SuperNeurons: Dynamic GPU Memory Management for Training Deep Neural Networks

Linnan Wang [§] Jinmian Ye [†] Yiyang Zhao [†] Wei Wu [‡] Ang Li [II] Shuaiwen Leon Song [II]
Zenglin Xu [†] Tim Kraska [◇§]

[§] Brown University
[†] Univ. of Electr. Sci. & Tech. of China
[‡] Los Alamos National Laboratory
[II] Pacific Northwest National Laboratory
[◇] Massachusetts Institute of Technology

Abstract

Going deeper and wider in neural architectures improves their accuracy, while the limited GPU DRAM places an undesired restriction on the network design domain. Deep Learning (DL) practitioners either need to change to less desired network architectures, or nontrivially dissect a network across multiGPUs. These distract DL practitioners from concentrating on their original machine learning tasks. We present SuperNeurons: a dynamic GPU memory scheduling runtime to enable the network training far beyond the GPU DRAM capacity. SuperNeurons features 3 memory optimizations, *Liveness Analysis*, *Unified Tensor Pool*, and *Cost-Aware Recomputation*; together they effectively reduce the network-wide peak memory usage down to the maximal memory usage among layers. We also address the performance issues in these memory-saving techniques. Given the limited GPU DRAM, SuperNeurons not only provisions the necessary memory for the training, but also dynamically allocates the memory for convolution workspaces to achieve the high performance. Evaluations against Caffe, Torch, MXNet and TensorFlow have demonstrated that SuperNeurons trains at least 3.2432 deeper network than current ones with the leading performance. Particularly, SuperNeurons can train ResNet2500 that has 10^4 basic network layers on a 12GB K40c.

CCS Concepts **• Computing methodologies** → *Parallel algorithms*; *Computer vision problems*

Keywords Neural Networks, GPU Memory Management, Runtime Scheduling

PPoPP '18, February 24–28, 2018, Vienna, Austria

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4982-6/18/02…$15.00

DOI: http://dx.doi.org/10.1145/3178487.3178491

1. Introduction

Deep Neural Network (DNN) is efficient at modeling complex nonlinearities thanks to the unparalleled representation power from millions of parameters. This implies scaling up neural networks is an effective approach to improve the generalization performance. The Deep Learning (DL) community now widely acknowledges either going deeper or going wider on the nonlinear architecture improves the quality of image recognition tasks. For example, 9-layer AlexNet won the 2012 ILSVRC (ImageNet Large-Scale Visual Recognition Challenge) with a top-5 error of 17%. GoogLeNet (inception v1) refreshed the top-5 error rate to 6.67% with 22 inception units in 2014 ILSVRC, and ResNet further reduced the error rate down to 3.57% in 2015 ILSVRC with 152 residual units.

While DL practitioners are enthusiastically seeking deeper and wider nonlinear networks, the limited size of GPU DRAM becomes a major restriction. Training a deep network is inherently a computation-intensive task. Almost every AI lab today, either in academia or industry, is deploying the network training on GPUs for the purposes of better performance [3]. Data need to be residing on GPU DRAM for the GPU computing, but the largest commercial GPU DRAM so far is 24 GB. This is still far from sufficient to accommodate a deep neural network. For example, the latest Inception v4 has 515 basic layers consuming 44.3 GB memory in the training. The deeper or wider we go, the higher memory usages will be. Therefore, this deep trend subjects the rigid GPU DRAM to the severe space insufficiency.

Major DL frameworks, such as Caffe or MXNet, have tried to alleviate the GPU memory shortage with several static memory reduction techniques. Those techniques, due to their static nature, are not well tuned to address the new data and dependency variations in non-linear networks. For example, Caffe and Torch do not fully support the data flow analysis on non-linear neural networks; the strategy of trading computation for memory in MXNet is limited because it ignores the memory variations across network layers. These limitations have motivated us to propose a dynamic approach for the emerging deep nonlinear neural architectures.

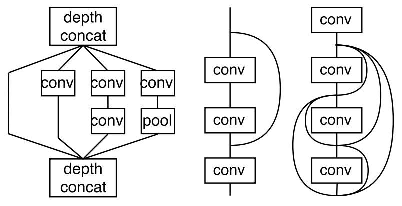
In this paper, we present the first dynamic GPU memory scheduling runtime for training deep non-linear neural networks. The runtime allows DL practitioners to explore a much deeper and wider model beyond the physical limitations of GPU memory. It utilizes tensors as the fundamental scheduling units to consist with the layer-wise computations enforced in DL performance primitives cuDNN [7]. The runtime seamlessly orchestrates the tensor placement, movement, allocation and deallocation so that the underlying memory operations are entirely transparent to users.

Our runtime guarantees the minimal peak memory usage, $peak_m = \max(l_i)$, at the layer-wise granularity. We denote the memory usage of the ith layer as l_i, and the superscript, e.g. l_i^f or l_i^b, as the forward/backward. The peak memory usage during the forward and backward computations is denoted as $peak_m$. First, *Liveness Analysis* recycles no longer needed tensors to reduce $peak_m$ from baseline $\sum_{i=1}^{N} l_i^f + \sum_{i=1}^{N} l_i^b$ to $\sum_{i=1}^{N} l_i^f + l_N^b$ (defined in Sec.3). Secondly, Unified Tensor Pool (UTP) offloads tensors in compute-intensive layers, referred to as checkpoints, to the external physical memory. This further reduces $peak_m$ from $\sum_{i=1}^{N} l_i^f + l_N^b$ to $\sum_{i=1}^{N}(l_i^f \notin checkpoints) + l_N^b$. Finally, *Cost-Aware Recomputation* drops the forward results of cheap-to-compute or none-checkpoints layers and reconstructs them to reduce $peak_m$ from $\sum_{i=1}^{N}(l_i^f \notin checkpoints) + l_N^b$ to $peak_m = \max(l_i)$. The final $peak_m$ indicates the largest computable network is bounded by the maximal memory usage among layers.

Our runtime also features three performance optimizations to improve the efficiency of *Liveness Analysis* and *UTP*. First, GPUs require memory allocations to create tensors and deallocations to free tensors. Thus, the highly frequent large tensor allocations/deallocations incur the non-negligible overhead in *Liveness Analysis* [25]. The runtime successfully amortizes the cost by directly reusing memory segments from a huge pre-allocated memory pool, managed by a heap based GPU memory management utility. Secondly, UTP swaps tensors among different physical memory spaces, while modern GPUs equip with independent Direct Memory Access (DMA) engine exposing opportunities to hide communications under computations. The runtime also meticulously overlap communications with computations. However, the overlapping opportunity is limited given the fixed amount of computations. We propose a LRU based Tensor Cache built on GPU DRAM to minimize total communications by tensor reusing.

This paper claims the following contributions:

- We demonstrate the new memory scheduling challenges in nonlinear neural networks, and discuss the key limitations of existing approaches.



(a) fan (b) join

Figure 1: The non-linear connections in inception v4 (fan), ResNet (join, left) and DenseNet (join, right). DenseNet utilizes a full-join.

- By dynamically allocating memory for convolution workspaces, SuperNeurons delivers the leading performance among state-of-art DL systems on the GPU.
- We design and implement SuperNeurons to enable DL practitioners to explore deep neural networks; and the largest computable network of SuperNeurons is only bounded by the maximum memory usage among layers.

