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

J. Gregory Pauloski
University of Chicago

Qi Huang
University of Texas at Austin

Lei Huang
Texas Advanced Computing Center

Shivaram Venkataraman
University of Wisconsin, Madison

Kyle Chard
University of Chicago
Argonne National Laboratory

Ian Foster
University of Chicago
Argonne National Laboratory

Zhao Zhang
Texas Advanced Computing Center

ABSTRACT

Kronecker-factored Approximate Curvature (K-FAC) has recently been shown to converge faster in deep neural network (DNN) training than stochastic gradient descent (SGD); however, K-FAC's larger memory footprint hinders its applicability to large models. We present KAISA, a **K**-FAC-enabled, **A**daptable, **I**mproved, and **S**c**A**lable second-order optimizer framework that adapts the memory footprint, communication, and computation given specific models and hardware to improve performance and increase scalability. We quantify the tradeoffs between memory and communication cost and evaluate KAISA on large models, including ResNet-50, Mask R-CNN, U-Net, and BERT, on up to 128 NVIDIA A100 GPUs. Compared to the original optimizers, KAISA converges 18.1–36.3% faster across applications with the same global batch size. Under a fixed memory budget, KAISA converges 32.5% and 41.6% faster in ResNet-50 and BERT-Large, respectively. KAISA can balance memory and communication to achieve scaling efficiency equal to or better than the baseline optimizers.

KEYWORDS

Machine Learning, Distributed Computing, Second-Order Optimization, K-FAC, Data-Parallel Algorithms

ACM Reference Format:
J. Gregory Pauloski, Qi Huang, Lei Huang, Shivaram Venkataraman, Kyle Chard, Ian Foster, and Zhao Zhang. 2021. KAISA: An Adaptive Second-Order Optimizer Framework for Deep Neural Networks. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3458817.3476152

1 INTRODUCTION

Deep neural networks (DNNs) have driven breakthroughs in many research domains, including image classification [22, 24], object detection and segmentation [21], machine translation [53], and language modeling [17]. DNNs are typically trained with stochastic gradient descent (SGD), or variants thereof. As models and training datasets become larger, training must increasingly be performed in parallel on many CPUs, GPUs, or TPUs [30, 56]. For example, the BERT [17] model with 330 million parameters would take weeks to months to train on a single GPU, and GPT-3 [11] with 175 billion parameters cannot fit in the memory of any commercially available GPU. While distributed training on many processors can reduce training time, the need to communicate weight updates and other information among processors can limit scalability [39].

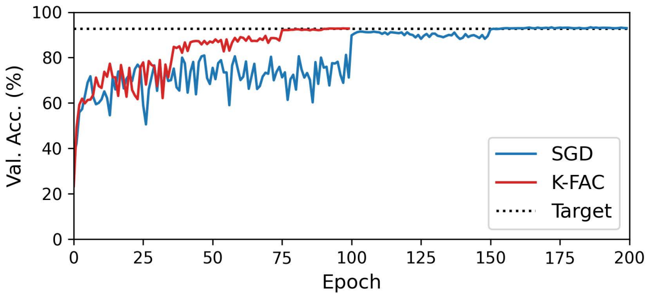


Figure 1: SGD vs. K-FAC training for ResNet-32 with the CIFAR-10 dataset. K-FAC reduces iterations needed for convergence.

Recent theoretical [20, 35, 36] and empirical [8, 42, 50] studies have shown that Kronecker-factored Approximate Curvature (K-FAC) can accelerate training by enabling convergence with fewer iterations than SGD—for example, achieving baseline validation accuracy in 40% fewer epochs than SGD on ResNet-32 with the CIFAR-10 dataset (see Figure 1). Technically, researchers use K-FAC to approximate the inverse of the Fisher Information Matrix (FIM)—an approximation of the Hessian—and precondition the gradients before parameter updates [36]. The K-FAC approximation is 1) compute intensive, due to the required inverse or eigen decomposition calculations with $O(N^3)$ complexity (N is the number of parameters); 2) memory intensive, as it stores per-layer activations and gradients, which may take $O(N^2)$ space compared to $O(N)$ for SGD; and 3) communication intensive for distributed training because the Kronecker factors and accompanying inverses or eigen decompositions must be communicated to all workers. To overcome these overheads and achieve faster training times than SGD at scale, K-FAC updates are often decoupled from gradient preconditioning so the expensive computations are performed less frequently.

Existing K-FAC systems either cache all Kronecker factor eigen decompositions needed to precondition gradients locally [44] or distribute gradient preconditioning across processors [42]. Neither approach enables K-FAC DNN deployments that are both memory and communication efficient, thus inhibiting scalability. The first approach avoids communication when local memory is sufficient. The second approach reduces memory footprint by not caching eigen decompositions locally but increases communication.

These various considerations make it difficult to use K-FAC efficiently in practice given the specific requirements of the DNN model or hardware and require users to choose between optimizing memory or communication. To address these concerns, we propose KAISA, a **K**-FAC-enabled, **A**daptable, **I**mproved, and **S**c**A**lable second-order optimizer framework that can adapt execution to a given model size and memory limit. KAISA allows tuning the ratio between communication and memory footprint to optimally apply K-FAC to distributed training by controlling the fraction of processes with local access to data. This adaptation is made possible by grouping the processes, then distributing and communicating data within each group. K-FAC distribution strategies described in other work are effectively special cases of KAISA in which either all processes [44] or one process [42] cache data.

We evaluate KAISA on a range of classification, segmentation, and language modeling applications, including ResNet-50 [22], Mask R-CNN [21], U-Net [46], and BERT [17], on clusters of 448 NVIDIA Tesla V100s and 192 Ampere A100 GPUs. Our results show that: 1) with fixed global batch size, KAISA trains deep neural networks 18.1–36.3% faster than the original optimizers for these applications; 2) with a fixed memory budget, KAISA trains ResNet-50 and BERT-Large phase 2 in 32.5% and 41.6% less time compared to momemtum SGD and Fused LAMB, respectively; 3) by varying the number of gradient workers, the K-FAC memory overhead can be reduced by 1.5–2.9×; 4) for high communication applications such as ResNet-50, extra processor memory can be used to train 24.4% faster; and 5) using state-of-the-art NVIDIA A100 GPUs, KAISA converges in fewer iterations at all scales, and KAISA's scaling performance is on-par with SGD even with KAISA's additional communication overhead. Our code is open source with the MIT license and available at https://github.com/gpauloski/kfac_pytorch.

Our contributions in this paper are:

- An adaptive second-order optimizer framework that trains faster than SGD and its variants, while preserving convergence.
- A quantitative study of the tradeoff between memory footprint and communication and its impact on training time in K-FAC design.
- The first large scale evaluation of K-FAC convergence and speedup relative to SGD for Mask R-CNN, U-Net, and BERT on up to 128 A100 GPUs.

The rest of the paper is as follows. We present the mathematical background and distributed implementation of K-FAC in §2. We describe KAISA's design in §3 and implementation in §4. We present our experiments and results in §5. In §6, we summarize existing DNN optimization frameworks and memory management techniques. Finally, we conclude in §7.

2 BACKGROUND

We first introduce K-FAC and its distributed implementation. K-FAC is an efficient approximation of the Fisher Information Matrix (FIM), which has been shown to be equivalent to the Generalized Gauss-Newton (GGN) matrix in specific cases and can be viewed as an approximation of the Hessian [36]. In a standard SGD update step, the weights are updated using the gradients of the loss, as illustrated in Equation 1. The K-FAC update step in Equation 2 uses the FIM F to precondition the gradients prior to update [44].

$w^{(k)}$ is the weight at iteration k, $\alpha^{(k)}$ is the learning rate at iteration k, n is the mini-batch size, $\nabla L_i(w^{(k)})$ is the gradient of the loss function L_i for the i^{th} example with regard to $w^{(k)}$, and $F^{-1}(w^{(k)})$ is the inverse of the FIM.

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

$$\text{K-FAC: } w^{(k+1)} = w^{(k)} - \frac{\alpha^{(k)} F^{-1}(w^{(k)})}{n} \sum_{i=1}^{n} \nabla L_i(w^{(k)}) \tag{2}$$

It has been shown empirically that training DNNs with the K-FAC second-order method enables convergence with fewer iterations than with SGD alone. Theoretical understandings of the convergence rates of natural gradient methods, such as K-FAC, are an area of active research. Previous work has shown that K-FAC [57] has linear convergence to the global minimum given a sufficiently over-parameterized model. In strongly-convex problems, natural gradient methods have a quadratic convergence rate compared to the linear convergence of SGD [10]. The strongly-convex case provides some understanding of how K-FAC can improve convergence in non-convex cases. Further, it has been shown that natural gradient methods enable larger learning rates improving the rate of convergence [57]. K-FAC makes greater per-iteration progress in minimizing the objective function at the cost of more computationally expensive iterations.

2.1 K-FAC Approximation

K-FAC is based on the Kronecker product, a block matrix factorization that can reduce a large matrix inverse into two smaller matrix inverses. K-FAC exploits the properties of the Kronecker product and the geometry of the FIM for DNNs to greatly reduce the complexity of computing the approximate FIM inverse.

