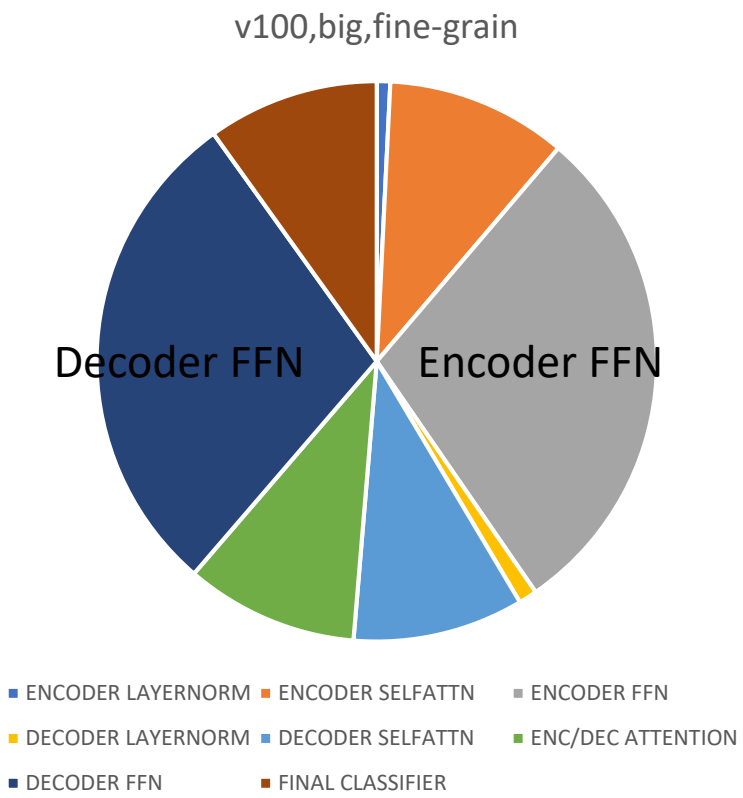
- MXNet Profiler is initialized & enabled for set window in training script
- Profiler produces a JSON file of TraceEvents with pid, timing info, and (fused) API call
- JSON can be viewed using chrome tracing tool, decent visualization to determine which operator handles correspond to different portions of the model execution
- Lillie Pentecost wrote a set of scripts to manually parse and collect summary statistics, filtering by operator handles and by thread + process id to gather GPU activity

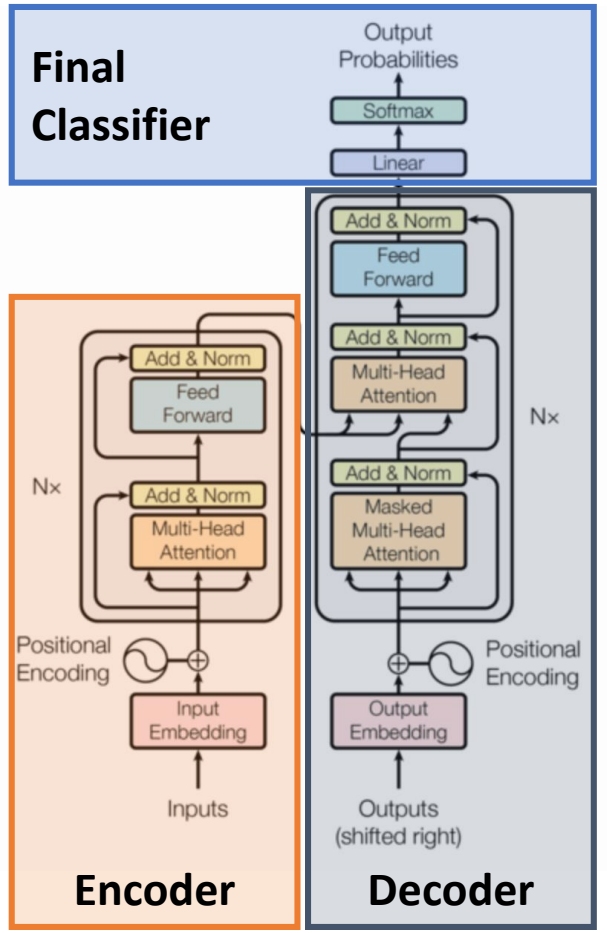

```
    {
        "name": "[FullyConnected,SliceChannel,_mul_scalar,Reshape,transpos
e,Reshape,Reshape,transpose,Reshape,batch_dot,broadcast_add,softmax,Dr",
        "cat": "operator",
        "ph": "B",
        "ts": 1532968109736230,
        "pid": 28,
        "tid": 1869324207769125107
    }
,
    {
        "name": "[FullyConnected,SliceChannel,_mul_scalar,Reshape,transpos
e,Reshape,Reshape,transpose,Reshape,batch_dot,broadcast_add,softmax,Dr",
        "cat": "operator",
        "ph": "E",
        "ts": 1532968109736849,
        "pid": 28,
        "tid": 1869324207769125107
    }
```

