Baechi: Fast Device Placement of Machine Learning Graphs

Beomyeol Jeon[†], Linda Cai[*], Pallavi Srivastava[◇], Jintao Jiang[‡], Xiaolan Ke[†], Yitao Meng[†], Cong Xie[†], Indranil Gupta[†]
[†]University of Illinois at Urbana-Champaign [*]Princeton University [◇]Microsoft
[‡]University of California, Los Angeles

ABSTRACT

Machine Learning graphs (or models) can be challenging or impossible to train when either devices have limited memory, or the models are large. Splitting the model graph across multiple devices, today, largely relies on learning-based approaches to generate this placement. While it results in models that train fast on data (i.e., with low step times), learning-based model-parallelism is time-consuming, taking many hours or days to create a placement plan of operators on devices. We present the Baechi system, where we adopt an algorithmic approach to the placement problem for running machine learning training graphs on a small cluster of memory-constrained devices. We implemented Baechi so that it works modularly with TensorFlow. Our experimental results using GPUs show that Baechi generates placement plans in time 654×–206K × faster than today's learning-based approaches, and the placed model's step time is only up to 6.2% higher than expert-based placements.

CCS CONCEPTS

• **Computer systems organization → Distributed architectures**.

KEYWORDS

Machine Learning Systems, Placement Algorithms, Constrained Memory, TensorFlow, Distributed Systems

SoCC '20, October 19–21, 2020, Virtual Event, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8137-6/20/10…$15.00
https://doi.org/10.1145/3419111.3421302

ACM Reference Format:
Beomyeol Jeon[†], Linda Cai[*], Pallavi Srivastava[◇], Jintao Jiang[‡], Xiaolan Ke[†], Yitao Meng[†], Cong Xie[†], Indranil Gupta[†]. 2020. Baechi: Fast Device Placement of Machine Learning Graphs. In *ACM Symposium on Cloud Computing (SoCC '20), October 19–21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3419111.3421302

1 INTRODUCTION

Distributed Machine Learning frameworks use more than one device in order to train large models and allow for larger training sets. This has led to multiple open-source systems, including TensorFlow [1], PyTorch [39], MXNet [11], Theano [47], Caffe [22], CNTK [41], and many others [28, 42, 53]. These systems largely use *data parallelism*, wherein each device (GPU) runs the entire model, and multiple items are inputted and trained in parallel across devices.

Yet, the increasing size of Machine Learning (ML) models and scale of training datasets is quickly outpacing available GPU memory. For instance the vanilla implementation of a 1000-layer deep residual network required 48 GB memory [12], which is much larger than the amount of RAM available on a typical COTS (Commercial Off-the-Shelf) device. Even after further optimizations to reduce memory cost, the ML model still required 7 GB memory, making it impossible to run an entire model on a single device with limited memory, as well as prohibitively expensive on public clouds like AWS [4], Google Cloud [16], and Azure [33].

At the same time, today, ML training is gravitating towards being run among small collections of *memory-constrained* devices. These include small groups of cheap devices like edge devices (for scenarios arising from Internet of Things and Cyberphysical systems), Unmanned Aerial Vehicles (UAVs or drones), and to some extent even mobile devices. For instance, real-time requirements [32, 55], privacy needs [9, 10], or budgetary constraints, necessitate training only using nearby or in-house devices with limited resources.

These two trends—increasing model graph sizes and growing prevalence of puny devices being used to train the model graph—together cause scenarios wherein a single device is insufficient and results in an Out of Memory (or OOM) exception. For example, we found that the Google Neural Machine Translation (GNMT) [52] model OOMs on a 4 GB GPU even with conservative parameters: batch size 128, 4 512-unit long short-term memory (LSTM) layers, 30K vocabulary, and sequence length 50.

This problem is traditionally solved by adopting *model parallelism*, wherein the ML model graph is split across multiple devices. Today, the most popular way to accomplish model parallelism is to use *learning-based approaches* to generate the placement of operators on devices, most commonly by using Reinforcement Learning (RL) or variants. Significant in this space are works from Google [34, 35] and the Placeto system [2]. A learning-based approach iteratively learns (via RL) and adjusts the placement on the target cluster, with the goal of minimizing execution time for each training step in the placed model, i.e., its *step time*.

While learning-based approaches achieve step times around those obtained by experts, these approaches still take a very long time to generate a placement plan. For instance, using one state-of-the-art learning-based approach [34], NMT models required 562,500 steps during the learning-based placement, and the runtime per learning step was 1.94 seconds—giving a total placement time of 303.13 hours.

Such long waits are cumbersome and even prohibitive at model development time, when the software developer needs to make many quick and ad-hoc deployments [2]. In fact, studies of analytics clusters reveal that most analytics job runs tend to be short [13, 51]. For instance, the step time for a typical model graph (e.g., NMT or Inception-V3), to train on a single data batch, is O(seconds) on a typical GPU. Overwhelming this time with learning-based placement times which span hours, significantly inhibits the developer's agility.

In fact, the series of works on learning-based placement techniques attempt to address exactly this shortcoming. They progressively improve on training time to place the model [2, 34, 35]. Yet the fastest placement times still run into many hours. One might possibly apply parallelization techniques [23, 24, 50] to the learning model being used to perform placement, in order to speed it up, but the total incurred resource costs would stay just as high—hence, parallelization is orthogonal to our discussion.

Additionally, a learning-based placement run works for a target cluster and a given model graph with fixed hyperparameters (e.g., batch size, learning rate, etc.). If the model graph were to be transitioned to a different cluster with different GPU specs, the learning has to be repeated all over again, incurring high overhead. Next, consider a developer who is trying to find the right batch size for a target cluster.

Table 1: Terms and Notations. *Used in the Paper.*

G	Machine Learning graph to be placed (Classical: Dependency graph of tasks to be placed)
n	Number of operators (or tasks) in G
m	Number of devices in a cluster
M	Memory available per device
d_i	Size of memory required by operator (task) i
ρ	Ratio between maximum operator-to-operator (task-to-task) communication time and minimum per-operator (per-task) computation time
SCT assumption	Small communication time assumption: Ratio between maximum operator-to-operator (task-to-task) communication time and minimum per-operator (task-to-task) computation time is ≤ 1
makespan	Training time for one data mini-batch, i.e., runtime for executing a ML graph on one input mini-batch

This process of exploration is iterative, and every hyperparameter value trial needs a new run of the learning-based technique, making the process slow.

For the model development process to be agile, nimble, and at the same time coherent with future real deployments, what is needed is a new class of placement techniques for model parallelism, that: i) are significantly faster in placement than learning-based approaches, and yet ii) achieve fast step times in the placed model.

In this paper we adopt a traditional *algorithmic* approach for the placement of ML models on memory-constrained clusters. Our contributions are:

- We adapt classical literature from parallel job scheduling to propose two memory-constrained algorithms, called *m-SCT (memory-constrained Small Communication Times)* and *m-ETF (memory-constrained Earliest Task First)*. We also present *m-TOPO (memory-constrained TOPOlogical order)*, a strawman. We focus on the static version of the problem.
- We prove that under certain assumptions, m-SCT's step time is within a constant factor of the optimal.
- We present the *Baechi* system (Korean for *placement*, pronounced "Bay-Chee"). Baechi incorporates m-SCT/m-ETF into TensorFlow, and focuses on design decisions necessary for efficient performance. In spite of apparent similarities with other techniques, no past work [2, 34, 35] deals with *scheduling-related* issues like Baechi does.
- We present experimental results from a real deployment on a small cluster of GPUs, which show that Baechi generates placement plans in time 654×–206K × faster than today's learning-based approaches, and yet the placed model's step time is only up to 6.2% higher than expert-based placements.

2 NEW ALGORITHMS FOR MEMORY-CONSTRAINED PLACEMENT

We present the problem formulation and our three placement techniques. For each technique, we first discuss the classical approach (not memory-aware), and then our adapted memory-constrained algorithm. Where applicable, we prove optimality.

Our three approaches are: 1) a placer based on topological sorting (TOPO) 2) a placer based on Earliest Task First (ETF), and 3) a placer based on Small Communication Time (SCT).

2.1 Problem Formulation

Given m devices (GPUs), each with memory size M, and a Machine Learning (ML) graph G that is a DAG (Directed Acyclic Graph) of operators, the device placement problem is to place nodes of G (operators) on the devices so that the makespan is minimized. Makespan, equivalent to step time, is traditionally defined as the total execution time to train on one input mini-batch (i.e., unit of training data). Table 1 summarizes key terms used throughout the paper. When discussing classical algorithms, we use the classical terms "tasks" instead of operators.

If one assumes devices have infinite (sufficient) memory, *scheduling with communication delay* is a well-studied theoretical problem. The problem is NP-hard even when under the simplest of assumptions [19], such as infinite number of devices and unit times for computation and communication.