2. Background and Motivation

2.1 Challenges for Processing Super Deep Neural Networks

Traditional Convolutional Neural Networks (CNN) [16, 17, 20] are typically composed of several basic building layers, including Convolution (CONV), Pooling (POOL), Activation (ACT), Softmax, Fully Connected (FC), Local Response Normalization (LRN), Batch Normalization (BN), and Dropout. For linear CNNs, these layers are independent and inter-connected to their neighbors in a sequential manner: $1 \leftrightarrow 2 \leftrightarrow \cdots \leftrightarrow n$. Recently, several deep non-linear neural architectures have been proposed to further improve the state-of-the-art accuracy on the 1K ImageNet recognition challenge, e.g., Inception v4[22], ResNet[12], and DenseNet[13]. These prominent network designs (especially the one that solves the classic gradient vanishing [4] problem) pave the algorithmic foundation for DL practitioners to harness the unparalleled representation power brought forth by the super deep non-linear neural architectures. For example, the latest inception v4 delivers 95% top-5 accuracy with 515 basic building layers while ResNet151[1] achieves 94.3% top-5 accuracy with 567 layers. In Figure 1, we illustrate two classic types of non-linear connections: fan and join. Compared with the linear connection pattern, the sparse fan-out connection (Figure 1a) avoids one huge computing-inefficient dense layer [21] while the join connection prevents gradients from quickly vanishing in the back-propagation [12].

Training these super deep and complex non-linear neural architectures is a computation-intensive task. Due to its DL-driven novel architecture designs and massive parallelism,

[1] 151 represents the number of convolutional units.

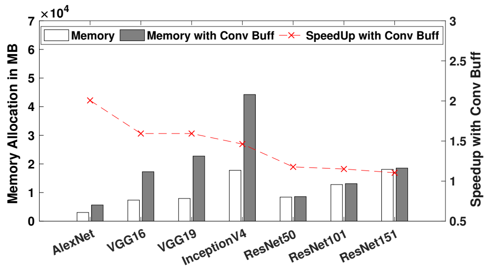


Figure 2: The left axis depicts the memory usages of networks. The batch size of AlexNet is 200, and the rest use 32. The right axis and red x marks depict the speedup (imgs/s) with and without convolution workspaces.

GPUs have been widely adopted in today's industry and academia for the efficient neural network training. However, there are critical issues for efficiently training in these newly-developed super deep non-linear neural architectures: *limited GPU resident memory* and *a high degree of variation in computational dependencies*.

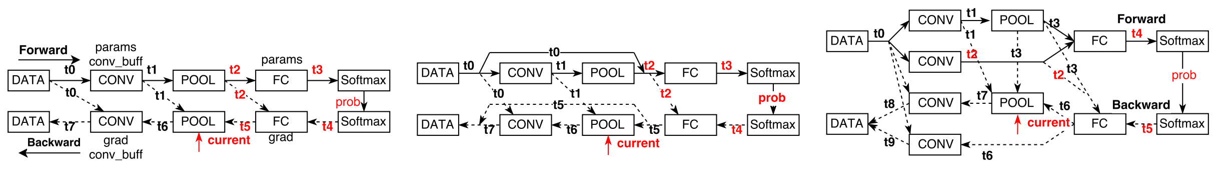
Challenge I: Limited GPU Resident Memory. The prominent deep neural architectures share a common feature: high memory demand and computation intensity. Figure 2 illustrates the network-wide memory usages of several recent DNNs in training with and without convolution workspaces (buffer). Among them, AlexNet and VGG are linear networks while the others are non-linear. We can observe that the non-linear networks demand a significant amount of GPU memory, e.g., ResNet152 and Inception v4 require up to 18.5GB and 44.3 GB at only the batch size of 32, respectively. However, these sizes are either similar to or surpass the resident memory sizes of commercial GPUs on the market today. For instance, the newest generations of NVIDIA Pascal and Volta GPUs only have 16GB with HBM2 enabled (e.g., P100 and V100) while the one with the most memory available in recent generations is Maxwell P40 with 24GB GDDR5. This limitation poses a major bottleneck for deep learning practitioners for exploring deep and wide neural architectures [18, 21, 22]. The most straightforward solution is to split the network across GPUs, i.e. Model Parallelism. However, splitting either the computations of a network or a layer incurs excessive intra-network and intra-layer communications that drastically deteriorate the performance. For example, recent work has suggested the deficiency of applying model parallelism for deep neural networks: it compromises at least 40% speed when training a network with 1.3 billion parameters from 36 GPUs to 64 GPUs [8]. To address the performance issues from Model Parallelism, Data Parallelism has been widely adopted in today's mainstream deep learning frameworks such as Caffe[14], TensorFlow[2], Torch[9], and MXNet[5]. In this model, each GPU holds a network replica; and one GPU computes one sub-gradient with a sub-batch. Subsequently, all sub-gradients are aggregated as one global gradient to update the network parameters [24]. Although this process does not incur intra-network or intra-layer communications besides necessary gradient exchanges, it requires the network training to fit in the limited GPU DRAM. In this paper, we focus on addressing the GPU memory shortage issue for training deep neural networks under data parallelism model while taking the training performance into design considerations.

Challenge II: Variations in Computational Dependencies for Nonlinear Networks. Nonlinear networks exhibit a high degree of dependency variations while linear networks follow a fixed sequential execution pattern with predictable data dependencies [19]. Fig.3 illustrates the data dependency graph for linear (a) and nonlinear (b and c) neural architectures. One typical training iteration consists of two phases: forward and backward propagation. For linear networks, data is sequentially propagated in the forward pass; and a layer's backward computation is simply contingent upon the previous layer as illustrated in Figure 3a. Thus their computation and dependency patterns are static regardless of the total layers involved.

However, for nonlinear networks, a high degree of variation in computational dependencies appear. Fig.3b and 3c show two simple examples of join and fan nonlinear connections. Join connections forward a layer's output tensor to another layer, creating a dependency between two layers. For example, the join in Fig.3b forwards `t0` from DATA layer to FC layer in the forward pass. The dependency of join-based non-linear networks is non-deterministic as any two layers can be connected with a join, e.g., in DenseNet. For fan connections, it creates multiple branches in the execution flow: DATA layer forks two branches and joins them before FC layer. Separate branches, each with a different number of layers, have to finish before joining them back to the original branch, making this execution sequence nonlinear. Although the two basic nonlinear scenarios shown here are intuitive, a typical deep nonlinear network today has hundreds of joins and fans convoluted together, resulting in a complex network architecture. These significantly complicate runtime resource-management compared to the static computational pattern in linear ones. Therefore, the memory scheduling of deep non-linear neural networks demands a dynamic solution to effectively address these variations in both the execution flow and computation dependencies.

2.2 Limitations of GPU Memory Management in Mainstream Deep Learning Frameworks

Several static memory reduction techniques have been implemented in today's deep learning frameworks to address the GPU memory shortage at data parallelism level. For example, Caffe and Torch directly reuse the forward data tensors for the backward data propagation, which saves up to 50% of memory on a linear network [1]. Although this technique works well on linear networks, it requires extra tensors to hold the future dependencies for training non-linear



(a) linear (b) join (nonlinear) (c) fan (nonlinear)

Figure 3: Data dependencies of different neural architectures. Tensors in red are ready to free when the computation back propagates to the POOL layer. Solid lines represent forward dependencies and dashed lines represent backward dependencies.

networks, thereby limiting the effectiveness and efficiency. Also, these frameworks still have to fit the entire network into GPU DRAM without leveraging NUMA architectures, and this level of reuse is arguably not adequate for contemporary deep nonlinear neural networks. MXNet and TensorFlow are built with a Directed Acyclic Graph (DAG) execution engine [27]. Users explicitly define the computation flow and tensor dependencies, which provide necessary information for the DAG engine to analyze the life span of tensors. Both systems then free tensors that are no longer needed in order to save memory. MXNet also implements a per-layer-based re-computation strategy that is similar to Resilient Distributed Datasets (RDD) in Spark [28]. Basically it frees the tensors produced by computation-inexpensive layers in the forward pass, and recomputes the freed dependencies for the backward pass by doing another forward. However, this method neglects non-uniform memory distribution of network layers, consequentially demanding large unnecessary memory usages. TensorFlow swaps long-lived data tensors from GPU DRAM to CPU DRAM, but it fails to optimize data communications between the two (e.g., utilizing pinned data transfer) which compromises at least 50% of communication speed.

