KAISA: An Adaptive Second-Order Optimizer Framework for Deep Neural Networks

J. G. Pauloski [1], Q. Huang [2], L. Huang [3], S. Venkataraman [4], K. Chard [1,5], I. T. Foster [1,5], Z. Zhang [3]

[1] University of Chicago
[2] University of Texas at Austin
[3] Texas Advanced Computing Center
[4] University of Wisconsin, Madison
[5] Argonne National Laboratory

HPC and Machine Learning

Training with **large batch sizes** (e.g., 100K to 1M) is key to machine learning training on HPC.

- ++ Used to **scale out** to more nodes.
- -- Leads to **worse generalization** performance.

How can we better enable large batch training on HPC?

Second-Order Optimization

A good candidate for large batch, distributed training!

1. Larger batches are more representative of the dataset's distribution.

   → *infrequent second-order information updates*

2. Gradient noise limits batch size and increases throughout training (McCandlish, 2018).

   → *second-order methods optimize noise-independent terms better* (Martens, 2014)

3. Higher computation-to-communication ratio in second-order methods.

   → *enables more advanced distribution schemes*

Kronecker-Factored Approximate Curvature

- Second-order methods incorporate the curvature of the parameter space.
  - ++ More progress optimizing the objective function per-iteration
  - -- **Expensive to compute!**

- K-FAC **efficiently approximates** the Fisher Information Matrix (FIM) for preconditioning the gradients (Martens+, 2015).

$$\text{SGD: } w^{(k+1)} = w^{(k)} - \frac{\alpha^{(k)}}{n} \sum_{i=1}^{n} \nabla L_i(w^{(k)}) \qquad \text{K-FAC: } w^{(k+1)} = w^{(k)} - \frac{\alpha^{(k)} F^{-1}(w^{(k)})}{n} \sum_{i=1}^{n} \nabla L_i(w^{(k)})$$

- Generalizes better with **large batch sizes** and **converges in fewer iterations** than first-order methods (Ba+, 2017)
  - Scales to extremely large batch sizes, e.g., 131k for ImageNet training (Osawa+, 2019)

Goals

- Investigate the **tradeoffs between memory and communication** in distributed second-order optimization.
- Design a K-FAC distribution scheme that can **adapt the ratio of memory and communication**.
- Implemented as a gradient preconditioner than can be **used in place with existing optimizers**.
- First to show **K-FAC provides speedups** in wide variety of applications and hardware environments.

Kronecker Product

$$A \otimes B = \begin{bmatrix} a_{11}B & \dots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \dots & a_{mn}B \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \otimes \begin{bmatrix} 5 & 6 \\ 7 & 8 \\ 9 & 0 \end{bmatrix} = \begin{bmatrix} 1\times5 & 1\times6 & 2\times5 & 2\times6 \\ 1\times7 & 1\times8 & 2\times7 & 2\times8 \\ 1\times9 & 1\times0 & 2\times9 & 2\times0 \\ 3\times5 & 3\times6 & 4\times5 & 4\times6 \\ 3\times7 & 3\times8 & 4\times7 & 4\times8 \\ 3\times9 & 3\times0 & 4\times9 & 4\times0 \end{bmatrix}$$

