Expediting Deep Learning with High Performance Computing

Dr. Kunal Banerjee, SMIEEE

Staff Data Scientist
Walmart Global Tech, Bangalore, India

Ex-Research Scientist
Intel Labs, Bangalore, India

Doctoral Student
Comp Sc & Engg, IIT Kharagpur

Overview

Deep Learning & HPC

Deep Learning is now ubiquitous:
self-driving cars, voice-activated assistants, automatic machine translation, image recognition, cancer detection, market price forecasting, etc.

Table: Training time and top-1 validation accuracy with ImageNet/ResNet-50

	Batch Size	Processor	DL Library	Time	Accuracy
He et al.	256	Tesla P100×8	Caffe	29 hours	75.30%
Goyal et al.	8K	Tesla P100×256	Caffe	21 hours	76.30%
Smith et al.	16K	full TPU Pod	TensorFlow	30 mins	76.10%
Akiba et al.	32K	Tesla P100×1024	Chainer	15 mins	74.90%
Jia et al.	64K	Tesla P40×2048	TensorFlow	6.6 mins	75.80%
Mikami et al.	68K	Tesla V100×2176	NNL	224 secs	75.03%

Source: `news.developer.nvidia.com/sony-breaks-resnet-50-training-record-with-nvidia-v100-tensor-core-gpus/`

"Finally, after decades of research, deep learning, the abundance of data, the powerful computation of GPUs came together in a big bang of modern AI."

– Jensen Huang, CEO of Nvidia

Deep Learning & HPC

Deep Learning is now ubiquitous:
self-driving cars, voice-activated assistants, automatic machine translation, image recognition, cancer detection, market price forecasting, etc.

Table: Training time and top-1 validation accuracy with ImageNet/ResNet-50

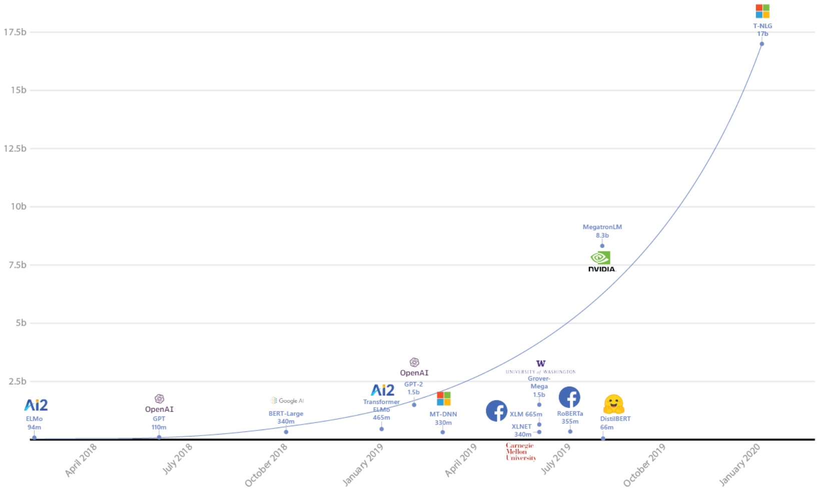
	Batch Size	Processor	DL Library	Time	Accuracy
He et al.	256	Tesla P100×8	Caffe	29 hours	75.30%
Goyal et al.	8K	Tesla P100×256	Caffe	21 hours	76.30%
Smith et al.	16K	full TPU Pod	TensorFlow	30 mins	76.10%
Akiba et al.	32K	Tesla P100×1024	Chainer	15 mins	74.90%
Jia et al.	64K	Tesla P40×2048	TensorFlow	6.6 mins	75.80%
Mikami et al.	68K	Tesla V100×2176	NNL	224 secs	75.03%

Source: `news.developer.nvidia.com/sony-breaks-resnet-50-training-record-with-nvidia-v100-tensor-core-gpus/`

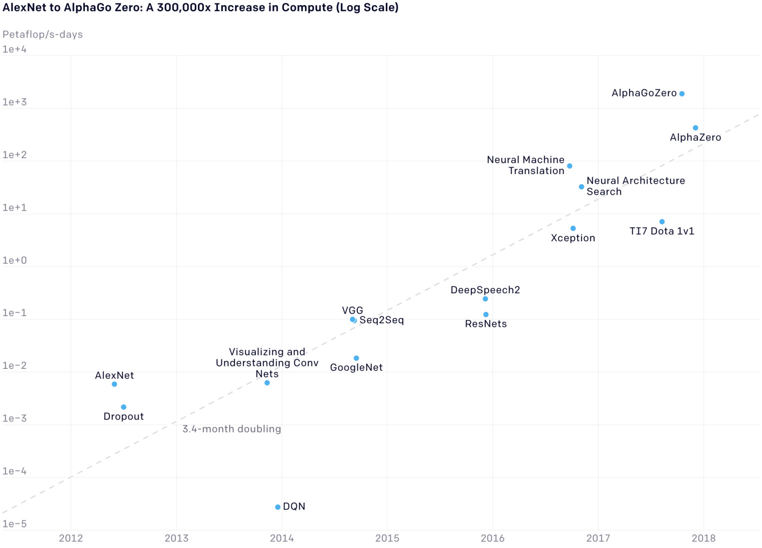
"Finally, after decades of research, deep learning, the abundance of data, the powerful computation of GPUs came together in a big bang of modern AI."

– Jensen Huang, CEO of Nvidia

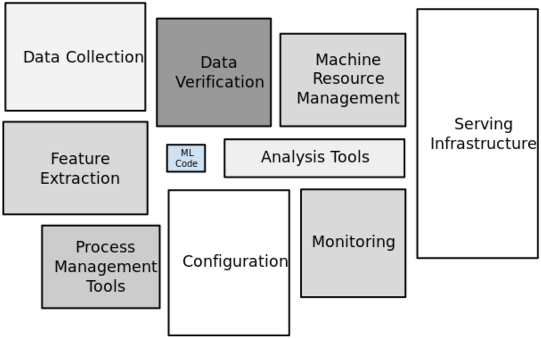
Growing Size of Deep Learning Models



Growing Compute of Deep Learning Models



Only 5% ML Code Exists in an ML System



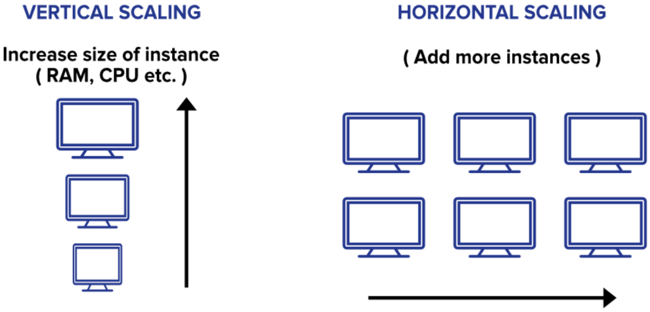
Source: Scullery et al., Hidden Technical Debt in Machine Learning Systems, NeurIPS 2015

