

# Scaling Distributed Deep Learning Workloads beyond the Memory Capacity with KARMA

Mohamed Wahib<sup>\*§</sup>, Haoyu Zhang<sup>†</sup>, Truong Thao Nguyen<sup>\*</sup>, Aleksandr Drozd<sup>§</sup>, Jens Domke<sup>§</sup>, Lingqi Zhang<sup>†</sup>,

Ryousei Takano<sup>\*</sup>, Satoshi Matsuoka<sup>§†</sup>

<sup>\*</sup> National Institute of Advanced Industrial Science and Technology, Japan

{mohamed.attia, nguyen.truong, takano-ryousei}@aist.go.jp

<sup>†</sup> miHoYo Inc, (This work was done while the coauthor worked in Tokyo Institute of Technology) lynkzhang@gmail.com

<sup>‡</sup> Tokyo Institute of Technology, Tokyo, Japan zhang.l.ai@m.titech.ac.jp

<sup>§</sup> RIKEN Center for Computational Science, Kobe, Japan {aleksandr.drozd, jens.domke}@riken.jp, matsu@acm.org

**Abstract**—The dedicated memory of hardware accelerators can be insufficient to store all weights and/or intermediate states of large deep learning models. Although model parallelism is a viable approach to reduce the memory pressure issue, significant modification of the source code and considerations for algorithms are required. An alternative solution is to use out-of-core methods instead of, or in addition to, data parallelism.

We propose a performance model based on the concurrency analysis of out-of-core training behavior, and derive a strategy that combines layer swapping and redundant recomputing. We achieve an average of 1.52x speedup in six different models over the state-of-the-art out-of-core methods. We also introduce the first method to solve the challenging problem of out-of-core multi-node training by carefully pipelining gradient exchanges and performing the parameter updates on the host. Our data parallel out-of-core solution can outperform complex hybrid model parallelism in training large models, e.g. Megatron-LM and Turning-NLG.

**Index Terms**—Deep Neural Networks, Out-of-core, GPUs

## I. INTRODUCTION

Training Deep Neural Networks (DNNs) is increasingly becoming one of the main HPC workloads. As model and dataset sizes for Deep Learning (DL) become increasingly large, the memory requirement for training Neural Networks (NNs) increases dramatically. Even though the latest generation of Nvidia GPUs have up to 32 GiB (V100), this capacity remains a major bottleneck in a lot of the cases [1]. For example, with a large network such as ResNet-200 [2], the local batch-size for training cannot be larger than six ImageNet samples, and in ResNet-1001 the local batch size drops down to two samples. This problem is also a challenge for models that require tens of billions of parameters [3], [4], at which the model will not fit into a single GPU and programmers are forced to employ complex model partitioning methods [1].

Distributed training can be used if multiple GPUs are available. Data parallelism is a commonly used scheme in which the model is replicated and the training data is distributed. However, this scheme does not reduce memory pressure from model parameters and activations, forcing users to resort to partitioning a model on several devices (*model parallelism*). For instance, the Megatron-LM model [3] has 8.3 billion parameters which need at least 16 GPUs to fit, assuming 16 GiB memory per GPU. In addition, even if a given model is

relatively small, there are cases where even a single training sample is too large to be processed on a single GPU. Such cases include high resolution medical or satellite images which can go up to 2 GiB per sample [5]. In comparison, the widely used ImageNet dataset [6] has images that are smaller than 100 KiB per sample (re-sized to  $224 \times 224$ ).

Although model parallelism could be a solution, construction of a cost model and significant modification of the code is needed for every model/dataset/system combination [1], [7], [8]. Another general solution to this memory capacity problem, that we discuss in this paper, is to use out-of-core methods, without or with redundant recompute, to break the GPU memory limitation [9]–[14].

