# TFLMS: Large Model Support in TensorFlow by Graph Rewriting

Tung D. Le
IBM Research - Tokyo
Tokyo, Japan
tung@jp.ibm.com

Haruki Imai
IBM Research - Tokyo
Tokyo, Japan
imaihal@jp.ibm.com

Yasushi Negishi
IBM Research - Tokyo
Tokyo, Japan
negishi@jp.ibm.com

Kiyokuni Kawachiya
IBM Research - Tokyo
Tokyo, Japan
kawatiya@jp.ibm.com

## ABSTRACT

While accelerators such as GPUs have limited memory, deep neural networks are becoming larger and will not fit with the memory limitation of accelerators for training. We propose an approach to tackle this problem by rewriting the computational graph of a neural network, in which swap-out and swap-in operations are inserted to temporarily store intermediate results on CPU memory. In particular, we first revise the concept of a computational graph by defining a concrete semantics for variables in a graph. We then formally show how to derive swap-out and swap-in operations from an existing graph and present rules to optimize the graph. To realize our approach, we developed a module in TensorFlow, named TFLMS. TFLMS is published as a pull request in the TensorFlow repository for contributing to the TensorFlow community. With TFLMS, we were able to train ResNet-50 and 3DUnet with $4.7x$ and $2x$ larger batch size, respectively. In particular, we were able to train 3DUNet using images of size of $192^3$ for image segmentation, which, without TFLMS, had been done only by dividing the images to smaller images, which affects the accuracy.

## 1 INTRODUCTION

Deep neural networks together with deep learning are effective for solving complex signal-processing problems such as those in computer vision, speech recognition, and natural language processing. However, training a neural network is time-consuming, often taking days to weeks. The training is mainly based on matrix multiplications; therefore, it is often accelerated using accelerators such as GPUs. In 2012, GPUs were used for training a neural network for the first time. It was a deep convolutional neural network of 16 layers, called AlexNet [9], achieving outstanding image classification results in the ILSVRC-2012 competition [1] with a top-5 test error rate of 15.3%. Since then, GPUs have been popular for deep learning.

After the success of AlexNet in the ILSVRC-2012 competition, deep learning has evolved quickly for a broader spectrum of applications. Neural networks are deeper (including more layers) and larger, e.g., ResNet-1001 consists of 1001 layers and is much deeper than AlexNet [7]. Thus, neural networks are sometimes too large to be fit with the memory limitation of GPUs for training.

From the hardware viewpoint, GPUs should be designed to have a larger physical memory, but increasing physical memory is expensive. From the software viewpoint, there are three main approaches

to solving this problem. The first one is reducing memory consumption by reusing memory regions [13] for different computations, compressing a neural network [3] or using low precision [5], the second is re-computing some of the computations from checkpoints [2], the third is using an external memory such as CPU memory for temporarily storing intermediate results during training [10, 11].

We pursued the third approach of using an external memory because it often helps with training a larger model compared to the other approaches and it can be generally applied to any neural networks. Different from the previous studies involving swapping data from GPU memory to an external memory, and vice versa, in an ad-hoc manner, we propose an approach based on formal rules for graph rewriting, which is provable. Our contributions in this paper are as follows:

- We revised the concept of a computational graph of a neural network. Our definition of a computational graph is inspired by that in TensorFlow [1], a popular framework for deep learning. Different from a computational graph in TensorFlow, variables in our computational graph are first-class citizens and consistent with the concept of operations in a computational graph.
- We formally derived swap-out and swap-in operations from an existing graph, those used to exchange intermediate results between GPUs and CPUs. Derivation is based on some rules in program transformations with correctness guarantee, which helps us understand the nature of swapping operations.
- We presented two strategies for finding control operations that are used to control when data are swapped in from an external memory to GPU memory, which helps improve performance.
- To realize our approach, we developed a module in TensorFlow, called TFLMS. TFLMS is published as a pull request in the TensorFlow repository for contributing to the TensorFlow community. With TFLMS, we were able to train ResNet-50 [6] and 3DUnet [4] with a $4.7x$ and $2x$ larger batch size, respectively. In particular, we were able to train 3DUNet using images of size of $192^3$ for image segmentation, which, without TFLMS, had been done only by dividing the images to smaller images.

The rest of the paper is organized as follows. In Section 2, we discuss related work. In Section 3, we discuss our proposed approach

---

[1]http://www.image-net.org/challenges/LSVRC/2012/
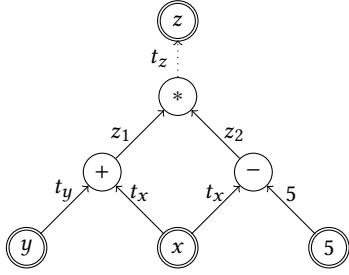
**Figure 1: Computational graph for** $z = (x + y) * (x - 5)$**. Vertices are operations and edges are tensors. Continuous arrows represent "read" edges and dotted arrows represent "update" edges. Double circles are parameterized operations including variables and constants.**

involving revising the concept of a computational graph and presenting the semantics of the graph. In Section 4, we discuss the rules to derive swap-out and swap-in operations and optimizations. In Section 5, we discuss our TFLMS module that implements our approach in TensorFlow. In Section 6, we present the experimental results. Section 7 summarizes the key points and discusses future work.

## 2 RELATED WORK

The most intuitive method for training large models is using Unified Memory [12], a single memory address space accessible from both CPUs and GPUs. Enabling Unified Memory is simple, but its performance is very poor compared to custom methods that manually offload and prefetch data. Shirahata et al. [13] proposed a reduction approach of reusing, during the backward phase, the memory regions allocated for the forward phase. Rhu et al. [11] proposed a different approach of managing runtime memory by virtualizing the memory usage of neural networks against both GPU and CPU memory. During training, only the current layer is active and consumes GPU memory while the other layers' data are swapped out to the CPU memory. This approach performed better than using Unified Memory. Meng et al. [10] took the same approach as [11] for TensorFlow by swapping tensors from GPU memory to CPU memory and vice versa. However, the authors did not discuss how to derive swap-out and swap-in operations [10]. Besides, we could not find their TensorFlow source code. We borrowed Meng et al.'s idea but formally defined transformation rules for graph rewriting so that the correctness of the transformed computational graph is provable. Apart from using CPU memory as a temporary memory for computation, Chen et al. [2] proposed an approach of gradient-checkpointing, in which checkpointing vertices in a computational graph are automatically defined using graph partition. Parts of the graph in between checkpointing vertices are re-computed during the backward phase. The forward phase is generally computed twice. Wang et al. [14] combined both swapping and recomputation in a single framework.