(Top-Down) Levels of Applying Optimization

- **Node-level:** Scaling to multiple nodes – **scope:** $> 100\times$
- **Model-level:** Explore alternate models and/or model compression
  E.g.: Use EfficientNet-B0 instead of Resnet-50 – **scope:** $\sim 10\times$
- **Compiler/Kernel-level:** Execute alternate optimized kernels
  E.g.: FFT/Winograd based convolution – **scope:** $\sim 3 - 4\times$
- **Precision-level:** Apply precision levels below FP32
  Nvidia Volta – peak FLOPS: 15 TF vs peak OPS (in FP16): 120 TF – **scope:** $8\times$
  Nvidia Ampere – peak FLOPS: 19.5 TF vs peak OPS (in FP16, BFLOAT16): 312 TF (624 TF with sparsity) – **scope:** $32\times$

☞ We explored all levels for optimization; however, model-level was explored comparatively less

Scaling



- Vertical Scaling = Scaling Up, and Horizontal Scaling = *Scaling out*
- Henceforth, we are going to talk about horizontal scaling only

Scaling (contd.)

Amdahl's Law

$$speedup = \frac{1}{1 - p + \frac{p}{N}}$$

where, p is the parallelizable portion, and N is the number of nodes

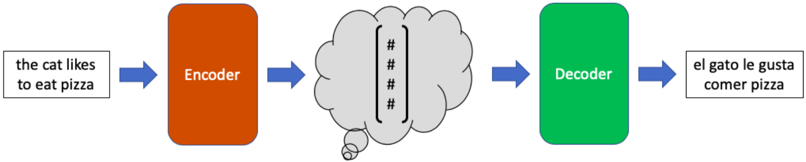
Example: If 80% of the process is parallizable (i.e., $p = 0.8$) and we have 4 nodes, then

$$speedup = \frac{1}{1 - 0.8 + \frac{0.8}{4}} = \frac{1}{0.2 + 0.2} = \frac{1}{0.4} = 2.5$$

In words, the *speedup* will be 2.5 times compared to a single node execution.

- **Strong Scaling:** How the solution time varies with the number of processors for a fixed *total* problem size.
- **Weak Scaling:** How the solution time varies with the number of processors for a fixed problem size *per processor*.

Scaling GNMT on an Intel CPU Cluster

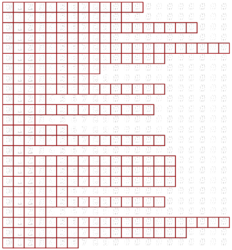


Translating English to Spanish

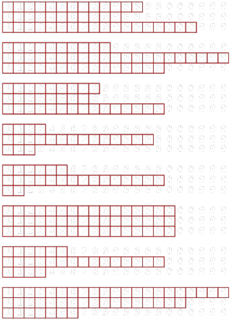
- Neural Machine Translation (NMT) is an approach to machine translation that uses an artificial neural networks
- Google's NMT (GNMT) is an LSTM based machine translation model
- Major steps taken to expedite training of GNMT workload:
  1. LSTM optimizations on x86 architecture (will cover in kernel-level)
  2. Load balancing dataset
  3. Distributed training with Horovod-MLSL

Load Balancing Dataset

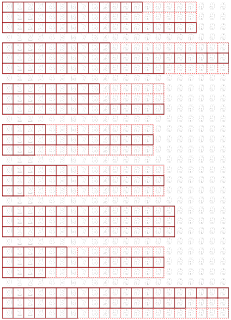
Training Dataset – Load Imbalance Problem



List of sentences



After naive batching (BS=3)

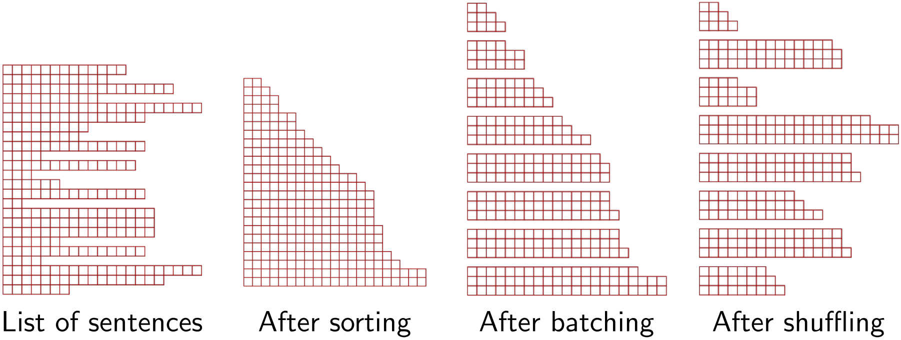


After padding

Lots of wasted compute due to padding to max sentence length within current batch

Load Balancing Dataset

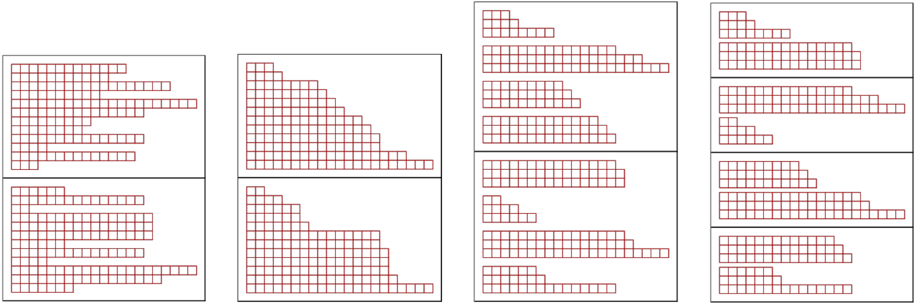
Training Dataset – Bucketing



Grouping similar length sentences together in a batch reduces wasted compute due to padding