The first challenge that KARMA must address is how to first derive an efficient out-of-core strategy that reduces the stall in the execution pipeline, i.e. address the device occupancy bottleneck. Prefetching and data swapping in general, is a well-researched area. That being said, general prefetching and swapping techniques are not efficient for out-of-core DL since they don't provide a comprehensive considerations of the concurrency requirements and occupancy w.r.t. DL workloads [9], [10]. The main challenge in deriving an efficient prefetching and swapping strategy is to build a robust model for projecting the minimum required concurrency to keep device utilization as close as possible to maximum. This requires taking into consideration specific features and requirements in DL training: reuse of intermediate results from the forward phase in the backward phase, orchestrating complex pipelines in case of distributed training, non-linear dependency between layers, and memory footprint to compute imbalance (i.e. compute is not linearly correlated to the memory footprint). We propose a performance model to derive a capacity-based out-of-core strategy by the means of assuring a minimum concurrency, i.e., available parallelism, that keeps the device at the highest possible utilization. In addition, we identify and utilize any opportunities at which redundantly recomputing layers reduces the stalls in the prefetching pipeline. More specifically, we generate a schedule to interleave the layers designated for swapping with the recompute of layers that are not swapped.

The second challenge that KARMA must address is how to enable multi-GPU training; none of the existing out-of-

core methods support multi-GPU, since the layers are split between the GPU and CPU, and hence the weight update becomes complicated. To overcome that, we use a pipeline of overlapped gradient exchanges, by groups of layers, and orchestrate the update of the weights to be executed on the CPU, in a heterogeneous fashion, before swapping the layers in and out of the device for the following iteration.

In summary, our paper contributes the following:

- We propose a performance model based on occupancy analysis to estimate the training time of out-of-core methods.
- Based on our performance mode, we propose *KARMA*, an out-of-core method for training DNNs with the PyTorch framework [15]. *KARMA* includes a novel approach for interleaving capacity-based layer swapping with redundant recompute to assure peak device occupancy with minimum stalls in the training pipeline. Using *KARMA*, we achieve an average of 1.52x speedup over the state-of-the-art out-of-core and recompute methods.
- We further extend the capability of *KARMA* to support data parallelism in multi-GPU training, i.e. the first out-of-core method to support multi-GPUs, by orchestrating weight updates of swapped-out layers to happen on the CPU side.
- We demonstrate how data parallel *KARMA* outperforms the model plus data parallel approaches in training models with billions of parameters, e.g. Megatron-LM [3] (8.3B parameters) and Turing-NLG [4] (17B parameters) trained on 2,048 GPUs. Finally, we demonstrate cases at which data parallel *KARMA* is the more cost effective methodology when scaling the mini-batch size above the memory limit.

## II. BACKGROUND AND RELATED WORK

Deep neural networks are made up of a network of neurons (nodes) that are organized in layers to form a model. A DNN is trained iteratively by updating the weights of connections between nodes in order to reduce the prediction error of labeled datasets. A DNN is trained on a dataset of samples to calculate the model weights for which the loss function is minimized. DNNs are trained in two stages. First, during the forward phase, the samples pass through the entire network. Next, in the subsequent backward phase (back propagation), the gradients are computed and weights are updated. Commonly, the samples are randomly chosen from the dataset in batches (known as mini-batch). The training process is performed on the batches of samples, in an iterative fashion, by using an optimization algorithm such as the Stochastic Gradient Descent (SGD). Training with the entire set of training samples is typically repeated, in what is called epochs, until the model converges to a desired accuracy.

### A. Related Work

Speeding up DNNs is extensively researched, but few works address the memory barrier in distributed training. We do not consider comparisons to basic virtual memory methods (i.e. CUDA Unified Memory) since several works report they perform less favorably than dedicated methods [9], [10].

1) **Out-of-core (Virtualization):** vDNN++ [10] acts as a runtime memory manager that virtualizes the memory usage of DNNs in order to enable simultaneous training of DNNs with both GPU and CPU memories when the DNN cannot fit into GPU memory. Synchronization of the computation and swap-in/out of data at the end of each layer can cause inefficiencies to some extent in vDNN++. The library ooc\_cuDNN [11] is an extension of Nvidia’s DNN library (cuDNN). It uses an out-of-core approach to apply cuDNN-compatible operators even when a layer exceeds GPU memory capacity. In ooc\_cuDNN, the feature map of a single layer can be divided over any of the three dimensions: batch, channels, and filters. This means that the swapping of tensors in ooc\_cuDNN is limited to the scope of a single layer, i.e., no prefetching is applied.

2) **Gradient Checkpointing (Recompute):** Gradient checkpointing [16], [19], [20] is a method that enables fitting larger models into memory by redundantly recomputing activations from checkpoints in the back propagation (at the cost of increased compute time).