2.1.1 *Kronecker Product.*

The Kronecker product is written as $A \otimes B$ where A has size $m \times n$ and B has size $p \times q$. The resulting matrix has shape $mp \times nq$.

$$A \otimes B = \begin{bmatrix} a_{11}B & \dots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \dots & a_{mn}B \end{bmatrix} \tag{3}$$

The Kronecker product has two convenient properties:

$$(A \otimes B)^{-1} = A^{-1} \otimes B^{-1} \tag{4}$$

$$(A \otimes B)\vec{c} = B^{\top}\vec{c}A. \tag{5}$$

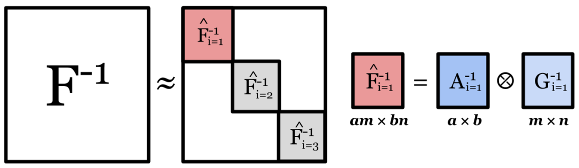


Figure 2: The K-FAC approximation of the Fisher information matrix. ⊗ is the Kronecker product [44].

2.1.2 K-FAC Approximation. The FIM for a deep neural network is a block matrix where each block maps to layers in the model. The block corresponding to the i and j^{th} layers is approximated as

$$F_{i,j} \approx a_{i-1}a_{j-1}^\top \otimes g_i g_j^\top . \tag{6}$$

Here, a_{i-1} and g_i are the activation of the $i-1^{\text{th}}$ layer and the gradients of the i^{th} layer in the model, respectively [36, 44].[1]

A fundamental assumption of K-FAC is the independence between layers [36, 44].[2] Using this assumption, K-FAC approximates the FIM as a diagonal block matrix $\hat{F}$.

$$\hat{F} = \mathtt{diag}(\hat{F}_1, ..., \hat{F}_i, ..., \hat{F}_L) \tag{7}$$

As shown in Figure 2, the inverse of F is a diagonal block matrix composed of the inverses of each diagonal block $\hat{F}_i$:

$$\hat{F}^{-1} = \mathtt{diag}(\hat{F}_1^{-1}, ..., \hat{F}_i^{-1}, ..., \hat{F}_L^{-1}) \tag{8}$$

where

$$\hat{F}_i = a_{i-1}a_{i-1}^\top \otimes g_i g_i^\top = A_{i-1} \otimes G_i . \tag{9}$$

We refer to A_{i-1} and G_i as the Kronecker factors. In practice, A_{i-1} and G_i are estimated with a running average of the factors computed over training batches [36, 44].

Due to the layer-wise independence of K-FAC, the gradient preconditioning and weight update for a single layer i at iteration k can be written as:

$$w_i^{(k+1)} = w_i^{(k)} - \alpha^{(k)} \hat{F}_i^{-1} \nabla L_i(w_i^{(k)}). \tag{10}$$

We can apply properties 4 and 5 to reduce the gradient preconditioning, $\hat{F}_i^{-1}\nabla L_i(w_i^{(k)})$, to an efficient form where the smaller Kronecker factors, rather than the large FIM, are inverted.

$$\hat{F}_i^{-1}\nabla L_i(w_i^{(k)}) = G_i^{-1}\nabla L_i(w_i^{(k)})A_{i-1}^{-1} \tag{11}$$

Tikhonov regularization is used to avoid ill-conditioned matrix inverses with K-FAC by adding a *damping parameter* γ to the diagonal of $\hat{F}_i$ [20, 42]. In most implementations, instead of computing $\hat{F}^{-1}$, we compute $(\hat{F}_i + \gamma I)^{-1}$ as:

$$(\hat{F}_i + \gamma I)^{-1} = (A_{i-1} + \gamma I)^{-1} \otimes (G_i + \gamma I)^{-1}. \tag{12}$$

Thus, the final update step for the parameters of layer i at iteration k is:

$$w_i^{(k+1)} = w_i^{(k)} - \alpha^{(k)} (\hat{F}_i + \gamma I)^{-1} \nabla L_i(w_i^{(k)}) \tag{13}$$

$$= w_i^{(k)} - \alpha^{(k)} (G_i + \gamma I)^{-1} \nabla L_i(w_i^{(k)}) (A_{i-1} + \gamma I)^{-1}. \tag{14}$$

[1]Formally, F represents an expected value, and the expectation of a Kronecker product is not equivalent to the Kronecker product of the expected factors. However, this approximation still reasonably represents the structure of the FIM [36].
[2]This assumption is sufficient to produce an effective approximation for F and necessary to produce a tractable algorithm.

2.1.3 Alternative Approximation. It has been shown that an empirically more stable[3] approximation for $(\hat{F} + \gamma I)^{-1}\nabla L_i(w_i^{(k)})$ can be computed using an eigen decomposition of the Kronecker factors, A and G, from Equation 9 [20, 44]. Given Q_A and Q_G, the eigenvectors of the factors, and v_A and v_G, the eigenvalues of the factors, the preconditioned gradient can be computed as follows.

$$V_1 = Q_G^\top \nabla L_i(w_i^{(k)}) Q_A \tag{15}$$

$$V_2 = V_1 / (v_G v_A^\top + \gamma) \tag{16}$$

$$(\hat{F}_i + \gamma I)^{-1} \nabla L_i(w_i^{(k)}) = Q_G V_2 Q_A^\top \tag{17}$$

The composition of the factors A and G in Equation 9 guarantees the factors are symmetric and therefore the factor eigen decompostions have real eigenvalues and orthogonal eigenvectors. In this work, we use the eigen decomposition method for gradient preconditioning.

2.1.4 Infrequent K-FAC Updates. A common strategy in second-order optimization methods is to update the second-order information every few iterations [36, 42, 44]. Intuitively, second-order information does not change as rapidly from one iteration to the next like first-order information. The K-FAC update interval parameter controls the number of iterations between second-order updates, i.e., iterations between eigen decomposition recomputations. Larger K-FAC update intervals result in more stale information, so tuning this parameter is key to achieving fast training with K-FAC. Practically, K-FAC can maintain convergence with K-FAC update intervals of 100–2000 iterations [44].

2.2 Distributed Implementation

Existing distributed K-FAC implementations are hybrid-parallel, with first-order information (e.g., gradients) computed in data-parallel and second-order information (e.g., K-FAC approximation) in model-parallel. Figure 3 outlines the model-parallel K-FAC computation performed between standard data-parallel training steps.

We refer to the two existing distributed K-FAC implementation strategies as MEM-OPT (memory optimized K-FAC) and COMM-OPT (communication optimized K-FAC). Both work by replicating the DNN across all processes and assigning a random local batch of training data to each process at each iteration. The data-parallel forward pass, backward pass, and SGD weight update stages outlined in Figure 3 are the same in both methods. The key differences between the two approaches are the model-parallel computation implementations in the gradient preconditioning stage.

2.2.1 MEM-OPT. MEM-OPT [42] assigns each layer in the DNN to a different process during the preconditioning stage. Each process computes the eigen decompositions of A_{i-1} and G_i needed for preconditioning the gradients using Equations 15–17. Each process then broadcasts the preconditioned gradients to all processes such that subsequent SGD weight updates can be done in data-parallel.

MEM-OPT has a low memory footprint because no eigen decompositions are duplicated: each layer's eigen decompositions are stored on only one process. MEM-OPT has communication in three places: a) gradient allreduce, b) factor allreduce, and c)

[3]In this context, stable means produces more consistent validation results across batch sizes and hyperparameter settings.

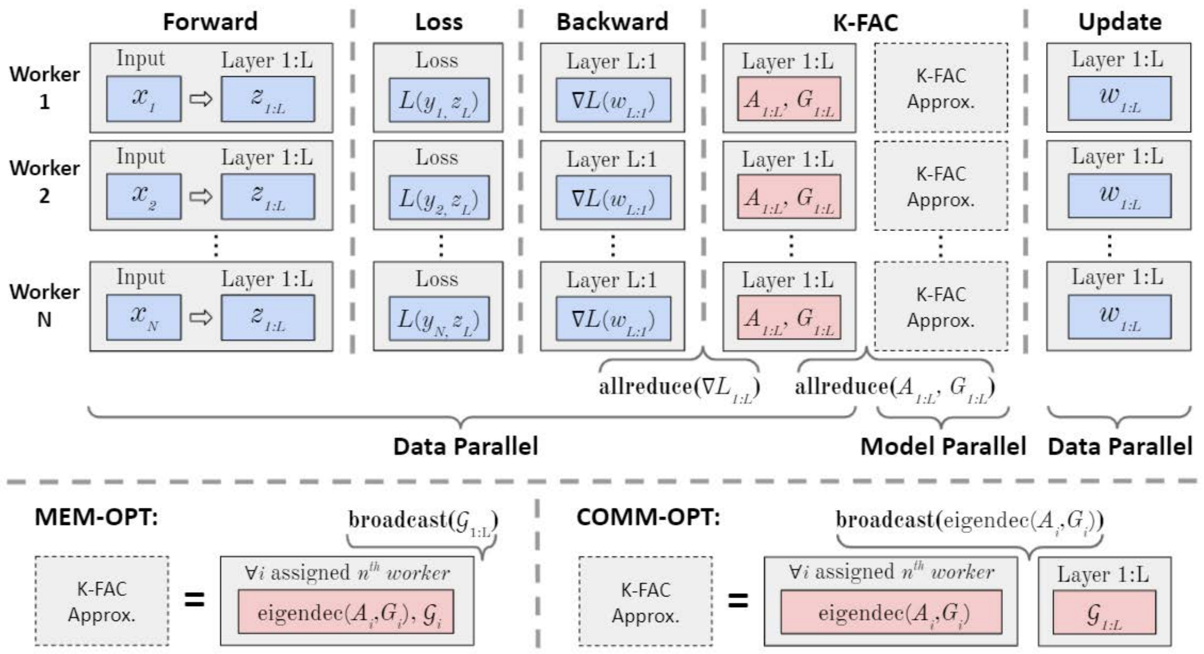

Forward
Loss
Backward
K-FAC
Update
Worker 1
Worker 2
Worker N
Input
Layer 1:L
Loss
Layer L:1
K-FAC Approx.
allreduce($\nabla L_{1:L}$)
allreduce($A_{1:L}$, $G_{1:L}$)
Data Parallel
Model Parallel
Data Parallel
MEM-OPT:
broadcast($\mathcal{G}_{1:L}$)
$\forall i$ assigned n^{th} worker
eigendec(A_i,G_i), $\mathcal{G}_i$
COMM-OPT:
broadcast(eigendec(A_i,G_i))
eigendec(A_i,G_i)