Load Balancing Dataset

Multi-node Training Dataset



Divided list of sentences across nodes

After sorting on each node

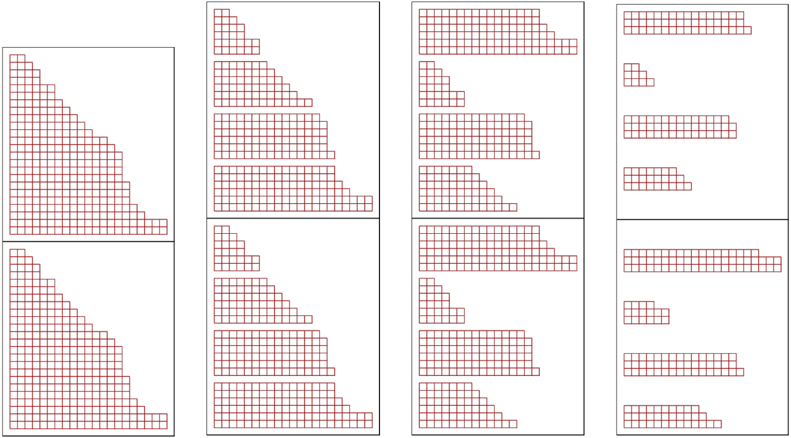
After batching & shuffling

View of global minibatch

Synchronization at gradient reduction causes wasted cycles due to different sequence length on different nodes

Load Balancing Dataset

Load Balanced Multi-node Training Dataset



Each node processes full list of sentences and picks global minibatch from a given bucket – then selects its own portion from global batch

Horovod-MLSL

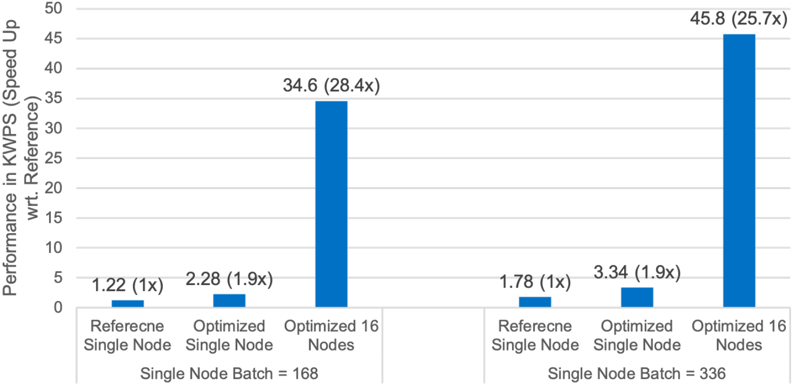
Horovod

- Originally designed by Uber for distribiuted deep learning using TensorFlow
- Provides concise API to modify TensorFlow model and make it run on multi-node clusters
- By default uses MPI as communication backend
- Additional communication optimizations, e.g., fusing several communication calls

Intel® Machine Learning Scaling Library (MLSL)

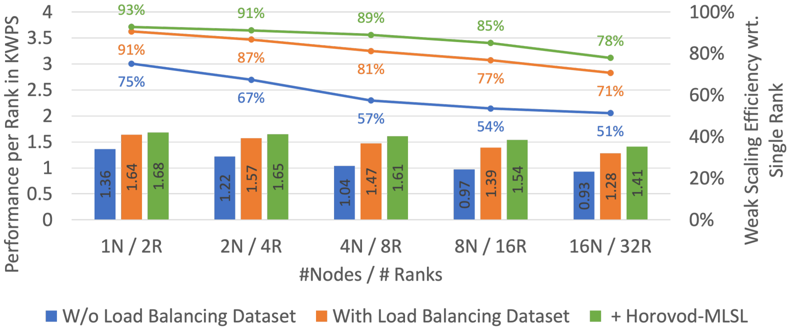
- We changed the MPI backend for Horovod with MLSL backend
- Trade off compute for performance by dedicating several cores to communication
- Advantageous for communication-bound models
- Currently, replaced by Intel® oneAPI Collective Communications Library (oneCCL)

Summary of Performance Optimizations



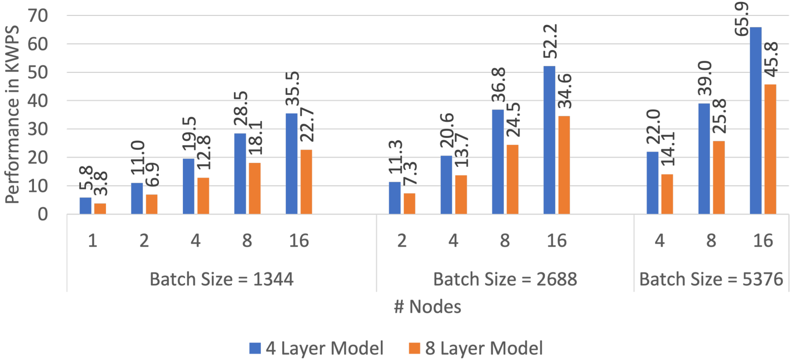
★ D Kalamkar, K Banerjee et al., "Training Google Neural Machine Translation on an Intel CPU Cluster," CLUSTER 2019

Weal Scaling up to 16 nodes



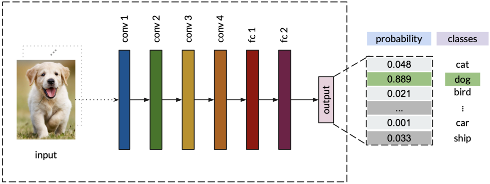
☞ 78% weak scaling efficiency from 1 node → 16 nodes

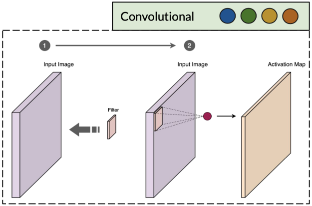
Strong Scaling up to 16 nodes

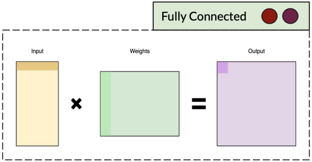


☞ 81% strong scaling efficiency from 1 node → 16 nodes

Convolution Neural Networks

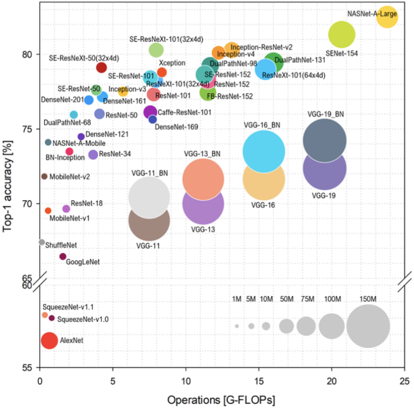






Squeezing Convolution Neural Networks

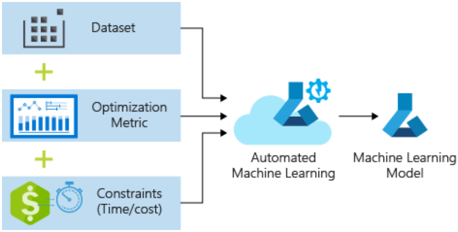
- Iandola et al., “SqueezeNet: AlexNet-level Accuracy with 50x Fewer Parameters and <0.5MB Model Size”, 2016
- Researchers use *model compression* and/or variants of convolution layer (e.g., depthwise separable convolution in MobileNet)