$m \times n \otimes p \times q \longrightarrow mp \times nq$

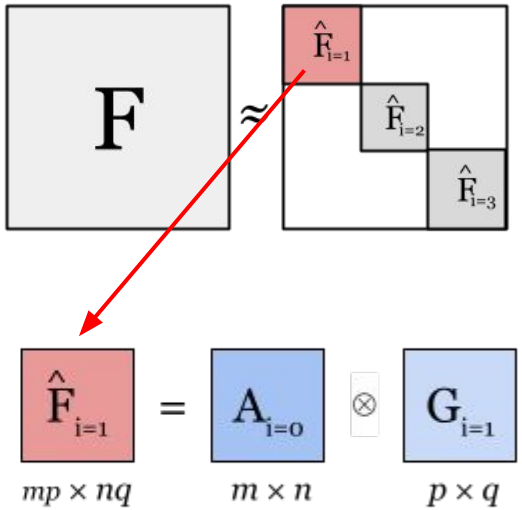
Efficient F Approximation

Step 1: Approximate the FIM as a block diagonal matrix*

$$\hat{F} = \texttt{diag}(\hat{F}_1, ..., \hat{F}_i, ..., \hat{F}_L)$$

Step 2: Decompose each block as the Kronecker Product of the activations of the previous layer with the gradient w.r.t. the output of the current layer

$$\hat{F}_i = a_{i-1}a_{i-1}^\top \otimes g_i g_i^\top = A_{i-1} \otimes G_i$$



**Recall inverse of block diagonal matrix is composed of the inverses of each block*

Efficient Gradient Preconditioning

Step 3: Apply properties of Kronecker Product to weight update equation

Properties: $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$ $\quad (A \otimes B)\vec{c} = B^{\top}\vec{c}A$

Weight Update: $w_i^{(k+1)} = w_i^{(k)} - \alpha^{(k)} \hat{F}_i^{-1} \nabla L_i(w_i^{(k)})$

$$\hat{F}_i^{-1} \nabla L_i\left(\omega_i^{(k)}\right) = G_i^{-1} \nabla L_i(\omega_i^{(k)})\, A_{i-1}^{-1}$$

Preconditioned Gradient

**Alternatively, gradient preconditioning can be performed with eigen decompositions of the factors.*

Distributed K-FAC

Ba, Grosse, and Martens

(ICLR 2017)

- TensorFlow Parameter Server
- Scaled to 8 GPUs on a single node

Osawa et al.

(CVPR 2019)

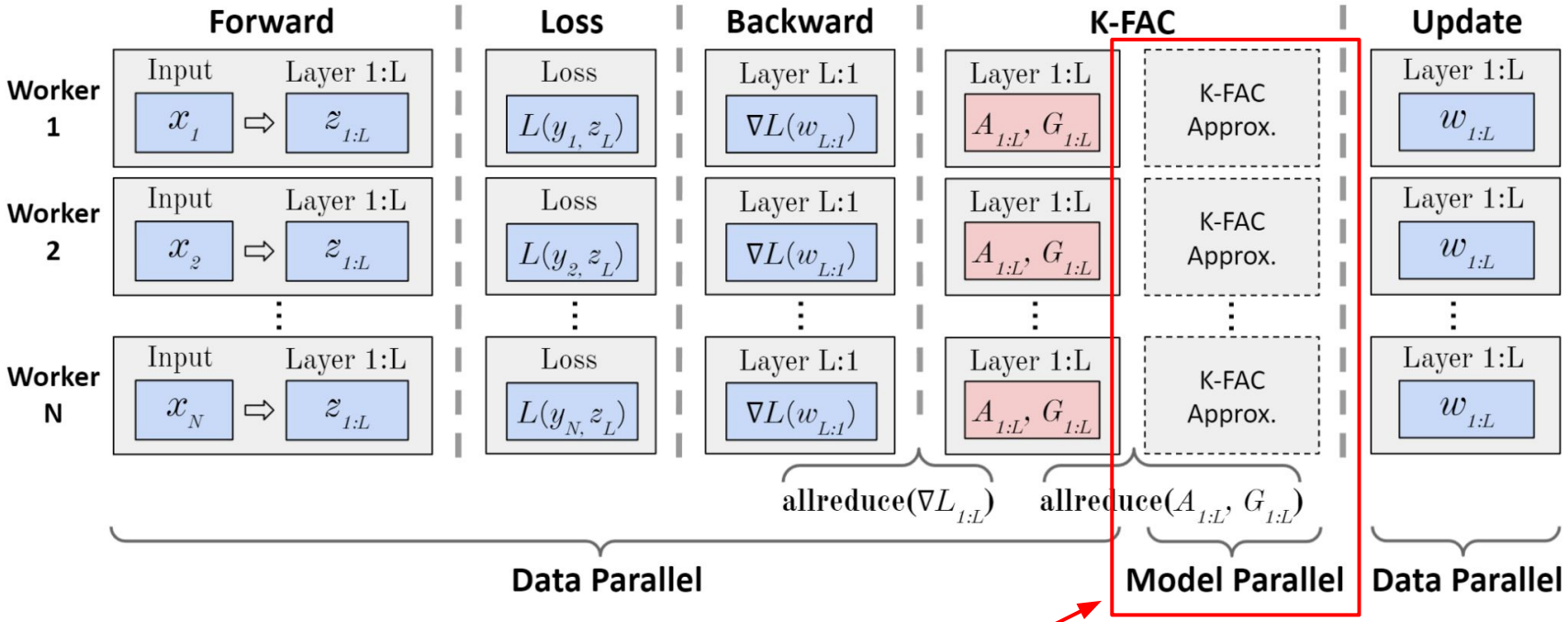
- Synchronous, data-parallel scheme in Chainer
- Layer-wise distribution scheme for matrix inversion/preconditioning
- 75% validation accuracy for ResNet-50/ImageNet in 978 iterations w/ 131k batch size
- Later work converges to 76% on ResNet-50/ImageNet in 2 minutes on 2048 V100 GPUs
- **Lower memory footprint**