Since gradient checkpointing, by definition, trades performance for memory capacity, it could not be used to improve performance in distributed training (unlike out-of-core, as will be shown later). In addition, gradient checkpointing has a lower bound on the memory consumption for a model of  $N$  layers of  $\mathcal{O}(\sqrt{N})$  (or  $\mathcal{O}(\log_2 N)$  under a special recursive scheme, so may not be used commonly [16]). Hence, it might enable training larger models, yet is still bounded by a memory requirement, unlike out-of-core solutions that in theory have no bounds.

3) **Out-of-core Plus Recompute:** *SuperNeurons*, *Capuchin*, and *Pooch*: *SuperNeurons* [12] uses a simple method to combine swapping and recomputing based on the target layer, e.g. activations of convolution layers are swapped out while batch normalization layers are recomputed. However, this does not take the actual execution time and memory usage during training into account. *Pooch* [13] extends a DL framework to enable out-of-core DNN training by mixing layer swapping and recomputing based on a schedule. *Pooch* decides on the schedule based on the classification of operators in a given model, yet the target layers for swapping or recomputing are determined based on runtime profiling. *Capuchin* [14] uses dynamic tracking of tensor access patterns at runtime. It combines out-of-core with recompute to achieve the same amount of footprint reduction as the sole out-of-core approach but with better performance (7%).

It is important to emphasize that all out-of-core methods do not support multi-GPU, to the authors knowledge. That is since the weight update regime for multi-GPUs conflicts with swapping-out layers to CPU. With the massive increase in the resources required for SoTA models (specially models used in NLP), an out-of-core solution supporting multi-GPU training is beneficial for democratizing SoTA NLP research.

4) **Model Parallelism:** Model parallelism is increasingly gaining traction as a method to avoid the memory capacity limitation by splitting the model over different GPUs [7], [8], [18], [21], [22]. FlexFlow [18] notably enables model parallelism by

TABLE I: Limitations and Restrictions of Related Approaches; Label Explanation: *OOC* = Out-of-core, *RECOMP* = Redundant Recomputation, *MP* = Model Parallelism, *N* = Number of Layers, *P* = Number of Model Parameters, *MN* is Multi-node

Name	Approach	Min.Req. Memory	Universal	Multi-node	Strong Scaling (MN)	Fault Tolerance (MN)	Ref.
vDNN++	OOC	None	✗	✗	N/A	N/A	[10]
ooc_cuDNN	OOC	None	✗	✗	N/A	N/A	[11]
Gradient Checkpoint	RECOMP	$\mathcal{O}(\sqrt{N})$	✓	✓	✗	✓	[16]
SuperNeurons	OOC & RECOMP	$\mathcal{O}(\sqrt{N})$	✗	✗	N/A	N/A	[12]
Pooch	OOC & RECOMP	$\mathcal{O}(\sqrt{N})$	✗	✗	N/A	N/A	[13]
Graph Partitioning	Implicit MP	None	✓	✗	✗	✗	[17]
FlexFlow	Explicit MP	$\mathcal{O}(\sqrt{P})$	✗	✓	✓	✗	[18]
<b>KARMA (This work)</b>	OOC & RECOMP	None	✓	✓	✓	✓	

splitting any of the dimensions for any type of DNNs. Model parallelism approaches are invasive and require alteration of how the model is implemented—on a case by case bases—depending on how the model tensors are split. Furthermore, using model parallelism ties the minimum number of GPUs to be used to the model size. For instance, the Megatron-LM model [3] requires a minimum of 16 GPUs of 16 GiB memory capacity. Even solutions that are highly optimized for reducing the number of GPUs for extremely large models still require tens or hundreds of GPUs [4]. In summary, model parallelism approaches are complex, intrusive, and enforce a non-trivial bound on minimum number of GPUs for large models.

Implicit model parallelism methods partition the dataflow graph over the GPUs [17], [23], [24]. One limitation of this pure DAG approaches is the complexity in scaling to multi-nodes, hence DAG solution so far have been limited to a single node. In addition scheduling DAGs is an NP-hard problem with  $\mathcal{O}(V^2)$  complexity, hence work based on that approach report significant time and resource overhead [17], [24].