AutoML (Neural Architecture Search, AutoAI)

Goals of AutoML

1. Preprocess and clean the data
2. Select an appropriate model family
3. Optimize model hyperparameters
4. Design the topology of neural networks (if deep learning is used)
5. Analyze the results obtained

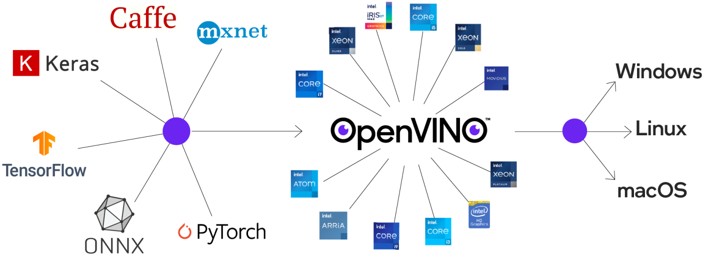


Source: `https://softwareengineeringdaily.com/2019/05/15/introduction-to-automated-machine-learning-automl/`

Compilers for Deep Learning

"The next war is compilers for the frameworks."

– Soumith Chintala, Distinguished Engineer, Facebook AI



Google: Tensorflow XLA & MLIR (absorbed by LLVM), **Amazon**: NNVM
Facebook: Glow, **Intel**: nGraph (now moved to OpenVino), Apache TVM

Bounds on Performance in HPC

- **Compute-bound:** Kernel spends most of its time in calculations, e.g., matrix multiplication – compute: $O(n^3)$, memory: $O(n^2)$
- **Bandwidth-bound:** Kernel spends most of its time in fetching the data, e.g., matrix addition – compute: $O(n^2)$, memory: $O(n^2)$
- **Latency-bound:** Kernel spends most of its time in memory fetches without saturating the global memory bus, e.g., accessing `A[B[i]]`

☞ Bandwidth-bound and latency-bound together constitute **memory-bound** kernels

☞ (Normal) Convolution is compute-bound except 1×1 convolutions, which being element-wise multiplications are memory-bound

☞ Personally, never worked with latency-bound kernels although these sometimes occur in HPC, e.g., FM-index based sequence search in computational-biology

☞ There are other types of bounds as well, e.g., I/O-bound

Bounds on Performance in HPC

- **Compute-bound:** Kernel spends most of its time in calculations, e.g., matrix multiplication – compute: $O(n^3)$, memory: $O(n^2)$
- **Bandwidth-bound:** Kernel spends most of its time in fetching the data, e.g., matrix addition – compute: $O(n^2)$, memory: $O(n^2)$
- **Latency-bound:** Kernel spends most of its time in memory fetches without saturating the global memory bus, e.g., accessing `A[B[i]]`

☞ Bandwidth-bound and latency-bound together constitute **memory-bound** kernels

☞ (Normal) Convolution is compute-bound except 1×1 convolutions, which being element-wise multiplications are memory-bound

☞ Personally, never worked with latency-bound kernels although these sometimes occur in HPC, e.g., FM-index based sequence search in computational-biology

☞ There are other types of bounds as well, e.g., I/O-bound

Convolution (direct)

Naive convolution loop structure

```
S1: for n = 0 ... N-1 do //no. of images (minibatch)
  S2: for k = 0 ... K-1 do //output feature maps
    S3: for c = 0 ... C-1 do //input feature maps
      S4: for oj = 0 ... P-1 do //output height
        S5: for oi = 0 ... Q-1 do //output weight
          S6: ij = stride * oj
          S7: ii = stride * oi
            S8: for r = 0 ... R-1 do //weight height
              S9: for s = 0 ... S-1 do //weight width
                S10: O[n][k][oj][oi] += I[n][c][ij+r][ii+s] * W[k][c][r][s]
```

- Output feature maps computed independently (data parallel fashion)
  - Block C and K loops by a factor of VLEN
  - Vectorize fused multiply-add (FMA) in line S10
- Register blocking in loops P and Q
  - Improve data reuse from registers
  - Decrease L1 traffic
  - Hide the latency of the FMA instructions

Convolution (direct) – after optimization

Convolution with vectorization and register blocking

```
S1: C_b = C / VLEN
S2: K_b = K / VLEN
S3: P_b = P / RB_P
S4: Q_b = Q / RB_Q
S5: for n = 0 ... N-1 do
  S6: for k_b = 0 ... K_b-1 do
    S7: for c_b = 0 ... C_b-1 do
      S8: for oj_b = 0 ... P_b-1 do
        S9: for oi_b = 0 ... Q_b-1 do
          S10: ij = stride * oj_b * RB_P
          S11: ii = stride * oi_b * RB_Q
          S12: oj = oj_b * RB_P
          S13: oi = oi_b * RB_Q
            S14: for r = 0 ... R-1 do
              S15: for s = 0 ... S-1 do
                S16: for k = 0 ... VLEN do
                  S17: for c = 0 ... VLEN do
                    S18: for p = 0 ... RB_P do
                      S19: for q = 0 ... RB_Q do
                        S20: ij' = ij + stride * p
                        S21: ii' = ii + stride * q
                          S22: O[n][k_b][oj+p][oi+q][k] += I[n][c_b][ij'+r][ii'+s][c] * W[k_b][c_b][r][s][c][k]
```

Tensor Layouts:

$I[N][C_b][H][W][VLEN]$, $O[N][K_b][P][Q][VLEN]$, $W[K_b][C_b][R][S][VLEN][VLEN]$

Convolution (direct) – after further optimization

Convolution with microkernel and layer fusion

```
S1: for n = 0 ... N-1 do
  S2: for k_b = 0 ... K_b-1 do
    S3: for c_b = 0 ... C_b-1 do
      S4: for oj_b = 0 ... P_b-1 do
        S5: for oi_b = 0 ... Q_b-1 do
          S6: ij = stride * oj_b * RB_P
          S7: ii = stride * oi_b * RB_Q
          S8: oj = oj_b * RB_P
          S9: oi = oi_b * RB_Q
          S10: CONV(&I[n][c_b][ij][ii][0], &W[k_b][c_b][0][0][0][0], &O[n][k_b][oj][oi][0])
          S11: if fuse(L()) AND c_b == C_b-1 then
            S12: APPLY (L(), &O[n][kb][oj][oi][0]))
```