Transformer – Training Timing Summary

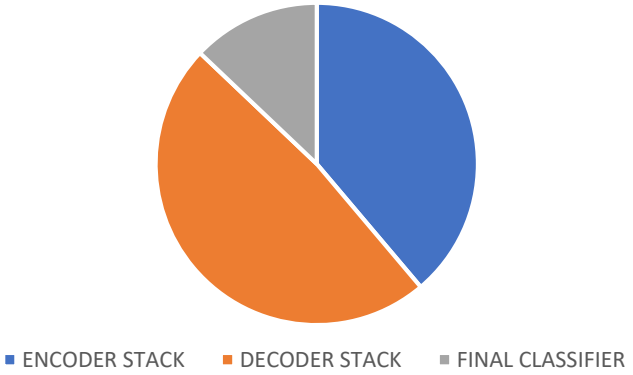




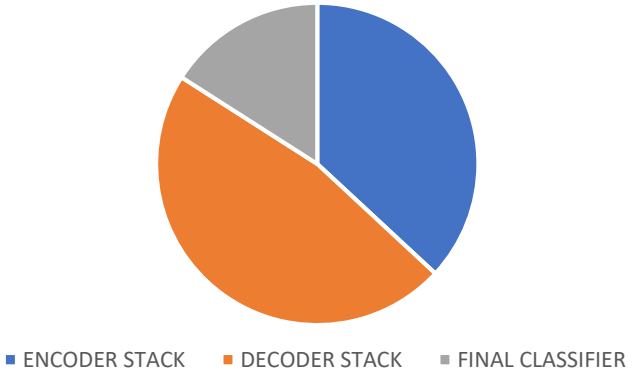


Transformer – Across GPU Cards, base

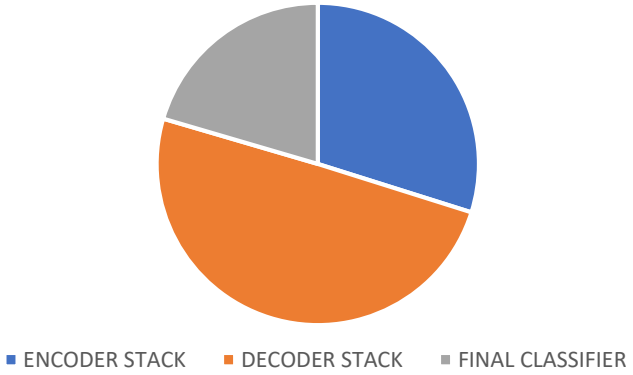
p40,base,coarse-grain



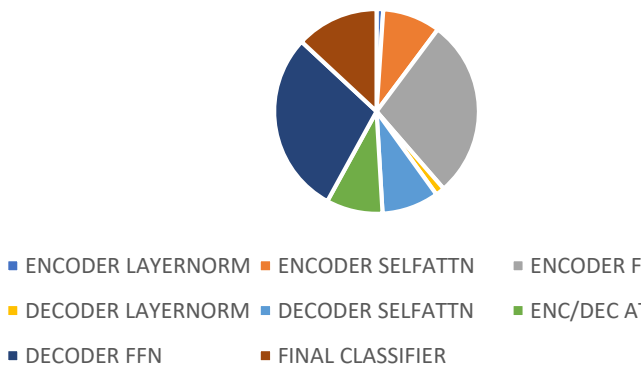
p100,base,coarse-grain



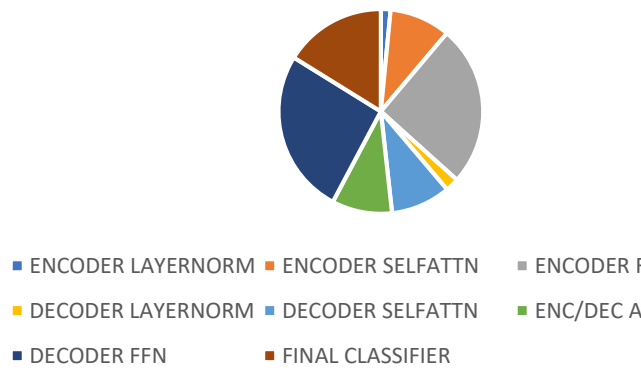
v100,base,coarse-grain



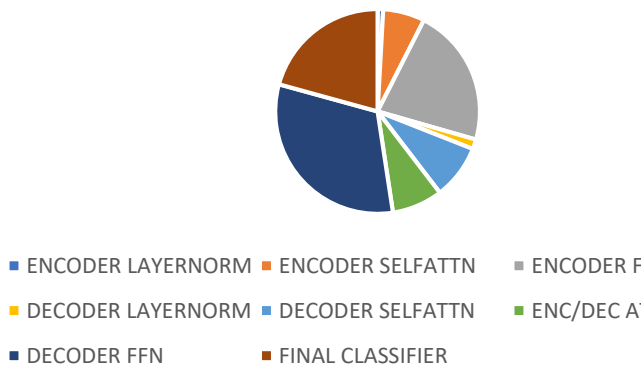
p40,base,fine-grain



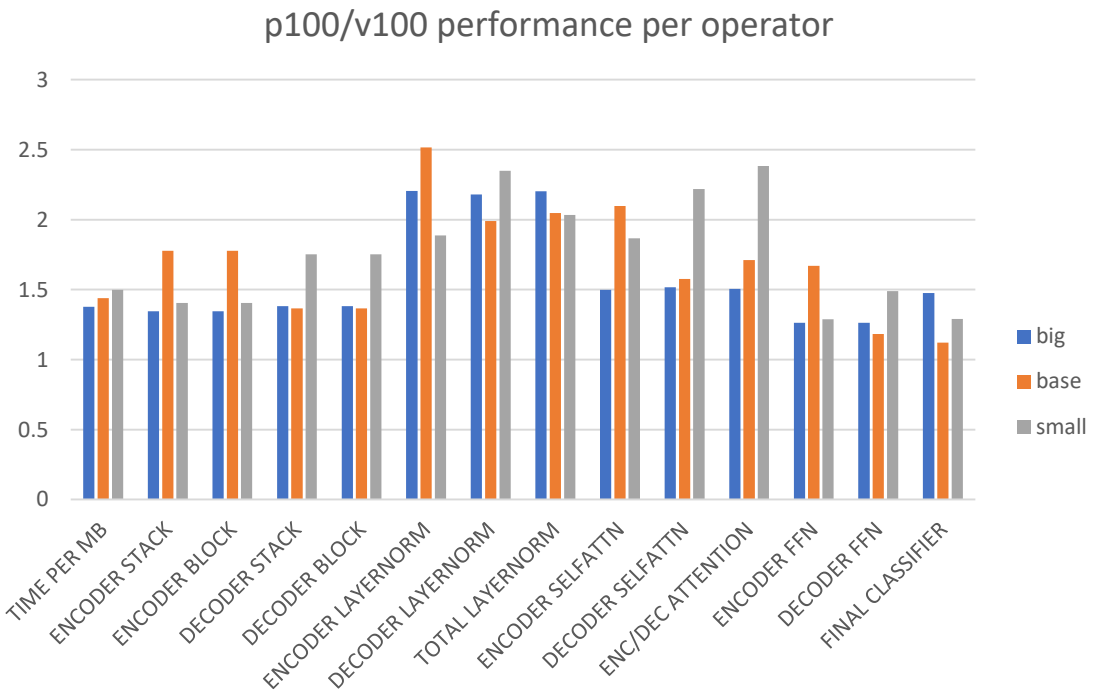
p100,base,fine-grain



v100,base,fine-grain



Transformer – Across GPU Cards (FP32)