Out of the three best-performing solutions to the infinite memory problem, we choose the two that map best to ML graphs: 1) *Earliest Task First or ETF* [21, 49], and 2) *Small Communication Time or SCT* [17]. SCT is provably close to optimal when the ratio of maximum communication time between any two tasks to minimum computation time for any task is ≤ 1. We excluded a third solution, UET-UCT [37], since it assumes unit computation and communication times, but ML graphs have heterogeneous operators.

2.2 m-TOPO: Topological Sort Placer

Background: Topological Sort (Not Memory-Aware). Topological sort [25] is a linear ordering of vertices in a DAG, such that for each directed edge $u \to v$, u appears before v in the linear ordering.

New Memory-Constrained Version (m-TOPO). Our modified version, called *m-TOPO*, works as follows. It calculates the maximum load-balanced memory that will be used per device, by dividing total required memory by number of devices, and then accounting for outlier operators. Concretely, this per-device cap is $Cap = (\sum_{i\in[n]} d_i/m + \max_{i\in[n]} d_i)$. Then m-TOPO works iteratively, and assigns operators to devices in increasing order of device ID. For the current device, m-TOPO places operators until the device memory usage reaches Cap. At that point, m-TOPO moves on to the next device ID, and so on. At runtime, m-TOPO executes the operators at a device in the topologically sorted order.

2.3 m-ETF: Earliest Task First Placer

Background: ETF (Not Memory-Aware). ETF [21] maintains two lists: a sorted task list I containing unscheduled tasks, and a device list P. In I, tasks are sorted by *earliest schedulable time.* The earliest schedulable time of task i is the latest finish time of i's parents in the DAG, plus time for their data to reach i. In P, each device is associated with its earliest available time, i.e., last finish time of its assigned tasks (so far).

ETF iteratively: i) places the head of the task queue I at that device from P which has the earliest available time, ii) then updates the earliest available time of that device to be the completion time of the placed task, and iii) updates earliest schedulable time of that task's dependencies in queue I (if applicable).

The earliest schedulable time of task j on device p is the later of two times: (i) device p's earliest available time ($free(p)$), and (ii) all predecessor tasks of j have completed *and* have communicated their data to j. More formally, let: a) $\Gamma^-(j)$ be the set of j's predecessors; b) for i: start time is s_i, computation time is k_i; c) $x_{ip} = 0$ when task i is on device p, otherwise $x_{ip} = 1$. Then, the earliest schedulable time of task j across all devices is:

$$\min_{p\in P}\left[\max\left(free(p), \max_{i\in\Gamma^-(j)}\left(s_i + k_i + c_{ij}x_{ip}\right)\right)\right]. \quad (1)$$

Under the SCT assumption (Table 1), ETF's makespan was shown [21] to have an approximation ratio of $(2 + \rho - \frac{1}{m})$ within optimal, where ρ is the ratio of the maximum communication time to minimum computation time, and m is the number of devices. This approximation ratio approaches 3 when ρ approaches 1 and $m \gg 1$.

New Memory-Constrained Version (m-ETF). Our new modified algorithm, called *m-ETF*, maintains a queue Q of operator-device pairs (i, p) for all unscheduled operators and all devices. Elements (i, p) in Q are sorted in increasing order of the earliest schedulable time for operator i on device p. This earliest schedulable time takes into account dependent parents of i as well as the earliest time that device p is available, given operators already scheduled at p. The reader will notice that m-ETF's modified queue can also be used for ETF–the key reason to use (i, p) pairs is for m-ETF to do fast searches, since the earliest available device(s) may have insufficient memory.

m-ETF iteratively looks at the head of the queue. If the head element (i, p) is not schedulable because device p's leftover memory is insufficient, then the head is removed. If the head is schedulable, then operator i is assigned to start on device p at that earliest time, and we: i) update p's earliest available time to be the completion time of i, and ii) update i's child operators' earliest schedulable times in queue Q (if applicable).

2.4 m-SCT: Small Communication Time Placer

Background: SCT (Not Memory-Aware). The classical SCT algorithm [17] first develops a solution assuming an infinite number of available devices, and then specializes for a finite number of devices. We elaborate details, as they are relevant to our memory-constrained version.

Classical SCT: Infinite Number of Devices. SCT uses integer linear programming (ILP). The key idea is to find the *favorite child* of a given task i, and prioritize its scheduling on the same device as task i. For a task i, denote its *favorite child* as $f(i)$.

The original ILP specification from [17] solves for variables $x_{ij} \in \{0,1\}$, where $x_{ij} = 0$ if and only if j is i's favorite child. For completeness, we provide this full ILP specification below [17] (Section 3.2 in that paper). Below, the machine learning graph is $G = (V, E)$; and i, j refer to operators.

$$\left\{\begin{array}{ll}
\min w^{\infty} & \text{Minimize makespan.} \\
\forall i \rightarrow j \in E(G),\ x_{ij} \in \{0,1\} & x_{ij} = 0 \text{ when } j \text{ is } i\text{'s favorite child.} \\
\forall i \in V(G),\ s_i \geq 0 & \text{All tasks start after time=0.} \\
\forall i \in V(G),\ s_i + k_i \leq w^{\infty} & \text{All tasks should complete before makespan.} \\
\forall i \rightarrow j \in E(G),\ s_i + k_i + c_{ij}x_{ij} \leq s_j & \text{Given edge } i \rightarrow j \text{, then } j \text{ must start after } i \text{ completes. If on different devices, communication cost should be added.} \\
\forall i \in V(G),\ \sum_{j \in \Gamma^+(i)} x_{ij} \geq |\Gamma^+(i)| - 1 & \text{Every task has at most one favorite child.} \\
\forall i \in V(G),\ \sum_{j \in \Gamma^-(i)} x_{ij} \leq |\Gamma^-(i)| - 1 & \text{Every task is the favorite child of at most one predecessor.}
\end{array}\right. \quad (2)$$

We modify the above as follows. We allow x_{ij} to take any real value between 0 and 1, thus making the ILP a relaxed LP. This can be solved in polynomial time using the interior point method [26]. Then the SCT algorithm simply rounds the LP solution x_{ij} to be 1 if $x_{ij} \geq 0.1$, setting it to 0 otherwise. x_{ij} can be used to determine the favorite child of each task: j is i's favorite child if and only if after rounding, $x_{ij} = 0$.

This infinite device algorithm's makespan was shown [17] to achieve an approximation ratio $\frac{2+2\rho}{2+\rho}$ within optimal, where ρ is the ratio of the maximum communication time to the minimum computation time.

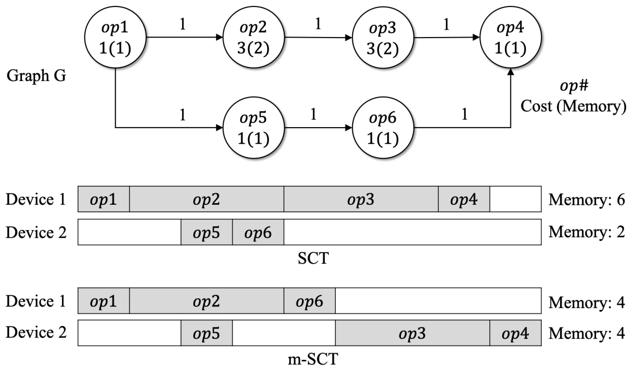

Figure 1: Classical SCT vs. m-SCT. *When per-device memory is limited to 4 memory units, SCT OOMs but m-SCT succeeds. m-SCT's training time (makespan) is only slightly higher (9) than SCT with infinite memory (8).*

We note that the ILP has a meaningful LP relaxation if and only if: (i) infinite number of devices are available, and (ii) the SCT assumption is satisfied, i.e., the ratio of the maximum inter-task communication time to the minimum task computation time is ≤ 1. Nevertheless, even if this assumption were not true for an ML graph and devices, we show later that SCT can still be attractive.

Classical SCT: Extension to Finite Number of Devices. For a finite number of devices, SCT schedules tasks similar to ETF [21], but: a) prefers placing the favorite child of a task i on the same devices as i (each task has at most one favorite child, and at most one favorite parent), and b) prioritizes urgent tasks, i.e., a task that can begin right away on any device.

It was proved that SCT's makespan has an approximation ratio of $(\frac{4+3\rho}{2+\rho} - \frac{2+2\rho}{m(2+\rho)})$ within optimal [17], which is strictly better than ETF's (Section 2.3). For instance, when ρ approaches 1 and $m \gg 1$, then SCT is within $\frac{7}{3}$ of optimal while ETF is 3 times worse than optimal.

New Memory-Constrained Version (m-SCT). Our proposed memory-constrained algorithm, called *m-SCT*, works as follows. First, m-SCT determines scheduling priority and selects devices in the same way as the finite case SCT algorithm just described. Second, when a device p runs out of available memory, m-SCT excludes p from future operator placements.