Figure 3: Hybrid-parallel distributed K-FAC implementation overview. Blue boxes are standard computations in data-parallel training. Red boxes are computations required by K-FAC. Workers maintain identical copies of the model. The output of each layer z is computed during the forward pass for the local batch x. Then, the loss between the true output y and predicted output z_L is calculated and used to compute gradients in the backward pass. The gradients are then allreduced across workers. During the forward/backward pass, K-FAC caches intermediate data for computing factors. In the K-FAC stage, workers compute factors A and G in data-parallel and allreduce the results. The eigen decompositions and preconditioned gradients $\mathcal{G}$ are computed in the K-FAC approximation stage. The existing K-FAC methods, MEM-OPT and COMM-OPT, implement the model-parallel stage differently as shown at the bottom of the figure. After the K-FAC stage, all workers have $\mathcal{G}$ and can update weights locally using $\mathcal{G}$ and a standard optimizer (e.g., SGD or ADAM).

preconditioned gradient broadcast, all shown in Figure 3. In non-K-FAC update steps (e.g., steps where the eigen decompositions are not updated), MEM-OPT avoids the factor allreduce because eigen decompositions from a previous step are reused; however, the preconditioned gradient broadcast is still required because the gradient being preconditioned changes every iteration.

2.2.2 *COMM-OPT.*

COMM-OPT [44] decouples the eigen decomposition from gradient preconditioning. Instead of assigning each layer to a process, individual factors are assigned to a process to be eigen decomposed in parallel. The eigen decompositions are broadcast back to all processes such that every process holds a copy of all eigen decompositions. Each process computes all preconditioned gradients locally prior to weight updates.

COMM-OPT has a larger memory-footprint because every process must maintain a copy of the eigen decompositions. COMM-OPT has communication in three places: a) gradient allreduce, b) factor allreduce, and c) eigen decomposition broadcast, also shown in Figure 3. Decoupling the eigen decompositions from the gradient preconditioning achieves two goals: 1) A_{i-1} and G_i can be computed in different processes, doubling the maximum worker utilization, and 2) in non-K-FAC update steps, the communication in (b) and (c) can be avoided because every worker can locally precondition gradients with the cached eigen decompositions. Thus, in non-K-FAC update intervals, COMM-OPT has no additional communication overhead compared to SGD. COMM-OPT has been shown to be 4–16% faster than MEM-OPT for ResNet-50 training on 16–256 V100 GPUs [44].

These two implementations convey the fundamental tradeoff between caching the decompositions locally to avoid communication and communicating the preconditioned gradients every iteration to avoid additional memory overheads. This tradeoff impacts the communication, computation, and memory overhead of K-FAC and motivates our research in this paper.

3 DESIGN

We introduce features of KAISA's design that enable its tunable memory footprint and explain how KAISA performs load balancing and half precision training to improve scalability.

3.1 Tunable Memory Footprint

KAISA introduces HYBRID-OPT, a configurable distributed K-FAC strategy that can exploit tradeoffs between memory usage and communication overhead.

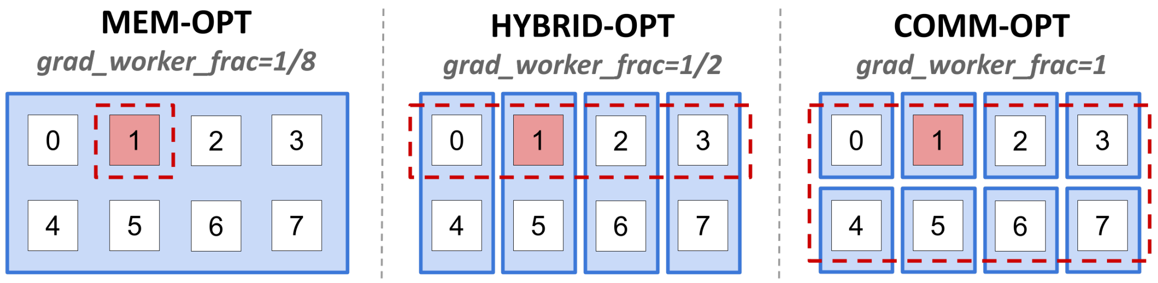


Figure 4: A comparison of three *grad_worker_frac* values on eight processes. Process 1, shaded in red, computes the eigen decompositions for this layer and communicates the result to all gradient workers, the processes inside the dashed red box. Gradient workers compute and broadcast the preconditioned gradient to a subset of the remaining processes, denoted by the light blue box. In MEM-OPT, there is one gradient worker that broadcast the preconditioned gradient to all processes. In HYBRID-OPT, there are four gradient workers that send the preconditioned gradient to one of the four remaining processes (e.g., process 0 sends the preconditioned gradient to process 4). In COMM-OPT, the preconditioned gradient need not be communicated because all processes are gradient workers.

In this strategy, each layer has a subset of the processes assigned to be gradient preconditioners, referred to as the gradient workers. *grad_worker_frac* defines the size of this subset, i.e., the gradient worker count is $\texttt{max}(1, \mathit{grad\_worker\_frac} \times \mathit{world\_size})$. One of the gradient workers is also responsible for computing the eigen decompositions for the layer and broadcasting the results to the remaining gradient workers. At the end of the eigen decomposition stage, each gradient worker has a copy of the eigen decompositions and can precondition the gradient.

Gradient receivers are the processes not assigned as gradient workers for the layer. Each gradient worker is responsible for broadcasting the preconditioned gradient to a subset of the gradient receivers. With multiple gradient workers, the broadcast groups are smaller and simultaneous broadcasts are possible, reducing overall communication time. For example, given two gradient workers and two gradient receivers, each gradient worker sends the preconditioned gradients to just one receiver; both gradient workers perform this communication at the same time.

After the gradient broadcast, all processes have a copy of the preconditioned gradient with which local weights can be updated.

Observe that KAISA unifies existing distributed K-FAC strategies because COMM-OPT and MEM-OPT are special cases of HYBRID-OPT. COMM-OPT is the case where $\mathit{grad\_worker\_frac} = 1$ (all processes precondition the layer's gradient), and MEM-OPT is the case where $\mathit{grad\_worker\_frac} = 1/\mathit{world\_size}$ (a single process preconditions a layer's gradient and broadcasts the result). Figure 4 compares MEM-OPT, COMM-OPT, and HYBRID-OPT in an eight process environment.

As *grad_worker_frac* increases, more processes cache the eigen decompositions and the memory footprint increases. Visually, the number of processes that cache the eigen decompositions is represented by the processes in the dashed red box in Figure 4.

Continuing with Figure 4, HYBRID-OPT has a lower preconditioned gradient broadcast cost than MEM-OPT because the broadcast groups are smaller and broadcasts are overlapped. HYBRID-OPT requires four separate broadcasts to groups of size two in comparison to MEM-OPT which requires one large broadcast to a group of eight. Since these broadcasts involve non-overlapping processes, we can execute all four broadcast calls simultaneously in the HYBRID-OPT case. Given the complexity of broadcasting using the minimum spanning tree algorithm is $O(\log p)$, where p is the number of processes in the broadcast group, the complexity is reduced from $O(\log 8)$ in MEM-OPT to $O(\log 2)$ in HYBRID-OPT. Note in steps where the eigen decompositions are updated, HYBRID-OPT incurs an additional eigen decomposition broadcast with $O(\log 4)$ complexity; however, as mentioned in §2.1.4, eigen decompositions are updated infrequently so the average complexity for HYBRID-OPT is still less than that of MEM-OPT.

The problem addressed by KAISA's HYBRID-OPT strategy is akin to that of 2.5D matrix multiplication [48]. Both algorithms can utilize extra processor memory to reduce communication costs by controlling the replication factor of data. In 2.5D matrix multiplication, a parameter c determines the number of data copies, and in KAISA, the *grad_worker_frac* determines the number of workers that store the eigen decompositions.

3.2 Greedy Factor Distribution

The eigen decompositions required for K-FAC are expensive to compute. Distributed K-FAC optimizes this stage by computing eigen decompositions in a model-parallel fashion. To most efficiently use available resources, the eigen decomposition computations should be distributed in a manner that minimizes the makespan T, the time it takes for all processes to complete their assigned computations. We use the longest processing time greedy algorithm which produces an assignment with makespan $T \leq \frac{3}{2}T^*$ where T^* is the optimal makespan [28]. The longest processing time algorithm sorts jobs by decreasing length and then iteratively assigns each job to the worker with the lowest current work load.