## 3 COMPUTATIONAL GRAPHS

A computational graph is a core concept in TensorFlow. Neural networks defined by users are represented by a computational

graph of operations. TensorFlow then executes optimizations over the graph before invoking operations in the graph. In this section, we revise the concept of a computational graph in TensorFlow [1] to make its semantics more consistent.

### 3.1 Definition

DEFINITION 1. *(Computational graph) Let* $G = (V, E, \lambda, \tau)$ *be a vertex and edge-labeled directed graph, where* $V$ *is the set of vertices in* $G$, $E \subseteq V \times V$ *is the set of edges in* $G$, $\lambda : V \rightarrow (O, Bool)$ *is a function mapping each vertex to a tuple of an operation* $o \in O$ *and a Boolean value indicating whether the operation is parameterized or not, and* $\tau : E \rightarrow (T, ACT)$ *is a function mapping each edge to a tuple of a value of data type* $T$ *and an action in* $ACT$ *where* $ACT = \{"read", "update", "control"\}$.

Computational graphs are a way to express mathematical expressions in which each vertex is an operation with inputs of incoming edges and outputs of outgoing edges. In deep learning, computational graphs are used to express computations in neural networks that consist of operations whose input and output are often multi-dimensional arrays. Multi-dimensional arrays are often called *tensors*. Tensors that are used to store the internal states of a neural network, e.g., learning weights and bias in hidden layers in a neural network, are updated regularly. Hence, we classify operations into *normal operations* and *parameterized operations* where parameterized operations have internal states that can be updated. A *variable* is a special parameterized operation that is to update its internal variable using the identity operation[2]. A *constant* is a special case of a variable where its value is set once and is never updated. Each edge has a value indicating an action related to the tensor on the edge. There are three actions: "read", "update", and "control". Considering an edge $(u, v)$ from an operation $u$ to an operation $v$, actions "read" and "update" mean $v$ reads and updates the tensor, respectively; and action "control" means $u$ triggers the execution of $v$, and $u$ is called a *control dependency operation*.

Figure 1 shows a computational graph for an expression $z = (x + y) * (x - 5)$. In this example, there are three variables $x, y$, and $z$. An outgoing edge emanating from a variable means reading the variable value, and an incoming edge to a variable means updating a tensor to the variable (denoted with a dotted arrow).

### 3.2 Notations and semantics

Table 1 lists the notations to represent different vertices and edges in a graph. Function composition is denoted as "∘", and, from its definition, we have $(f_2 \circ f_1)x = f_2(f_1(x))$. Function "._i" is to take the $i$-th element in a tuple, e.g., $(a, b)._2$ returns $b$.
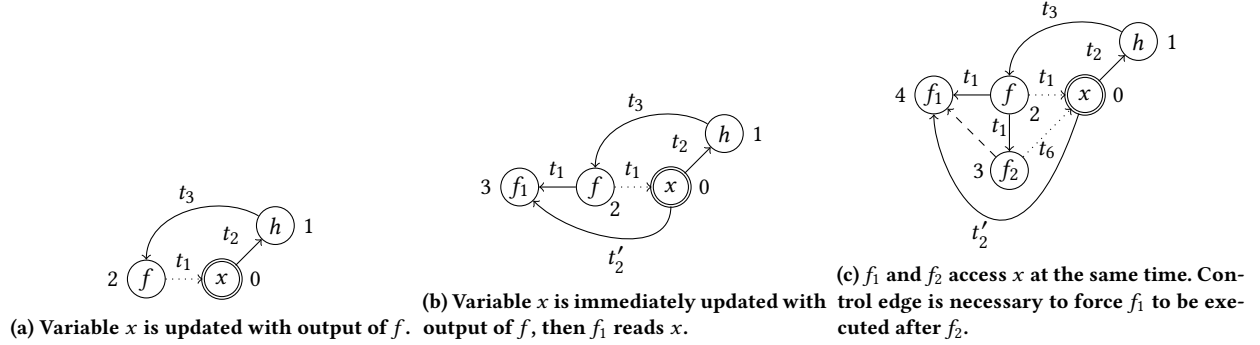
An operation in a computational graph is generally triggered to execute when all of its incoming edges have data. The operation generates data on its outgoing edges then other operations are repeatedly triggered in the same manner. This procedure ends when all of the reachable operations are executed and all of the reachable edges are filled with data. In other words, each of the reachable operations, except variables, is executed once.

However, there is no way to trigger the execution of a graph. At the beginning of computation, there is no way to set a value

---

[2]Identity function accepts a value and returns the same value

**Table 1: Notations**

| | Notation | Definition | Meaning |
|---|---|---|---|
| **Vertices** |  | $\lambda(f) = (f, False)$ <br> $t_4 = f([t_1, t_2, t_3])$ | Normal operation $f$, taking list of inputs $t_1, t_2, t_3$ and returning output $t_4$. |
| |  | $\lambda(x) = (x, True)$ <br> $t = x$ | Parameterized operation or variable $x$. |
| **Edges** |  | $\tau(f, g) = (t, "read")$ | $g$ reads $t$ as input. $t$ is output of $f$. This edge represents $g \circ f$. |
| |  | $\tau(f, x) = (t, "update")$ <br> $x = t$ | $f$ produces output $t$ that is used to update variable $x$. $t$ and $x$ must have the same type. |
| |  | $\tau(f, g) = (\_, "control")$ | $g$ cannot be executed unless $f$ finished. |



(a) Variable $x$ is updated with output of $f$.

(b) Variable $x$ is immediately updated with output of $f$, then $f_1$ reads $x$.

(c) $f_1$ and $f_2$ access $x$ at the same time. Control edge is necessary to force $f_1$ to be executed after $f_2$.

**Figure 2: Examples about graphs regarding variables. Integers above or below vertex are order of that vertex in topological ordering.**

for an edge. Furthermore, computational graphs are acyclic graphs, and there are some operations with no incoming edges. These operations cannot be triggered. This problem is resolved using variables.

Variables in a computational graph are used to store learnable parameters, input and output data, and are used to trigger computation of the graph. Variables are special and make a computational graph for deep learning different from a general dependency graph. Because a variable has an internal state, defining its semantics is non-trivial in the context of the graph. At the beginning, variables are initialized with values input by users or random values generated by a distribution. During training, they are updated by a learning optimizer. This leads to a variable being visited more than once, and may introduce cycles if its semantics is ambiguous. The remainder of this section introduces a clear semantics for variables.