In spite of the seemingly small difference, Figure 1 shows that m-SCT can succeed where SCT fails. SCT achieves a makespan of 8 time units with infinite memory but OOMs for finite memory. With finite memory, m-SCT succeeds and increases makespan to only 9 time units.

2.5 Optimality Analysis of m-SCT

We now formally prove that m-SCT's approximation ratio to optimal is an additive constant away from SCT's approximation ratio. Since SCT itself was known to be within a constant factor of optimal [17], our result means that m-SCT is also within a constant factor of optimal.

Let $K = \frac{m \cdot M}{\sum_{i=1}^{n} d_i}$, where for any operator (task) i in G, d_i is the size of memory required by i. Intuitively, K is the ratio of the total memory available from all devices to the total memory required by the model.

Theorem 1. *Let the approximation ratio given by the finite SCT algorithm be α, then m-SCT has approximation ratio $\leq \alpha + (1 + \frac{2+2\rho}{(2+\rho)\cdot m}) \cdot \frac{1}{K-1}$.*

Proof. Given $M = \frac{K}{m}\sum_{i=1}^{n} d_i$—from a total of m devices, at most $\frac{m}{K}$ devices would be full (hence dropped) at any time. Thus, at least $\frac{(K-1)\cdot m}{K}$ devices are not dropped throughout the algorithm.

Denote w^{∞} as the makespan of the infinite SCT algorithm, and w^{∞}_{OPT} as the optimal solution. Let w^{m}_{OPTm} and w^{m}_{OPT} respectively be the optimal solutions to the m-device variant with and without memory constraint. Then we have: $w^{\infty}_{OPT} \leq w^{m}_{OPT} \leq w^{m}_{OPTm}$. Intuitively, this is because as one goes from left to right in this inequality, one constrains the problem further.

Since at least $\frac{(K-1)\cdot m}{K}$ devices are always available for scheduling in m-device m-SCT, the m-device m-SCT algorithm generates a makespan T' at least as good as the makespan T generated by running a $(\frac{(K-1)\cdot m}{K})$-device finite SCT algorithm.

We also know from [17] that the m-device finite *SCT* algorithm's makespan $w \leq \frac{1}{m}\sum_{i=1}^{n} k_i + (1 - \frac{1}{m})w^{\infty}$, where $k_i =$ task i's compute time. Hence, the $(\frac{(K-1)\cdot m}{K})$- device finite *SCT* algorithm has makespan T such that:

$$\begin{aligned} T &\leq \frac{1}{\frac{(K-1)m}{K}} \cdot \sum_{i=1}^{n} k_i + (1 - \frac{1}{\frac{(K-1)m}{K}})w^{\infty} \\ &\leq \frac{K}{K-1} w^{m}_{OPT} + (1 - \frac{K}{(K-1)m})w^{\infty}. \end{aligned} \tag{3}$$

The second inequality arises as $\sum_i k_i \leq m \cdot w^{\infty}_{OPT}$, since the device that finishes last needs at least $\frac{1}{m}\sum_i k_i$ time. The approximation ratio of m-device finite SCT algorithm is: $\alpha = 1 + (1 - \frac{1}{m})\beta$, where $\beta = \frac{2+2\rho}{2+\rho}$ [17]. The makespan T' of m-device m-SCT is bounded as:

$$\begin{aligned} T' \leq T &\leq \Big(\frac{K}{K-1} + (1 - \frac{K}{(K-1)m})\beta\Big) w^{m}_{OPTm} \\ &\leq \Big(\frac{1}{K-1} + \frac{\beta}{(K-1)m} + 1 + (1 - \frac{1}{m})\beta\Big) w^{m}_{OPTm}. \end{aligned} \tag{4}$$

Thus *m-SCT* has an approximation ratio $\alpha + (1 + \frac{2+2\rho}{(2+\rho)\cdot m}) \cdot \frac{1}{K-1}$. □

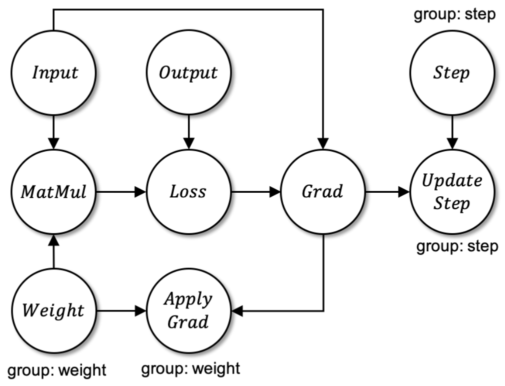

Figure 2: Working Example. *ML Graph for Linear Regression.*

Corollary 1.1. *m-SCT has an approximation ratio $\Big(\frac{K}{K-1} + (1 - \frac{K}{(K-1)m})\frac{2+2\rho}{(2+\rho)}\Big)$ with respect to optimal. When ρ approaches 1, and $m \gg 1$, then the approximation ratio of m-SCT approaches $(\frac{7}{3} + \frac{1}{K-1})$.*

3 BAECHI DESIGN

Our Baechi system has to tackle several challenges in order to incorporate the three placement algorithms just described. For concreteness, we built Baechi to work modularly with TensorFlow [1]. Baechi's challenges are: 1) Satisfying TensorFlow's colocation constraints, 2) Minimizing Data Transfer via Co-Placement, 3) Optimizations to reduce the number of operators to be placed, and 4) Accommodating Sequential and Parallel Communications. Baechi solves these using a mix of both new ideas (Sections 3.1,3.3,3.4) and ideas similar to past work (Sections 3.2,3.3).

Working Example. We use Figure 2 as a working example throughout this section. It is a simplified TensorFlow graph for linear regression training with stochastic gradient descent (SGD).

3.1 TensorFlow Colocation Constraints

The first challenge arises from the fact that TensorFlow (TF) *requires* certain operators to be colocated. For instance, TensorFlow offers a variable operator, `tf.Variable`, which is

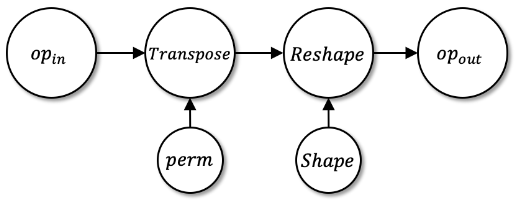

Figure 3: Co-Placement. *Subgraph of* `tf.tensordot` *Generating Data Transfers by m-ETF.*

used to store persistent state such as an ML model parameter. The assignment and read operators of a variable are implemented as separate operators in TensorFlow, but need to be placed on the same device as the variable operator. TensorFlow represents this placement requirement as a *colocation group* involving all these operators. E.g., in Figure 2 there are two colocation groups: one containing `Weight` and `ApplyGrad`, and another containing `Step` and `UpdateStep`.

Baechi's initial placement (using the algorithms of Section 2) ignores colocation requirements. Our first attempt was to *post-adjust placement*, i.e., to "adjust" the device placement, which was generated ignoring colocation, by "moving" operators from one device to another, in order to satisfy TF's colocation constraints. We explored multiple post-adjustment approaches including: i) preferring the device on which the compute-dominant operator in the group is placed, ii) preferring the device on which the memory-dominant operator in the group is placed, and iii) preferring the device on which a majority of operators in the group are placed. We found all these three approaches produced inconsistent performance gains, some giving step times up to 406% worse than the expert. We concluded that post-adjusting was not a feasible design pathway.

Baechi's novel contribution is to *co-adjust placement*, using colocation constraint-based grouping *while* creating the schedule. (In comparison, e.g., ColocRL [35] groups *before* placement.) Concretely, whenever Baechi places the *first* operator from a given colocation group, all other operators in that group are immediately placed on that same device. Baechi tracks the available memory on each device given its assigned operators. If the device cannot hold the entire colocation group, then Baechi moves to the algorithm's next device choice. We found this approach the most effective in practice, and it is thus the default setting in Baechi.

3.2 Co-Placement Optimization

Different from TensorFlow's colocation constraints (Section 3.1), Baechi further prefers to do *co-placement* of certain operators. This is aimed at minimizing data transfer overheads. Common instances include: (i) groups of communicating operators whose computation times are much shorter

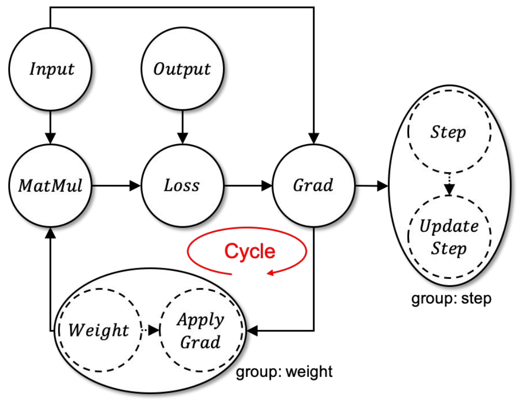

Figure 4: Operator Fusion. *Fused ML Graph Example.*

than their communication times, and (ii) matched forward and backward (gradient-calculating) operators.