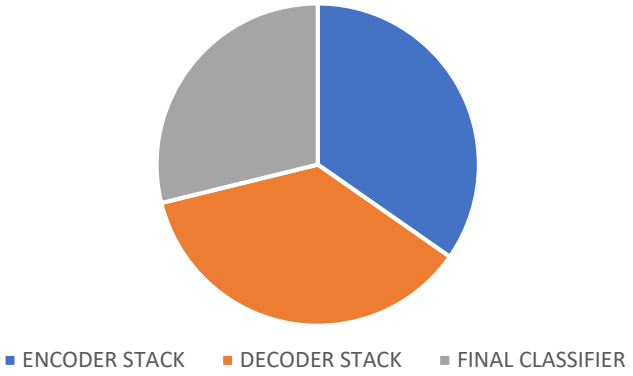


GPU PERFORMANCE COMPARISON

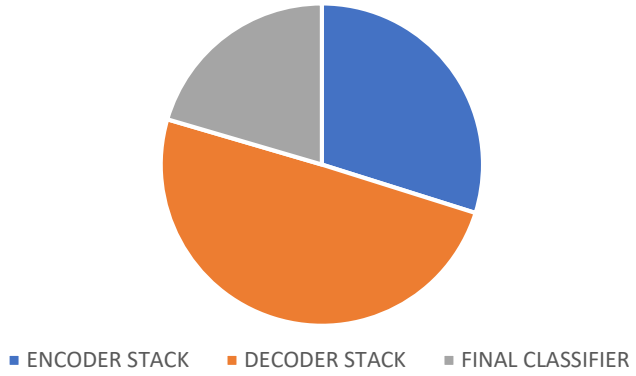
	P100	V100	Ratio
DL Training	10 TFLOPS	120 TFLOPS	12x
DL Inferencing	21 TFLOPS	120 TFLOPS	6x
FP64/FP32	5/10 TFLOPS	7.5/15 TFLOPS	1.5x
HBM2 Bandwidth	720 GB/s	900 GB/s	1.2x
STREAM Triad Perf	557 GB/s	855 GB/s	1.5x
NVLink Bandwidth	160 GB/s	300 GB/s	1.9x
L2 Cache	4 MB	6 MB	1.5x
L1 Caches	1.3 MB	10 MB	7.7x
GPU Memory	16 GB HBM2	16 GB HBM2	1x