Pauloski et al.

(SC 2020)

- Synchronous, data-parallel scheme in PyTorch/Horovod
- Decoupled matrix inversion from preconditioning
- Used eigen decomposition preconditioning method
- K-FAC converges 18--25% faster than SGD across scales
- **Less communication**

Hybrid-Parallel K-FAC

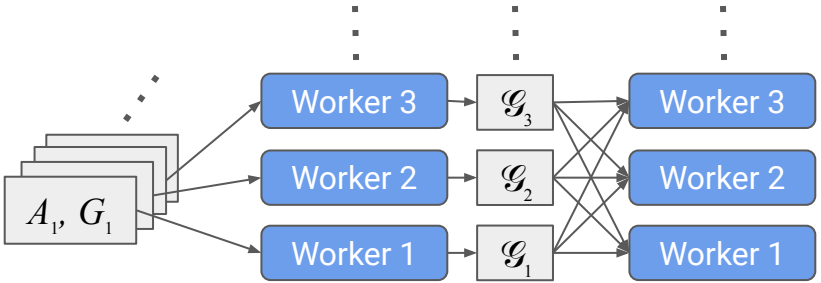


Performed infrequently (e.g., every *t*=100 iterations)

Model Parallel K-FAC Stage

MEM-OPT

Osawa et al. (CVPR 2019)

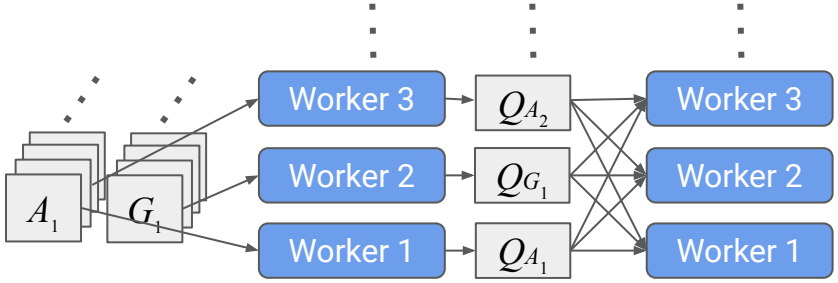


++ Lower memory usage

-- Communication required every iteration

COMM-OPT

Pauloski et al. (SC20)



++ Communication every *t* iterations

-- Higher memory usage due to caching

An Adaptable K-FAC Framework

Given more available memory, such as moving from a 16GB V100 to a 40GB A100, should you?

a) increase the batch size, or
b) use more memory for K-FAC (i.e., MEM-OPT ➡ COMM-OPT)?