Figure 3 shows an example for case (i). This subgraph generated by `tf.tensordot` API is a frequent pattern occurring inside TensorFlow graphs. The subgraph permutes the dimensions of $\texttt{op}_{\texttt{in}}$ output according to the `perm`'s output (`Transpose`) and then changes the tensor shape by `Shape`'s output (`Reshape`).

When m-ETF places this subgraph on a cluster of 3 devices, it places $\texttt{op}_{\texttt{in}}$, `perm`, and `Shape` on different devices. Computation costs for `perm` and `Shape` are very short (because they process predefined values), whereas subsequent communication times are much larger. Thus, m-ETF's initial placement results in a high execution time.

Baechi's co-placement heuristic works as follows. If the output of an operator is only used by its next operator, we place both operators on the same device. This is akin to similar heuristics used in ColocRL [35]. In Figure 3, Baechi's co-placement optimization places all of the operators on one device, avoiding any data transfers among the operators.

For case (ii), to calculate gradients in the ML model, TensorFlow generates a backward operator for each forward operator. Baechi co-places each backward operator on the same device as its respectively-matched forward operator.

Upon placing the first operator in a colocation group, Baechi uses both the co-placement heuristic and the colocation constraints (Section 3.1) to determine which other operators to also place on the same device. Co-placement not only minimizes communication overheads but also speeds up the placement time by reducing the overhead of calculating schedulable times on devices.

3.3 Operator Count Minimization

Placement time can be decreased by reducing the number of operators/groups to be placed. We do this via two additional methods:

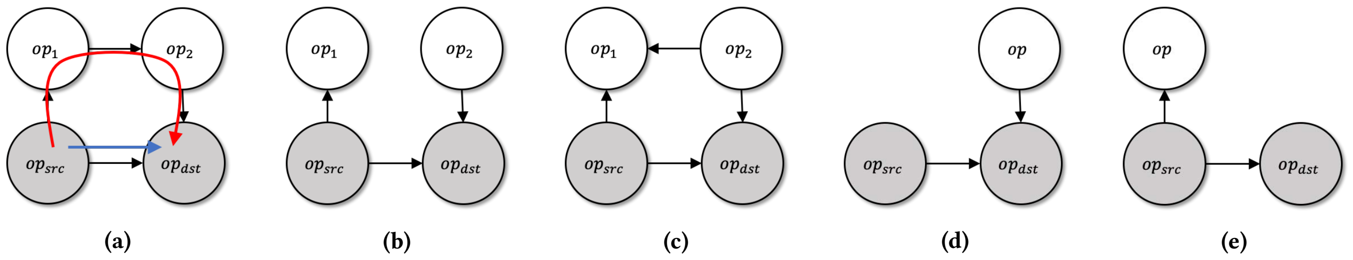

Figure 5: Operator Fusion Without Creating Cycles. *When op_{src} and op_{dst} are fused, some scenarios create a cycle (a), while others do not (b, c, d, e). Baechi fuses operators in a subset of "safe" cases, particularly (d, e).*

i) *Operator Fusion*: Fusing operators that are directly connected and in the same co-placement group; and
ii) *Forward-Operator-Based Placement:* Placing operators by only considering the forward operators.

Operator Fusion. Baechi fuses *operators* using either the colocation constraints (Section 3.1) or co-placement optimizations (Section 3.2). This is new and different from TensorFlow's fusion of *operations*. One challenge that appears here is that this may introduce *cycles* in the graph, violating the DAG required by our algorithms.

Figure 4 shows an example resulting from Figure 2—a cycle is created when Step and UpdateStep are fused into a new meta-operator, and Weight and ApplyGrad are fused.

Consider two nodes–source and destination–with an edge from source to destination. Merging source and destination creates a cycle if and only if there is *at least one additional path from source to destination,* other than the direct edge. Note that there cannot be a reverse destination to source path as this means the original graph would have had a cycle. In Figure 5a, fusing $\mathrm{op}_{\mathrm{src}}$ and $\mathrm{op}_{\mathrm{dst}}$ creates a cycle. Unfortunately, we found that pre-checking existence of such additional paths before fusing two operators is unscalable, because the model graph is massive.

Instead, Baechi realizes that a *necessary* condition for an additional path to exist is that the source has an out-degree at least 2 *and* the destination has an in-degree at least 2 (otherwise there wouldn't be additional paths). Thus Baechi uses a conservative approach wherein it fuses two operators only if the negation is true, i.e., *either* the source has an out-degree of at most 1, *or* the destination has an in-degree of at most 1 (Figures 5d, 5e). This fusion rule misses a few fusions (Figures 5b, 5c) but it catches common patterns we observed, like Figure 5d.

Forward-Operator-Based Placement. When memory is sufficient (i.e., one device could run the entire model), Baechi considers only forward operators for placement and thereafter co-places each corresponding backward (gradient) operators on the same respective device as their forward counterparts. This is a commonly-used technique [2, 35]. This significantly cuts placement time. When device memory is insufficient, Baechi runs the placement algorithms using both forward and backward operators, forcing corresponding pairs to be co-placed using the heuristic of Section 3.2.

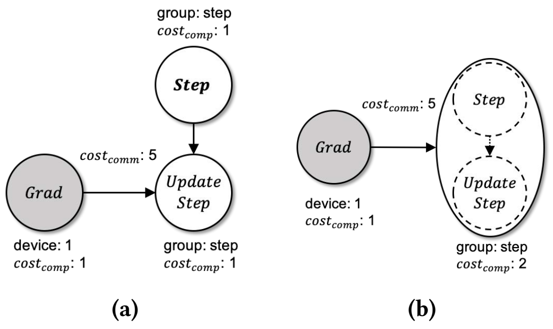

Figure 6: Operator Fusion. *Avoiding Data Transfer Example. (a) Before Fusion. (b) After Fusion.*

Example: Benefits of Fusion. Figure 6a shows the placement of a subgraph of Figure 2 on two devices. Baechi first places Grad on device-1. Baechi places the next operator, Step on the idle device-2, and colocates (due to TF constraints) UpdateStep on device-2. This creates communication between the devices. Assuming operators' compute costs are 1, and communication cost between Grad and UpdateStep is 5, this results in an execution time of 7 time units.

On the other hand, Figure 6b shows that Baechi merges Step and UpdateStep with operator fusion. Since this meta-operator's schedulable time on device-1 is earlier than on device-2 due to communication overhead, Baechi places it on device-1. Fusion lowers total execution time to 3 time units.

Loops in the Original Model Graph. Different from the cycles discussed above, some network graphs consist of loops, e.g., RNNs. We use the unrolled ML graph [3] to turn the graph into a DAG, and then apply Baechi's techniques.

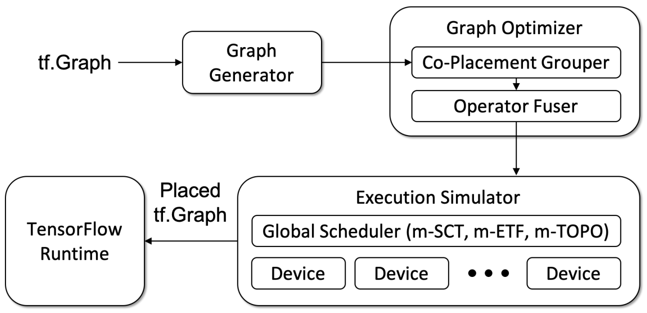

Figure 7: Baechi Input and Output.

3.4 Sequential vs. Parallel Communication

Our algorithms from Sections 2.3 and 2.4 assume that each operator can send data simultaneously to its children. Baechi also proposes a new way to deal with environments involving constrained networks (including our deployment in Section 5), where data transfer is sequential. For networks that limit each device to do at most one transfer at a time (out or in), Baechi assumes communication queues at devices. Concretely, when a data transfer between two devices is requested, Baechi assumes the request is put into the respective devices' communication queues and processed sequentially at both ends. During placement, Baechi calculates the wait time at the communication queues and adds it to the earliest schedulable time computed for the operator. Specifically the queue wait time is added to equation (1) in Section 2.3. Otherwise, normal m-SCT/m-ETF apply, as described earlier.

4 IMPLEMENTATION

In order to integrate modularly with TensorFlow (TF) [1] v1.12, Baechi adopts the workflow shown in Figure 7. Baechi executes the following steps: 1) its *Graph Generator* takes as input a TF graph containing operators, 2) Baechi's *Graph Optimizer* uses the design of Section 3 to account for TF colocation constraints, and applies co-placement and operator fusion, 3) Baechi's *Execution Simulator (ES)* executes our placement algorithms (m-TOPO, m-ETF, or m-SCT), and 4) finally Baechi outputs a TF graph with operators assigned to devices. This output graph is fed to TF runtime.