### B. Summary of Limitations and Issues in Related Approaches

We highlight the limitations and issues in existing approaches (see Table I for summary):

- All prior out-of-core methods are limited to a single GPU. However, unless out-of-core solutions support data parallel multi-node training, they will not be considered a practical approach for training large models and/or datasets.
- Using recompute with data parallel training always adds an overhead. Recomputing layers is used with some of the out-of-core methods to further relax the memory limitation, yet in most cases they add an overhead. Contrarily, we use recomputing to reduce the runtime by reducing the stalls in the pipeline.
- Both gradient checkpointing and model parallelism enforce a lower bound on required memory, but this can be an issue when the lower bound is itself higher than the memory capacity per device.
- Fault tolerance is a concerning issue with single-GPU out-of-core methods and model parallel methods. On the contrary, out-of-core data parallelism (i.e. our KARMA methodology) could potentially adapt to faults by either relaunching with a smaller worker pool [25] or shrinking the worker pool [26].

## III. KARMA: OUT-OF-CORE DISTRIBUTED TRAINING OF DEEP NEURAL NETWORKS

### A. Overview of KARMA

KARMA enables distributed training of DNNs beyond memory capacity. As shown in the overview in Figure 1, KARMA builds a dependency graph of the model and conducts various examinations on the model to extract metadata that would be used in the performance model of device occupancy (❶ and ❷ in the figure). Next, KARMA splits the layers to groups of blocks that optimize for reducing the total runtime. We reduce the total runtime by formulating a two-tier constrained optimization problem that maximizes the device occupancy, which in-turn requires an efficient strategy to reduce the stalls due to data movement to minimum (❸ and ❹). We use a novel scheduling strategy that involves a capacity-based layer swapping policy interleaved with recomputation. Next, KARMA generates an execution plan based on the identified optimal blocking and recompute strategy (❺), and finally replaces the original model code with the new one.

It is important to note that KARMA supports multi-GPU training. In the case of multi-GPU, steps ❹ and ❺ change to introduce a 5-stage pipelined heterogeneous approach for which the orchestration and execution plan is based on pipelining phased gradient exchanges with CPU-side weight update.

### B. Assumptions and Restrictions

The model in this paper can be generalized to discrete accelerators, and is not limited to GPUs. The work in this paper is based on the following assumptions:

- We distinguish two levels of memory: GPU memory is referred to as *near memory* and host memory is *far memory*.
- The interconnect between the two memories is bi-directional (e.g. PCIe or NVLink).
- Recomputation, i.e. redundant computation, can be utilized by the performance model.
- Swapping plus recompute strategy and scheduling is applied on blocks of layers<sup>1</sup>.
- In most cases, the amount of time required to compute a block is less than the time required to swap-in/out this block.
- The following families of models are supported: CNNs, Transformer-based architectures, and fully convolutional (e.g. U-Net).

<sup>1</sup>For the remainder of this paper we use the word *block* to describe a set of consecutive layers that are bundled together when they are computed, swapped, and their weights are being updated