Transformer Timing – Scaling with Model Size

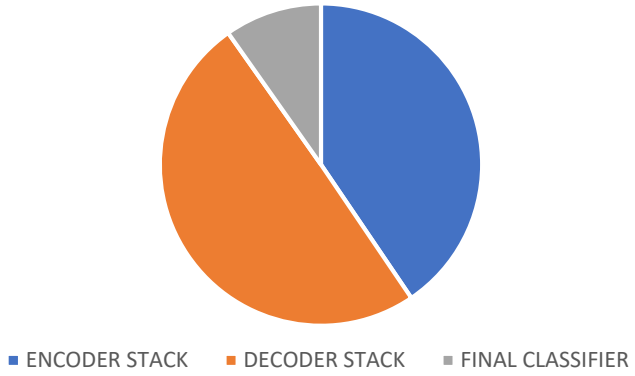
v100,small,coarse-grain



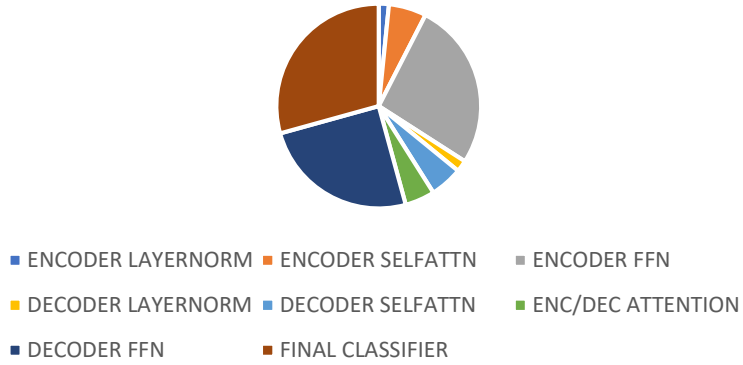
v100,base,coarse-grain



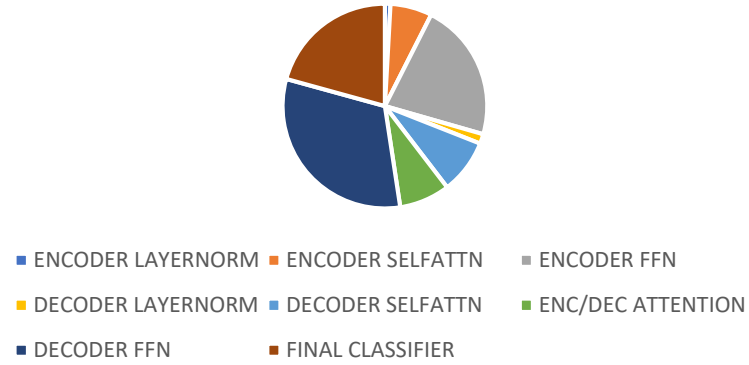
v100,big,coarse-grain