4.1 Graph Generator and Graph Optimizer

Baechi parses the input TF graph and generates an equivalent NetworkX [40] graph. The NetworkX format allows Baechi to both store operator execution metadata (computation and communication times, memory needed, etc.), and to easily manipulate the graph (e.g., fuse operators).

Baechi uses the standard TensorFlow profiling tool to obtain computation time and memory allocation for each operator. TF profiler returns allocation information for temporary, persistent, and output tensor memory. The temporary memory is allocated at the beginning of an operation and deallocated when the operation finishes. The persistent memory is allocated and used over the entire execution, e.g., to store persistent states such as weights.

For communication time, we use a linear model proportional to data size. We implemented a microbenchmark tool atop TensorFlow to measure communication times for various data sizes, and generated a communication cost function through the linear regression.

Then Baechi applies the co-placement optimizations (Section 3.2), and optimizations like operator fusion (Section 3.3) on the graph.

4.2 Execution Simulator

Baechi's Execution Simulator (ES) takes as input the operator graph from the Graph Optimizer and outputs the TensorFlow graph in which all operators are assigned to devices. Our first attempt was in fact to try repurposing TensorFlow's simulator, but it assumed operators were already placed, assumed zero communication cost, and ignored caching. This motivated the design of Baechi's new ES uniquely for memory-constrained placement.

The ES consists of: a) a global scheduler, and b) simulated devices (with specs identical to deployment). The global scheduler maintains a single queue with operators that are ready to run. The scheduler extracts operators from its queue and applies our scheduling algorithms (m-TOPO, m-ETF, m-SCT) to place them on devices.

In ES, each device has two FIFO queues, one for operators and one for data transfer. This allows data transfer to overlap with operator execution. When a device receives a tensor from another device, it caches the tensor to avoid duplicate data transfer.

Dynamic Memory Allocation. Calculating a device's memory usage as the sum total of all its assigned operators results in an over-estimate. For example, Inception-V3 with batch size 32 can execute using 4 GB even though its operators' memory needs add up to 22 GB.

Baechi's ES tracks an estimate of memory usage during its placement. When an operator executes on a device, the device allocates temporary memory, and separate memory for its output tensors. The temporary memory is deallocated when the operator finishes. The output tensor memory is deallocated after all its successors finishes except the persistent memory of the operator. If a device's memory becomes full, the device can be removed–this never happens in practice as usually a device has at least a few bytes left. This loosely parallels the way in which TF manages memory. Baechi reserves memory for a colocation group at device p when the first operator is placed on p (Section 3.1). The reserved memory is deallocated when all the operators in the group finish.

Linear Programming Solver. To solve the SCT LP problem, we use the interior point method. This is preferable over other solvers such as simplex [7] as it guarantees polynomial execution time [48]. Concretely, we use the primal dual interior-point solver via Mosek optimization [5], which has a run time complexity of $O(n^{3.5}L)$, where L is the maximum number of bits in the LP input, and n is the number of variables.

4.3 Miscellaneous Issues

We discuss a few key miscellaneous aspects.

LP Modifications. The ILP solutions (Section 2.4) resulted in more than one favorite child (or parent) being selected for certain nodes. In Baechi we lowered the rounding threshold from 0.5 to below 0.2. This eliminated all violations, and avoided nodes from having multiple favorite children. (We use threshold = 0.1 in practice.)

Ignoring Bootstrap Steps in Profiling. 1) In a training run of a model graph, step times are initially high due to TF bootstrapping. We estimate step times in steady state, after a few iterations have passed. 2) Some TF operators are implemented with multiple GPU kernels. When profiling these operators, we include multiple kernel executions, in order to avoid underestimation. This is similar to TF's cost model [1].

5 EVALUATION

Our evaluation answers the following five questions:
1. How fast is Baechi's *placement time*, i.e., how quickly do our algorithms find placements? (Section 5.2)
2. How fast are the *step times* of the placement generated by Baechi, i.e., training time per step of the placed model? (Section 5.3)
3. How do the Baechi's step times change when there is *insufficient memory* per GPU? (Section 5.4)
4. How do the step times for Baechi compare to *single GPU* and *expert placements*? (Section 5.3)
5. How much is the benefit due to Baechi's *optimizations* from Sections 3.2 and 3.3? (Section 5.5)

5.1 Experimental Settings

We use two popular ML benchmarks: Inception-V3 and Google Neural Machine Translation System (GNMT). These two models were chosen because: i) they are respectively considered the best representatives of vision and Natural Language Processing (NLP) models, and ii) Past work [2, 34, 35] used Inception-V3 and NMT (GNMT is a more complex version), thus allowing us to compare.

Benchmark: Inception-V3. Inception-V3 [44] is a convolutional neural network architecture that is widely used for image classification. This model is composed of multiple blocks called Inception modules. The Inception modules consist of branches of convolutional and pooling operators. To train the model, we use RMSProp [18] and batch sizes of both 32 and 64.

Benchmark: GNMT. Google Neural Machine Translation System (GNMT) [52] is a language model for automated translation. GNMT consists of: encoder and decoder modules, each a stack of recurrent neural networks (RNNs); and the attention module to process long sequences effectively. We use 4 long short-term memory (LSTM) layers of the encoder and the decoder layers with residual connections, and the Bahdanau attention mechanism [6]. We use the LSTM hidden size of 512, the vocabulary size of 30,000, the unrolled RNNs with the sequence length of 40 and 50, and the batch size of 128 and 256. Baechi applies the co-placement optimization to LSTM cell operators and also to attention operators.

Compared to Inception-V3, GNMT has fewer barriers (sync points) inside its model graph, indicating that GNMT has a higher potential to benefit from Baechi's parallel placements.

Machine Setup. All experiments are run on our local server that has 4 NVIDIA GTX 2080 GPUs, with 8 GB per-GPU memory (the machine also has an Intel i9-7960X CPU, but this is not used to execute operators). GPUs are connected to CPUs via PCIe 3.0 x16 (we do not use NVLink [38]). All data transfers go through the host memory (no P2P communication among GPUs). This results in a slow IO bus, and we believe this high ratio of communication overhead to computation overhead is representative of realistic scenarios like the kinds outlined in Section 1.

To generate operator execution information, we sample 5 TensorFlow profiling runs after 5 warm-up training steps. We average these to generate compute costs. We place all GPU-supported operators only on GPUs.

Approach to Comparison. To compare Baechi, we extract the best performing numbers from each of the three previous works [2, 34, 35]. Directly running these other systems was complicated by lack of uniform availability of working code—Placeto's code [2] missed key optimizations; ColocRL [35] is proprietary; only HierarchicalRL's code [34] was available, but it was slow and generated inefficient placements. E.g., For GNMT, HierarchicalRL took 12 hours+ to run placement (batch size 128, length 50) and the resultant step time was much higher than expert's, contrary to HierarchicalRL paper's claims. Essentially, direct comparison would be unfair to these other papers without knowing the exact hyperparameters they used to achieve their "best" performance. In light of this, our comparison gives the benefit of doubt to,

Table 2: Placement Time. *Time to Generate a Placement for our target machine with 4 GPUs.*

Model	HierarchicalRL [34]	Placeto [2]	Baechi (m-SCT)
Inception-V3	11 hrs 50 mins	1 hr 49 mins	1-10 seconds
NMT (GNMT)	1 day 21 hrs 14 mins	2 days 20 hrs 40 mins	1.2-48 seconds

and uses the best performance from, these learning-based placement papers. All the above papers compared step times to experts, and we do too.

5.2 Placement Time

Table 2 shows both: 1) measured placement times of Baechi, and 2) calculated placement times for two learning-based techniques, namely: HierarchicalRL [34] and Placeto [2]. The numbers for HierarchicalRL and Placeto are normalized quantities, both derived from numbers reported in Addanki et al. [2]. For these two systems, we multiply the fastest step time among its reported placements, by the number of placement samples[1]. For instance, HierarchicalRL's [34] Inception-V3 placement training time is derived as a product of the reported final step time (1.19 s) and the number of samples (35,800), giving 42,602 s, or 11 hrs 50 mins.

Hence, the numbers for these learning-based placers are their best-case performance. In comparison, we use the worst-case placement times from Baechi, specifically from m-SCT which took the longest to generate a placement. Note that all times in Table 2 exclude time to profile the graph, as profiling is a common baseline encountered by all the three approaches shown. We find the profiling time to be low, about 10–12 s total for Inception-V3 and GNMT. This breaks down as 2-4 s for warmup execution, 1–3 s for graph execution for profiles, and less an 1 s for parsing profile results.