- RB_P, RB_Q depend on convolution, VLEN depends on architechture $\Longrightarrow$ JITed microkernel
- Software Prefetching (to hide latency)
  - L1 cache prefetches — by same microkernel
  - L2 cache prefetches — by different microkernel
- Change loop ordering to maximize data reuse
  - Eg. For R=1, S=1 convolutions, pull in C_b loop $\Longrightarrow$ Increase output tensor reuse by a factor of C_b
- Parallelization strategy – $N \times K_b \times P_b \times Q_b$ independent microkernel invocations

Convolution (direct) – Results

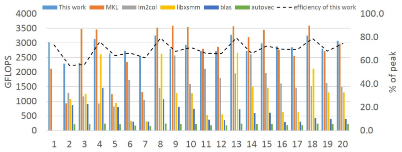


Figure: Forward Propagation on Skylake

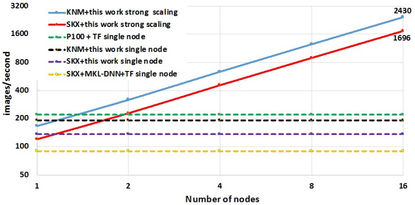
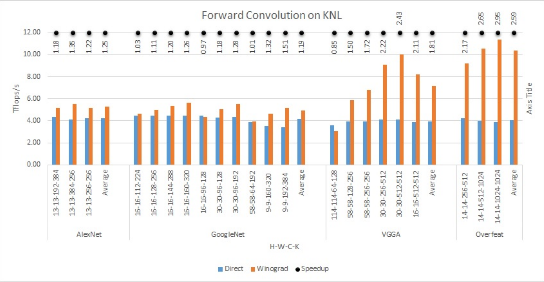


Figure: ResNet-50 training performance results

★ "Anatomy Of High-Performance Deep Learning Convolutions On SIMD Architectures," SC 2018

Convolution using Winograd

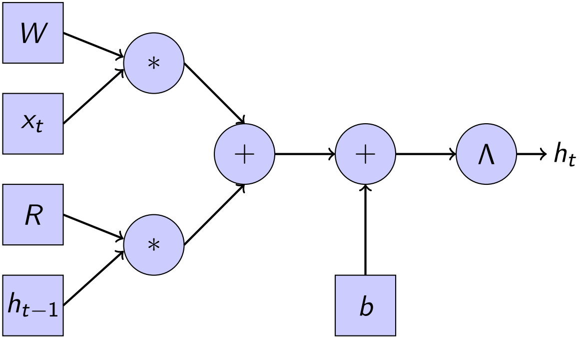


- Winograd – a special case of Convolution having filter size 3×3
- Complexity of multiplication: N.(H/m).(W/n).C.K.(m+R-1).(n+S-1), N - minibatch, (H, W) - (height, width) of feature map, (m, n) - (height, width) of tile, C - #input channels, K - #output channels, $R = S = 3$ (filter size)

★ "Understanding the Performance of Small Convolution Operations for CNN on Intel Architecture.," SC Poster 2017

Recurrent Neural Network (RNN)

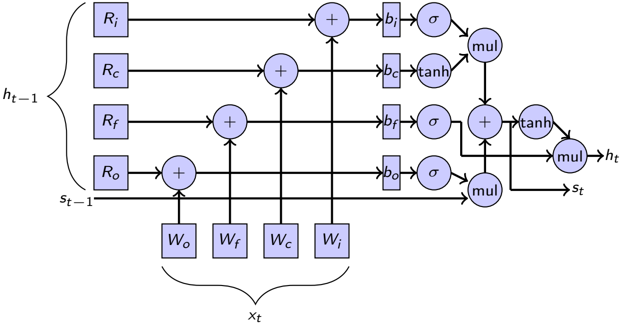
$$h_t = \Lambda(W * x_t + R * h_{t-1} + b)$$



$\Lambda \in \{\sigma, \tanh, \mathrm{ReLU}\}$, i.e., a non-linear activation function

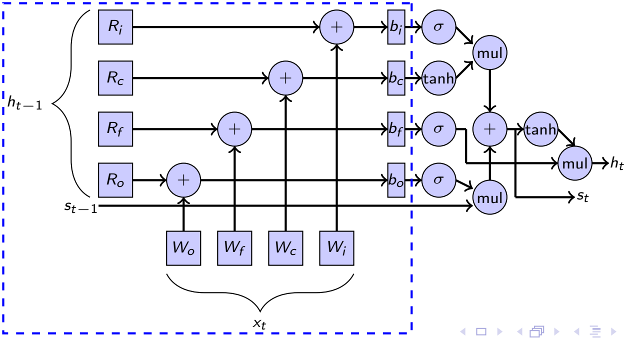
Long Short-Term Memory (LSTM)

$$
\begin{aligned}
i_t &= \sigma(W_i * x_t + R_i * h_{t-1} + b_i) \\
c_t &= \tanh(W_c * x_t + R_c * h_{t-1} + b_c) \\
f_t &= \sigma(W_f * x_t + R_f * h_{t-1} + b_f) \\
o_t &= \sigma(W_o * x_t + R_o * h_{t-1} + b_o) \\
s_t &= f_t \circ s_{t-1} + i_t \circ c_t \\
h_t &= o_t \circ \tanh(s_t)
\end{aligned}
$$



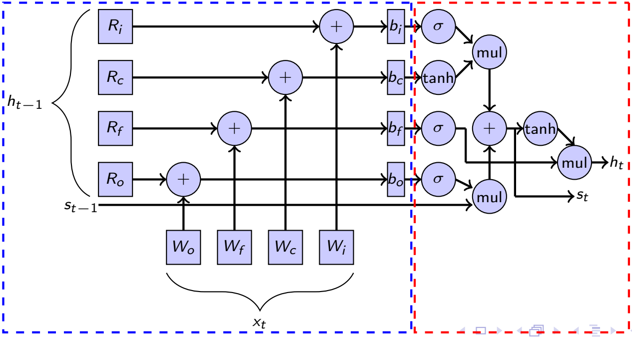
Long Short-Term Memory (LSTM)

$$i_t = \sigma(W_i * x_t + R_i * h_{t-1} + b_i)$$
$$c_t = \tanh(W_c * x_t + R_c * h_{t-1} + b_c)$$
$$f_t = \sigma(W_f * x_t + R_f * h_{t-1} + b_f)$$
$$o_t = \sigma(W_o * x_t + R_o * h_{t-1} + b_o)$$
$$s_t = f_t \circ s_{t-1} + i_t \circ c_t$$
$$h_t = o_t \circ \tanh(s_t)$$