To describe the semantics of a computational graph containing variables, we first define a topological ordering over a computational graph.

DEFINITION 2. *(Topological ordering) Given a computational graph $G = (V, E, \lambda, \tau)$, let $N$ be the number of vertices in the graph, topological ordering is a mapping of vertices to an integer, $\gamma : V \rightarrow \{0, 1, \ldots, N - 1\}$, satisfying*

- $\gamma(v) = 0, \forall v \in V \wedge \lambda(v)._2 = \text{True}$, *and*
- $\gamma(u) < \gamma(v), \forall (u, v) \in E \wedge \lambda(v)._2 = \text{False}$.

In general, a topological ordering represents the order of execution of operations in a graph. Given two operations $u$ and $v$, if $\gamma(u) < \gamma(v)$, $u$ is executed before $v$. If $\gamma(u) = \gamma(v)$, $u$ and $v$ are executed in parallel. In this paper, variables always have order of 0, which means variables will be executed first, and incoming edges ("update" edges) to them do not change their order. Later executions of a variable depend on its incoming operations, and are independent of the variable's order. These executions alone do not trigger the variable's outgoing operations.

Let us consider the example graph in Figure 2a. The graph has the following execution ordering: "$x \rightarrow h \rightarrow f \rightarrow x$". First, variable $x$ is initialized by users then it triggers operation $h$. Then, $h$ is executed and triggers operation $f$. Finally, $x$ is updated with the

output of $f$, and the computation finishes. Operation $h$ depends on $x$ only, and $x$ itself can not trigger $h$ again.

The example graph in Figure 2b may have two possible execution orderings: "$x \rightarrow h \rightarrow f \rightarrow x \rightarrow f_1$", or "$x \rightarrow h \rightarrow f \rightarrow f_1 \rightarrow x$". Operation $f_1$ is triggered based on the availability of tensors $t_1$ and $t_2'$. It is easy to see that $f_1$ must be executed after $f$ and after $x$. However, $x$ is executed multiple times. It is important to know which output of $x$ is used as input to $f_1$.

To avoid ambiguity, we present the following convention regarding variables:

- An operation is always using the latest value of a variable.
- Variables always have the highest priority of execution among operations consuming the same tensor.

This convention helps us ensure that $f_1$ is executed after updating $x$ with the output from $f$.

The execution order of an operation not only depends on data availability on incoming edges but also control dependency edges. "Control" edges do not have data. In other words, they are not inputs for the operation. "Control" edges are used to control the execution order of an operation. Adding a "control" edge into a graph will alter the topological ordering of the graph. If $(u, v)$ is a "control" edge, $v$ must be executed after $u$, and $\gamma(u) < \gamma(v)$. By this definition, there is no control edge to a variable.

The example graph in Figure 2c has a new operation, $f_2$, that consumes the output of $f$, executes computation and, updates variable $x$. Without the control edge from $f_2$ to $f_1$, after $f$ is executed, $f_1$ and $f_2$ can be executed in parallel because they do not depend on each other. Because they both access variable $x$, i.e, $f_1$ reads $x$ and $f_2$ writes to $x$, a control edge is necessary to ensure that they access $x$ in order. The "control" edge from $f_2$ to $f_1$ states that $f_1$ will be executed after finishing $f_2$ and updating $x$.

## 3.3 Training using back-propagation

Training a neural network involves minimizing an objective function $f$ measuring the distance between a ground truth value and predicted value. The objective function is a composition of multiple functions with learnable parameters, and the gradient descent algorithm is often used to minimize the function. Optimization is an iterative procedure updating learnable parameters so that the objective function is minimized, in which each training iteration consists of three phases: *forward phase* to compute the objective function, *backward phase* to compute gradients of the objective function with respect to learnable parameters, and *update phase* to update learnable parameters using the gradients. Backward phase is done via back-propagation for efficiency, starting from the objective function and propagating back gradients through the functions. At the beginning of an iteration, tensors are cleaned up except variables for learnable parameters. Variables for input tensors are fed with new data and trigger the iteration. Because a training dataset is often very big, each iteration takes only a subset (*batch*) of examples extracted from the training dataset as its input tensor. The number of examples in a batch (or *batch size*) will affect the size of the input tensor and also other tensors in the computational graph. In general, increasing batch size will make a model larger.

Figure 3 shows how learnable parameters (represented by variables) are updated during training. In the forward phase, variable
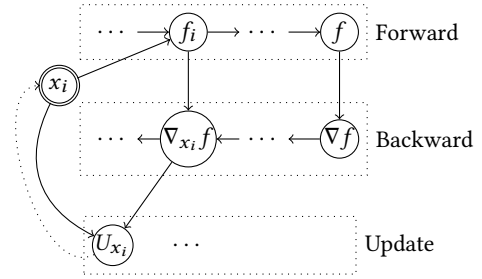


**Figure 3: How variable is used and updated in training.**

$x_i$ is an input to function $f_i$, outputs from $f_i$ are used in the later function, finally a loss value is produced by objective function $f$. In the backward phase, we compute gradients of $f$ with respect to learnable parameters. Function $\nabla_{x_i} f$ computes the gradient of $f$ with respect to $x_i$, which requires $f_i$'s output as one of its inputs. Finally, $x_i$ is updated by a function $U_{x_i}$ during the update phase.

## 3.4 Device placement

In TensorFlow, each operation in the computational graph is placed on a device such as a GPU, CPU, FPGA. Communication between two devices automatically occurs if an operation on one device consumes a tensor produced by another operation on the other device. In fact, TensorFlow adds a pair of two operations, "send" and "receive", to the graph for exchanging a tensor. In this paper, we do not show these communication operations when drawing graphs.

## 3.5 Garbage collection

If a tensor is no longer used in TensorFlow, it is released by TensorFlow garbage collection. Every tensor is assigned a reference count, which is the number of operations. Each time a tensor is consumed by an operation, its reference count is decreased by one. If the reference count reaches zero, the tensor is available to be released. In other words, the lifetime of a tensor is from the operation generating it to the last operation consuming it. Let $ts$ be a tensor produced by an operation $u$, and $v_1, v_2, \ldots, v_k$ be $k$ operations consuming $ts$. The life time of $ts$ is computed as $\max\{\gamma(v_1), \gamma(v_2), \ldots, \gamma(v_k)\} - \gamma(u)$.