Table 2 shows that Baechi places ML models orders of magnitude faster than the learning-based approaches. For Inception-V3, Baechi reduces placement time, from 1.8–11.8 hours, to under 10 s. Thus Baechi is 654×–42.6K× faster at placing Inception-V3. For GNMT, Baechi reduces placement time from several days to under 48 s. Thus Baechi is 3392×–206K× faster at placing GNMT.

Overall, Baechi is 654×–206K× faster at placement compared to today's learning-based approaches [2, 34].

5.3 Placement with Sufficient Memory

We next evaluate the effectiveness of the generated placement by measuring the step time of the placed model, i.e., its time to execute 1 training step on an input data batch. We first explore the scenario when each GPU has sufficient memory to run the entire model. We compare against both: i) step time on a single GPU, which might be fast because it avoids the overheads of communication, and ii) an *expert*-based placement scheme for placement on multiple GPUs.

The expert is a manual process and we do it as follows. For GNMT, we use the technique of Wu et al. [52]. Each LSTM layer in the encoder and decoder modules are placed on different GPUs. The embedding layer is placed on the same GPU as the first LSTM layer. The output projection layer is placed on the same GPU as the last decoder LSTM layer. For Inception-V3, the expert is the single GPU placement, similar to HierarchicalRL [34].

m-ETF, m-SCT –VS.– Single GPU, Expert. Table 3 shows the step times for the three algorithms in Baechi–namely m-TOPO, m-ETF, and m-SCT—as well as the single GPU and expert. We show numbers for 2 batch sizes in each model, and 2 sequence lengths in GNMT. We observe that for Inception-V3: 1) m-ETF and m-SCT find the same device placements as the expert, i.e., place all operators in a single GPU, and 2) m-TOPO has 6.1–6.3% higher step time than the expert. This occurs because m-TOPO splits the neural network between the Inception blocks, and hence the next inception block(s) are unable to run until the previous block(s) finish.

In GNMT, first, compared to single GPU placement, m-ETF's placements have step times that are 12.1–33.9% faster. The step time speedups for m-SCT over single GPU are between 18.4–28.5%. These observations show that Baechi's m-ETF and m-SCT are able to extract benefits of parallelism in spite of communication overheads.

Second, in GNMT, compared to the expert, m-ETF is between 4.5% slower and 6.2% faster in step times. Compared to the expert, m-SCT is between 6.2% slower and 1.9% faster. These observations show that Baechi's m-ETF and m-SCT are able to generate placements with step times in the same ballpark as the expert, while taking significantly less time to create a placement than the manual expert which takes minutes to hours.

m-TOPO. Baechi's m-TOPO is significantly slower than m-ETF and m-SCT. m-TOPO's step times are 5.8%–26.4% slower than m-ETF and 5.8%–23.3% slower than m-SCT. After a deep dive into m-TOPO we found that it places most of the encoder's LSTM layers at the first two GPUs, and most of the decoder LSTM layers at the other two GPUs. However, this parallelization is offset negatively by the high data transfers between the kernel weight and the LSTM cell operators for LSTM layers.

m-SCT vs. m-ETF. Our theoretical analysis in Section 2.5 predicts m-SCT beating m-ETF. In practice, m-ETF's step times are faster than m-SCT's for three out of 4 settings in GNMT (it is faster only under sequence length 40, batch size 128).

[1]Even if one were to parallelize the learning-based placers, their resource usage would be similar to the normalized time metric we show.

Table 3: Baechi with Sufficient Memory. *Average Step Times (Training) in seconds of Placed Model Graphs, and Speedup over Single GPU and Expert Placements. 4 GPUs (unless otherwise mentioned).*

							Speedup over			
							Single GPU		Expert (4 GPUs)	
Model	Batch Size	Single GPU	Expert	m-TOPO	m-ETF	m-SCT	m-ETF	m-SCT	m-ETF	m-SCT
Inception-V3	32	0.269	0.269	0.286	0.269	0.269	0.00% (1 GPU Expert)			
	64	0.491	0.491	0.521	0.491	0.491	0.00% (1 GPU Expert)			
GNMT (length: 40)	128	0.251	0.214	0.265	0.224	0.212	12.1%	18.4%	-4.5%	0.9%
	256	0.474	0.376	0.481	0.354	0.369	33.9%	28.5%	6.2%	1.9%
GNMT (length: 50)	128	0.319	0.259	0.348	0.264	0.267	20.9%	19.5%	-1.9%	-3.0%
	256	0.618	0.484	0.609	0.502	0.516	23.1%	19.8%	-3.6%	-6.2%

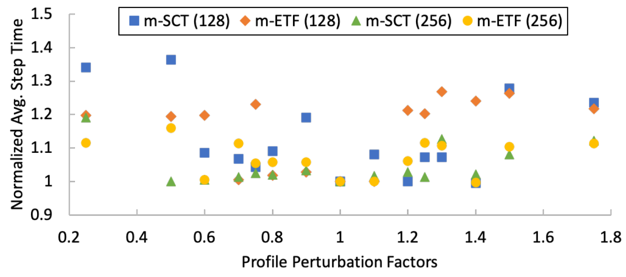

Figure 8: Baechi Sensitivity to Profiling Errors (GNMT). *X-axis shows perturbation multiplied into profile.*

Table 4: Baechi with Insufficient Memory. *Average Step Times (Training) in seconds of Placed Model Graphs (Parentheses show Slowdown compared to Sufficient Memory).*

Model	Single GPU	Expert	m-TOPO	m-ETF	m-SCT
Inception-V3	OOM	OOM	0.690 (58.6%)	0.312 (13.8%)	0.292 (7.9%)
GNMT	OOM	0.221 (3.2%)	0.272 (2.6%)	0.230 (2.6%)	0.212 (0.0%)

This practical suboptimal behavior of m-SCT is because of two reasons. First, SCT's optimality proof relies on the assumption that the minimum operator computation time is larger than or equal to the maximum communication time. This does not hold in our experimental machine—a 4 B GPU-GPU transfer takes 50–200 ms while many operators execute within 1 ms, and 67% of Inception-V3's operators take under 50 ms. Second, the m-SCT LP model (Section 2.4) assumes parallel data transfers from an operator to all its children. Our experimental machine only allows sequential transfers (Section 3.4)[2]. Overall, m-SCT and m-ETF are comparable in practice, with m-ETF having a slight edge in both placement time and step time.

Profile Sensitivity. To measure Baechi's sensitivity to profiling errors, we perform runs where in each run we multiply the computation and communication profiles by a perturbation factor. Figure 8 shows that: (i) both m-SCT and m-ETF are largely insensitive to profiling errors—step times do not deviate much up to 20% profile error, and from 20%-80% profile error step times deviate only up to 30%; (ii) m-ETF is less sensitive than m-SCT; and (iii) higher batch sizes (256 vs. 128) imply lower sensitivity. We explain the reasons behind observations (ii) and (iii) as follows. Observation (ii) occurs because m-ETF's heuristic is less sensitive to profile errors than m-SCT's favorite child selection—all else being equal, increasing profile perturbation affects m-ETF later than m-SCT. Observation (iii) is because increasing batch sizes increases tensor sizes. Consequently, communication times rise much faster relative to computation time increase. This raises wait times for tensor arrival during execution, and this wait time intuitively acts as a buffer that reduces the effect of profiling errors.

5.4 Placement with Insufficient Memory

Next, we limit per-GPU memory to 30% of total available memory, i.e., from 8 GB down to 2.4 GB. Table 4 shows results for Inception-V3 with batch size of 32, and GNMT with batch size of 128 and sequence length 40.

A few notes on configuration changes in the experiments. For GNMT, co-placement (Section 3.2) remains enabled and we use the same configuration as Section 5.3. For Inception-V3, we disable co-placement as otherwise it generated a massive operator group, causing an Out of Memory error

[2]Faster data transfers between GPUs, e.g., via NVLink [38], have the potential to make m-SCT more competitive than m-ETF, but this is outside our scope.

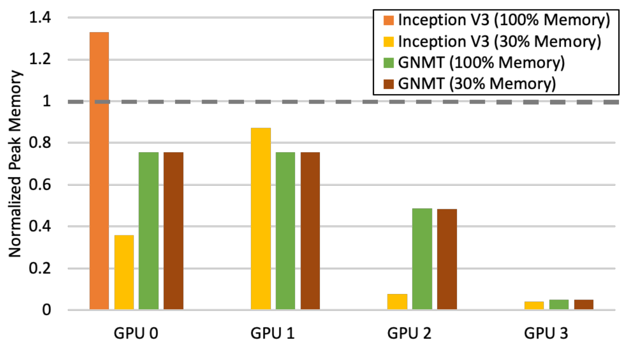

Figure 9: Baechi Load Balance of Memory Usage. *Dashed line is memory limit for each GPU (normalized).*

(OOM). Disabling co-placement increases the number of operators to be placed from 2,620 to 7,077, and placement time from 1 s to 10.3 s.