Long Short-Term Memory (LSTM)

$$
\begin{aligned}
i_t &= \sigma(W_i * x_t + R_i * h_{t-1} + b_i) \\
c_t &= \tanh(W_c * x_t + R_c * h_{t-1} + b_c) \\
f_t &= \sigma(W_f * x_t + R_f * h_{t-1} + b_f) \\
o_t &= \sigma(W_o * x_t + R_o * h_{t-1} + b_o) \\
s_t &= f_t \circ s_{t-1} + i_t \circ c_t \\
h_t &= o_t \circ \tanh(s_t)
\end{aligned}
$$



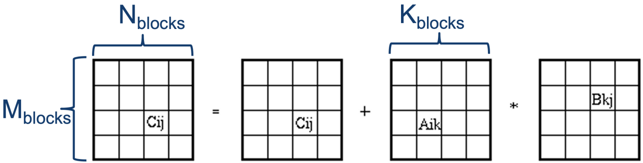
Implementation of RNN, LSTM, GRU

- Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) are special cases of RNN – same methodology applies in general
- Main computation consists of GEMMs $W_{\{i,f,o,c\}} * x_t$ and $R_{\{i,f,o,c\}} * h_{t-1}$
- Element-wise operations are applied to the GEMM results
- Analogous equations for back-propagation kernels
- Perform two large GEMMs ($W * x$ and $R * h$) or one larger GEMM ($WR * xh$), then perform element-wise operations
  - ✓ Easy to implement – rely on vendor-optimized GEMM
  - ✗ Element-wise operations exposed as bandwidth-bound kernel (vs in-cache reuse of GEMM outputs)

Implementation of RNN, LSTM, GRU

- Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) are special cases of RNN – same methodology applies in general
- Main computation consists of GEMMs $W_{\{i,f,o,c\}} * x_t$ and $R_{\{i,f,o,c\}} * h_{t-1}$
- Element-wise operations are applied to the GEMM results
- Analogous equations for back-propagation kernels
- Perform two large GEMMs ($W * x$ and $R * h$) or one larger GEMM ($WR * xh$), then perform element-wise operations
  - ✓ Easy to implement – rely on vendor-optimized GEMM
  - ✗ Element-wise operations exposed as bandwidth-bound kernel (vs in-cache reuse of GEMM outputs)

Batch-Reduce GEMM



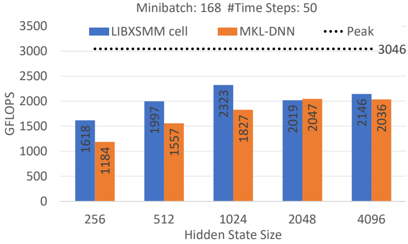
$$C_{ij} = C_{ij} + \sum_{k=1}^{K_{blocks}} A_{ik} * B_{kj}$$

Batch reduce GEMM

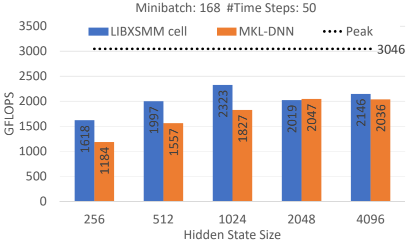
- ☞ The single batch-reduce GEMM kernel can act as the innermost kernel for a variety of deep learning topologies (CNN, RNN, LSTM, GRU, MLP) delivering SOTA performance
- ☞ Boosts **programmer productivity** which otherwise is spent tuning various kernels across different topologies
- ☞ All code available in `https://github.com/hfp/libxsmm`
- ★ E Georganas, K Banerjee et al., "Harnessing Deep Learning via a Single Building Block," IPDPS 2020

Comaprison with Intel® MKL-DNN

☞ Intel® MKL-DNN is an open source performance library from Intel



Forward pass results

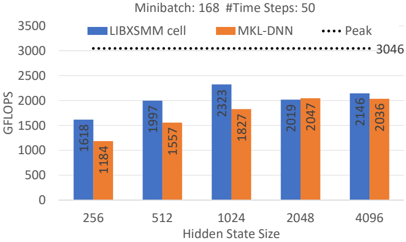


Backward/weight update pass results

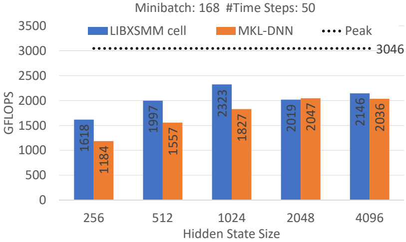
- **Machine:** Single socket Xeon Platinum 8180 with 28 Cores
- For small/medium sized problems, forward pass is up to 1.4× faster, while for backward/weight update it is up to 1.3× faster
- For large weight matrices the two approaches have similar performance because GEMM has *cubic* scaling whereas element-wise has *quadratic* scaling

Comaprison with Intel® MKL-DNN

☞ Intel® MKL-DNN is an open source performance library from Intel



Forward pass results



Backward/weight update pass results

★ K Banerjee et al., "Optimizing Deep Learning LSTM Topologies onIntel Xeon Architecture," ISC 2019 (Best Research Poster – AI & ML track)

★ K Banerjee et al., "Optimizing Deep Learning RNN Topologies on Intel Architecture," JSFI 2019 (invited paper)

Using Off-the-shelf Library: DeepSpeed

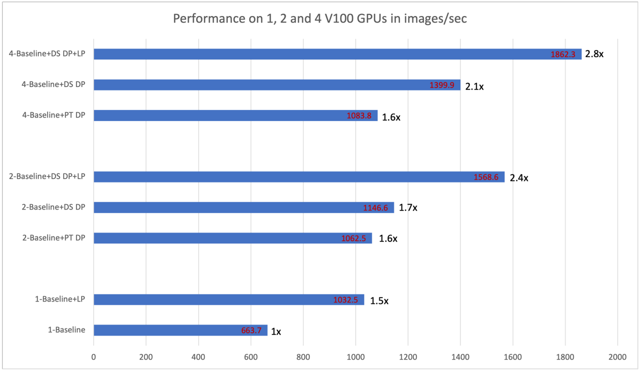


Figure: Performance of DeepSpeed's data parallelism and low-precision

- ☞ One need not be an expert – just pick off-the-shelf components
- ★ "From Pixels To Words: A Scalable Journey Of Text Information From Product Images To Retail Catalog," CIKM 2021