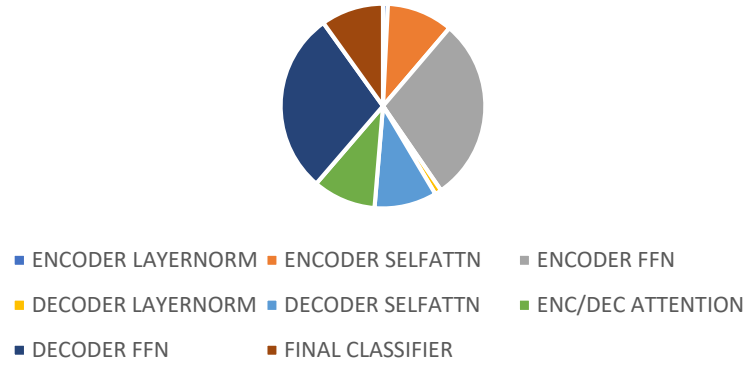
v100,small,fine-grain



v100,base,fine-grain

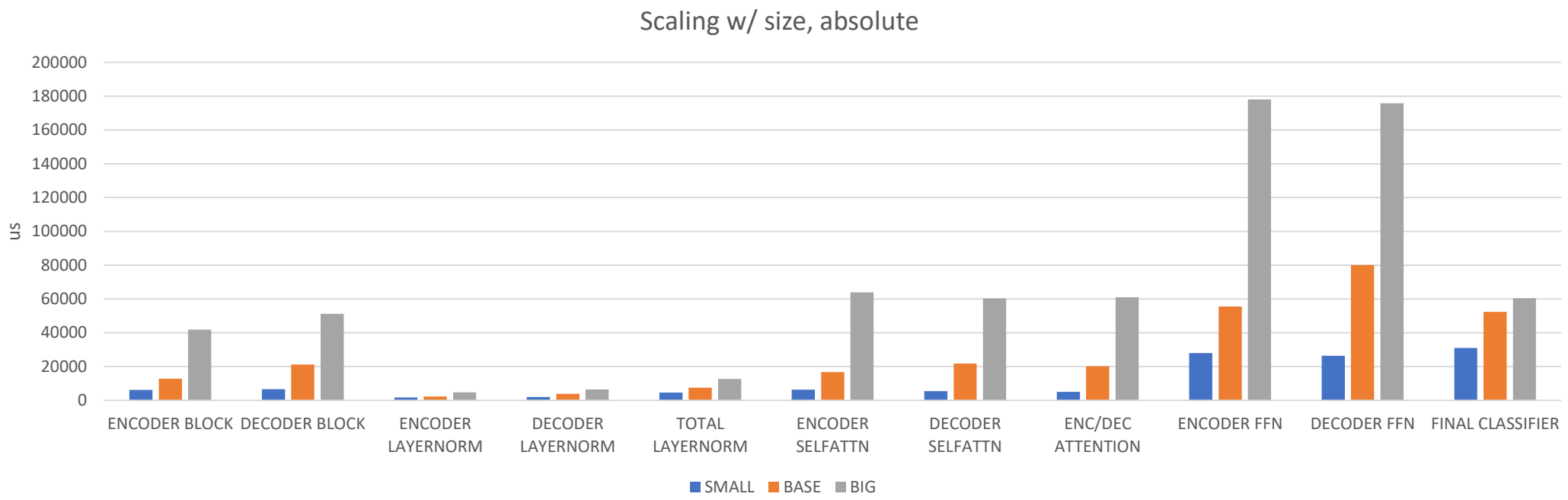


v100,big,fine-grain



Transformer – Scaling with Model Size

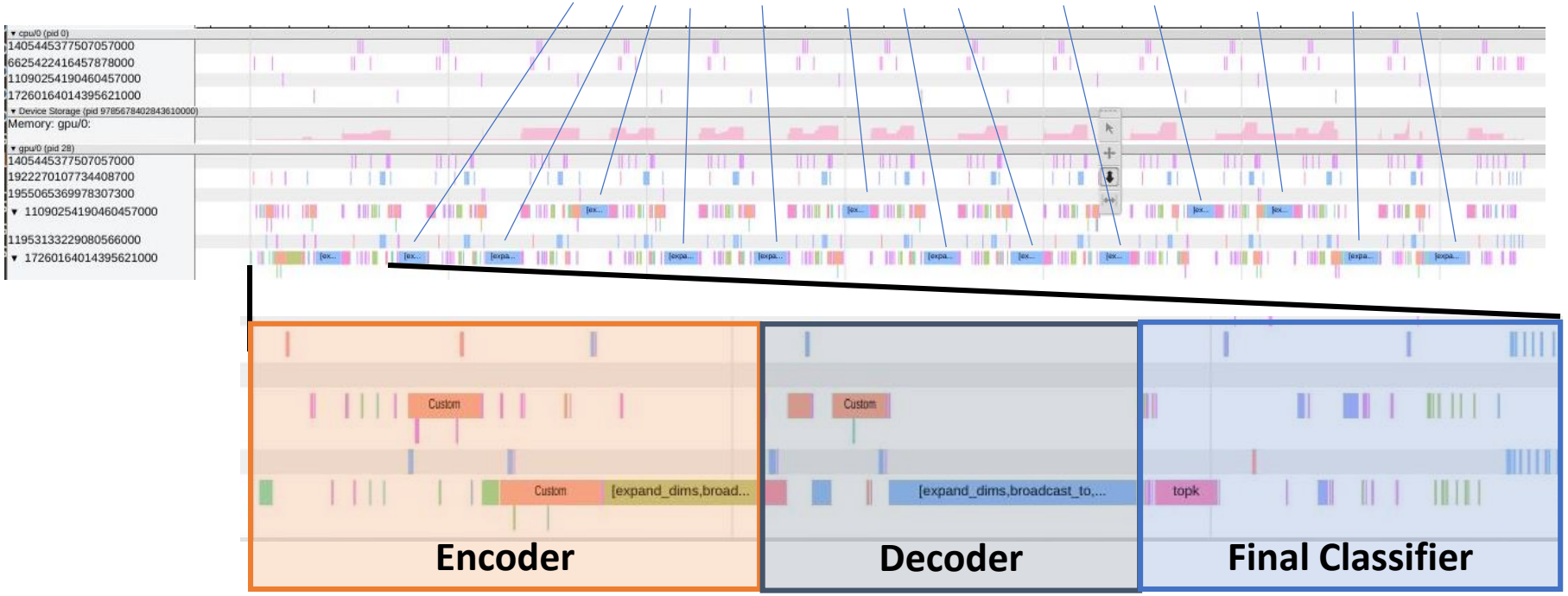
- (v100)