## 4 GRAPH REWRITING

A computational graph or a neural network model is said being large to be trained with the memory limitation of GPUs if there are many tensors that are kept in the GPU memory at a time so that they consume more memory than the GPU memory. Hence, an out-of-memory error often happens when training such a large graph. This is essentially because there are many tensors with a long lifetime in a computational graph. In this section, we will show how to rewrite a large graph so that training them is possible with a limited GPU memory. In general, our idea is temporally sending "long lifetime" tensors in a GPU to a CPU and sending them back to the GPU when necessary.
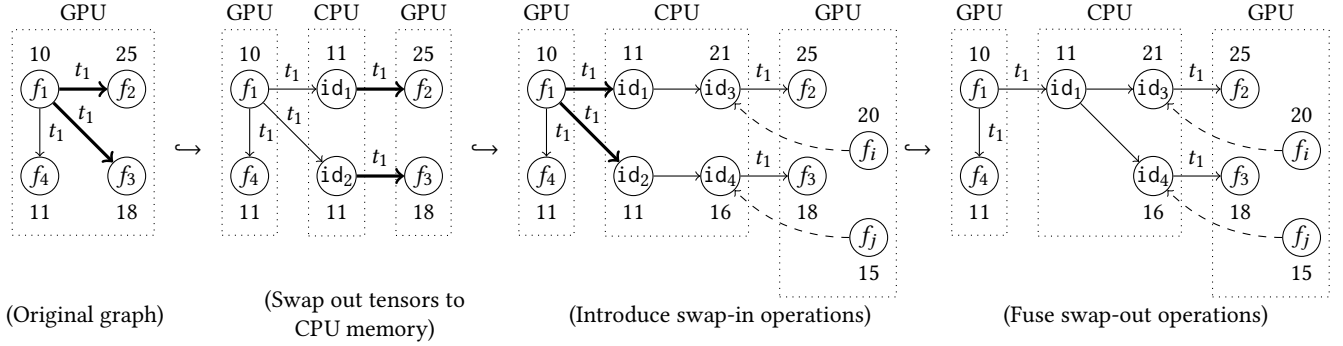
**Figure 4: Example of graph rewriting for supporting large models. Thick edges in left subgraph are rewritten to produce right subgraph. Integers above or below vertex are order of that vertex in topological ordering. In this example, threshold ($\alpha$) to trigger graph rewriting is 5, so edges from $f_1$ to $f_2$ and $f_3$ are rewritten. $f_i$ and $f_j$ are control dependency operations that trigger executions of swap-in operations $\text{id}_2$ and $\text{id}_3$, respectively.**

## 4.1 Swapping out tensors to CPU memory

To put a tensor residing in GPU memory on CPU memory, we derive operations to automatically send the tensor to the CPU and send it back to the GPU. Let us consider an edge $(f_1, f_2)$ where $\tau(f_1, f_2) = (t, \_)$ and $f_1, f_2$ are executed using a GPU. Computation for this edge is

$$f_2^G \circ f_1^G \tag{1}$$

where the superscript $G$ stands for GPU. This computation can be rewritten into:

$$f_2^G \circ \text{id}^C \circ f_1^G \tag{2}$$

where the superscript $C$ stands for CPU, and id is an identity function that is $\text{id}(x) = x$.

Since id is executed using a CPU, the output tensor of $f_1$ will be swapped out to the CPU memory for id immediately after $f_1$ finishes, and GPU memory is released. The output tensor of id will be swapped in to the GPU for $f_2$ when $f_2$ is triggered. We call function id in Equation 2 *a swap-out operation*.

Using Equation 2, we are able rewrite a graph so that GPU memory consumption is reduced. However, not all edges are needed to rewrite. For edges $(u, v)$ where $\gamma(v) - \gamma(u) = 1$, $v$ is executed immediately after $u$. Hence, there is no need to swap the tensor on such edges. We can define a threshold $\alpha$ and graph rewriting for an edge $u, v$ is triggered if $\gamma(v) - \gamma(u) \geq \alpha$.

## 4.2 Optimization

Equation 2 is not optimized due to two reasons: it is too late to swap the output tensor of id in, and $f_2$ must wait for the tensor sent from CPU memory to GPU memory; and the tensor may be swapped out and swapped in multiple times since there may be multiple operations apart from $f_2$ reading it. In this section we present three rules to optimize Equation 2. Figure 4 shows computational graphs obtained by each of optimization rules.

*4.2.1 Introduce swap-in operations.* To swap a tensor in early, we need an additional operation. An Identity function can be rewritten as the composition of a function and its inverse function, that is,

$$\text{id} = f^{-1} \circ f \tag{3}$$

Equation 2 becomes:

$$f_2^G \circ (f^{-1})^C \circ f^C \circ f_1^G \tag{4}$$

Since id also has the inverse function, i.e, id, we choose id for $f$ (if one would like to reduce the memory consumption on the CPU, a pair of encoding and decoding functions can be used for $f$ instead of id),

$$f_2^G \circ \text{id}_2^C \circ \text{id}_1^C \circ f_1^G \tag{5}$$

In Equation 5, $\text{id}_2$ will be used to swap a tensor in to a device, and we call function $\text{id}_2$ *a swap-in operation*. It is worth noting that we must manually trigger $\text{id}_2$ in a good order; otherwise, $\text{id}_2$ is executed immediately after $\text{id}_1$. To do this, a control edge from an operation to $\text{id}_2$ must be added. We present two strategies for choosing a control operation in Section 4.3.

*4.2.2 Fuse swap-out operations.* A tensor produced by an operation is often used by multiple operations, and it is redundant if the tensor is swapped out to CPU memory multiple times. Hence, it is recommended to always fuse swap-out operations of the same tensor into a single swap-out operation.

*4.2.3 Fuse swap-in operations.* Consider a situation that multiple swap-in operations swap a tensor multiple times for multiple consuming operations. If the tensor is large and the consuming operations are close to each other, then swapping the tensor multiple times would introduce more overhead. In this case, it is better to fuse the swap-in operations into one swap-in operation. The tensor is swapped in only once and resides in GPU memory to be reused by the consuming operations. For example, in the right-most graph in Figure 4, if $f_2$ and $f_3$ are close and $t_1$ is large, then we fuse $\text{id}_3$ and $\text{id}_4$ into a singe swap-in operation. To determine how close two operations are, we may define a threshold for the distance between them.