For each factor to be eigen decomposed, the processing time is approximated as $O(N^3)$ where each factor is an $N \times N$ matrix [16]. Alternatively, memory usage can be optimized for by using $O(N^2)$ as the approximation since N^2 is the size of the factor. KAISA uses the longest processing time strategy at the start of training to assign each factor to a process.

```python
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.

3.3 Half Precision Storage and Computation

Mixed precision training is a common strategy to reduce training times and memory usage on supported hardware [37]. Popular frameworks for automatic mixed precision (AMP) training, such as NVIDIA AMP and PyTorch AMP, cast forward and backward pass operations to half precision where possible. Gradient calculations in the backward pass can often produce very small values that would be clipped when cast to half precision, so these frameworks scale gradients to a larger value during the backward pass and unscale the gradients appropriately before the optimization step.

KAISA adapts to the precision being used for training. When training with AMP, factors are stored in half precision, reducing memory costs. K-FAC computations are performed in half precision where possible. Eigen decompositions are generally unstable in half precision so factors are cast to single precision before decomposition. KAISA can store eigen decompositions in half precision to further reduce memory consumption if needed.

Half precision training has become a staple in achieving state-of-the-art results in deep learning, and KAISA can particularly benefit from half precision training due to the K-FAC computation and communication overhead.

3.4 K-FAC Usage

We design KAISA to implement K-FAC as a preconditioner to standard optimizers with support for Conv2d and Linear layers. KAISA has an easy-to-use interface and can be incorporated into existing training scripts in two lines: one to initialize and one to call *KFAC.step()* prior to *optimizer.step()* (see Listing 1). KAISA automatically registers the model and determines the distributed communication backend (e.g., Torch, Horovod, single-process).

A call to *KFAC.step()*: 1) computes the factors using the forward/backward pass data, 2) computes the eigen decompositions in parallel and broadcasts the results, 3) computes the preconditioned gradients and broadcasts the results if necessary, and 4) scales the preconditioned gradients.

4 IMPLEMENTATION

We implement KAISA using PyTorch [43], with communication, interface, large-batch training, and gradient preconditioning implemented to collectively enable efficient second-order optimization.

4.1 AMP and Distributed Training

We use PyTorch AMP for mixed precision training. With PyTorch AMP, the *GradScaler* object responsible for scaling and unscaling the gradient in the backward pass can be passed to KAISA. KAISA uses the *GradScaler* to correctly unscale the G factors, since the scale factor can change from iteration to iteration, causing problems when computing the running average of G over the course of training. All communication operations are performed in the precision of the data to reduce bandwidth requirements.

KAISA supports `torch.distributed` and Horovod [47] for distributed training. In this work, we use `torch.distributed` for all experiments because the `DistributedDataParallel` model wrapper overlaps gradient communication with backpropogation, works seamlessly with PyTorch AMP, and provides a broadcast group abstraction needed for HYBRID-OPT.

4.2 Factor Accumulation

Processor memory (e.g., GPU VRAM) limits the maximum per-processor batch size during training. A common strategy to achieve larger effective batch sizes is gradient accumulation, a method where gradient values for multiple forward and backward passes are accumulated between optimization steps. For example, if processor memory limits the batch size to 8, an effective batch size of 32 can be achieved by accumulating gradients over four forward/backward passes prior to the optimization step. This strategy is common in applications such as BERT, where effective batch sizes are often $> 2^{15}$, but modern GPUs are limited to local batch sizes $< 2^7$.

The forward/backward pass data needed by KAISA to compute the factors is accumulated over each mini-batch between calls to *KFAC.step()*. As the number of gradient accumulation steps increases, the memory needed to accumulate the forward/backward pass data grows linearly. KAISA can efficiently support gradient accumulation by computing the factors for the current mini-batch during the forward/backward pass instead of during *KFAC.step()*. The factor communication is still performed during *KFAC.step()*.

4.3 Triangular Factor Communication

Previous K-FAC work has exploited the symmetric nature of the Kronecker factors to reduce communication volume by sending only the upper triangle for each factor [49, 50]. Our implementation supports extracting the upper triangle for the factor allreduce and reconstructing the full factor before the eigen decomposition stage. For the models studied in this work, this optimization did not yield performance improvements for two reasons. First, network latency impacted overall communication time more than bandwidth; second, this optimization has an additional overhead for extracting the upper triangle and reconstructing the factor. For models with larger individual layers—and therefore factors—this optimization could yield greater benefits.

4.4 Gradient Preconditioning

The largest overhead of K-FAC in non-K-FAC update steps is computing the preconditioned gradients. This process, described by Equations 15–17, involves a series of matrix additions, divisions, and multiplications. Observe that the gradient $\nabla L_i(w_i^{(k)})$ is the only variable that changes between non-K-FAC update iterations. In particular, the computation involving the outer product of the eigenvalues, $1/(v_G v_A^\top + \gamma)$, in Equation 16 does not need to be recomputed every iteration—only after the eigen decompositions are updated. We also observe that when *grad_worker_frac* $> 1/world\_size$, multiple processes perform this computation redundantly.

To reduce the total number of operations during the preconditioning stage, we move the computation of the outer product into the eigen decomposition stage. The process assigned to eigen decompose G computes $1/(v_G v_A^\top + \gamma)$ and broadcasts the result to all gradient workers instead of broadcasting v_A and v_G. This ensures that $1/(v_G v_A^\top + \gamma)$ is computed once (on a single worker) and then reused many times by other workers. In practice, this reduced the time to precondition the gradients for a single layer by up to 53%.

5 EXPERIMENTS

We report on experiments that address four issues: 1) convergence to baseline evaluation metrics; 2) time to convergence with and without KAISA to validate the design and implementation; 3) exploration of the memory and communication tradeoff using KAISA to develop a quantitative understanding of the *grad_worker_frac* configuration; and 4) evaluation of KAISA's scaling performance.

5.1 Hardware and Software Stack

We performed experiments on two computers:

- The GPU subsystem of the Frontera supercomputer at the Texas Advanced Computing Center, which has 112 nodes, each powered by IBM Power9 processors, with four 16 GB NVIDIA V100 GPUs (448 GPUs in total). Nodes are connected by an InfiniBand EDR network. We use PyTorch 1.6, CUDA 10.2, CUDNN 7.6.5, and NCCL 2.5.6, and MVAPICH2-GDR 2.3.4 to launch processes on multiple nodes for distributed training.
- The GPU subsystem of the Theta supercomputer at Argonne National Laboratory. This system has 24 NVIDIA DGXA100 nodes with eight 40GB A100 GPUs each (192 A100 GPUs in total). On DGXA100 nodes, we use PyTorch 1.7, CUDA 11.0, CUDNN 8.0.4, and NCCL 2.7.8.

We distinguish between these two systems in the following text by specifying either V100 or A100 GPUs.

5.2 Applications

We evaluate KAISA for classification, segmentation, and language modeling applications.

Classification: We use ResNet-50 [22] with the ImageNet-1k dataset [29], which has 1000 categories with approximately 1.3M training images and 50K validation images. We use K-FAC to precondition all convolutional and linear layers in ResNet-50 and SGD for the weight updates.

Segmentation: We explore two segmentation tasks. First, we use the NVIDIA reference PyTorch implementation of Mask R-CNN [5, 21] with the Common Objects in Context (COCO) 2014 dataset [32]. We use K-FAC to precondition the convolutional and linear layers in the region of interest (ROI) heads of Mask R-CNN and SGD for the weight updates.

Table 1: Baseline performance and hardware summary for ResNet-50, Mask R-CNN, U-Net, and BERT-Large. "val acc" is validation accuracy. "mAP" is mean average precision. "DSC" is Dice similarity coefficient.

App	Ref	Baseline	GPU	# GPUs
ResNet-50	[4]	75.9% val acc	V100	64
			A100	8
Mask R-CNN	[4]	0.377 bbox mAP, 0.342 segm mAP	V100	32
				64
U-Net	[2]	91.0% val DSC	A100	4
BERT-Large	[5]	90.8% SQuAD v1.1 F1 score	A100	8

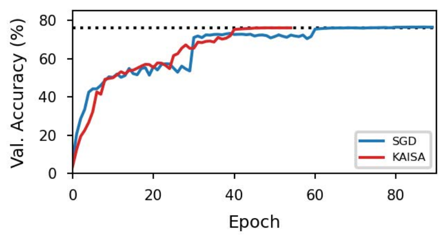
Second, we use a U-Net [46] architecture for segmenting brain tumor sub-regions. We extend a Kaggle competition implementation [2] to enable multi-GPU training. The test case was run on the LGG Segmentation Dataset [3], which contains Magnetic Resonance (MR) images of the brain from 110 patients across five hospitals. Images from a random subset of 100 patients are used as the training dataset and the remaining 10 patients are used for validation. We apply K-FAC to all convolutional layers in the model.

Language Modeling: We train the BERT-Large Uncased model using a modified version of the NVIDIA reference PyTorch implementation for BERT [5, 17] with the English Wikipedia [52] and Toronto BookCorpus datasets [59]. Each transformer in BERT is implemented using a series of Linear layers, and we apply K-FAC to all linear layers. We do not use K-FAC to precondition the embedding layer and prediction head because both of these layers have a Kronecker factor with shape $vocab\_size \times vocab\_size$, and since the vocab size for BERT-Large is 30K, these factors cannot be efficiently eigen decomposed. Fused LAMB is used as the optimizer [56]. The strategy outlined in §4.2 is used to reduce memory consumption since gradient accumulation is used for training.

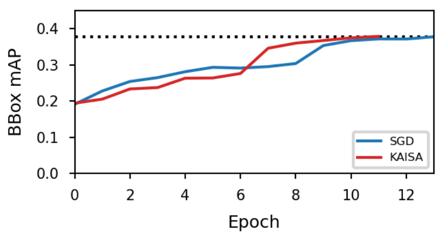
5.3 Convergence with Fixed Batch Size