Transformer – Inference Results

- Single translation example, beam size 1, German-English translation
- Fitting all decoder + final classifier parameters on-chip for performance win on inference

Inference, base model, v100 – “Toys R Us is planning to hire fewer workers this Christmas season .”



NVProf profiles activity on the GPU only

Slows down code by very large factor (~150X) if things like FP operation counts are collected
Not so bad if only time is collected
Output is CSV, example below is post processed to add some information about the batching and so on

==115269== Profiling application: python tf_cnn_benchmarks.py --model alexnet --forward_only True --local_parameter_device=gpu --num_gpus=1 --batch_size 1 --num_batch 100 --display_every 1000
==115269== Profiling result:
==115269== Metric result:

Device	Kernel	Invocations	Metric Name	Metric Description	Min	Max	Avg	Total
Tesla P4 (0)	void Eigen::internal::EigenMetaKernel<Eigen::TensorEvaluator<Eigen::'	108	flop_count_sp	Floating Point Operations(Single	154,587	154,587	154,587	16,695,396
Tesla P4 (0)	void cudnn::maxwell::gemm::setOutputKernel<float>(int, float*, float)	53	flop_count_sp	Floating Point Operations(Single	0	0	0	0
Tesla P4 (0)	void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::m	159	flop_count_sp	Floating Point Operations(Single	9,216	46,656	29,440	4,680,960
Tesla P4 (0)	void Eigen::internal::EigenMetaKernel<Eigen::TensorEvaluator<Eigen::'	54	flop_count_sp	Floating Point Operations(Single	154,587	154,587	154,587	8,347,698
Tesla P4 (0)	void Eigen::internal::EigenMetaKernel<Eigen::TensorEvaluator<Eigen::'	159	flop_count_sp	Floating Point Operations(Single	1,001	4,096	3,064	487,176
Tesla P4 (0)	maxwell_scudnn_128x64_interior_nn	53	flop_count_sp	Floating Point Operations(Single	143,327,232	143,327,232	143,327,232	7,596,343,296
Tesla P4 (0)	void tensorflow::BiasNCHWKernel<float>(int, float const *, float const ,	265	flop_count_sp	Floating Point Operations(Single	43,264	193,600	101,324	26,850,860
Tesla P4 (0)	void cudnn::winograd::generateWinogradTilesKernel<int=0, float, float	159	flop_count_sp	Floating Point Operations(Single	4,276,224	8,552,448	6,176,768	982,106,112
Tesla P4 (0)	maxwell_scudnn_winograd_128x128_tile148n_nt	159	flop_count_sp	Floating Point Operations(Single	157,827,072	314,720,256	227,453,610	36,165,123,990
Tesla P4 (0)	cudnn::maxwell::gemm::computeOffsetsKernel(cudnn::maxwell::gem	107	flop_count_sp	Floating Point Operations(Single	0	0	0	0
Tesla P4 (0)	maxwell_scudnn_128x64_stridedB_splitK_small_nn	53	flop_count_sp	Floating Point Operations(Single	472,743,936	472,743,936	472,743,936	25,055,428,608
Tesla P4 (0)	void Eigen::internal::EigenMetaKernel<Eigen::TensorEvaluator<Eigen::'	371	flop_count_sp	Floating Point Operations(Single	0	0	0	0
Tesla P4 (0)	void gemv2N_kernel_val<float, float, float, int=128, int=8, int=4, int=4, i	53	flop_count_sp	Floating Point Operations(Single	8,529,311	8,529,311	8,529,311	452,053,483
Tesla P4 (0)	void tensorflow::functor::SwapDimension1And2InTensor3Simple<unsi	54	flop_count_sp	Floating Point Operations(Single	0	0	0	0
Tesla P4 (0)	cudnn::maxwell::gemm::computeBOffsetsKernel(cudnn::maxwell::gen	53	flop_count_sp	Floating Point Operations(Single	0	0	0	0
Tesla P4 (0)	void gemv2N_kernel_val<float, float, float, int=128, int=4, int=4, int=4, i	106	flop_count_sp	Floating Point Operations(Single	34,222,080	76,984,320	55,603,200	5,893,939,200
Tesla P4 (0)	void tensorflow::functor::SwapDimension0And2InTensor3Simple<float	266	flop_count_sp	Floating Point Operations(Single	0	0	0	0
				Total flop count				76,202,056,779
				Batch size				1
				Num batches executed				54
				Num warmup batches				0
				Total batches				54
				Total images				54
				Flop per image				1,411,149,200