## 4.3 Strategies to add control edges

Control edges to swap-in operations are added to a computational graph to control when swap-in operations are trigger. They are important to reduce the overhead of communication of swapping tensors in. Consider Equation 5, a control operation for the swap-in

**Algorithm 1** Direct-order strategy

**Input:** source operation $f_1$, target operation $f_2$, lower-bound $\sigma_l$, upper-bound $\sigma_u$

**Output:** an operation $g$

1: $l \leftarrow \max\{\gamma(f_2) - \sigma_u + 1, \gamma(f_1)\}$ ▷ Lowest order
2: **for** $i \leftarrow \sigma_l$ to $\sigma_u$ **do**
3:      $k \leftarrow \gamma(f_2) - i$ ▷ $i$ operations before $f_2$
4:      **if** $k \leq l$ **then** ▷ Out of range
5:          **return** null
6:      **end if**
7:      $T \leftarrow \{f \mid f \in V, \gamma(f) = k\}$ ▷ All operations of order $i$
8:      $T \leftarrow \{f \mid f \in T, f_2 \in \text{Reach}(f)\}$ ▷ $f_2$ is reachable from $f$
9:      **if** $T$ is not empty **then**
10:          $g \leftarrow \text{GET}(T)$ ▷ Randomly get one item in $T$
11:          **return** $g$
12:      **end if**
13: **end for**

---

operation $\text{id}_2$ must be chosen from a set of operations, $V_c$, where $\forall v \in V_c, \gamma(\text{id}_1) < \gamma(v) < \gamma(f_2)$ to guarantee the correctness of the computational graph. Let $k = \gamma(f_2) - \gamma(v)$ be the distance between $f_2$ and $v$. If $k$ is too small, a tensor is swapped in too late, and $f_2$ has to wait for the tensor. If $k$ is too large, a tensor is swapped in too early, and the tensor is kept in the device for a long time before being actually used by $f_2$.

An ideal solution for choosing a control operation is having a cost model for computational graphs and using the model to prioritize operations. However, in TensorFlow, the shape of the input and output tensors of an operation is generally unknown at the beginning unless data are fed into the graph then trigger the operation. This means that, at the time a graph is rewritten, there is no information about the actual size of tensors, and it fails to compute operation cost statically.

In a context of statically modifying a computational graph, we introduce two parameters: *lower-bound* $\sigma_l$ and *upper-bound* $\sigma_u$ to handle choosing control operations. Let us assume that an edge $(f_1, f_2)$ is rewritten using a swap-out operation so and swap-in operation si:

$$f_2^G \circ \text{si}^C \circ \text{so}^C \circ f_1^G \tag{6}$$

We present two strategies to find a control operation for si.

*4.3.1 Direct-order strategy.* The direct-order strategy involves directly using the topological ordering to obtain a set of candidates for control operation, starting from the target operation $f_2$ and going back to $f_1$. Lower-bound and upper-bound are relative to $f_2$.

Algorithm 1 shows the algorithm of this strategy. Candidates are operations whose distance to $f_2$ is in the range of $\sigma_l$ to $\sigma_u$ (Line 7) and there exists a path from them to $f_2$ (Line 8). The algorithm stops once it has found one operation satisfying the above conditions (Lines 9–12).

*4.3.2 Chain-rule strategy.* The chain-rule strategy involves starting from the source operation $f_1$ and going down along the forward phase to find corresponding backward operations as candidates for control operations. Breadth-first search is used to traverse operations in the forward phase in which lower-bound and upper-bound

**Algorithm 2** Chain-rule strategy

**Input:** source operation $f_1$, target operation $f_2$, lower-bound $\sigma_l$, upper-bound $\sigma_u$

**Output:** an operation $g$

1: $S_1 \leftarrow \{f_1\}; S_2 \leftarrow \emptyset; S_c \leftarrow \emptyset$
2: **while** $S_1$ is not empty **do**
3:      **if** $\sigma_u = 0$ or $\sigma_l > \sigma_u$ **then**
4:          **return** null
5:      **end if**
6:      $s \leftarrow \text{GET}(S_1)$ ▷ Get one item in $S_1$
7:      $T \leftarrow \text{OUT}(s)$ ▷ Outgoing operations of $s$
8:      **if** $\sigma_l \leq 0$ **then** ▷ Inside the range
9:          $B \leftarrow \{f \mid f \in T, f \text{ is a backward operation}\}$
10:          $B \leftarrow \{f \mid f \in B, \gamma(f) > \gamma(f_1), \gamma(f) < \gamma(f_2)\}$
11:          $B \leftarrow \{f \mid f \in B, f_2 \in \text{Out}(f)\}$ ▷ $f_2$ is reachable from $f$
12:          **if** $B$ is not empty **then**
13:              $g \leftarrow \text{GET(B)}$ ▷ Randomly get one item in $B$
14:              **return** $g$
15:          **end if**
16:      **end if**
17:      $N \leftarrow \{f \mid f \in T, f \text{ is a forward operation}\}$
18:      **for** $f$ in $N$ **do**
19:          **if** $f \in S_c$ **then** ▷ $f$ is visited
20:              **continue**
21:          **end if**
22:          **if** $f \notin S_2$ **then**
23:              $S_2 \leftarrow S_2 \cup \{f\}$
24:          **end if**
25:      **end for**
26:      $S_c \leftarrow S_c \cup \{s\}$ ▷ mark $s$ as visited
27:      **if** $S_1$ is empty **then** ▷ go down one level
28:          $\sigma_l \leftarrow \sigma_l - 1; \sigma_u \leftarrow \sigma_u - 1$
29:          $S_1 \leftarrow S_2; S_2 \leftarrow \emptyset$
30:      **end if**
31: **end while**

---

are used to limit the search space of forward operations. In other words, lower-bound and upper-bound are relative to the source operation $f_1$.

Algorithm 2 shows the algorithm of this strategy. For breadth-first search, we maintain two open sets $S_1$ and $S_2$, and one closed set $S_c$. The $S_1$ contains current forward operations, and $S_2$ contains forward operations for the next level (including all outgoing operations of operation in $S_1$). The $S_c$ contains visited operations. Starting from $f_1$, once the algorithm is in the range of $\sigma_l$ to $\sigma_u$ (Line 8), it obtains outgoing backward operations of a current operation (Line 9), then checks the validity of these backward operations (Lines 10–11). If there is one valid operation, it is a candidate and the algorithm returns it. Otherwise, the algorithm goes to the next level (Lines 27–30).