We compare KAISA performance (converged accuracy, epochs to convergence, and time to convergence) on ResNet-50, Mask R-CNN, U-Net, and BERT-Large against the baseline implementations listed in Table 1. For ResNet-50 and Mask R-CNN, we use MLPerf benchmark target results [4]. For U-Net, we use the baseline validation Dice similarity coefficient (DSC) from the model's GitHub repository [2]. For the BERT-Large baseline, researchers have reported F1 scores of 91.08% [5], 91.0% [1], and 90.4% [56] for fine-tuning on the SQuAD v1.1 dataset. We use the best reported F1 score (i.e., the NVIDIA implementation [5]) with the LAMB optimizer. Due to the partial unavailability of the Toronto BookCorpus training dataset, our measurement only converges to 90.8%. (The Toronto BookCorpus dataset is no longer available online as a holistic package; we could recover only 14,155 of the 16,846 books.)

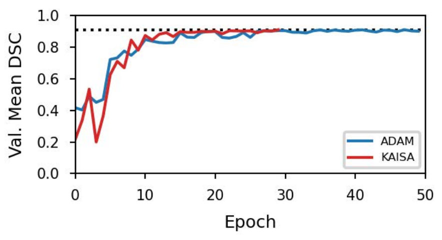
In all comparisons, we use the same global batch size for KAISA and the original optimizers to isolate the improvement from second-order information. Table 2 summarizes the hyperparameters for each application. For ResNet and K-FAC specific hyperparameters, we use values from [44] to provide direct performance comparisons to previous K-FAC works. For Mask R-CNN and BERT-Large, we use the NVIDIA reference hyperparameters [5]. Further performance



(a) ResNet-50. The time-to-convergence is 268.1 mins for K-FAC and 354.0 mins for momentum SGD.



(b) Mask R-CNN. The time-to-convergence is 115.8 mins for K-FAC and 136.1 for SGD.



(c) U-Net. The time-to-convergence is 10.9 mins for K-FAC and 14.6 for ADAM.

Figure 5: Validation Metric Curve Comparison between KAISA and SGD/ADAM on ResNet-50, Mask R-CNN, and U-Net. The dotted lines represent the target metric.

Table 2: Summary of hyperparameters used for each application. BS = global batch size, LR = learning rate, WU = warm up iterations, K_freq = number of iterations between eigen decomposition re-computations, F_freq = iterations between factor updates. *grad_worker_frac* = 1 and damping = 0.003 for all cases.

App	BS	LR	WU	K_freq	F_freq
ResNet-50	2,048	0.8	3,130	500	50
Mask R-CNN	64	8e-2	800	500	50
U-Net	64	4e-4	500	200	20
BERT-Large	65,536	5e-5	103	100	10

improvements could be gained through more extensive hyperparameter tuning; however, this was not necessary to achieve results better than the original optimizers.

ResNet-50: We train ResNet-50 for 55 and 90 epochs for KAISA and momentum SGD, respectively, using FP16 precision on eight NVIDIA A100 GPUs. Figure 5(a) shows the validation accuracy curve using the two optimization methods. KAISA converges to the baseline validation accuracy at epoch 46 and momentum SGD at epoch 65. The time-to-convergence is 268.1 minutes for KAISA: 24.3% less than the 354.0 minutes for momentum SGD.

Mask R-CNN: Our baseline measurement of Mask R-CNN on 32 NVIDIA V100 GPUs takes 25,640 iterations to converge to 0.377 bbox mAP and 0.342 segm mAP in FP32. With K-FAC enabled for the ROI heads, training converges to 0.379 bbox mAP and 0.350 segm mAP in 21,000 iterations. The bbox mAP comparision for SGD and KAISA is shown in Figure 5(b). KAISA reduces the training time from 136.1 minutes to 115.8 minutes, a 14.9% improvement. With a batch size of 128 on 64 V100 GPUs, KAISA converges in 12,000 iterations compared to 15,000 with SGD and reduces training time by 18.1%.

U-Net: The reference U-Net implementation [2] with ADAM converges to 91.0% validation DSC within 50 epochs with four NVIDIA A100 GPUs using FP32 training. Using KAISA, the training converges above 91.0% in 30 epochs as seen in Figure 5(c). KAISA reduces the training time by 25.4% (10.9 minutes vs. 14.6 minutes).

BERT: BERT pretraining has two phases. The first trains with maximum sequence length of 128 for 7038 iterations and the second trains with maximum sequence length of 512 for 1,563 iterations. We fine-tune the pretrained BERT model for three epochs using the SQuAD dataset. All phases and fine-tuning are done in FP16. Tuning the hyperparameters of BERT is expensive as each run can take over 130 hours with 8 A100 GPUs, so we showcase the effectiveness of KAISA with the second phase of BERT pretraining. For phase two pretraining, we start with the same model pretrained with LAMB during phase one. We train with KAISA for {800, 1,000, 1,200} iterations then fine-tune SQuAD. Table 3 summarizes the validation SQuAD F1 scores after fine-tuning and the pretraining performance improvements.

KAISA converges to the 90.8 F1 baseline in 800 iterations, 47.9% less iterations than required for LAMB, and KAISA takes 36.3% less time than LAMB to converge.

Table 3: BERT performance comparison: KAISA vs. LAMB

		KAISA, with iterations:		
Metric	LAMB	1200	1000	800
F1	90.8	91.0	91.0	90.8
Stdev	0.13	0.15	0.17	0.24
Iterations	1,536	1,200	1,000	800
Time (hrs)	47.7	41.5	34.4	30.4

Table 4: Time to convergence and hyperparameters for each optimizer. BS = global batch size, LR = learning rate, g_frac = gradient worker fraction, K_f = iterations between eigen decomposition re-computations, F_f = iterations between factor updates, T_conv is time to convergence in minutes.

App	Opt	BS	LR	g_frac	K_f	F_f	T_conv
ResNet-50	SGD	8K	3.2	—	—	—	N/A
	KAISA	5K	2.0	1/64	200	20	96
	KAISA	5K	2.0	1/2	200	20	83
BERT-Large	LAMB	24K	3e-3	—	—	—	2,917.6
	KAISA	32K	4e-3	1/2	100	10	1,702.5
	KAISA	32K	4e-3	1	100	10	1,703.5

5.4 Convergence with Fixed Memory Budget

To understand how a *grad_worker_frac* can enable second-order optimization in memory-constrained environments, we compare KAISA's performance against baselines with fixed memory budgets. In particular, we train ResNet-50 on 64 V100 GPUs and BERT-Large phase two on eight A100 GPUs. For each experiment, we use the maximum possible local batch size and measure the convergence, epochs to convergence, and time to convergence. Table 4 summarizes the hyperparameters and time to convergence.

ResNet-50: With momentum SGD, the maximum local batch size is 128 per GPU and the global batch size is 8,192. We train ResNet-50 for 90 epochs using momentum SGD which achieves 75.0% validation accuracy—0.9% lower than the MLPerf baseline. Next, we use KAISA with *grad_worker_frac* = 1 and a local batch size of 80; however, the training runs out of memory. Lowering *grad_worker_frac* to 1/2, training takes 83 minutes to converge to 75.9% in the 48th epoch. The complete 55 epoch training takes 95 minutes and the validation accuracy reaches 76.0%. So, even if momentum SGD converges to 75.9% by the 90th epoch, KAISA still reduces the time to convergence by 32.5%. Finally, we run the same training process with MEM_OPT by setting *grad_worker_frac* to 1/64. It converges at the 47th epoch and the time to convergence is 96 minutes. The complete 55 epoch training takes 111 minutes. This experiment highlights the benefit of KAISA in cases where compute resource can not be efficiently utilized with the original optimizers. With a *grad_worker_frac* value of 1/2, KAISA offers benefits over COMM_OPT and MEM_OPT by enabling second-order optimization under a tight memory budget that is 13.9% faster than COMM_OPT and not feasible with MEM_OPT.

BERT: With LAMB, the maximum possible local batch size per GPU is 12 for the second phase of this BERT implementation [5], and the global batch size is 24,576. For KAISA, we use a local batch size of 8 and global batch size of 32,768. This experiment uses the same hyperparameters as in §5.3, so all cases with LAMB and BERT should converge to the baseline, thus we only project the training time with the first 100 steps. As the global batch size with LAMB is 24,576, 2,084 training steps are required to finish three epochs, and the training time is 2,917.6 minutes. KAISA, with *grad_worker_frac* = 1/2, takes 3,268.8 minutes to finish the three epochs. However, KAISA converges to baseline after 800 steps, as shown in Table 3, so the time to converge is 1,702.5 minutes—41.6% faster than LAMB. Setting *grad_worker_frac* = 1 takes 1,703.5 minutes to converge for KAISA. The performance is comparable to the case with *grad_worker_frac* = 1/2. We conduct a detailed study on this convergences phenomena for different *grad_worker_frac* values in §5.5.

5.5 Memory vs. Communication

To understand the impact of *grad_worker_frac* on training times, we train ResNet-{18, 50, 101, 152}, Mask R-CNN, and BERT-Large on 64 V100 GPUs for *grad_worker_frac* ∈ {1/64, 1/32, 1/16, 1/8, 1/4, 1/2, 1}. For each experiment, we record the average iteration time, i.e., time between weight updates, and the GPU memory usage. We refer to the *K-FAC overhead* as the memory required to store the factors and eigen decompositions, and the K-FAC overhead is computed as the difference between the K-FAC and SGD memory usage. For all ResNet models, we use the same hyperparameters used for ResNet-50 (Section 5.3) except for ResNet-152 where the local batch size was lowered to 24. The results are presented in Figure 6, and a summary of the memory usage is provided in Table 5.

The K-FAC memory overhead increases linearly as a function of *grad_worker_frac* for all models. KAISA requires 1.5–45.8% more memory than SGD depending on the application and value of *grad_worker_frac* (Table 5). The maximum K-FAC overhead (i.e., when *grad_worker_frac* = 1) is 1.5–2.9× that of the minimum K-FAC overhead (i.e., when *grad_worker_frac* = 1/64).

Table 5: Summary of per-GPU memory usage for training, in MB. Abs. is the absolute memory required for training. Δ is the %-increase in memory required over SGD. The K-FAC overhead is the K-FAC abs. memory minus the SGD abs. memory.

		SGD	K-FAC Min		K-FAC Max	
Model	Precision	Abs.	Abs.	Δ	Abs.	Δ
ResNet-18	FP32	2454	2838	16.7%	3260	32.8%
ResNet-50	FP32	4762	5396	13.3%	6608	38.8%
ResNet-101	FP32	6313	7463	18.2%	8755	38.7%
ResNet-152	FP32	6620	8204	23.9%	9092	37.3%
Mask R-CNN	FP32	6553	6650	1.5%	6743	2.9%
BERT-Large	FP16	8254	9555	15.8%	12038	45.8%

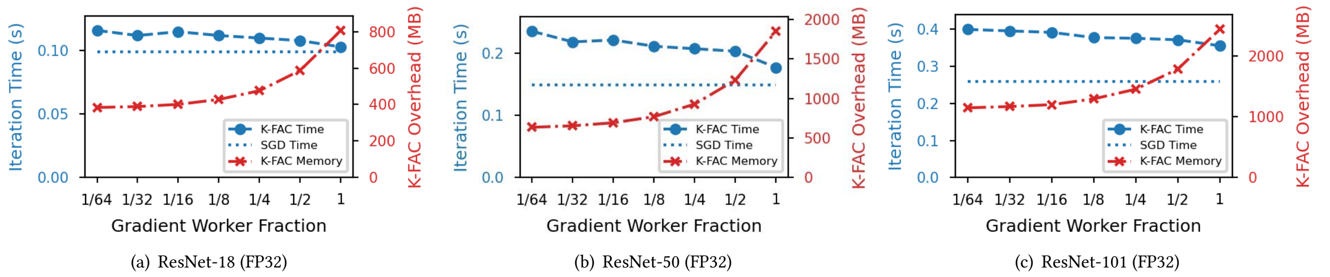
With respect to iteration times, the ResNet models scale well with the number of gradient workers with ResNet-50 scaling the best. For ResNet-50, the speedup from a gradient worker count of 1 to 64 is 24.4% for FP32 with a 22% increase in total memory usage. The average iteration times for Mask R-CNN and BERT-Large remain constant as the number of gradient workers is increased.

For comparison, the same ResNet-50 experiment with 64 V100s in [44] only shows a 7.6% speedup when increasing the gradient worker count from 1 to 64. This improvement in KAISA over previous work is due to the unique contributions presented in §3.2, §3.3, §4.1, and §4.4. These promising results for ResNet-50, a de facto standard benchmark for deep learning systems, are important as the performance characteristics of ResNet-50 represent a large set of commonly used models (e.g. VGG16, U-Nets, etc.).

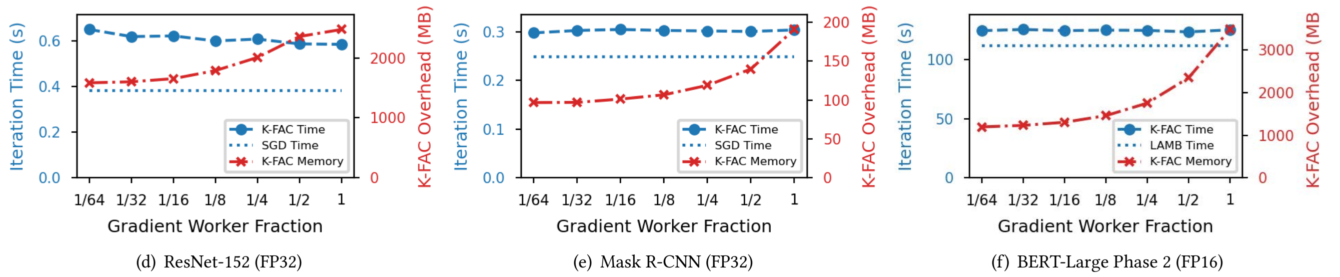