Effect on Step Time. Table 4 shows that the single GPU placer always suffers an OOM (Out of Memory) error. The expert placer OOMs for Inception-V3, but succeeds for GNMT. In comparison, all three variants of Baechi succeed in placing under insufficient memory (m-TOPO, m-ETF, m-SCT).

For Inception-V3, only Baechi succeeds in placement. m-ETF and m-SCT provide step times that are only 13.8% and 7.9% worse respectively than the sufficient memory cases (i.e., Table 3). m-TOPO degrades by 58.6% because of its disabled co-placement, which ballooned communication along the graph's critical path.

For GNMT, the overheads of all three Baechi algorithms and the expert are small, making them comparable to the sufficient memory numbers.

Load Distribution. Figure 9 shows the peak memory usage, normalized to the memory limit for each GPU (insufficient memory case). For Inception-V3, with a 30% memory cap, a single GPU does not suffice, and that m-SCT relies on a mix of multiple GPUs. In particular, 2 of the 4 GPUs appear to be used more. This is because Inception-V3 has more barriers (sync points) than GNMT, limiting Inception-V3's ability to parallelize effectively.

For GNMT, Baechi's m-SCT is able to more evenly load balance (than Inception-V3) across the 4 GPUs, even when memory is sufficient. In fact, we found that m-SCT generates an identical placement in both cases with sufficient and with insufficient memory. While this fact is also true for the expert, m-TOPO, and m-ETF, their step times are 2.6–3.2% higher than the sufficient memory cases (Table 3). This slowdown is because of TensorFlow runtime memory optimizations. When the memory usage approaches its limit, the TensorFlow runtime resorts to certain memory optimizations to decrease peak memory usage. For the expert placement, peak

Table 5: Benefits of Baechi Optimizations. *Number of Operators to be Placed, Placement Times in seconds, and Average Step Times in seconds. m-SCT.*

Model	Un-Optimized			Optimized		
	Num. Ops	Placement (seconds)	Step (seconds)	Num. Ops	Placement (seconds)	Step (seconds)
Inception-V3	6884	68.0	0.302	17	0.9	0.269
GNMT (length: 40)	18050	275.1	0.580	542	1.2	0.212
GNMT (length: 50)	22340	406.1	0.793	706	2.4	0.267

memory usage for one GPU device decreases from 2 GB (83% of the memory limit) to 1.45 GB and thus the number of memory operations increases 6% under insufficient memory. These memory optimizations do not kick in for m-SCT, making it faster than the expert.

5.5 Benefit of Baechi Optimizations

Table 5 shows the benefit from the combined optimizations of Section 3.2 and 3.3. Inception-V3 with batch size 32 and GNMT with batch size of 128 are used. We use the m-SCT variant of Baechi. The experimental setup has 4 GPUs with sufficient memory.

Overall, we observe that Baechi's combined optimizations achieve 75.6×–229.3× speedup in placement times, and 1.1×–3.0× speedup in step times. We discuss a few interesting aspects. Operator fusion (Section 3.3) reduces both number of operators to be placed and thus also placement time. Forward-operator-based placement (Section 3.3) significantly speeds up placement. Concretely the latter optimization reduces the number of operators to be placed 2.7× for Inception-V3 and 6.5×–7.0× for GNMT. This accelerates the placement times 13.7× for Inception-V3 and 20.2×–31.4× for GNMT.

Co-placement (Section 3.2) is efficient because it clusters operators. This reduces step times. While co-placement does not change the operator count to be placed, it decreases placement time by reducing the overhead of calculating schedulable times.

6 RELATED WORK

Data Parallelism. Data parallelism refers to training the same model replicas with multiple partitioned data in parallel. This is motivated by increasing sizes of datasets. MALT [31] is a fault-tolerant, network-cost effective solution for data parallel ML. Another common data parallelism framework is NESL [8], a first-order functional language that enables developers to put irregular-parallel program in parallel devices. OptiML [43] is a domain-specific language (DSL). Most major ML frameworks offer support for data parallelism [1, 11, 39].

Model Parallelism. Compared to data parallelism, relatively fewer solutions exist for model parallelism. DistBelief [14] and STRADS [27] require the user to manually specify device placement, while the systems in [29, 30] do not generalize to arbitrary ML models.

As discussed in Section 1, reinforcement-learning based approaches have been popular lately to perform placement for model parallelism, including work from Google [34, 35] and the Placeto system [2]. ColocRL [35] trains a sequence-to-sequence model by RL to generate placements of manually grouped subsets of TensorFlow operators. HierarchicalRL [34] substitutes the human intervention for grouping operators with an ML model and jointly trains the ML models for operator grouping and device placements. Placeto [2] proposes an approach that transfers learned device placement models to new ML models in order to minimize training times for the new model placements.

Classical Parallel Scheduling. Classical parallel scheduling, e.g., ETF [21] and SCT [17], has been widely used in task scheduling on multiple computers. ETF and SCT are used as baselines by many subsequent works [15, 20, 36, 49, 54]. None of these address memory constraints and a finite number of devices. For instance, Eyraud-Dubois et al. [15] investigate the execution of tree-shaped task graphs using multiple processors, but without always obeying memory restrictions.

TensorFlow Graph Optimizations. Existing techniques [45, 46] work only *after* the graph has been placed—e.g., to improve operations' performance—and thus are inapplicable. E.g., Running Grappler (TensorFlow's graph optimizer) generates an optimized graph protobuf, but it is unusable as it lacks certain metadata. Baechi's targeted problem is harder as we have to both optimize the graph and do placement.

7 CONCLUSIONS

Summary. We have proposed algorithmic solutions to model parallelism, useful in scenarios where devices are memory-constrained or neural networks are massive. Among our three algorithms (m-ETF, m-TOPO, m-SCT), the m-SCT algorithm is provably within a constant factor of the optimal achievable training time. We have implemented these algorithms into our new Baechi system, which can be used alongside TensorFlow.

Experimental results on a 4 GPU setup showed our approaches reduce placement time by a factor of between 654×–206000× compared to today's state-of-the-art placement approaches which are learning-based, while increasing step time (makespan) by only up to 6.2% compared to expert placers. When memory is constrained further, while single GPU and expert placers suffer OOM errors, Baechi's algorithms, especially m-SCT and m-ETF, were able to place successfully while suffering only up to a 13.8% increase in step time compared to sufficient memory. Baechi's optimizations help reduce placement time by 75.6×–229.3×, and step time by 1.1×–3.0×. We also conclude that m-SCT and m-ETF perform comparably, with m-ETF having a slight edge for slower networks.

Retrospective. When we first implemented m-ETF and m-SCT, the placed models had very high step times because communication-intensive operators violated the SCT assumption (Table 1). We whittled away at this with a persistent effort at systems design and optimizations (outlined in Section 3), which played a major role in bringing the step times down. Although our exploration was efficient and we cycled new techniques and optimizations on a weekly basis, it took 1 man-year of effort to converge to what now appears in this paper. This is indicative of the difficulties associated with implementing scheduling algorithms on today's open-source ML systems (and in a sense shows why existing learning-based approaches are so attractive!). Nevertheless, our results show that the benefits of algorithmic design are worth our exploratory pain.

We believe that our work opens up a new direction for solid algorithmic exploration in the problem of model parallelism, with the promise of speed, generalizability, predictability, and analyzability. For instance, federating learning introduces the need to solve the dynamic version of the memory-constrained problem, where devices continuously join, leave, and fail.

Code. Baechi's code is openly available at the following link: http://dprg.cs.uiuc.edu/downloads.php

ACKNOWLEDGMENTS

This work was supported in part by the following grants: NSF IIS 1909577, and NSF CNS 1908888; as well as by generous gifts from Capital One, Schlumberger, and Microsoft.

We thank our shepherd Sundar Dev and our anonymous reviewers for their invaluable feedback that helped improve this paper.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, 265–283.

[2] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. 2019. Learning Generalizable Device Placement Algorithms for Distributed Machine Learning. In *Advances in Neural Information Processing Systems 32 (NeurIPS '19)*. Curran Associates, Inc., 3981–3991.

[3] Frances. E. Allen and John Cocke. 1972. A Catalogue of Optimizing Transformations. *Design and Optimization of Compilers* (1972), 1–30.

[4] Amazon. 2020. Amazon Web Services (AWS). https://aws.amazon.com

[5] Erling D. Andersen and Knud D. Andersen. 2000. The Mosek Interior Point Optimizer for Linear Programming: An Implementation of the Homogeneous Algorithm. In *High Performance Optimization*. Springer, 197–232.