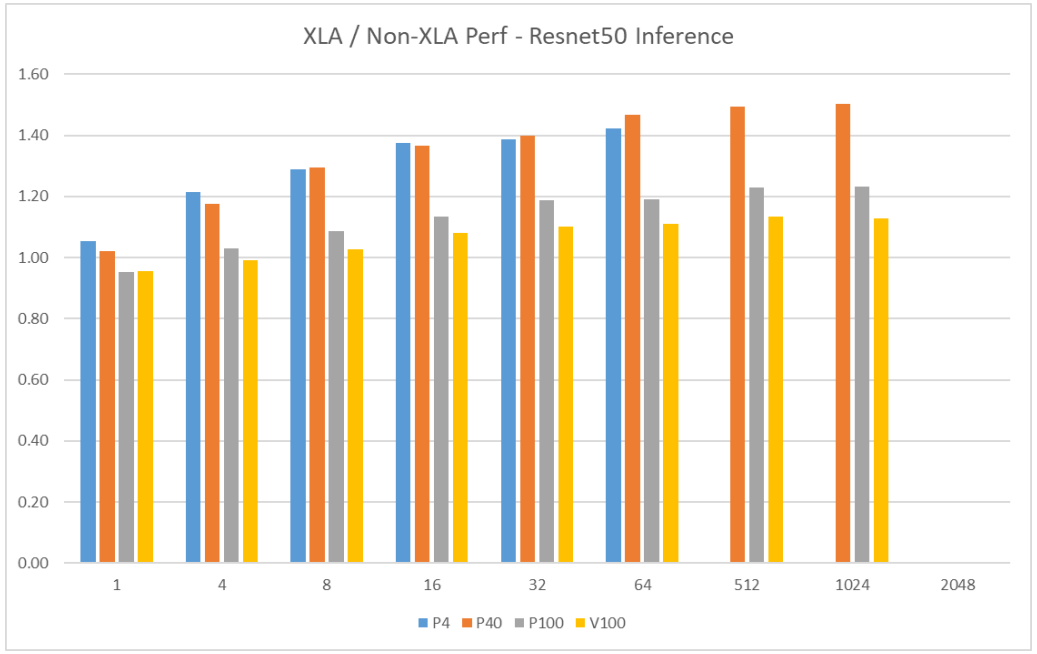
Framework profilers have issues

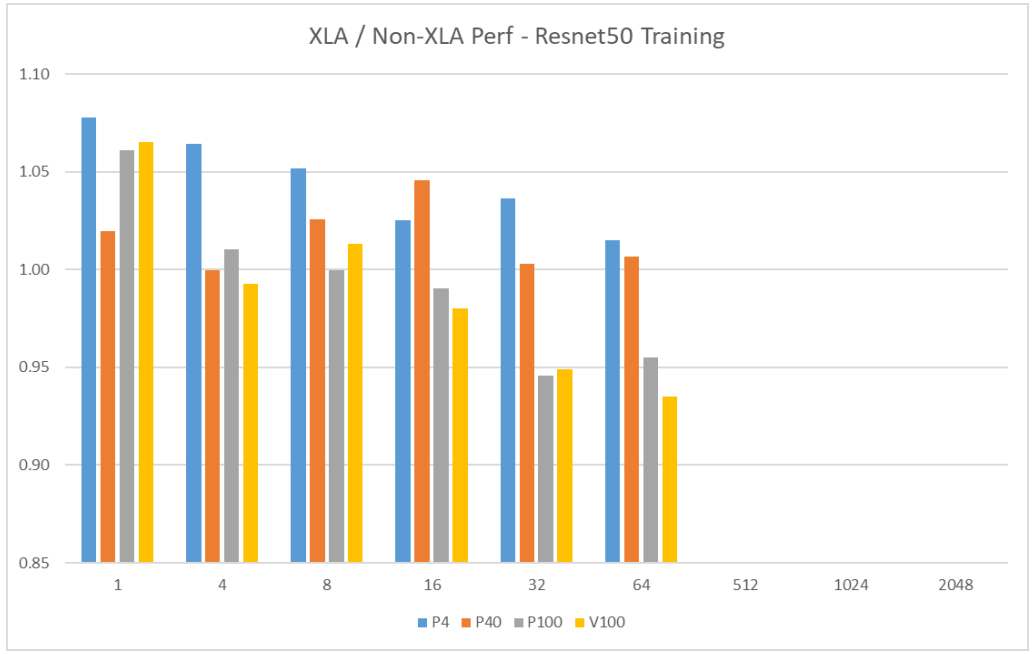
- Not clear TF profiler works
- MxNet profiler has issues with symbols/long names
- Python profilers have issues seeing into compiled libraries
  - ie the frameworks
- HW profilers
  - Nvprof requires binary instrumentation (and 165X slow down) for anything beyond cycles

Intermediate representations

- Intermediate representations for deep neural networks
  - Create a framework independent representation
    - Simplifying multi framework support from HW vendors
  - Enable rational approaches to network calculation optimization
  - Multi layer fusion
    - Ex: conv layer followed by relu layer followed by max pooling layer
    - Combine to a single layer to avoid data movement
    - Multi layer fusion is also done independently from IR ex: Nvidia TensorRT
- XLA and ONNX are currently popular approaches

Impact of XLA on Resnet50 @ fp32

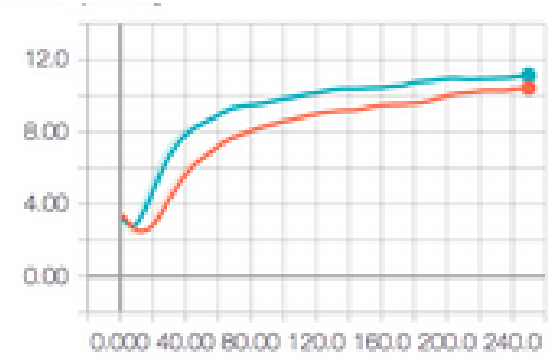




TF R1.7, Cuda 9.1, cudnn 7.1.2

Numerical stability

- MLPerf submissions require 5 runs with random seed 1->5
  - Timing is defined as a time to threshold quality
  - Quality is measured once per epoch
- Because of the granularity of the evaluation, a small variation in convergence leads to a large variation in time
  - for translators each epoch takes ~ 6 hours



Numerical Stability: MLPerf RNN