More importantly, none of aforementioned DL frameworks utilize a dynamic scheduling policy that provisions necessary memory space for deep nonlinear network training while at the same time optimizing the training speed given the existing GPU DRAM resource. In other words, these static memory-saving techniques aggressively reduce the GPU memory usage at the expense of speed. Users either painstakingly tune the performance or suffer from insufficient memory during the execution. Additionally, these frameworks either have no optimization strategy or adopt a naive method on allocating the convolution workspace (see Section 3.5), which is a decisive factor determining CNN training speed on the GPU. In summary, these challenges motivate us to design a dynamic scheduling runtime to provision necessary memory for the training while maximizing the memory for convolution workspaces to optimize the training speed.

3. Design Methodologies

This section elaborates on three memory optimization techniques and their related performance issues in SuperNeurons. From a high-level perspective, SuperNeurons provision necessary memory spaces for the training while maximizing the speed by seeking convolution workspaces within the constraint of native GPU memory size.

Notations and Baseline Definition: To facilitate the analysis of proposed techniques, we denote the forward memory usage of the ith layer as l_i^f, the backward as l_i^b. We denote the peak memory usage as $peak_m$. We use the naive network-wide tensor allocation strategy as the baseline, which allocates an independent tensor for each memory requests. Thus, the $peak_m$ of baseline is $\sum_{i=1}^{N} l_i^f + \sum_{i=1}^{N} l_i^b$. We also denote the maximal memory usage among layers as $l_{peak} = max(l_i)$, where $i \in [1, N]$, and N represents the network length. t_i represents the ith tensor.

First, *Liveness Analysis* reduces the baseline $peak_m$ to $\sum_{i=1}^{N} l_i^f + l_N^b$ by recycling free tensors amid back-propagation, demonstrating up to 50% of the memory saving. This technique is guaranteed to work on various nonlinear architectures, and it is constructed in $\mathcal{O}(N^2)$. *Liveness Analysis* involves high-frequent memory operations on the large chunk memory, while native memory utilities, e.g. cudaMalloc and cudaFree, incur the nontrivial overhead. We address this issue with a preallocated heap managed by the runtime.

Secondly, *Unified Tensor Pool(UTP)* further reduces $peak_m$ to $\sum_{i=1}^{N}(l_i^f \notin checkpoints) + l_N^b$, where checkpoints represent the compute-intensive layers such as FC and CONV. UTP provides a consolidated memory abstraction to external memory pools to supply for the training. Instead of using naive on-demand data transfers, it hides communications under computations. While the overlapping opportunity is limited given the fixed amount of computations, UTP further introduces a *Tensor Cache* built on GPU to reduce communications.

Finally, *Cost-Aware Recomputation* reduces $peak_m$ to $max(l_i)$, the minimum at the layer-wise granularity. The method keeps track of memory distributions among checkpoints to minimize the extra computations while ensuring $peak_m \leq max(l_i)$.

3.1 Prerequisites

A typical DNN network layer computes on a 4-dimension tensor indexed by batches (N), image channels (C), height (H) and width (W) (Fig.5). Since cuDNN operates at the

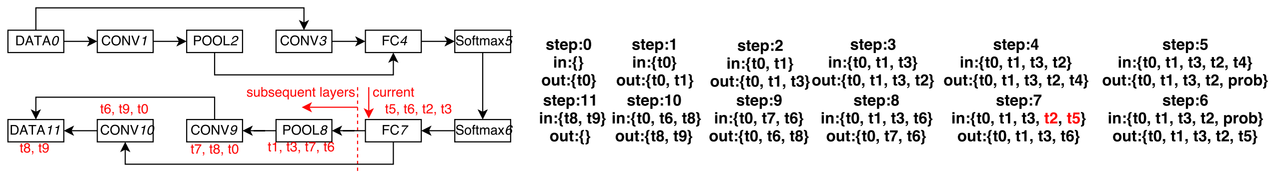

DATA0
CONV1
POOL2
CONV3
FC4
Softmax5
DATA11
CONV10
CONV9
POOL8
FC7
Softmax6
subsequent layers
current
t6, t9, t0
t5, t6, t2, t3
t8, t9
t7, t8, t0
t1, t3, t7, t6
step:0 in:{} out:{t0}
step:1 in:{t0} out:{t0, t1}
step:2 in:{t0, t1} out:{t0, t1, t3}
step:3 in:{t0, t1, t3} out:{t0, t1, t3, t2}
step:4 in:{t0, t1, t3, t2} out:{t0, t1, t3, t2, t4}
step:5 in:{t0, t1, t3, t2, t4} out:{t0, t1, t3, t2, prob}
step:11 in:{t8, t9} out:{}
step:10 in:{t0, t6, t8} out:{t8, t9}
step:9 in:{t0, t7, t6} out:{t0, t6, t8}
step:8 in:{t0, t1, t3, t6} out:{t0, t7, t6}
step:7 in:{t0, t1, t3, t2, t5} out:{t0, t1, t3, t6}
step:6 in:{t0, t1, t3, t2, prob} out:{t0, t1, t3, t2, t5}

Figure 4: Applying *Liveness Analysis* on the nonlinear network shown in Fig.3c. The number after the layer name (e.g., DATA0, CONV1, etc.) represents the step, which are calculated by Alg. 1. We mark the prerequisite tensors for a layer in red, such that t_7, t_8, t_0 are required by CONV9. Each *in* and *out* set tracks live tensors before and after the layer's computations. We can free $t2$ and $t5$ at step 7 since no subsequent dependencies from POOL8, CONV9, CONV10, and DATA11.

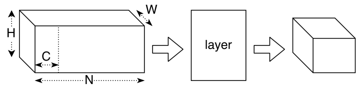


Figure 5: The structure of tensors used in DNN.

Algorithm 1: Construct execution steps for nonlinear neural architectures

```
Data: neural architecture definitions
Result: execution order
1  Function RouteConstruct(layer)
2      if layer is NULL then
3          return
4      $layer \rightarrow counter\_inc()$;
5      if layer→get_counter < size of prev layers then
6          return
7      $computation\_route.push(layer)$;
8      $next\_layers = b \rightarrow get\_next()$;
9      for $next\_l \in next\_layers$ do
10         RouteConstruct(next_l);
11     $reset\ layer \rightarrow counter\ to\ 0$
```

layer granularity, we use tensors as the basic memory scheduling unit.

Alg.1 describes how SuperNeurons constructs execution steps for nonlinear neural architectures. The input is the first network layer; then Alg.1 recursively explores the subsequent layers in Depth-First Searching (DFS), except that it reaches a join where all prior layers must finish before proceeding. The behavior is achieved by the counter in each layer that tracks the input dependencies (line 5 → 6 in Alg.1).

Fig.6 demonstrates an example execution route for a nonlinear network constructed by Alg.1. Each box represents a network layer indexed from **a** to **j**. Note that this network has two fan structures (layer **b**, **c**, **d** and layer **f**, **g**, **h**) nested together. Alg.1 successfully identifies layers **e**, **g** and **h** as the prerequisites for executing **i**.

3.2 Liveness Analysis and Its Related Issues

Liveness analysis enables different tensors to reuse the same physical memory at different time partitions. Our runtime implements a simple yet effective variant of the traditional

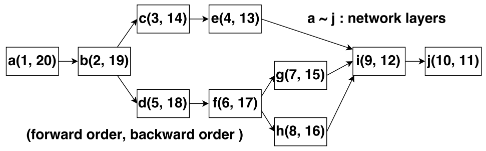


Figure 6: Execution route created by Algorithm 1 on a nonlinear network. The left digit represents the forward step, while the right digit represents the backward step.

data flow analysis constructed in $\mathcal{O}(N^2)$ for various nonlinear neural networks. The general procedures are as follows:

1. We construct an *in* and *out* set for every layers to track the live tensors before and after the layer, which cost $\mathcal{O}(N)$, where N is the network length.
2. The runtime populates a layer's *in* and *out* sets by checking the dependencies of subsequent layers. It eliminates tensors in *in* from *out* if no subsequent layers need them. The cost is $\frac{N(N-1)}{2} \sim \mathcal{O}(N^2)$ as each check costs $N-1$, $N-2$, ..., 2, 1, respectively.