We can understand why ResNet model performance varies across *grad_worker_frac* values while the Mask R-CNN and BERT-Large performance remains constant by considering the bandwidth requirements of these applications. The bandwidth required by KAISA is a function of the size of the factors, eigen decompositions, and frequency of K-FAC updates.

With 64 V100s, ResNet-50 calls *KFAC.step()* frequently (4–6 calls/second) and incurs a K-FAC memory overhead between 634 MB and 1.8 GB. In comparison, Mask R-CNN calls *KFAC.step()* with a lower frequency (3 calls/second) and has a much smaller K-FAC overhead (100–200 MB). Thus, the changes in how KAISA communicates data with respect to *grad_worker_frac* are less apparent in Mask R-CNN. BERT-Large has the lowest bandwidth requirements of all applications even though it has the largest K-FAC overhead. BERT-Large uses gradient accumulation to achieve very large batch sizes (32K for phase 2) and as a result only calls *KFAC.step()* every ~120 seconds.

While the iteration times for low-communication models such as BERT and Mask R-CNN are invariant to the *grad_worker_frac* value in KAISA, KAISA still produces faster-than-SGD training times with small increases in memory-overhead. This is due to KAISA's unique features outlined in §3 and §4. Further, practitioners training these models at larger scales, e.g., 100s or 1000s of GPUs, where communication becomes a greater bottleneck will benefit more from the flexibility KAISA provides to adapt training to environments with increasing communication costs.



(a) ResNet-18 (FP32) (b) ResNet-50 (FP32) (c) ResNet-101 (FP32)



(d) ResNet-152 (FP32) (e) Mask R-CNN (FP32) (f) BERT-Large Phase 2 (FP16)

Figure 6: Average iteration time and K-FAC memory overhead across *grad_worker_frac* values on 64 V100 GPUs. Dotted lines are the baseline iteration times without K-FAC.

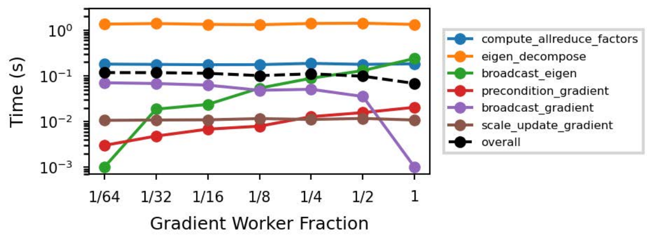


Figure 7: Average function execution time during calls to *KFAC.step()* for ResNet-50 on 64 GPUs.

Tuning the *grad_worker_frac* hyperparameter, to determine an optimal balance between iteration time and memory usage, is simple as it only requires profiling the average iteration time for each *grad_worker_frac* value over a few iterations.

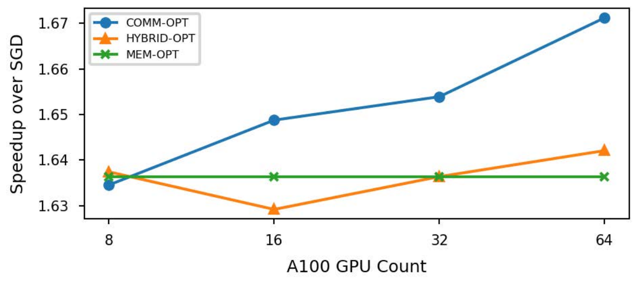
We find that with respect to training times, *grad_worker_frac* has the most impact in applications that spend a larger proportion of time doing communication. To further understand how *grad_worker_frac* impacts training times, we analyze the execution time for each section within *KFAC.step()* with ResNet-50 on 64 V100s. Figure 7 provides the time spent in each section for all layers in the model during a call to *KFAC.step()*. Times are averaged over 10,000 iterations and across all workers. Eigen decompositions are updated every 500 iterations. As shown in Figure 7, factor computation and communication, eigen decomposition, and scaling and updating the gradients, are invariant to the *grad_worker_frac*.

The time required to broadcast the eigen decompositions increases substantially as the number of gradient workers is increased. Referring back to Figure 4, the more gradient workers there are, the more processes that need to receive the eigen decompositions. However, the eigen decompositions, in this case, are only recomputed every 500 iterations so the eigen decomposition broadcast has a negligible effect on the average iteration time.

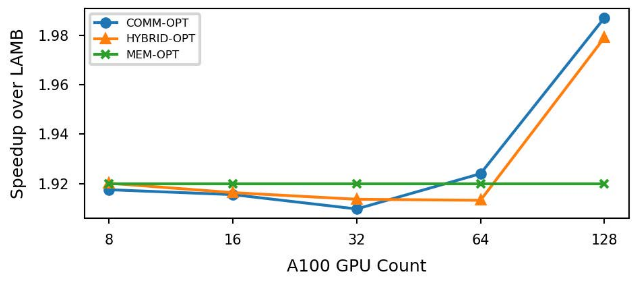
On the other hand, the gradient preconditioning and broadcast occur every iteration regardless of if the factors or eigen decompositons are updated and have the greatest influence on average iteration time. We see that the time to precondition the gradients increases with the gradient worker count because each process is assigned as a gradient worker for more layers. This discovery highlights the importance of the gradient preconditioning optimizations made in §4.4. The time to broadcast the preconditioned gradients decreases to 0 as the gradient worker count approaches *world_size*, and notably, the time decreases at a *faster rate* than the increase in time required in the preconditioning stage. This trend is a result of each gradient worker needing to send the results to fewer other processes as the gradient worker count increases.

5.6 Scaling

To examine the scaling characteristic of KAISA, we measure the average time per epoch for ResNet-50 and average time per iteration for BERT-Large phase 2. We choose ResNet-50 and BERT-Large because they represent a high-communication and low-communication model, respectively (see §5.5). We study three KAISA variants: COMM-OPT, MEM-OPT, and HYBRID-OPT with a *grad_worker_frac* of 1/2, and we report the projected end-to-end training time speedup for the KAISA variants over the base optimizers (SGD and LAMB) in Figure 8. For ResNet-50, we project the training time to 90 epochs for SGD and 55 for KAISA. For BERT-Large, we project the phase 2 training time to 1,563 steps for LAMB and 800 for KAISA based on the study in §5.3. The hyperparameters in Table 2 are used, and for ResNet-50, the K-FAC update frequency is scaled inversely with the global batch size to keep the number of K-FAC updates per training



(a) ResNet-50



(b) BERT-Large Phase 2

Figure 8: Speedup for the ResNet-50 and BERT-Large applications on A100 nodes.

samples constant. We store and communicate the factors and eigen decompositions in FP16 for BERT-Large.

In Figure 8(a) and 8(b), MEM-OPT maintains a constant speedup over SGD and LAMB, respectively, across all scales. In contrast, the speedup for COMM-OPT improves in both cases as the scale increases indicating the tradeoff of more memory for reduced communication does have scaling benefits. HYBRID-OPT sees performance improvements on par with COMM-OPT with BERT-Large while using less GPU memory. Overall, HYBRID-OPT has the best balance of scaling and memory usage for large-scale BERT pretraining.

As a whole, KAISA achieves better speedups with BERT-Large which is likely due to ResNet-50 being more communication bound than BERT-Large as noted in §5.5. Further scaling experiments on a machine with more GPUs is needed to understand the characteristics and limits of KAISA's speedup over SGD.

6 RELATED WORK

The emergence of DNNs has motivated revisiting many aspects of system design. TensorFlow [6], PyTorch [43], and MXNet [14] are examples of deep learning frameworks that support end-to-end training. TensorFlow Serving [41], DLHub [13], and Clipper [15] focus on low-latency inference serving. Ray [38] provides a platform to integrate simulation, training, and serving for reinforcement learning applications. Researchers have also designed new scheduling approaches in GPU clusters with informed hardware heterogeneity [40], sharing capability [55], and early user feedback [54] to optimize cost and hardware utilization. Our work here focuses on a lower-level question: given model architecture, communication bandwidth, processor count, and memory size, can the hybrid-parallel aspect of distributed K-FAC be adaptable and reduce training time? In the rest of this section, we briefly review other works in optimization frameworks.

Optimizer Frameworks: Distributed optimization frameworks take many forms. In a synchronous optimizer such as Horovod [47], all variables are updated in every iteration. Asynchronous optimizers relax variable update consistency, for example by passing values via parameter servers [31], to achieve higher performance than synchronous methods [33, 45]. The pipeline parallel paradigm, such as GPipe [25] and PipeDream [39] which hold multiple versions of a model partition in a processor and exploit asynchronous optimizers, has shown comparable training convergence. BytePS [27] proposes a unified interface for synchronous and asynchronous SGD. Generally, asynchronous SGD has a non-linear slowdown compared to synchronous SGD [7]. KAISA implements K-FAC in a synchronous manner as a preconditioner such that it can work with any SGD variant.

Memory Shortage and Remedies: Training DNNs is memory intensive. A common technique for training large models is to swap between processor memory and host memory, such as in SwapAdvisor [23] and SuperNeurons [51]. An alternative is to discard some activation tensors and rematerialize them when needed for back-propagation. Checkmate [26] formulates this tradeoff as an optimization problem and provides an optimal rematerialization schedule. In KAISA, we apply techniques such as precision relaxing to reduce the memory consumption of K-FAC and controlling the distribution of eigen decomposition results across processors to maintain a minimal cost in training time.

K-FAC Convergence: Prior work on distributed K-FAC has reported training convergence primarily on ResNet-like convolutional neural networks. One study [8] used asynchronous distributed K-FAC to train ResNet-50 with ImageNet 2× faster than standard SGD, but only achieved 70% validation accuracy, a 5.9% loss compared to the 75.9% MLPerf [4] baseline.

Another distributed K-FAC implementation [42] trained ResNet-50 with ImageNet to 74.9% validation accuracy in just 978 iterations; however, comparisons to SGD are not provided. Later work iterated on this implementation to achieve 75.0% validation accuracy on ImageNet with ResNet-50 on 2048 V100 GPUs in 2 minutes by carefully optimizing the baseline SGD training and introducing a 21-bit floating point (FP21) specification for the factors along with other optimizations from previous works such as triangular matrix communication and infrequent K-FAC updates. FP21 was introduced due to worse convergence when using FP16 for factor communication; however, with KAISA, we found similar validation accuracy with ResNet-50 for FP32 and FP16 factor communication. Differences in the numerical stability of the eigen decomposition in KAISA rather than matrix inversion could be a possible reason for the discrepancies.

A fourth study [44] trained ResNet-50 to the 75.9% MLPerf baseline in 18-25% less time than with SGD by replacing the factor inverse with eigen decomposition and optimizing for reduced communication in non-K-FAC update steps, at the cost of a high memory footprint. KAISA generalizes previous distributed K-FAC strategies via a tunable memory footprint to balance the memory and communication costs. This design enables efficient, distributed K-FAC research across a wider range of hardware. Further, we showcase KAISA's effectiveness on a variety of domains and models.