Low-Precision Datatype & bfloat16

Low-Precision: Any datatype below FP32

FP32	s	8 bit exp	23 bit mantissa
FP16	s	5 bit	10 bit
BF16	s	8 bit	7 bit

BF16 has several advantages over FP16:

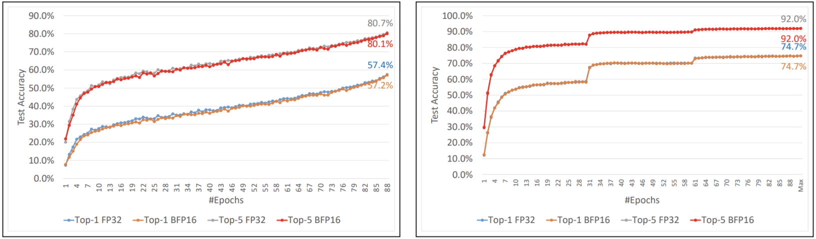
- It can be seen as a short version of FP32, skipping the least significant 16 bits of mantissa
- There is no need to support denormals; FP32, and therefore also BF16, offer more than enough range for deep learning training tasks
- Hardware exception handling is not needed

Source: https://software.intel.com/sites/default/files/managed/40/8b/bf16-hardware-numerics-definition-white-paper.pdf

Intel's 3rd Gen Intel® Xeon® Scalable processor (codenamed Cooper Lake) launched in 2020 includes bfloat16

Source: https://www.intel.in/content/www/in/en/products/docs/processors/xeon/3rd-gen-xeon-scalable-processors-brief.html

Deep Learning Training with bfloat16

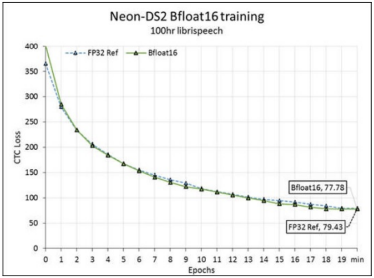


(a) AlexNet

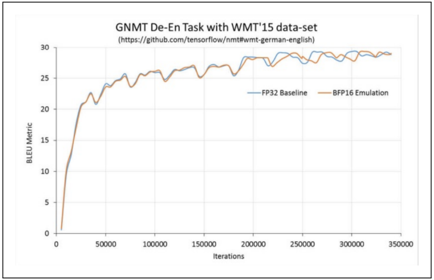
(b) ResNet-50

Figure: Imagenet-1K training, top-1 and top-5 validation accuracy plots for CNNs

Deep Learning Training with bfloat16



(a) DeepSpeech2



(b) GNMT

Figure: RNN training using bfloat16 data type

★ "A Study of BFLOAT16 for Deep Learning Training," 2019

Dynamic Fixed Point–16

☞ Integer based ALUs require less area and less power compared to foating point based ALUs

DFP16	s	15 bit mantissa	
	s	15 bit mantissa	
	s	15 bit mantissa	s 7 bit
	s	15 bit mantissa	

$E_{f_{max}} = E(\max\limits_{\forall f \in F} |f|), F$ is a floating point tensor
$E_s = E_{f_{max}} - (P - 2), P$ is the $\#$ of bits used
$\forall i_n \in I, f_n = i_n \times 2^{E_s}$, where $f_n \in F$
Multiplication: $i_{ab} = i_a \times i_b$ and exponent $E_s^{ab} = E_s^a + E_s^b$
Addition:

$$i_{a+b} = \begin{cases} i_a + (i_b >> (E_s^a - E_s^b)), & \text{if } E_s^a > E_s^b \\ i_b + (i_a >> (E_s^b - E_s^a)), & \text{if } E_s^b > E_s^a \end{cases}$$

and exponent $E_s^{a+b} = \max\limits_{E_s^a, E_s^b}$

CNN Training with DFP-16

Table: Training configuration and ImageNet-1K classification accuracy

Model	Batch-size/Epochs	Baseline		DFP-16	
		Top-1	Top-5	Top-1	Top-5
ResNet-50	1024/90	75.70%	92.78%	**75.77%**	**92.84%**
GoogLeNet-v1	1024/80	69.26%	**89.31%**	**69.34%**	**89.31%**
VGG-16	256/60	**68.23%**	**88.47%**	68.12%	88.18%
AlexNet	1024/88	**57.43%**	**80.65%**	56.94%	80.06%

☞ No change in hyper-parameters, and trained in as many iterations as FP32 baseline

☞ Batch norm layer is in FP32

☞ Speed up of 1.8× over FP32 baseline for Resnet-50 on Intel® XeonPhi™ Knights-Mill

★ "Mixed Precision Training of Convolutional Neural Networks using Integer Operations," ICLR 2018

A Perturbation Theory View of Low-Precision Inference

A Perturbation Theory View of Low-Precision Inference

fire engine (8a-2w)

A Perturbation Theory View of Low-Precision Inference

fire engine (8a-2w)

A Perturbation Theory View of Low-Precision Inference

ice bear (8a-4w)

A Perturbation Theory View of Low-Precision Inference

ice bear (8a-4w)

A Perturbation Theory View of Low-Precision Inference

old English sheepdog (8a-6w)

A Perturbation Theory View of Low-Precision Inference

old English sheepdog (8a-6w)

A Perturbation Theory View of Low-Precision Inference

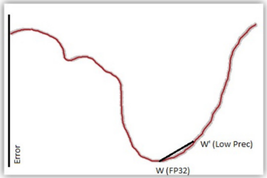
dalmatian (8a-8w)

A Perturbation Theory View of Low-Precision Inference

dalmatian (8a-8w)

A Perturbation Theory View of Low-Precision Inference

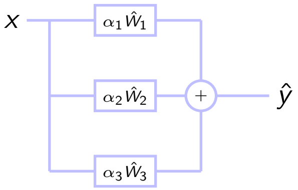
- Low precision (and sparsification): adding noise to weights/activations
- Need to preserve the output of each layer such that no/little loss of accuracy
- 8-bit weight and 8-bit activations (8a-8w) shown to work well
  - $\sim$1% loss of accuracy from FP-32 on very deep CNNs (e.g. ResNet101)
- Sub-8-bit weights and/or activations incur noticeable drop in accuracy
- Our observation:
  - Not all the weights may be well-represented by sub-8-bit precision
  - Not all the weights may need 8-bit precision
- Our approach: Keep low-precision weights in a neighborhood of FP32 weights using only 8-bit activations and Ternay weights



Ternary Residual Inference

Ternary Weights: $\alpha \hat{W} \simeq W, \hat{W}_i = sign(W_i)$, if $|W_i| > T$, and 0 otherwise

$E(\alpha, T) = \|W - \alpha \hat{W}\|_F^2, \alpha^*, T^* = \underset{\alpha \geq 0, T \geq 0}{argmin}\, E(\alpha, T), \alpha \geq 0, \hat{W}_i \in \{-1, 0, +1\}$



Theoretical understanding of sensitivity of weights and/or activations to final classification accuracy

- Highly sensitive layers (e.g. first conv layer) require more precision
- We want uniform low-precision operations rather than multi-precision

Ternary Residual Edge: add more ternary edges selectively s.t.