What if you need more fine-grained control of K-FAC memory usage?

KAISA: a **K**-FAC-enabled, **A**daptable, **I**mproved, and **S**c**A**lable distributed, second-order optimizer framework

Goals

- **Adaptable K-FAC** distribution scheme that **generalizes existing work**
- Implemented as a preconditioner **easily** plugged into **existing scripts**
- Show KAISA provides **speedups** across a **greater variety** of applications
- **Investigate tradeoffs** between memory footprint and communication

```
model = DistributedDataParallel(model)
optimizer = optim.SGD(model.parameters(), ...)
preconditioner = KFAC(model, grad_worker_frac=0.5)

for data, target in train_loader:
    optimizer.zero_grad()
    output = model(data)
    loss = criterion(output, target)
    loss.backward()

    preconditioner.step()
    optimizer.step()
```

Listing 1: Example K-FAC usage.

Distribution of Work

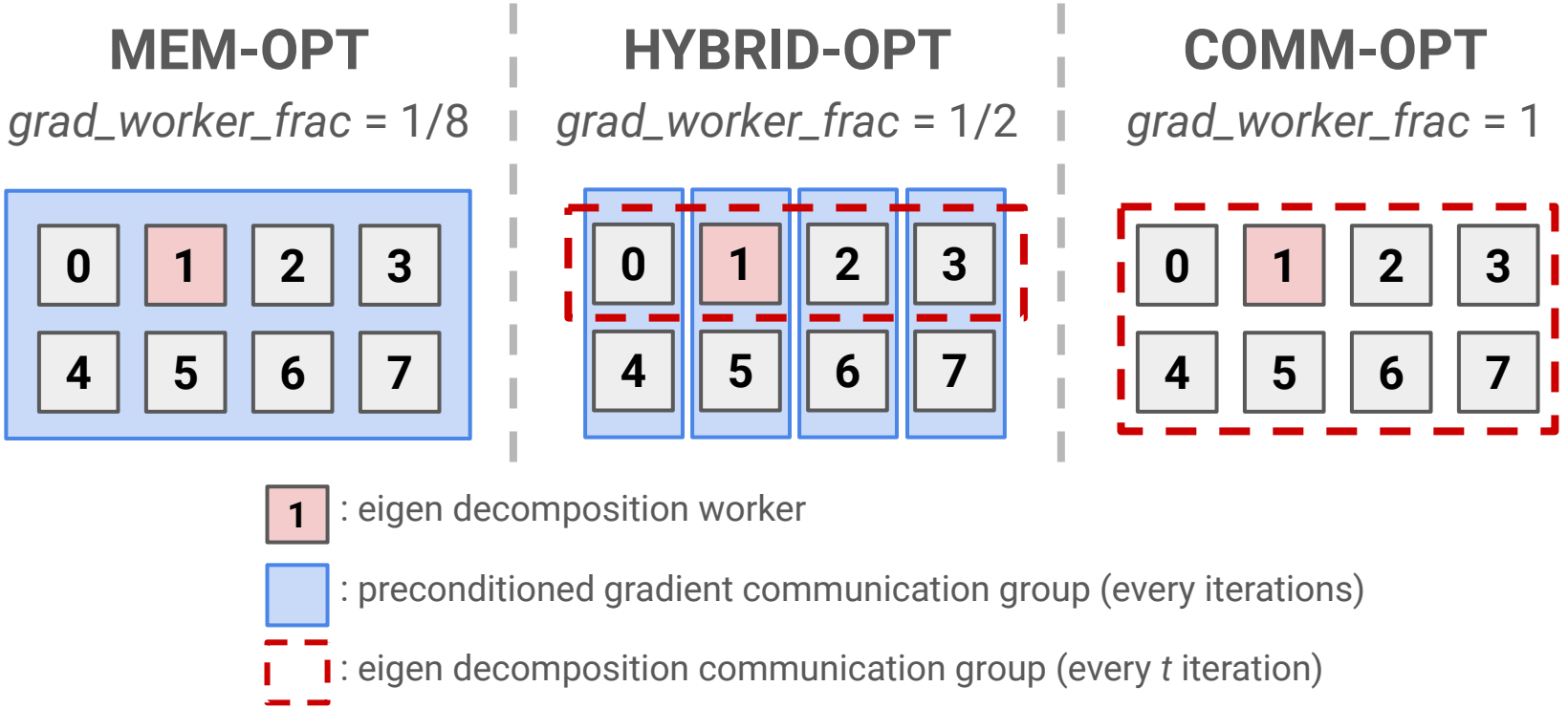
Worker Types for a Layer (*N* workers)