## 5 TFLMS MODULE IN TENSORFLOW

We developed a TensorFlow module, named *TFLMS*, based on our proposed approach. The module allows users to quickly turn their large model into one that can be trained with limited GPU memory.

**Table 2: Parameters in TFLMS**

| Parameter | Meaning | Default value |
|---|---|---|
| graph | The graph we will modify for LMS. This should be the graph of user-defined neural network. | required |
| optimizer_scopes | A set of scopes for the optimizers/solvers. | required |
| starting_scope | Tensors that are reachable from the operations in this scope will be swapped for LMS. Set this to the scope of the first layer if we would like to modify the whole graph. | None |
| starting_op_names | Tensors that are reachable from the operations with these names will be swapped for LMS. | None |
| excl_scopes | A set of scopes. Output tensors of operations in the scopes will not be swapped out to CPU memory. | empty |
| incl_scopes | A set of scopes. Output tensors of operations in the scopes will be swapped out to CPU memory. | empty |
| excl_types | A set of types. Output tensors of operations with these types will not be swapped out to CPU memory. | empty |
| incl_types | A set of types. Output tensors of operations with these types will be swapped out to CPU memory. | empty |
| n_tensors | The number of tensors for LMS, counting from the starting_scope. | -1 (all tensors) |
| lb | Lower-bound value for LMS. | 1 |
| ub | Upper-bound value for LMS. | 10000 |
| ctrld_strategy | Two strategies to find control dependency operations for swap-in operations: chain_rule and direct_order. | chain_rule |
| fuse_swapins | Fuse "close" swap-in operations into one operation. | False |
| swap_branches | If True, LMS will swap tensors in branches in the forward phase. | False |
| branch_threshold | A threshold for swapping branches in the forward phase. | 0 |
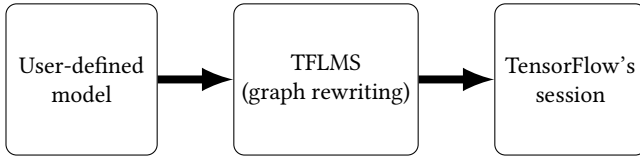


Figure 5: TFLMS module in TensorFlow.

In TensorFlow, users first define a neural network model. TensorFlow then automatically generates a computational graph from the model. Finally, users define a TensorFlow session to execute operations in the computational graph. Once a session is invoked, users cannot modify the computational graph. Hence, we implement TFLMS to statically modify the graph before a session starts.

Figure 5 shows how TFLMS is positioned in TensorFlow. TFLMS takes a computational graph and automatically modifies it using the transformation rules presented in Section 4. TFLMS uses APIs in the module "graph editor"[3] in TensorFlow to modify the graph. The modified graph is then executed by a TensorFlow session as normal. TFLMS's source code is publicly available as a pull request in the TensorFlow repository[4].

Listing 1 shows a brief example of using TFLMS in TensorFlow. While defining a neural network, users must define a scope for their

---

**Listing 1: Sample Python code to use TFLMS in TensorFlow.**

```python
# define a scope for the optimizer/solver
with tf.name_scope('adam_optimizer'):
    opt = tf.train.AdamOptimizer(1e-4)
    train_step = opt.minimize(cross_entropy)

# define a LMS instance and run it
from tensorflow.contrib.lms import LMS
lms_obj = LMS({'adam_optimizer'})
lms_obj.run(graph=tf.get_default_graph())

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
        batch = mnist.train.next_batch(50)
        train_step.run(feed_dict={x: batch[0],
                                    y_: batch[1]})
```

optimizer (Line 2). Users then define a LMS instance for that scope and run the instance to modify the computational graph of the neural network (Lines 7–9). After that, users create a TensorFlow session and train the network as usual.

## 5.1 Implementation

The important part of TFLMS is building a topological ordering. Given a graph, TFLMS uses the python package "toposort"[5] to build a topological order. The topological ordering, $\gamma$, is to decide

---

[3]Graph editor: https://www.tensorflow.org/api_guides/python/contrib.graph_editor
[4]https://github.com/tensorflow/tensorflow/pull/19845

[5]https://pypi.org/project/toposort/

which tensors are swapped out and when they are swapped in as shown Section 4. To rewrite edges, TFLMS traverses through the graph using the breadth-first search algorithm, starting from input variables. We do not rewrite incoming and outgoing edges of variables. In other words, learnable parameters are kept in GPU memory. Apart from an input of a computational graph, TFLMS allows users to pass other parameters to flexibly control how the graph is modified. Table 2 lists the parameters in TFLMS.

By default, TFLMS always rewrites edges between a forward operation and a backward operation. To determine operations in the backward phase, users should pass the scope[6] of solvers or optimizers that are used to train the model (via TFLMS parameter *optimizer_scopes*). Note that, it is possible to automatically rewrite the whole graph without *optimizer_scopes*. Using *optimizer_scopes* reduces unnecessary operations that are not helpful for large model support, e.g. operations in the update phase. If a model has many branches in the forward phase, users may want to use parameters *swap_branches* and *branch_threshold* to enable rewriting edges $(u, v)$ satisfying $\gamma(v) - \gamma(u) > branch\_threshold$. *branch_threshold* is the threshold $\alpha$ defined in Section 4.1. Swapping tensors in the forward phase may affect the performance of inferencing of a neural network because it introduces overhead of swapping the tensors out and in. However, if the neural network is still large for inferencing, swapping those tensors is necessary. Without enabling *swap_branches*, our modification does not cause any affect on the performance of inferencing because added swap-out and swap-in operations between the forward and backward phases are not executed during the inferencing. Inclusion or exclusion of an operation can be done via the operation's type or scope. Users can define a starting point for the breadth-first search by using the scope or name of operations via parameters *starting_scope* and *starting_op_names*. By default, TFLMS rewrites all reachable edges. However, users can define the number of tensors that are swapped via parameter *n_tensor*. Parameters *lb* and *ub* are lower-bound and upper-bound, respectively, as defined in Section 4.3. A strategy for choosing control operations is set by parameter *ctrld_strategy*. Parameter *fuse_swapins* is to enable the optimization of fusing swap-in operations.

## 5.2 Performance tuning

To get the maximum performance when using TFLMS, we need to find the combination of tuning parameters that provides the fastest training time with the model. The goal of the performance tuning is to swap out enough tensors to allow our training to run without out-of-memory errors, while not swapping too many such that the extra swapping communication overhead degrades performance.