- More ternary compute for certain parts of the augmented network
- Significant recovery of loss with overall less compute cost than 8a-8w

★ A Kundu, K Banerjee et al., "Ternary Residual Networks," SysML 2018

Ternary Residual Inference (contd.)

- Further fine-tuning via block-wise ternary residual
  - Partition the weights in disjoint blocks (of N elements)
  - Convert to ternary, and add r number of residual blocks (if necessary)
- N and r control the model size, accuracy, and ternary compute
- We can adjust the accuracy-compute trade-off on-the-fly during inference (unlike other models) by disabling least important residual blocks
- Results of Ternary Residual conversion of ResNet-101 pre-trained on ImageNet
  - Total number of blocks (without residual) is $1\times$
  - Overall $\sim 2\times$ savings in compute cost comparing to 8a-8w with similar accuracy

Block Size	$\sim 1\%$		$\sim 2\%$	
	#Blocks	mult reduction	#Blocks	mult reduction
$N = 64$	$2.4\times$	$26\times$	$2\times$	$32\times$
$N = 256$	$2.8\times$	$90\times$	$2.4\times$	$105\times$

Other Works

Function Approximation:

- $\sigma(x) = (1 + tanh(x/2))/2$
- $swish(x) = x.\sigma(x) = x.(1 + tanh(x/2))/2$
- $\mathrm{GELU}(x) = x.\Phi(x) \approx \frac{x}{2}(1 + tanh(\sqrt{2/\pi}(x + a.x^3)))$

★ "K-TanH: Efficient TanH for Deep Learning," arXiv 2020

- $\mathrm{softmax(z)}_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$
- Taylor series expansion till n^{th} order: $e^z \approx f^n(z) = \sum_{i=0}^{n} \frac{z^i}{i!}$
- Taylor $\mathrm{softmax(z)}_i = \frac{f^n(z_i)}{\sum_{j=1}^{K} f^n(z_j)}$

★ K Banerjee et al., "Exploring Alternatives to Softmax Function," DeLTA 2021 (nominated for Best Poster Award)

Fault Tolerance:

- Explore transient faults – single bit flip
- Compare resilience of pruned and quantized deep learning models

★ "Reliability Evaluation of Compressed Deep Learning Models," LASCAS 2020

Node-level Open Problems

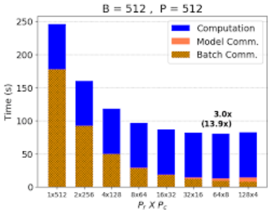
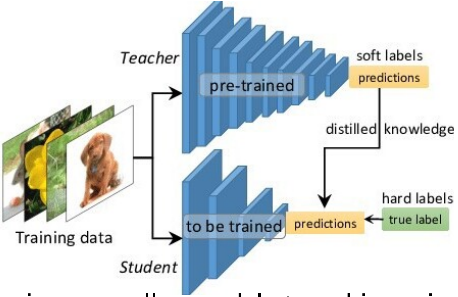


Figure: Communication overtakes computation during scaling

- Can we hide the communication time while doing computation?
- Can we decrease the amount of gradient exchange by:
  - compressing the gradients?
  - using low-precision?
  - sharing it intermittently?
- How to adapt DL training for **Federated learning** (data stored in local devices and not shared, e.g., in healthcare for data privacy)?

Model-level Open Problems



- How can we design a smaller models to achieve similar accuracy using knowledge distillation, low-precision and/or pruning?
- How can we devise better heuristics to avoid combinatorial explosion while:
    - searching appropriate layers?
    - applying low-precision?
    - tuning hyper-parameters?
- How to design models for **TinyML** (loaded into micro-controllers)?

Compiler/Kernel-level Open Problems

Temporal Definition

```
C(i, j) = 0;
C(i, j) += A(i, k) * B(k, j);
```

Spatial Mapping

```
C.update().tile(k, j, i, kk, jj, ii, KK, JJ, II);
                .isolate_producer_chain(A, A_loader, A_feeder)
                .isolate_producer_chain(B, B_loader, B_feeder)
                .isolate_consumer_chain(C, C_drainer, C_unloader);
A_loader.unroll(ii).remove(jj).vload(kk);
A_feeder.buffer(ii, Buffer::Double).unroll(ii);
B_loader.unroll(jj).remove(ii).vload(kk);
B_feeder.buffer(k, Buffer::Double).unroll(jj);
C.update().unroll(jj, ii)
               .forward(A_feeder, {1, 0}) .forward(B_feeder, {0, 1});
C_drainer.unroll(jj, ii).gather(C, {1, 0})
C_unloader.buffer(ii).unroll(ii).vstore(jj);
```

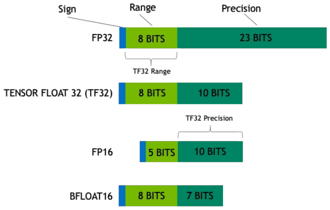
Temporal to Spatial ⟶ T2S

Figure: T2S, AutoSA, HeteroCL – libraries built primarily on top of Halide to automatically generate code for spatial architectures

- *Meta-compiler:* How can we automate the process of updating these libraries/compilers for new generations of hardware?
- How can we adapt compiler optimizations for low power (ideally, without sacrificing performance) to support **Green AI**?
- How can we formally/semi-formally verify these compilers?

★ K Banerjee et al., "A Quick Introduction to Functional Verification of Array-Intensive Programs," 2019

Precision-level Open Problems



- Can we develop a theoretical framework to argue about the various datatypes?
- We say that regularization induced by low precision sometimes lead to better accuracy – however, do we really understand it?
- Can we have hierarchical accumulators for low precision, e.g., FP8 $\rightarrow$ FP16 $\rightarrow$ FP32?

"Big data isn't about bits, it's about talent."
– Douglas Merrill, CEO, Zest AI

Thank you!

☞ https://kunalbanerjee.github.io/
✉ kunal.banerjee1@walmart.com