- MLPerf RNN Translator

use_bias=0					
epoch	seed=1	seed=2	seed=3	seed=4	seed=5
0	19.12	19.56	19.3	19.02	19.19
1	20.94	20.93	20.85	20.57	20.83
2	21.42	21.42	21.33	21.49	21.54
3	21.67	21.89	21.73	21.7	21.83
4	22.01		22.1	22.04	

- With use_bias=1 (adding bias terms to LSTM)

use_bias=1					
epoch	seed=1	seed=2	seed=3	seed=4	seed=5
0	19.33	19.37	19.11	19.34	19.26
1	20.93	20.93	20.7	20.68	20.89
2	21.63	21.53	21.41	21.43	21.24
3	21.64	21.74	21.56	21.67	21.67
4	21.88	22.09	21.82	21.86	21.56
5					22.08

Time per epoch vs Logging

- There are differences in run timing depending on whether one logs the output with tee vs redirect (>)
- Tee

RNN-P100				transformer-v100			
run	time	epochs	time/epoch	run	time	epochs	time/epoch
1	58861	4	14715.25	1	71734	4	17933.5
2	96995	5	19399	2	71736	4	17934
3	58856	4	14714	3	95361	45	2119.13333
4	87821	5	17564.2	4	71913	4	17978.25
5	85910	5	17182	5	71829	4	17957.25

RNN-V100-redirect			
run	time	epochs	time/epoch
1	50125	5	10025
2	50123	5	10024.6
3	50114	5	10022.8
4	49989	5	9997.8
5	60061	6	10010.1667
5	60021	6	10003.5

Some learnings

- Models are big and getting bigger
- Activations must be kept (or recalculated) for backpropagation
  - Increasing memory usage
- Memory capacity and bandwidth significantly impact performance
  - Optimal HW design is not obvious
- Defining a benchmark time is not straightforward due to random seeding