Alternative K-FAC Methods: Eigenvalue-corrected Kronecker-factorization (EK-FAC) [18], a more accurate approximation of the FIM, can perform cheap, partial updates. Noisy K-FAC [58] and noisy EK-FAC [9] are functionally similar to standard K-FAC but introduce adaptive weight noise. K-BFGS and K-BFGS(L) [19] apply BFGS [12] and L-BFGS [34] in an analogous method to K-FAC (e.g., block-diagonal, Kronecker-factored). These works have shown better-than-K-FAC performance in many small scale (e.g., single GPU) and small dataset/model (e.g., MNIST or CIFAR-10 with VGG16) cases. Kronecker-factor based FIM approximation variants are a growing area of research; however, large-scale studies have largely been limited to standard K-FAC. KAISA introduces a valuable, unified design paradigm that can be applied to these K-FAC variants to efficiently deploy and evaluate their effectiveness on large models at scale.

7 CONCLUSION

We have presented KAISA, a **K**-FAC-enabled, **A**daptable, **I**mproved, and **S**c**A**lable second-order optimizer framework. To enable scalable DNN training, KAISA adapts memory and communication usage, and appropriately distributes the complex K-FAC computations to best suit the model and hardware characteristics. We design KAISA to be adaptable to hardware with limited memory (e.g., gaming GPUs) or environments with high communication costs (e.g., Ethernet or massively parallel). We study the fundamental tradeoff between data access on local and remote memory, and evaluate KAISA's correctness and impact on training time by using four real-world applications. With the same global batch size, our experiments show a 18.1–36.3% training time reduction for ResNet-50, Mask R-CNN, U-Net, and BERT-Large, while preserving convergence to the baseline. Under the same memory budget, ResNet-50 and BERT phase 2 converge to the baseline in 32.5% and 41.6% less time compared to momentum SGD and Fused LAMB. In high-communication applications, such as ResNet models, we show that extra processor memory can be used to improve iteration times by reducing communication. In low-communication applications, such as Mask R-CNN and BERT-Large, we show the optimal KAISA usage and efficient scaling on par with SGD up to 128 GPUs.

8 ACKNOWLEDGEMENTS

This work was supported by NSF OAC-1931354 and OAC-1818253. Shivaram Venkataraman is supported by the Office of the Vice Chancellor for Research and Graduate Education at UW-Madison with funding from the Wisconsin Alumni Research Foundation.

REFERENCES

[1] BERT. https://github.com/google-research/bert.

[2] brain-segmentation-pytorch. https://www.kaggle.com/mateuszbuda/brain-segmentation-pytorch, https://github.com/mateuszbuda/brain-segmentation-pytorch.

[3] LGG segmentation dataset. https://www.kaggle.com/mateuszbuda/lgg-mri-segmentation.

[4] MLPerf. https://www.mlperf.org/.

[5] NVIDIA deep learning examples. https://github.com/NVIDIA/DeepLearningExamples.

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[7] Dan Alistarh, Christopher De Sa, and Nikola Konstantinov. The convergence of stochastic gradient descent in asynchronous shared memory. In *ACM Symposium on Principles of Distributed Computing*, pages 169–178. ACM, 2018.

[8] Jimmy Ba, Roger B. Grosse, and James Martens. Distributed second-order optimization using Kronecker-factored approximations. In *ICLR*, 2017.

[9] Juhan Bae, Guodong Zhang, and Roger B. Grosse. Eigenvalue corrected noisy natural gradient. *CoRR*, abs/1811.12565, 2018.

[10] Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *SIAM Review*, 60(2):223–311, 2018.

[11] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[12] C. G. BROYDEN. The Convergence of a Class of Double-rank Minimization Algorithms 1. General Considerations. *IMA Journal of Applied Mathematics*, 6(1):76–90, 03 1970.

[13] R. Chard, Z. Li, K. Chard, L. Ward, Y. Babuji, A. Woodard, S. Tuecke, B. Blaiszik, M. J. Franklin, and I. Foster. Dlhub: Model and data serving for science. In *IEEE International Parallel and Distributed Processing Symposium*, pages 283–292, Los Alamitos, CA, USA, may 2019. IEEE Computer Society.

[14] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[15] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, 2017.

[16] James Demmel, Ioana Dumitriu, and Olga Holtz. Fast linear algebra is stable. *Numerische Mathematik*, 108(1):59–91, Oct 2007.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[18] Thomas George, César Laurent, Xavier Bouthillier, Nicolas Ballas, and Pascal Vincent. Fast approximate natural gradient descent in a kronecker factored eigenbasis. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[19] Donald Goldfarb, Yi Ren, and Achraf Bahamou. Practical quasi-newton methods for training deep neural networks. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 2386–2396. Curran Associates, Inc., 2020.

[20] Roger Grosse and James Martens. A Kronecker-factored approximate Fisher matrix for convolution layers. In *33rd International Conference on International Conference on Machine Learning*, page 573–582.

[21] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *IEEE International Conference on Computer Vision*, pages 2961–2969, 2017.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[23] Chien-Chin Huang, Gu Jin, and Jinyang Li. SwapAdvisor: Pushing deep learning beyond the GPU memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1341–1355, 2020.

[24] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *CVPR*, volume 1, page 3, 2017.

[25] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, pages 103–112, 2019.

[26] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *Proceedings of Machine Learning and Systems*, 2:497–511, 2020.

[27] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation*, pages 463–479, 2020.

[28] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., USA, 2005.

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[30] Sameer Kumar, Victor Bitorff, Dehao Chen, Chiachen Chou, Blake Hechtman, HyoukJoong Lee, Naveen Kumar, Peter Mattson, Shibo Wang, Tao Wang, et al. Scale MLPerf-0.6 models on Google TPU-v3 Pods. *arXiv preprint arXiv:1909.09756*, 2019.

[31] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation*, pages 583–598, 2014.

[32] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft COCO: Common objects in context, 2015.

[33] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

[34] Dong C. Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *MATHEMATICAL PROGRAMMING*, 45:503–528, 1989.

[35] James Martens, Jimmy Ba, and Matt Johnson. Kronecker-factored curvature approximations for recurrent neural networks. In *International Conference on Learning Representations*, 2018.

[36] James Martens and Roger Grosse. Optimizing neural networks with Kronecker-factored approximate curvature. In *International conference on machine learning*, pages 2408–2417, 2015.

[37] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training, 2018.

[38] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation*, pages 561–577, 2018.

[39] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[40] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. *arXiv preprint arXiv:2008.09213*, 2020.

[41] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139*, 2017.

[42] Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Rio Yokota, and Satoshi Matsuoka. Large-scale distributed second-order optimization using Kronecker-factored approximate curvature for deep convolutional neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition*, June 2019.

[43] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[44] J Gregory Pauloski, Zhao Zhang, Lei Huang, Weijia Xu, and Ian T Foster. Convolutional neural network training with distributed K-FAC. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2020.

[45] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[46] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[47] Alexander Sergeev and Mike Del Balso. Horovod: Fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.

[48] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, pages 90–109, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[49] Yohei Tsuji, Kazuki Osawa, Yuichiro Ueno, Akira Naruse, Rio Yokota, and Satoshi Matsuoka. Performance optimizations and analysis of distributed deep learning with approximated second-order optimization method. In *Proceedings of the 48th International Conference on Parallel Processing: Workshops*, ICPP 2019, New York, NY, USA, 2019. Association for Computing Machinery.

[50] Yuichiro Ueno, Kazuki Osawa, Yohei Tsuji, Akira Naruse, and Rio Yokota. Rich information is affordable: A systematic performance analysis of second-order optimization using k-fac. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 2145–2153, New York, NY, USA, 2020. Association for Computing Machinery.

[51] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 41–53, 2018.

[52] Wikipedia. Wikipedia Corpus. https://meta.wikimedia.org/wiki/Data_dump_torrents#English_Wikipedia.

[53] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[54] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, 2018.

[55] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation*, pages 533–548, 2020.

[56] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training BERT in 76 minutes. *arXiv preprint arXiv:1904.00962*, 2019.

[57] Guodong Zhang, James Martens, and Roger B Grosse. Fast convergence of natural gradient descent for over-parameterized neural networks. In *Advances in Neural Information Processing Systems*, pages 8082–8093, 2019.

[58] Guodong Zhang, Shengyang Sun, David Duvenaud, and Roger Grosse. Noisy natural gradient as variational inference. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 5852–5861. PMLR, 10–15 Jul 2018.

[59] Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *IEEE International Conference on Computer Vision*, pages 19–27, 2015.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

For all experiments, we used the GPU subsystem of TACC's Frontera Supercomputer (112 nodes with 4 NVIDIA 16GB V100 GPUs, 256GB RAM, and an IBM Power9 CPU connected via an InfiniBand EDR network) and the GPU subsystem of the Theta Supercomputer at Argonne National Laboratory (24 DGXA100 nodes with 8 NVIDIA 40GB A100 GPUs per node).

On the V100 system, we used PyTorch 1.6,CUDA 10.2, CUDNN 7.6.5, and NCCL 2.5.6, and MVAPICH2-GDR 2.3.4. On the A100 system, we used PyTorch 1.7, CUDA 11.0, CUDNN 8.0.4, and NCCL 2.7.8.

Experiments/benchmarks included: 1) ResNet-50 with the ImageNet-1k dataset, 2) Mask R-CNN with the Common Object in Context 2014 dataset, 3) a generic U-Net architecture with the LGG Brain MRI Segmentation dataset from Kaggle, and 4) BERT-Large Uncased pretrained on the English Wikipedia and Toronto Bookscorpus datasets and finetuned on the SQuAD v1.1 dataset.

Author-Created or Modified Artifacts:

```
Persistent ID: DOI: 10.5281/zenodo.5163971
Artifact name: KAISA: PyTorch Distributed K-FAC
↪  Optimizer
Citation of artifact: J. Gregory Pauloski, Zhao
↪  Zhang, Lei Huang, Weijia Xu, and Ian T. Foster.
↪  2020. Convolutional neural network training with
↪  distributed K-FAC. In Proceedings of the
↪  International Conference for High Performance
↪  Computing, Networking, Storage and Analysis (SC
↪  20). IEEE Press, Article 94, 114

Persistent ID: DOI: 10.5281/zenodo.4895203
Artifact name: BERT PyTorch Training Code
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Longhorn Subsystem of the Frontera Supercomputer (TACC) with 112 16GB NVIDIA V100 x4 nodes, GPU Subsystem of the Theta Supercomputer (Argonne) with 24 40GB NVIDIA A100 x8 nodes

Operating systems and versions: Longhorn: Red Hat Enterprise Linux Server 7.6 running Linux Kernel 4.14.0-115.10.1.el7a.ppc64le; Theta: SUSE Linux Enterprise Server 15 SP1 running Linux Kernel 4.12.14-197.75-default

Compilers and versions: GCC 7.5.0

Applications and versions: ResNet-50 v1.5, Mask R-CNN, BERT-Large Uncased

Libraries and versions: PyTorch 1.6 and 1.7, CUDA 10.2 and 11.0, CUDNN 7.6.5 and 8.0.4, NCLL 2.5.6 and 2.7.8, MVAPICH2-GDR 2.3.4

Key algorithms: SGD, K-FAC

Input datasets and versions: ImageNet-1k, CIFAR-10, COCO-2014, LGG Segmentation Dataset (https://www.kaggle.com/mateuszbuda/lgg-mri-segmentation), Wikipedia English Dump (Oct '20), Toronto BookCorpus