Gradient Worker	Eigen Decomp Worker	Gradient Receivers
Compute preconditioned gradient with eigen decompositions	Compute eigen decompositions (one of the gradient workers)	Receives preconditioned gradients from a gradient worker
m per layer	**1 or 2** per layer	***N - m*** per layer

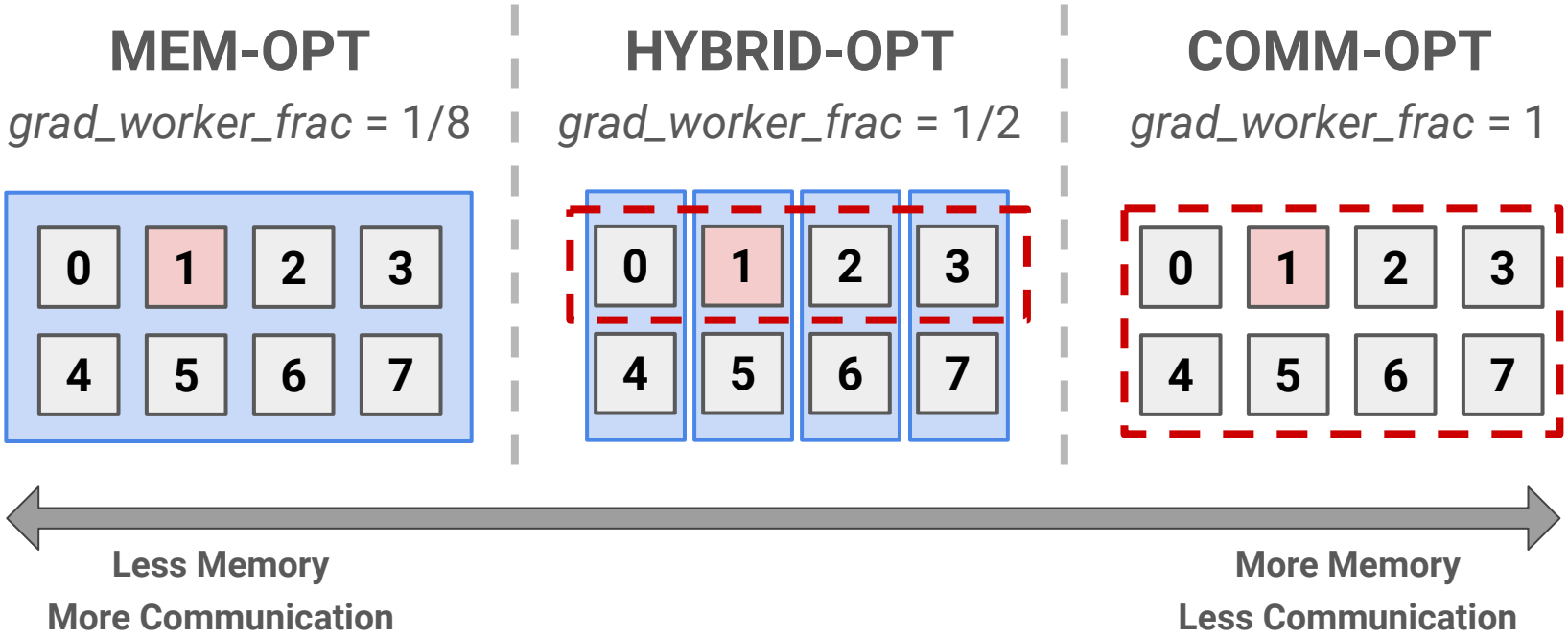
Gradient Worker Fraction (*grad_worker_frac*) = ***m / N***