Fig.4 demonstrates the detailed procedures of *Liveness Analysis* on the network shown in Fig.3c. It explicitly lists the content of *in* and *out* sets at each steps. For instance, for FC7, $in = \mathbf{t0}, \mathbf{t1}, \mathbf{t3}, \mathbf{t2}, \mathbf{t5}$. It needs to create tensor $\mathbf{t6}$ to finalize the current computation. Since $\mathbf{t2}$ and $\mathbf{t5}$ are no longer needed after FC7, runtime eliminates them from FC7's *out* set (step:7).

Liveness Analysis reduces the baseline $peak_m = \sum_{i=1}^{N} l_i^f + \sum_{i=1}^{N} l_i^b$ to $\sum_{i=1}^{N} l_i^f + l_N^b$. In order to simplify the analysis, let's assume identical memory usages on every layers, i.e. $l_i^f = l_i^b$ where $i \in [1, N]$. In the network training, the results of forward pass are needed by the backward propagation[2] [7, 26]. Therefore, the forward total memory usages at step k is $cost_k^f = \sum_{i=1}^{k} l_i^f$, where $k \leq N$. During the back-propagation, *Liveness Analysis* frees l_i^f and l_i^b where $i \in [k+1, N]$ at the backward step k since no future dependencies on them as demonstrated in Fig.4. Therefore, the backward total memory usages at step k is $cost_k^b = \sum_{i=1}^{k} l_i^f + l_k^b$ and $k \leq N$. Since $l_i > 0$, the $peak_m$

[2] Not all layers require the previous forward output for the back-propagation, again we simplify the case for the analysis.

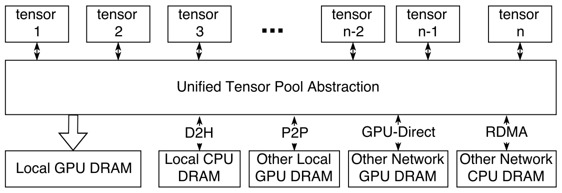


Figure 7: The unified tensor pool provides a consolidated memory abstraction to include various physical memory pools for tensor allocations.

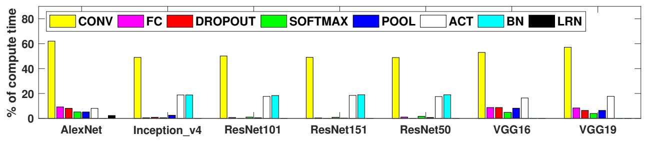
is $max(max(cost_k^f), max(cost_k^b)) = \sum_{i=1}^{N} l_i^f + l_N^b$. Therefore, *Liveness Analysis* saves up to 50% memory from the baseline.

3.2.1 Toward a High Performance Liveness Analysis

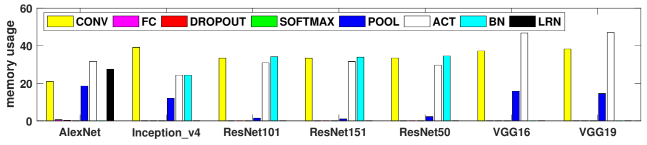
Both the empty initial in set at step 0 and the empty final out set at step 11 in Fig.4 demonstrates *Liveness Analysis* frequently stashes and frees tensors on the fly in a training iteration, while a typical training phase consists of millions of iterations and such intense memory operations incur nontrivial overhead if using the native *cudaMalloc* and *cudaFree* [25]. According to the experiment, ResNet50 wastes 36.28% of the training time on memory allocations/deallocations with *cudaMalloc* and *cudaFree*. To alleviate this performance issue, we implement a fast heap-based GPU memory pool utility. The core concept is to remove the allocation/deallocation overhead by preallocating a big chunk of GPU memory as a shared memory pool. Then we divide the entire GPU memory pool into 1KB blocks as the basic storage unit. The memory pool contains a list of allocated and empty memory nodes. Each node in the two lists contains memory address, occupied blocks and node ID. For an allocation request, the memory pool finds the first node with enough free memory from the empty list. After that, it updates the empty list and creates a new node in the allocated list to track the current allocation. For a deallocation request, the memory pool locates the node in the allocated list with the ID-to-node hash-table, then the pool places the node back to the empty list.

3.3 Unified Tensor Pool(UTP) and Its Related Issues

If the depth of a neural network goes to 10^3, the ImageNet training still consumes at least 10^2GB memory. Therefore, *Liveness Analysis* alone is inadequate for the emerging deep nonlinear neural architectures. We provide *Unified Tensor Pool (UTP)* to further alleviate the GPU DRAM shortage by asynchronously transferring tensors in/out of external memory. *UTP* is a consolidated memory pool abstraction for tensor allocations/deallocations, using various external physical memory such as CPU DRAM, DRAM of other GPUs, or remote CPU/GPU DRAM. In this paper, we focus on the scenario of using local CPU DRAM as an external pool for the fast and efficient interconnect, but the abstraction also applies to other cases shown in Fig.7. *UTP* intelligently manages the tensor placement, movement, allocation and deallocation, so that the underlying memory management is entirely transparent to DL practitioners.



(a) breakdown of execution time by layer types



(b) breakdown of memory usages by layer types

Figure 8: The percentages of execution time and memory usages by layer types in different networks. Note that the execution time includes both forward and backward passes.

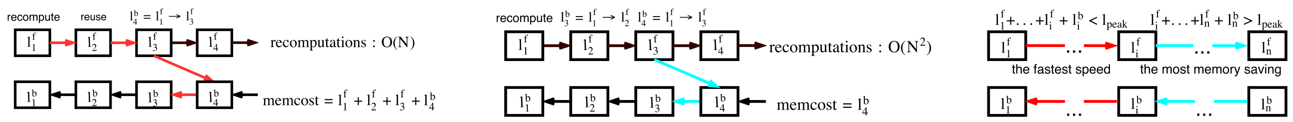
3.3.1 Basic UTP Memory Management: Memory Offloading and Prefetching

Not all the layers are suitable for *Offloading* and *Prefetching*. We define transferring tensors from GPU to external physical pools as *Offloading*, and the reversed operation as *Prefetching*. Fig.8a and Fig.8b demonstrate that POOL, ACT, BN and LRN all together occupy over 50% of the total memory, while their computations only account for an average of 20% of the entire workload. Thus, offloading these layers incurs a great overhead due to the insufficient overlapping of communications and computations. It is also not fruitful to offload on Dropout, Softmax and FC layers since they only use less than 1% of the total memory. Therefore, we only offload the tensors from CONV layers.

Offloading: the runtime asynchronously transfers the forward outputs of CONV layers to the preallocated pinned CPU memory. It records an event for this data transfer and frees the tensor's GPU memory once the event is completed. The runtime has an independent thread running in the background to check events of memory copies; and this enables GPU-to-CPU data transfers to overlap with the forward computations starting from the current CONV layer to the next one.

Prefetching: the runtime asynchronously brings the offloaded and soon to be reused tensors back to the GPU DRAM. At any CONV layers in the backward, the runtime asynchronously fetches the required tensors for the previous CONV layer. This enables the CPU-to-GPU data transfer to overlap with the backward computation starting from the current CONV layer to the previous one.