[6] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *3rd International Conference on Learning Representations (ICLR '15)*.

[7] Richard H. Bartels and Gene H. Golub. 1969. The Simplex Method of Linear Programming Using LU Decomposition. *Commun. ACM* 12, 5 (1969), 266–268.

[8] Lars Bergstrom and John Reppy. 2012. Nested Data-parallelism on the GPU. In *17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. ACM, 247–258.

[9] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konecny, Stefano Mazzocchi, H Brendan McMahan, et al. 2019. Towards Federated Learning at Scale: System Design. *arXiv preprint arXiv:1902.01046* (2019).

[10] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. 2017. Practical Secure Aggregation for Privacy-Preserving Machine Learning. In *24th ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, 1175–1191.

[11] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *arXiv Preprint arXiv:1512.01274* (2015).

[12] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets With Sublinear Memory Cost. *arXiv preprint arXiv:1604.06174* (2016).

[13] Yanpei Chen, Sara Alspaugh, and Randy Katz. 2012. Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. *Proceedings of the VLDB Endowment* 5, 12 (2012).

[14] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems 25 (NIPS '12)*. Curran Associates Inc., 1223–1231.

[15] Lionel Eyraud-Dubois, Loris Marchal, Oliver Sinnen, and Frédéric Vivien. 2015. Parallel Scheduling of Task Trees with Limited Memory. *ACM Transactions on Parallel Computing* 2, 2 (2015), 1–37.

[16] Google. 2020. Google Cloud Platform. https://cloud.google.com/gcp/

[17] Claire Hanen and Alix Munier. 1995. An Approximation Algorithm for Scheduling Dependent Tasks on m Processors with Small Communication Delays. In *4th INRIA/IEEE Symposium on Emerging Technologies and Factory Automation (ETFA '95)*, Vol. 1. IEEE, 167–189.

[18] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. 2012. Neural Networks for Machine Learning Lecture 6a Overview of Mini-Batch Gradient Descent. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[19] J.A. Hoogeveen, Jan K. Lenstra, and Bart Veltman. 1994. Three, Four, Five, Six, or the Complexity of Scheduling with Communication Delays. *Operations Research Letters* 16, 3 (1994), 129–137.

[20] Jinhua Hu, Jianhua Gu, Guofei Sun, and Tianhai Zhao. 2010. A Scheduling Strategy on Load Balancing of Virtual Machine Resources in Cloud Computing Environment. In *3rd International Symposium on Parallel Architectures, Algorithms and Programming (PAAP '10)*. IEEE, 89–96.

[21] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D. Anger, and Chung-Yee Lee. 1989. Scheduling Precedence Graphs in Systems with Interprocessor Communication Times. *SIAM Jornal on Computing* 18, 2 (1989), 244–257.

[22] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *22nd ACM International Conference on Multimedia (MM '14)*. ACM, 675–678.

[23] Zhihao Jia, Sina Lin, Charles R. Qi, and Alex Aiken. 2018. Exploring Hidden Dimensions in Accelerating Convolutional Neural Networks. In *35th International Conference on Machine Learning (ICML '18)*. PMLR, 2274–2283.

[24] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. In *2nd Conference on Machine Learning and Systems (MLSys '19)*. 1–13.

[25] Arthur B. Kahn. 1962. Topological Sorting of Large Networks. *Commun. ACM* 5, 11 (1962), 558–562.

[26] Narendra Karmarkar. 1984. A new Polynomial-Time Algorithm for Linear Programming. In *16th Annual ACM Symposium on Theory of Computing (STOC '84)*. ACM, 302–311.

[27] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A. Gibson, and Eric P. Xing. 2016. STRADS: A Distributed Framework for Scheduled Model Parallel Machine Learning. In *11th European Conference on Computer Systems (EuroSys '16)*. ACM, Article 5, 16 pages.

[28] Tim Kraska, Ameet Talwalkar, and John Duchi. 2013. MLbase: A Distributed Machine-Learning System. In *6th Biennial Conference on Innovative Data Systems Research (CIDR '13)*.

[29] Alex Krizhevsky. 2014. One Weird Trick for Parallelizing Convolutional Neural Networks. *arXiv preprint arXiv:1404.5997* (2014).

[30] Quoc V Le. 2013. Building High-Level Features Using Large Scale Unsupervised Learning. In *38th IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '13)*. IEEE, 8595–8598.

[31] Hao Li, Asim Kadav, Erik Kruus, and Cristian Ungureanu. 2015. MALT: Distributed Data-Parallelism for Existing ML Applications. In *10th European Conference on Computer Systems (EuroSys '15)*. ACM, Article 3, 16 pages.

[32] Mohammad Saeid Mahdavinejad, Mohammadreza Rezvan, Mohammadamin Barekatain, Peyman Adibi, Payam Barnaghi, and Amit P Sheth. 2018. Machine Learning for Internet of Things Data Analysis: A Survey. *Digital Communications and Networks* 4, 3 (2018), 161–175.

[33] Microsoft. 2020. Microsoft Azure. https://azure.microsoft.com/

[34] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V Le, and Jeff Dean. 2018. Hierarchical Planning for Device Placement. In *6th International Conference on Learning Representations (ICLR '18)*.

[35] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device Placement Optimization with Reinforcement Learning. In *34th International Conference on Machine Learning (ICML '17)*. PMLR, 2430–2439.

[36] Rolf H. Möhring, Markus W. Schäffter, and Andreas S. Schulz. 1996. Scheduling Jobs with Communication Delays: Using Infeasible Solutions for Approximation. In *4th Annual European Symposium on Algorithms (ESA '96)*. Springer, 76–90.

[37] Alix Munier and Jean-Claude König. 1997. A Heuristic for a Scheduling Problem with Communication Delays. *Operations Research* 45, 1 (1997), 145–147.

[38] NVIDIA. 2020. NVLink and NVSwitch. https://www.nvidia.com/en-us/data-center/nvlink/

[39] 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. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *33rd Conference on Neural Information Processing Systems (NeurIPS '19)*. Curran Associates, Inc., 8024–8035.

[40] Daniel A. Schult. 2008. Exploring Network Structure, Dynamics, and Function Using NetworkX. In *7th Python in Science Conference (SciPy '08)*. 11–15.

[41] Frank Seide and Amit Agarwal. 2016. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. ACM, 2135–2135.

[42] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska. 2013. MLI: An API for Distributed Machine Learning. In *13th IEEE International Conference on Data Mining (ICDM '13)*. IEEE, 1187–1192.

[43] Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. 2011. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *28th International Conference on Machine Learning (ICML '11)*. PMLR, 609–616.

[44] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. In *29th IEEE Conference on Computer Vision and Pattern Recognition (CVPR '16)*. IEEE, 2818–2826.

[45] TensorFlow Community. 2020. TensorFlow Graph Optimization with Grappler. https://www.tensorflow.org/guide/graph_optimization

[46] TensorFlow Community. 2020. XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla

[47] Theano Development Team. 2016. Theano: A Python Framework for Fast Computation of Mathematical Expressions. *arXiv Preprint arXiv:1605.02688* (2016).

[48] John A. Tomlin. 1989. A Note on Comparing Simplex and Interior Methods for Linear Programming. In *Progress in Mathematical Programming*. Springer, 91–103.

[49] Bart Veltman, B. J. Lageweg, and Jan K. Lenstra. 1990. Multiprocessor Scheduling with Communication Delays. *Parallel computing* 16, 2-3 (1990), 173–182.

[50] Minjie Wang, Chien-chin Huang, and Jinyang Li. 2019. Supporting Very Large Models Using Automatic Dataflow Graph Partitioning. In *14th European Conference on Computer Systems (EuroSys '19)*. ACM, Article 26, 17 pages.

[51] Mengdi Wang, Chen Meng, Guoping Long, Chuan Wu, Jun Yang, Wei Lin, and Yangqing Jia. 2019. Characterizing Deep Learning Training Workloads on Alibaba-PAI. In *22nd IEEE International Symposium on Workload Characterization (IISWC '19)*. IEEE, 189–202.

[52] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *arXiv preprint arXiv:1609.08144* (2016).

[53] Eric P. Xing, Qirong Ho, Wei Dai, Jin-Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. 2015. Petuum: A New Platform for Distributed Machine Learning on Big Data. In *21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '15)*. ACM, 1335–1344.

[54] Tao Yang and Apostolos Gerasoulis. 1994. DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. *IEEE Transactions on Parallel and Distributed Systems* 5, 9 (1994), 951–967.

[55] Engin Zeydan, Ejder Bastug, Mehdi Bennis, Manhal Abdel Kader, Ilyas Alper Karatepe, Ahmet Salih Er, and Mérouane Debbah. 2016. Big data Caching for Networking: Moving from Cloud to Edge. *IEEE Communications Magazine* 54, 9 (2016), 36–42.