MEM-OPT: *grad_worker_frac* = **1/N**
COMM-OPT: *grad_worker_frac* = **1**

Gradient Worker Fraction



Gradient Worker Fraction



Additional KAISA Improvements

- Greedy factor distribution: assign most “expensive” factor to worker with lowest current load.
  - Compute time: approximate eigen decomposition time as $O(n^3)$.
  - Memory usage: $O(n^2)$.
- Half-precision storage and computation for factors.
- Preconditioned Gradient Precomputation (up to 53% faster preconditioning)

$$V_1 = Q_G^\top \nabla L_i(w_i^{(k)}) Q_A$$
$$V_2 = V_1 / (v_G v_A^\top + \gamma)$$
$$(\hat{F}_i + \gamma I)^{-1} \nabla L_i(w_i^{(k)}) = Q_G V_2 Q_A^\top$$

Evaluation: Setup

Hardware

- GPU Subsystem of Frontera at TACC: 112 nodes with 4xV100 GPUs
- GPU Subsystem of Theta at Argonne National Lab: 24 DGXA100 nodes (8xA100 40GB GPUs)

Applications

- *Classification*: **ResNet**-{18,50,101,152} with ImageNet
- *Segmentation*: **Mask R-CNN** with COCO 2014 and **U-Net** with MRI tumor dataset
- *Language Modeling*: **BERT-Large** with English Wikipedia+Toronto BooksCorpus

Evaluation: Fixed Batch Size

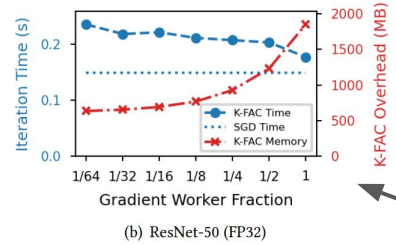
App	Default Optimizer	Baseline	# GPUs	Global Batch Size	Precision	KAISA Time-to-Convergence Improvement
ResNet-50	SGD	75.9% Val. Acc.	8 A100	2048	FP16	**24.3%**
Mask R-CNN	SGD	0.377 bbox mAP 0.342 segm mAP	64 V100	64	FP32	**18.1%**
U-Net	ADAM	91.0% Val. DSC	4 A100	64	FP32	**25.4%**
BERT-Large (Phase 2)	LAMB	90.8% SQuAD v1.1 F1	8 A100	65,636	FP16	**36.3%**

Evaluation: Fixed Memory Budget

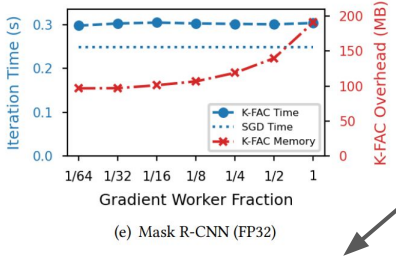
App	Optimizer	GPUs	Grad. Worker Frac.	Local Batch Size	Time-to-Convergence (minutes)
ResNet-50	SGD	64 V100	--	128	**123 (DNC)**
	KAISA		1/64	80	**96**
	KAISA		1/2	80	**83**
BERT-Large (Phase 2)	LAMB	8 A100	--	12	**2918**
	KAISA		1/2	8	**1703**
	KAISA		1	8	**1704**

* Use max possible local batch size for each experiment and measure time-to-convergence.