Offloading and *Prefetching* reduce $peak_m$ after *Liveness Analysis* to $\sum_{i=1}^{N}(l_i^f \notin checkpoints) + l_N^b$, where $checkpoints = \{CONV\}$. Since layers in $checkpoints$ are offloaded, the total memory consumption at each backward steps is $cost(k) = \sum_{i=1}^{k}(l_i^f \notin checkpoints) + l_k^b$,


recompute
reuse
$l_4^b = l_1^f \rightarrow l_3^f$
recomputations : O(N)
memcost = $l_1^f + l_2^f + l_3^f + l_4^b$
recompute $l_3^b = l_1^f \rightarrow l_2^f$ $l_4^b = l_1^f \rightarrow l_3^f$
recomputations : O(N^2)
memcost = l_4^b
$l_1^f + \ldots + l_i^f + l_i^b < l_{peak}$
$l_i^f + \ldots + l_n^f + l_n^b > l_{peak}$
the fastest speed
the most memory saving

(a) Speed-Centric Recomputation (b) Memory-Centric Recomputation (c) Cost-Aware Recomputation

Figure 9: The speed-centric strategy only recomputes the segment once, and other backward layers within the segment will reuse the recomputed tensors. Thus, it only incurs $\mathcal{O}(N)$ additional computations, but $memcost$ is $\sum_{i=1}^{seg} l_i^f + l_{seg}^b$. The memory-centric strategy recomputes forward dependencies every time for each backward layers. Though it incurs $\mathcal{O}(N^2)$ additional computations, $memcost$ is the lowest, i.e. l_i^b. Cost-Aware Recomputation profiles the memory usages across recomputation segments. It uses the speed-centric strategy (red) if $memcost$ of a segment is less than l_{peak}, and the most memory saving strategy (blue) otherwise.

Algorithm 2: The basic LRU operations

```
Data: Tensor ($T$) and $LRU$
Result: Tensor with the GPU memory.
Function $LRU.in$ ($T$)
    $T.Lock \leftarrow false$        /* A layer will lock its
        dependent tensors in the computation. */
    $LRU.insertFront(T)$
Function $LRU.out$ ($T$)
    $freedMem \leftarrow 0$
    while $freedMem < T.size$ do
        $T' = LRU.getLastUnlockedTensor()$
        $freedMem = freedMem + T'.size$
        $remove\ T'\ from\ LRU\ list$
        offload $T'.GA$ to $T'.CA$     /* CA is CPU Addr */
    $T.GA \leftarrow Malloc(T.size)$
Function $Check$ ($LRU$, $T$)
    $isFound \leftarrow LRU.find(T)$
    if $isFound = false$ then
        $T.GA \leftarrow Malloc(T.size)$  /* GA is GPU Addr */
        if $T.GA = \emptyset$ then
            $T.GA \leftarrow LRU.out()$
        $LRU.in(T)$                   /* cache miss */
    else
        $LRU.placeToFront(T)$           /* cache hit */
    return $T.GA$
```

where $k \in [1, N]$. The memory usage of each layers is non-negative, thus $peak_m = max(cost(k))$ is $\sum_{i=1}^{N}(l_i^f \notin checkpoints) + l_N^b$.

3.3.2 Caching Tensors on GPU DRAM

While the overlapping opportunity is limited given the fixed amount of computations in an iteration, the aforementioned on-demand *Prefetching/Offloading* protocol can quickly exhaust the chance. Nowadays CPU-to-GPU data movements over PCI-E, GPU-to-GPU data movements over the same PCI-E switch, and GPU-to-remote GPU over GPU-Direct RDMA deliver a practical speed of 8 GB/s, 10 GB/s, and 6 GB/s respectively but transferring Gigabytes data in each training iterations incurs the nontrivial overhead. Therefore, this on-demand tensor transfer protocol must be optimized. SuperNeurons proposes a *Tensor Cache* to exploit the temporal localities of tensors. It caches tensors on GPU DRAM to maximize their reuses and to minimize the global communications. With *Prefetching* and *Offloading*, the runtime only triggers data transfers when GPU DRAM is insufficient.

We adopt Least Recent Used (LRU) tensor replacement policy to build *Tensor Cache*. Since the back-propagation demonstrates the head-to-tail and tail-to-head computation pattern, it subjects the most recent used tensors to the earliest reusing as suggested in Fig.4. This motivates us to design *Tensor Cache* with a simple variant of LRU. While there are other sophisticated cache replacement policies that might better fit the scenario, thorough discussions of them fall out the scope of this paper.

Alg.2 demonstrates the three key operations of proposed LRU. 1) *LRU.in* function intends to place a tensor into LRU. Each tensor has a lock, and a tensor cannot be removed from LRU if locked. A layer will lock required tensors at calculations. LRU is implemented as a list with Most Frequently Used (MFU) at the front. 2) *LRU.out* function intends to remove enough bytes for a new tensor. It offloads the unlocked Least Recent Used tensors to CPU RAM until it has enough free memory for the new one. 3) *Check* function decides what operator to run on the tensor. It takes in a tensor to check if the tensor is in LRU based on the object address (line 2). If found, we place the tensor to the MFU position, i.e. the list front (line 9), and return the tensor's GPU address. This is the hit scenario. If not found, we call *LRU.out* to free enough memory for the new tensor before inserting it into LRU. This is the miss scenario.

3.4 Cost-Aware Recomputation

POOL, ACT, LRN and BN all together use an average of 50% memory, while their forward computations only account for less than 10% of the total time. This exposes an additional 50% memory savings with a fraction of performance loss by recomputing the forward dependencies in the back-propagation. Basically, the runtime frees the tensors in cheap-to-compute layers such as POOL for reconstructions. In general, there are memory-centric and speed-centric strategies for the recomputation for memory.

The speed-centric strategy keeps the recomputed tensors so that other backward layers can directly reuse them. Fig.9a denotes the procedures in red. At the backward step on l_4^b, it performs a forward pass from l_1^f to l_3^f to get dependencies for l_4^b. It keeps l_1^f, l_2^f so that they can be re-used for the backward computation on l_3^b and l_2^b. MXNet [6] adopts this strategy. It incurs the least $\mathcal{O}(N)$ additional computations, but $memcost$ is $\sum_{i=1}^{seg}(l_i^f) + l_{seg}^b$. $memcost$ will exceed l_{peak} if l_{peak} is within the segment.

The memory-centric strategy always recomputes the dependencies for each backward layer. In contrast to the speed-centric one, it fully exploits the memory-saving opportunity by freeing the recomputed intermediate results. For example, it recomputes $l_1^f \rightarrow l_3^f$ for l_4^b, while it recomputes $l_1^f \rightarrow l_2^f$ again for l_3^b as demonstrated by the blue lines in Fig.9b. The $memcost$ stays at l_i^b guaranteed to be $\leq l_{peak}$, but the strategy incurs $\mathcal{O}(N^2)$ additional computations.

We present a new *Cost-Aware Recomputation* that leverages the advantages of both methods. It is motivated by the observation that the memory costs of most recomputation segments are $\leq l_{peak}$, i.e. $\sum_{i=1}^{seg}(l_i^f)+l_{seg}^b \leq l_{peak}$. That implies we can leverage the least recomputations in the speed-centric strategy while still guaranteeing a memory usage of $\leq l_{peak}$ as in the memory-centric strategy. The general procedures of *Cost-Aware Recomputation* are as follows:

1. The runtime iterates over all the layers to find $l_{peak} = max(l_i)$ as the threshold.

2. In a recomputation segment, the runtime applies the speed-centric strategy (marked by red in Fig.9c) if $\sum_{i=1}^{seg}(l_i^f) + l_{seg}^b \leq l_{peak}$, and the memory-centric strategy (marked by blue in Fig.9c) otherwise.

Table.1 summarizes the extra computations for two basic strategies and *Cost-Aware Recomputation*. Our cost-aware method ensures $peak_m$ to be consistent with the memory-centric strategy, while the extra computations are comparable to the speed-centric strategy.

Cost-Aware Recomputation finally reduces $peak_m$ to $max(l_i)$. Previously, *Liveness Analysis* and *Offloading* jointly reduce the $cost_k^b$ to $\sum_{i=1}^{k}(l_i^f \notin checkpoints) + l_k^b$. Since *non-checkpoints* layers will be freed for recomputations, only the nearest *checkpoint* layer exists in the GPU memory. Thus, $cost_k^b = l_{checkpoint}$. During the recomputations, $cost_k^b$ can be either $\sum_{i=1}^{k}(l_i^f) + l_k^b \leq l_{peak}$ or l_i^b depending what recomputation strategies to use. On the other hand, *Cost-Aware Recomputation* guarantees $cost_k^b \leq l_{peak} = max(l_i)$ (see analyses above). Thus, the final network wide $peak_m = max(cost_k^b) = l_{peak}$, which is the minimal $peak_m$ achievable at the layerwise granularity.