The two tuning parameters we should focus on are *n_tensors* and *lb*. Since *n_tensors* controls the number of tensors that will be swapped, the higher this is set, the lower the peak GPU memory usage will be. The *lb* controls how soon the tensor is swapped back in before use. A low value *lb* can make the training on the GPU pause and wait while the swap-in finishes. This will degrade performance. A higher value *lb* allows the tensor swap-in to finish before it is needed and allows training to run without pause. The

---

[6]In TensorFlow, scope defines a name for a set of operations, similar to a folder in a file system.

**Table 3: Maximum batch size when swapping all reachable tensors. OOM stands for out-of-memory.**

| Model | Image | Without TFLMS | With TFLMS | Ratio |
|---|---|---|---|---|
| ResNet-50 | $224^2$ | 176 | 832 | 4.7 |
| 3DUnet | $128^3$ | 2 | 4 | 2 |
| 3DUnet | $192^3$ | OOM | 1 | |

downside to swapping in too early is that more tensors will be in the GPU memory at any point in time, resulting in higher peak GPU memory usage.

Tuning thus becomes finding the correct balance between *n_tensors* and *lb* that provides the best performance for a given model. To start the performance tuning it is suggested that *n_tensors* be set to -1, which will swap all reachable tensors, e.g., $N$ tensors. The *lb* should be set to the default of 1, which is the latest possible swap-in. It is useful to run with *n_tensors* = $N$ and then adjust it downward. If the model has branches similar to the 3UNet model, it is likely useful to set *swap_branches* to True and tune the branch threshold.

## 6 EXPERIMENTS

### 6.1 Experimental environment

Experiments were run on an IBM POWER8 NUMA-based machine [8] using one GPU. The machine has two 4GHz 10-core POWER8 processors, eight simultaneous multi-threads (SMTs) per core and 256 MB RAM per processor. There are four NVIDIA Tesla P100 GPUs (each with 16 GB memory). NVLinks are used for connections among GPUs and CPUs: one 80 GB/s duplex link between GPUs 0 and 1, one 80 GB/s duplex link between GPUs 2 and 3, two 80 GB/s duplex links from CPU 0 to GPUs 0 and 1, and two 80 GB/s duplex links from CPU 1 to GPUs 2 and 3. On the machine, we installed TensorFlow 1.8, CUDA Toolkit v9.0 and cuDNN 7.0.5.

We evaluated TFLMS using two popular neural networks: ResNet-50 for image recognition and 3DUNet for image segmentation. To make a model larger, we increase the batch size of each iteration. By default, we always fuse swap-out operations.

### 6.2 Maximum batch size

Table 3 shows the maximum batch size we are able to train using TFLMS. We let TFLMS swap all reachable tensors to reduce GPU memory consumption as much as possible. In total, TFLMS swapped all of 317 tensors for ResNet-50, all of 2779 tensors for 3DUNet with $192^3$ images and 2397 tensors for 3DUNet with $128^3$ images[7]. With TFLMS we were able to train ResNet-50 and 3DUNet with 4.7 and 2 times larger batch size, respectively. For 3DUnet, we were able to train the whole images of $192^3$ without resizing or splitting the images, which was impossible without TFLMS.

### 6.3 Training performance

Figure 6 shows the effectiveness of parameters n_tensors and lb on training performance of ResNet-50. We measured the number of images per second (images/sec) for each batch size. Without TFLMS, the maximum batch size we were able to train is 176. Performance

---

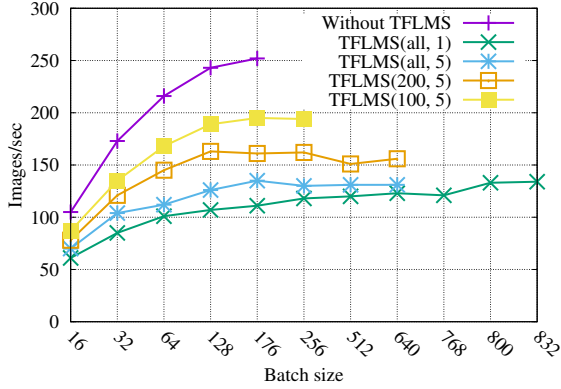[7]3DUnet architecture is changed according as image size.

**Figure 6: Effectiveness of n_tensors and lb on training performance of ResNet-50. TFLMS(x, y) means running TFLMS with n_tensors=x and lb=y. "all" means swapping all tensors, in this case n_tensors=317.**



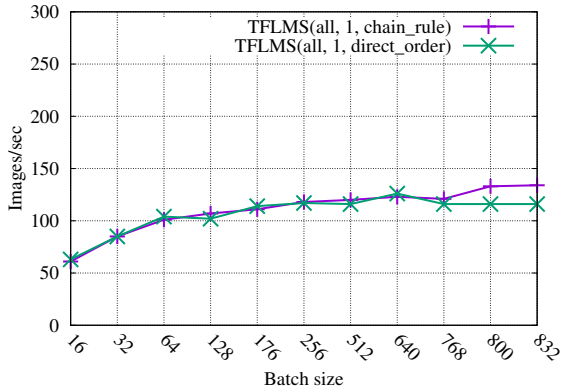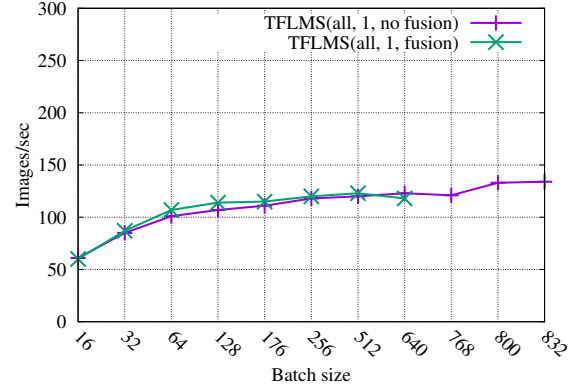**Figure 8: Effectiveness of fuse_swapins on training performance of ResNet-50. TFLMS(x, y, z) means running TFLMS with n_tensors=x, lb=y, fused_swapins=z. "all" means swapping all tensors, in this case n_tensors=317.**