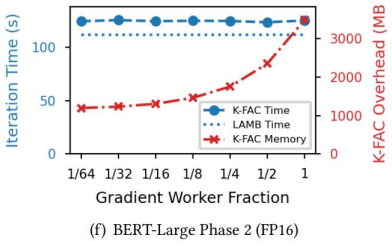
Evaluation: Memory vs. Communication



(b) ResNet-50 (FP32)



(e) Mask R-CNN (FP32)



(f) BERT-Large Phase 2 (FP16)

App	K-FAC Memory Overhead	Seconds between calls to KFAC.step()	K-FAC Bandwidth Requirements
ResNet-50	600 MB --- 1.8 GB	0.2 seconds	High
Mask R-CNN	100 MB --- 200 MB	0.33 seconds	Low
BERT-Large (Phase 2)	1.3 GB --- 3.8 GB	120 seconds	Low

High communication applications have **strong memory/communication tradeoffs**.

Low communication applications are **not as affected** by communication reductions with larger *grad_worker_frac* values.

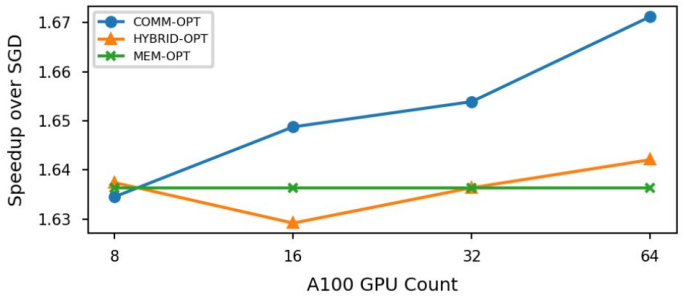
KAISA is still **faster than baseline optimizers** for all *grad_worker_frac*.

Evaluation: Scaling

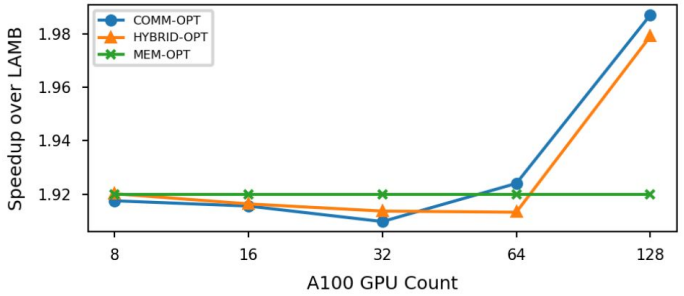
KAISA variant **speedups over SGD** with a **high communication** (ResNet-50) and **low communication** (BERT-Large) application.

Observations:

- 27-29% **faster than SGD** with ResNet-50
- 41-44% **faster than LAMB** for BERT-Large
- MEM-OPT has constant speedup
- HYBRID/COMM-OPT **improve with scale**
- HYBRID-OPT **best balance** of memory usage and scaling with BERT-Large



(a) ResNet-50



(b) BERT-Large Phase 2

KAISA Summary

We have presented ***KAISA***: a **K**-FAC-enabled, **A**daptable, **I**mproved, and **S**c**A**lable distributed, second-order optimizer framework.

- KAISA can adapt its distribution scheme to fit model and hardware characteristics.
- We study and characterize the tradeoff between caching and communication in distributed second-order training.
- First to show KAISA converges faster in fixed-batch size and fixed-memory budget environments for Mask R-CNN, U-Net, and BERT-Large.
- In high communication applications, extra memory can be used to reduce communication and improve training times.
- Show optimal scaling KAISA configuration for scaling up to 128 A100 GPUs.

Questions

Try out *KAISA* at

https://github.com/gpauloski/kfac_pytorch

Contact me at jgpauloski@uchicago.edu