3.5 Finding the Best Convolution Algorithm under the Memory Constraint

The speed of CONV layers significantly impacts the training as it accounts for over 50% of total computing time (Fig.8).

Table 1: The counts of recomputations (extra) and $peak_m$ using the speed-centric, the memory-centric and Cost-Aware Recomputation.

	speed-centric		**memory-centric**		**cost-aware**	
	extra	$peak_m$	extra	$peak_m$	extra	$peak_m$
AlexNet	14	993.018	23	886.23	17	886.23
ResNet50	84	455.125	118	401	85	401
ResNet101	169	455.125	237	401	170	401

cuDNN provides several convolution algorithms, e.g. using FFT, Winograd and GEMM, for different contexts. Some of them, FFT in particular, require temporary convolution workspaces to delivery the maximal speed as demonstrated in Fig.2. Therefore, the memory is also a critical factor to the high-performance training.

We implement a dynamic strategy for allocating convolution workspaces. It is dynamic because the memory left for convolution workspaces constantly changes in every step according to *Liveness Analysis*, *UTP* and *Cost-Aware Recomputation*. Since convolution workspaces do not affect the functionality, the allocations of functional tensors such as data and parameters are prioritized. The runtime then steps into each layer to profile the free bytes left in GPU DRAM after those memory techniques have been applied. With free memory information at individual steps, the runtime benchmarks all the memory-feasible convolution algorithms to pick the fastest one. Please note the runtime skips convolution algorithms that require more memory than it can provide. Each layer selects the fastest algorithm under the remaining GPU DRAM, and therefore maximize the performance of CONV layers and the entire training.

4. Evaluations

In this section, we present the results of our experimental studies that evaluate each of the memory and performance techniques in SuperNeurons. We also performed end-to-end evaluations against TensorFlow, MXNet, Caffe and Torch on various neural networks to justify the design.

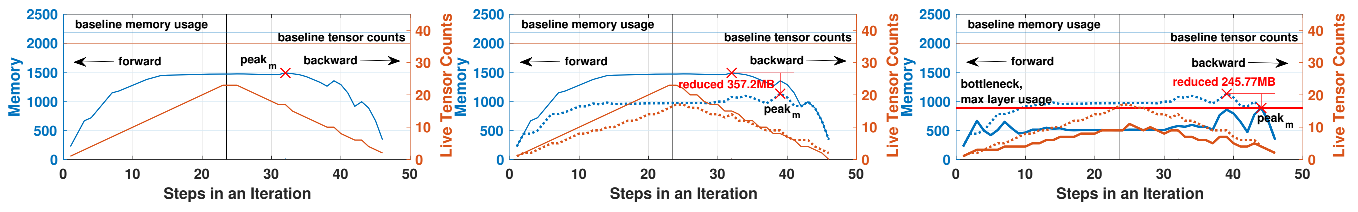
4.1 Components Evaluations

4.1.1 Memory Optimizations

We use the naive network-wide tensor allocation strategy as the baseline. Thus, the $peak_m$ of baseline is $\sum_{i=1}^{N} l_i^f + \sum_{i=1}^{N} l_i^b$, where N is the network length (defined in Sec.3). Since cuDNN operates at the layerwise granularity, $peak_m$ is bounded by the maximal memory usage among layers, i.e. l_{peak}.

Liveness Analysis reduces the baseline's $peak_m$ to $\sum_{i=1}^{N} l_i^f + l_N^b$. Fig.10a demonstrates how *Liveness Analysis* affect memory usages and live tensor counts at each forward/backward step on AlexNet. [3] Since AlexNet has 23 layers, there are

[3] the structure of AlexNet is CONV1→RELU1→LRN1→POOL1 →CONV2→RELU2 →LRN2→POOL2→CONV3→RELU3 →CONV4→RELU4→CONV5→RELU5→POOL5→FC1 →RELU6→Dropout1→FC2→RELU7→Dropout2 →FC3→Softmax



(a) liveness analysis (b) prefetching/offloading + liveness (c) recomputation + previous two

Figure 10: The evaluations of *Liveness Analysis*, *Prefetching/Offloading* and *Cost-Aware Recomputation* on AlexNet at the batch size of 200. AlexNet has 23 layers, and a training iteration consists of $1 \rightarrow 23$ forward steps and $24 \rightarrow 46$ backward steps. The blue curve (left axis) depicts memory usages at each step, while the orange curve (right axis) depicts live tensor counts at each step. (a) demonstrates how *Liveness Analysis* affects memory usages w.r.t the baseline (horizontal lines). (b) demonstrates how *Offloading/Prefetching* improve *Liveness Analysis* by comparing the memory usages of both techniques (blue dashed lines in (b)) with *Liveness* alone (solid blue curve in (b)). Similarly, (c) demonstrates how *Cost-Aware Recomputation* improve the previous two; and dashed lines in (c) are from (b).

23 forward steps and 23 backward steps. The central vertical line separates forward and backward, each of which contains 23 computational steps. The baseline allocates 36 data tensors consuming 2189.437MB, while *Liveness Analysis* uses up to 17 tensors with a peak memory usage of 1489.355MB. This demonstrates 31.9% improvement over the baseline in terms of $peak_m$. It is also observable that the location of $peak_m$ is not necessarily consistent with the peak tensor count. This confirms our claim that the memory is unevenly distributed across network layers.

To verify the cost model, i.e. $cost_k^b = \sum_{i=1}^{k} l_i^f + l_k^b$, we delve into the memory usages of the peak layer. Fig.10a suggests the 32th step reaches $peak_m$. This corresponds to the backward POOL5 in AlexNet, and $k = 14$ (or 46 - 32). The forward layers before POOL5 stash 5 tensors, consuming 1409.277MB ($\sum_{i=1}^{14} l_i^f$). Meanwhile the backward POOL5 stashes 3 tensors, consuming 80.078MB (l_{14}^b). Therefore, $cost_{14}^b$ = 1409.277 + 80.078 = 1489.355MB, which is consistent with the measured $peak_m$.

Prefetching and Offloading reduces the $peak_m$ after *Liveness Analysis* to $\sum_{i=1}^{N}(l_i^f \notin checkpoints) + l_N^b$. Fig.10b demonstrates the updated memory usages and live tensor counts after *Prefetching/Offloading* is applied on top of *Liveness Analysis*. We set CONV layers as checkpoints for offloading. The new $peak_m$ is 1132.155 MB at the 39th step or POOL2 backward. It further reduces 357.2MB off of the previous $peak_m$, for a total of 48.29% improvement over the baseline's $peak_m$. The new $peak_m$ shifts from POOL5 to POOL2 because of the number of CONV layers ahead of them. CONV1, CONV2, CONV3, and CONV4 are located before POOL5, and they consume 221.56MB, 142.38MB, 49.51MB and 49.51MB, respectively. The runtime offloads CONV $1 \sim 4$ to CPU RAM and prefetches CONV5. This leads the new memory usage of POOL5 to be 1489.355 - 221.56 - 142.38 - 49.51 = 1075.9MB, which is less than the measured new $peak_m$ 1132.155 MB at POOL2.

To verify the updated cost model, i.e. $\sum_{i=1}^{k}(l_i^f \notin checkpoints) + l_k^b$, we compare the calculated live tensor count from the model with the actual measurement. There are 2 checkpoints, CONV1 and CONV2, before POOL2, and the runtime prefetches CONV2 in the backward pass. As a result, the calculated live tensor count at POOL2 is 10 (measured live tensors before POOL2) - 1 (CONV1) = 9. This is the same as our actual measurement of 9 tensors at POOL2. Therefore, the updated cost model after *Prefetching/Offloading* is still valid.