**Figure 7: Effectiveness of ctrld_strategy on training performance of ResNet-50. TFLMS(x, y, z) means running TFLMS with n_tensors=x, lb=y, ctrld_strategy=z. "all" means swapping all tensors, in this case n_tensors=317.**
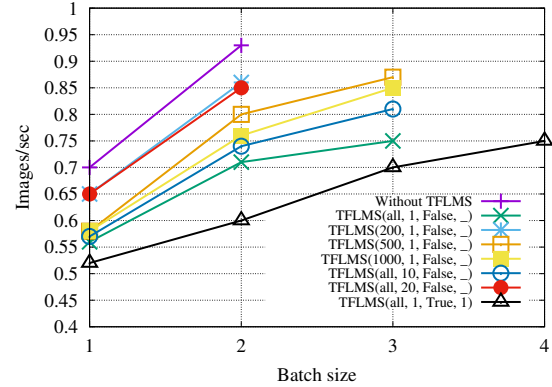


**Figure 9: Effectiveness of n_tensors, lb, swap_branches on training performance of 3DUnet. TFLMS(w, x, y, z) means running TFLMS with n_tensors=w, lb=x, swap_branches=y, branch_threshold=z. "all" means swapping all tensors, in this case n_tensors=2397. Input images are of size of $128^3$.**

for a smaller batch size was poor because GPU usage was small. With TFLMS, when we first swapped out all reachable tensors, i.e. 317 tensors, and set *lb* to 1 for swapping in a tensor as late as possible, the maximum batch size we were able to train is 832, 4.7 times larger than the one without TFLMS. However, performance was not good. We then tried to increase *lb* from 1 to 5 to swap in tensor earlier so that there were more overlap between computation and communication. It is clear that the higher *lb*, the better training performance, but the maximum batch size was decreased because there were more tensors residing in GPU memory at a time. Similarly, we decreased the number of tensors being swapped out, from 317 (all) to 200 or 100. We also obtained better performance. *n_tensors* was more effective than *lb* on training performance, and *lb* was less effective than *n_tensors* on the maximum batch size. Hence, there should be a tradeoff between *n_tensors* and *lb*.

Figure 8 shows the effectiveness of fusing swap-in operations. In both cases, we swapped out 317 tensors in total, but the numbers of swapping operations added to the graph with *fuse_swapins* enabled and disabled are 687 and 634, respectively. Fusing swap-in operations lead to better performance but smaller maximum batch size. This is because some tensors were kept in GPU memory for re-using as we mentioned in Section 4.2.3.

Figure 7 shows a comparison between two strategies "chain_rule" and "direct_order" for finding control dependency operations. Though the strategy "direct_order" is simple than "chain_rule", it sometimes had poorer performance for training ResNet-50. In particular, "direct_order" was much slow with batch sizes 768, 800 and 832.

Figure 9 shows results for 3DUnet. The maximum batch size we were able to train with TFLMS is twice as large as that without TFLMS. The effectiveness of Parameters n_tensors and lb for

3DUnet is similar to that for ResNet-50. In particular, when we decreased n_tensors from 2397 (all tensors) to 1000, we clearly saw better performance, but the maximum batch size was decreased from 4 to 3. We measured the effectiveness of swapping branches. We enabled swapping branches with threshold 20, the number of added operations was increased from 3895 to 4052 and the number of swapped tensors stayed the same. By swapping branches, we were able to train 3DUnet with the maximum batch size of 4 instead of 3. We also tried to train 3DUnet with large images, i.e. images of size of $192^3$. While without TFLMS we got out-of-memory errors, with TFLMS, we were able to train 3DUnet at 0.17 images/sec (Batch size=1, n_tensors=2779 (all), lb=1, swap_branches=True, branch_threshold = 20).

## 7 CONCLUSION

We have proposed a formal approach to deriving swap-out and swap-in operations for enabling large model support. We formally revised the concept of computational graph and borrowed the theory of program transformations to derive new operations as well as optimize the graph. Furthermore, We have proposed two strategies to statically find control dependency operations for triggering swap-in operations. The experimental results showed that our approach helped train very large models, i.e. 4.7 and 2 times larger for ResNet-50 and 3DUnet, respectively. Though our definition of computational graph is inspired by TensorFlow, it is still general enough to be applied to other computational graph based frameworks. In the future, we plan to incorporate the re-computation technique by introducing new transformation rules. Investigating a good heuristics to finding control dependency operations is an open problem.

## ACKNOWLEDGMENTS

## REFERENCES

[1] Martín Abadi, Michael Isard, and Derek G. Murray. 2017. A Computational Model for TensorFlow: An Introduction. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2017)*. ACM, New York, NY, USA, 1–7.

[2] T. Chen, B. Xu, C. Zhang, and C. Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. *ArXiv e-prints* (April 2016). arXiv:1604.06174

[3] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. 2018. Universal Deep Neural Network Compression. *CoRR* abs/1802.02271 (2018). http://arxiv.org/abs/1802.02271

[4] Özgün Çiçek, Ahmed Abdulkadir, Soeren S. Lienkamp, Thomas Brox, and Olaf Ronneberger. 2016. 3D U-Net: Learning Dense Volumetric Segmentation from Sparse Annotation. *CoRR* abs/1606.06650 (2016). http://arxiv.org/abs/1606.06650

[5] Julian Faraone, Nicholas J. Fraser, Giulio Gamberdella, Michaela Blott, and Philip Heng Wai Leong. 2017. Compressing Low Precision Deep Neural Networks Using Sparsity-Induced Regularization in Ternary Networks. *CoRR* abs/1709.06262 (2017). http://arxiv.org/abs/1709.06262

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). http://arxiv.org/abs/1512.03385

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. *Identity Mappings in Deep Residual Networks*. Springer International Publishing, 630–645.

[8] IBM. 2016. IBM Power System S822LC for High Performance Computing. http://www-03.ibm.com/systems/power/hardware/s822lc-hpc/.

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *International Conference on Neural Information Processing Systems*. 1097–1105.

[10] Chen Meng, Minmin Sun, Jun Yang, Minghui Qiu, and Yang Gu. 2017. Training deeper models by GPU memory optimization on TensorFlow. In *Proc. of ML Systems Workshop in NIPS*.

[11] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler. 2016. vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design. *ArXiv e-prints* (Feb. 2016). arXiv:1602.08124

[12] Nikolay Sakharnykh. 2017. Unified memory on Pascal and Volta. (2017). http://on-demand.gputechconf.com/gtc/2017/presentation/s7285-nikolay-sakharnykh-unified-memory-on-pascal-and-volta.pdf GTC.

[13] K. Shirahata, Y. Tomita, and A. Ike. 2016. Memory reduction method for deep neural network training. In *2016 IEEE 26th International Workshop on Machine Learning for Signal Processing (MLSP)*. 1–6.

[14] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. 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 (PPoPP '18)*. ACM, New York, NY, USA, 41–53.