Finally, ***Cost-Aware Recomputation*** reduces $peak_m$ to $\max(l_i)$. In theory, $\max(l_i)$ is the minimal $peak_m$ at the layerwise granularity as cuDNN needs to at least stash the tensors in a layer to compute. Fig.10c demonstrates stepwise memory usages and live tensor counts with all three techniques. We profile that $max(l_i) = 886.385MB$ at the backward LRN1 by iterating through every layer. Fig.10c demonstrates a $peak_m$ of 886 MB at the 44th step, which is the backward computation of LRN1. Therefore, the three proposed memory saving techniques successfully reduce the $peak_m$ from $\sum_{i=1}^{N} l_i^f + \sum_{i=1}^{N} l_i^b$ to $max(l_i)$.

4.1.2 Speed Optimizations

The runtime equips with a GPU Memory Pool and a Tensor Cache to improve the performance of memory techniques, and it also has a dynamic strategy for allocating convolution workspaces to accelerate the training speed. More specifically, the GPU Memory Pool amortizes the non-trivial overhead of high-frequency memory allocations/deallocations in *Liveness Analysis*, and Tensor Cache enables tensor reuse to minimize data transfers in *Prefetching/Offloading*. Fig.10c demonstrates how the GPU free space dynamically changes at each forward and backward step due to the three memory techniques. The runtime allocates convolution workspaces within the free memory at each step. As a result, the performance is optimized at individual layers under different stepwise memory constraints.

Table 2: The improvement of the GPU memory pool over cudaMalloc and cudaFree on various networks. The batch size for AlexNet is 128, while the rest is 16.

img/s	AlexNet	VGG16	InceptionV4	ResNet50	ResNet101	ResNet152
CUDA	359.4	12.1	6.77	21.5	11.3	7.46
Ours	401.6	14.4	10.0	32.9	18.95	13.2
speedup	1.12x	1.19x	1.48x	1.53x	1.68x	1.77x

Table 3: Communications with/without Tensor Cache. We benchmark the result on AlexNet by increasing the batch size from 256 to 1024.

Communications in GB	256	384	512	640	896	1024
Without Tensor Cache	2.56	3.72	4.88	6.03	8.35	9.50
Tensor Cache	0	0	0	0	0	0.88

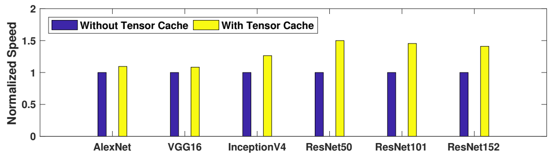


Figure 11: Normalized performance with and without *Tensor Cache*. The batch size of AlexNet is 128, and 32 for the rest.

GPU Memory Pool amortizes the non-trivial overhead of intensive memory operations in *Liveness Analysis* by preallocating a big chunk of GPU memory. Table 2 illustrates the performance improvement of using *GPU Memory Pool* over cudaMalloc and cudaFree. Linear networks such as AlexNet and VGG involve much fewer memory operations than non-linear ones such as InceptionV4 and ResNet50 $\rightarrow$ 152 due to the limited depth. Therefore, the speedups on nonlinear networks (ResNet 50$\rightarrow$152 and InceptionV4) are more significant than linear networks (AlexNet, VGG).

Tensor Cache intends to reduce unnecessary data transfers in *Prefetching/Offloading*. Specifically, the offloading is unnecessary if a network can fit into the GPU DRAM. In Table 3, we can see *Tensor Cache* successfully avoids communications at batch sizes of 256 $\rightarrow$ 896, while the communications, in the scenario without *Tensor Cache*, linearly increase along batch sizes. The training performance will deteriorate if communications outweigh computations. Fig.11 demonstrates up to 33.33% performance loss without using *Tensor Cache*. It is also noticeable that the speedup on linear networks (AlexNet, VGG16) is less significant than nonlinear ones (ResNet50$\rightarrow$152, Inception). In general, the computation intensity of a linear network layer is far greater than the non-linear one. Because their communications can overlap with computations in *Prefetching/Offloading*, *Tensor Cache* does not provide the comparable speed up for AlexNet and VCG16.

Dynamic Convolution Workspace Allocation intends to optimize each layer's training speed together with the three memory techniques. Convolution workspaces are critical for high performance, while the amount of free memory for convolution workspaces constantly changes at different computing steps as demonstrated in Fig.10c. The runtime picks the fastest memory-feasible convolution algorithm at a particular step.

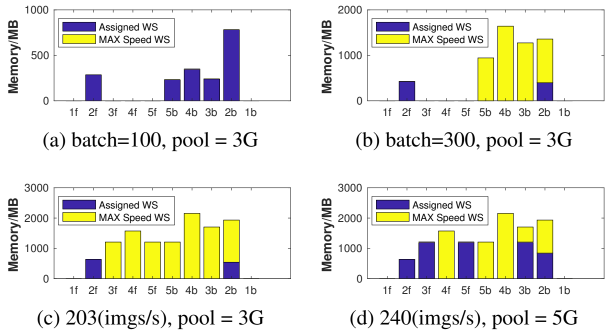


Figure 12: Dynamic Conv workspace allocations in the runtime. The digit in the x-axis represents the ith CONV layer, while the "f" or "b" represent the forward or backward computations.

Table 4: Going Deeper: the deepest ResNet that different frameworks can reach on a 12GB NVIDIA K40. The batch size is fixed at 16. ResNet has 4 for-loops to control its depth: $depth = 3*(n_1+n_2+n_3+n_4)+2$, where n_i is the upper limit of ith for-loop. We fix $n_1 = 6$, $n_2 = 32$, and $n_4 = 6$, while varying n_3 to increase the depth.

Depth	Caffe	MXNet	Torch	TensorFlow	SuperNeurons
ResNet	148	480	152	592	1920

Fig.12a and Fig.12b demonstrate that the runtime automatically reduces CONV workspaces to accommodate functional tensors with the increasing batch size. Specifically, the runtime prioritizes the functional tensor allocations at batch 300 under 3 GB memory pool (Fig.12b), while it provisions the most workspace for the maximal speed at batch 100 (Fig.12a). In general, a higher speed is observable with more convolution workspaces. Fig.12c and Fig.12d demonstrate that the training speed (images per second) increases from 203 img/s to 240 img/s with additional CONV workspaces.

4.2 Going Deeper and Wider

Our primary goal is to enable ML practitioners to explore deeper and wider neural architectures within the limited GPU DRAM. In this section, we conduct end-to-end comparisons to TensorFlow, MXNet, Caffe and Torch with several mainstream linear networks (AlexNet, VGG16) and non-linear ones (ResNet50 $\rightarrow$ 150, Inception V4) under the same experiment setup.

We increase the batch size to go wider. Table 5 presents the largest batch reachable by different frameworks before the GPU out-of-memory error. SuperNeurons consistently outperforms the mainstream frameworks on both linear and non-linear networks. On average, it handles 1.8947x

Table 5: Going Wider: the largest batch size that several mainstream neural architectures can reach in different frameworks with a 12GB NVIDIA K40.

peak batch	Caffe	MXNet	Torch	TensorFlow	SuperNeurons
AlexNet	768	768	1024	1408	1792
VGG16	48	64	48	80	224
InceptionV4	16	N/A	N/A	64	240
ResNet50	24	80	32	128	384
ResNet101	16	48	16	80	256
ResNet152	16	32	16	48	176

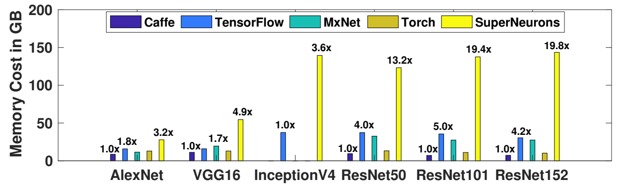


Figure 13: Going Wider: the corresponding memory usages for the batch size in TABLE 5.

larger batches than the second best. SuperNeurons can train ResNet101 at the batch of 256, which is 3x larger than the second best TensorFlow.

Fig.13 demonstrates the corresponding memory requirement to peak batches in Table 5. The translation is nonlinear because of the convolution workspace. We calculate the memory requirement with $\sum_{i=1}^{N} l_i^f + \sum_{i=1}^{N} l_i^b$, and l_i is the sum of the memory usages of all tensors in the layer. It is observable that SuperNeurons handles up to a 19.8x larger model than Caffe.

We add layers to go deeper. Table 4 demonstrates SuperNeurons trains 12.9730x, 12.6316x, 4.0000x, and 3.2432x deeper ResNet than Caffe, Torch, MXNet, and TensorFlow, respectively. Particularly, SuperNeurons can train a ResNet up to 2500 residual units having approximately 10^4 basic layers at the batch size of 1 on a 12GB GPU.

The training speed is measured by the number of processed images per second. Fig.14 presents an end-to-end training speed comparison of SuperNeurons to the mainstream DL systems. SuperNeurons consistently demonstrates the leading speed on various linear networks (AlexNet, VGG16) and nonlinear ones (ResNet50 $\rightarrow$ 152, Inception V4). The performance largely results from the abundant supply of convolution workspaces saved by the dynamic GPU memory scheduler. We can also observe that the speed has slowly deteriorated as batch size increases. This is due to the growing communications from more frequent tensor swapping between CPU and GPU DRAM. The performance will be at its worst when GPU memory can only accommodate one network layer. Then, the runtime has to constantly offload the current layer before proceeding to the next one.

5. Related Work

Several solutions have been proposed to address the GPU DRAM shortage for training large-scale neural networks.

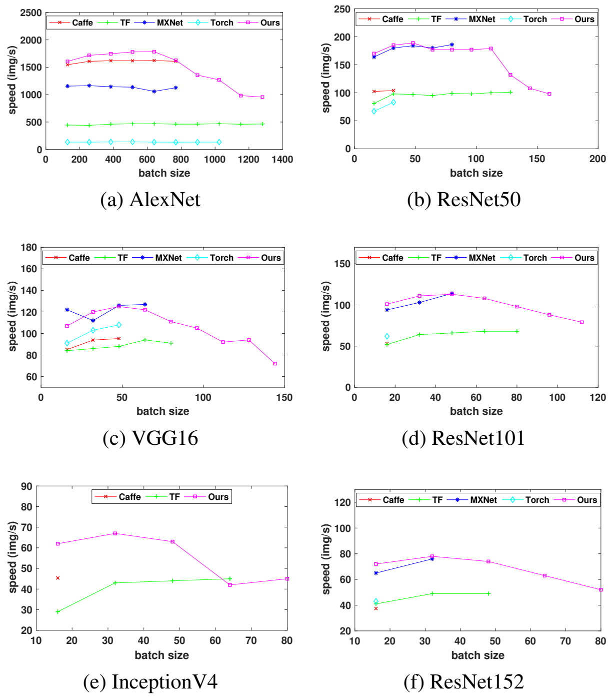


Figure 14: An end-to-end evaluation of different DL frameworks. We benchmark the data on a TITAN XP.

Model Parallelism provides a straightforward solution to the large network training. DistBelief [10] partitions a network across multiple machines so that each machine holds a segment of the original network. Coates et al [8] discuss another partition scheme on multi-GPUs. Model Parallelism demands huge intra-network communications for synchronization. Therefore, most DL systems parallelize the training with Data Parallelism for better performance [2, 5, 9, 14]. In this paper, we focus on the GPU DRAM shortage issue for Data Parallelism.

Under Data Parallelism, vDNN [19] proposes a prefetching and offloading technique to utilize the CPU DRAM as an external buffer for the GPU. It tries to overlap communications with computations by asynchronously swapping the data between CPU and GPU amid the back-propagation. The performance of this method largely depends on the communication/computation ratio. Some layers such as POOL are very cheap to compute, while the GPU processing speed is several orders of magnitude faster than the PCI-E 16x bus. In nonlinear networks, the performance will quickly deteriorate once computations are inadequate to overlap with communications. Chen et al [6] also introduces a recomputation strategy to trade computations for memory. However, their method fails to fully exploit the memory-saving opportunities and computation efficiency because it ignores the memory variations among layers.

Removing the parameter redundancy also reduces the memory usage. For example, network pruning [11] removes near zero parameters, and quantization [23] or precision reduction [15] utilize low precision floats to save memory. Although the parameter reduction has immense benefits in deploying neural networks on embedded systems, parameters only account for a negligible portion of memory usage in the training. Therefore, these approaches are quite limited to training only.

6. Conclusion

In this paper, we focus on the GPU memory scheduling problem for training deep neural networks, and we propose a novel dynamic scheduling runtime to tackle the issue. The runtime features three memory techniques to reduce $peak_m$ to $max(l_i)$, which is the minimum at the layer-wise granularity. We also propose several performance optimizations to guarantee high performance. Evaluations against state-of-the-art DL frameworks have demonstrated the effectiveness and efficiency of proposed dynamic scheduling runtime. It creates new opportunities for DL practitioners to explore deeper and wider neural architectures, allowing the opportunity for better accuracy with even deeper and wider designs.

7. Acknowledgements

This research is funded in part by the DARPA Award 16-43-D3M-FP-040 and gifts from Google, VMware, Mellanox, and Oracle. Zenglin Xu and Jianmian Ye were supported by a grant from the Natural Science Foundation of China (No. 61572111), and a Fundamental Research Fund for the Central Universities of China (No. ZYGX2016Z003).

References

[1] Mxnet's graph representation of neural networks. `http://mxnet.io/architecture/note_memory.html`.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[3] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah. Comparative study of caffe, neon, theano, and torch for deep learning. 2016.

[4] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[5] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[6] T. Chen, B. Xu, C. Zhang, and C. Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[7] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[8] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep learning with cots hpc systems. In *International Conference on Machine Learning*, pages 1337–1345, 2013.

[9] R. Collobert, S. Bengio, and J. Mariéthoz. Torch: a modular machine learning software library. Technical report, Idiap, 2002.

[10] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[11] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. Eie: efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 243–254. IEEE Press, 2016.

[12] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[13] G. Huang, Z. Liu, K. Q. Weinberger, and L. van der Maaten. Densely connected convolutional networks. *arXiv preprint arXiv:1608.06993*, 2016.

[14] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[15] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, N. E. Jerger, and A. Moshovos. Proteus: Exploiting numerical precision variability in deep neural networks. In *Proceedings of the 2016 International Conference on Supercomputing*, page 23. ACM, 2016.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[17] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[18] G. Pleiss, D. Chen, G. Huang, T. Li, L. van der Maaten, and K. Q. Weinberger. Memory-efficient implementation of densenets. *arXiv preprint arXiv:1707.06990*, 2017.

[19] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016.

[20] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[21] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[22] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, pages 4278–4284, 2017.

[23] V. Vanhoucke, A. Senior, and M. Z. Mao. Improving the speed of neural networks on cpus. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, volume 1, page 4, 2011.

[24] L. Wang, W. Wu, G. Bosilca, R. Vuduc, and Z. Xu. Efficient communications in training large scale neural networks. *arXiv preprint arXiv:1611.04255*, 2016.

[25] L. Wang, W. Wu, Z. Xu, J. Xiao, and Y. Yang. Blasx: A high performance level-3 blas library for heterogeneous multi-gpu computing. In *Proceedings of the 2016 International Conference on Supercomputing*, page 20. ACM, 2016.

[26] L. Wang, Y. Yang, R. Min, and S. Chakradhar. Accelerating deep neural network training with inconsistent stochastic gradient descent. *Neural Networks*, 2017.

[27] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra. Hierarchical dag scheduling for hybrid distributed systems. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 156–165. IEEE, 2015.

[28] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.