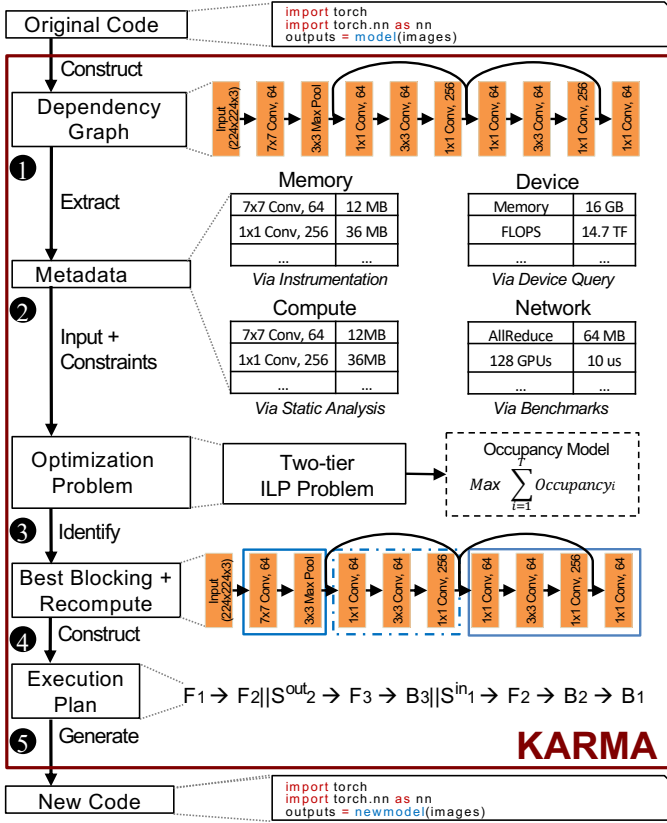


Fig. 1: Overview of KARMA's workflow

### C. Computational Cost of Blocks

We base our performance model and swapping strategy on the compute and memory requirements of blocks. A multitude of optimizations are used in DL frameworks [15]. Recent studies of optimizations in DL frameworks show that those optimizations have minimal effect on the aggregate number of operations when combining layers together [27]. Few exceptions remain, such as fusing convolution layers with average pooling using summed array tables [28]. Therefore, our proxy of the computational cost for each block is the aggregate number of arithmetic operations (i.e. FLOPS) for all layers in the block. We then use the concurrency capability of the target architecture to project the throughput of processing the blocks on the target architecture.

In the following, we list the compute requirements for layers that most commonly appear in DNNs<sup>2</sup>:

1) **Convolution Layer**: For the kernel size  $K \cdot K$  and  $C$  channels, a convolution layer needs  $K \cdot K \cdot C_i$  multiply and add operations. The number of operations for direct convolution of one sample is:

$$|Y_i| \cdot K \cdot K \cdot C_i = W_{i+1} \cdot H_{i+1} \cdot C_{i+1} \cdot K \cdot K \cdot C_i$$

. Whenever other convolution algorithms are used in cuDNN to compute the layer, e.g. GEMM-based, we adjust the number of operations accordingly based on the algorithm type.

<sup>2</sup>Less common layers are outside the scope of this paper. However, our performance model is generic: it allows adding new layers, if required.

2) **ReLU Layer**: The output of a ReLU layer is  $y_i = \max(0, x_i)$ . This requires  $|Y_i|$  comparison operations.

3) **Pooling Layer**: The total number of operations is:

$$|Y_i| \cdot K \cdot K \cdot C_i \cdot c = W_{i+1} \cdot H_{i+1} \cdot C_{i+1} \cdot K \cdot K \cdot C_i$$

where  $c$  is a multiplier adjusted to the number of operations based on the pooling type (i.e. max vs. average)

4) **Batch Normalization Layer**: Computing the mini-batch mean requires  $|B|$  operations and the mini-batch variance requires  $2 \cdot |B|$  operations. The normalization and the scaling/shifting requires  $4 \cdot |X_i|$  and  $2 \cdot |Y_i|$ , respectively. The total number of operations is:  $3 \cdot |B| + 4 \cdot |X_i| + 2 \cdot |Y_i|$ .

5) **Recurrent Neural Network Layer**: Taking LSTM [29] as the most common type of layer in RNN, four tensors that form the output of the layer are combined together as input to form the state of the cell by five operations: the total number of operations is  $20 \cdot |Y_i|$ . It is important to note that we adapt the number of operations to the specific RNN variant we use in the model we test. For instance, in attention models [30] used in machine translation, the decoder RNN additionally receives a weighted sum of encoder hidden states (attention mechanism), rather than just the last hidden state of the encoding stage.

6) **Self-Attention Layer**: Given an input with  $Q$  queries',  $K$  keys', and  $V$  values' matrices, for queries and keys of  $d_k$  dimensions and values of  $d_v$  dimensions and with a dot-product as a compatibility function ( $q.k = \sum_{i=1}^{d_k} q_i k_i$ ) [30], the total operations required are  $4d_k^3 + d_k^2 + 2|d_k|$ , where  $\text{Attention}(Q, K, V) = \text{Softmax}(\frac{QK^T}{\sqrt{d_k}})V$ .

7) **Fully-Connected Layer**: Since  $y_j \leftarrow f(\sum_i (x_i w_{i,j}) + \text{bias}_{i,j})$ ,  $|WT_i| = |X_i| \times |Y_i|$ , and  $|Y_i| = C_i$ , the total number of operations is:  $|WT_i|$ .

8) **Softmax Layer**: Since the softmax layer normalizes its input to a probability distribution, the total number of operations is  $2|X_i|$ .

9) **Other**: We do not list here the number of operations for other layers/operators which we support in our performance model since they can be simply inferred: dropout, tensor rescaling/reshaping, tensor element-wise, and tensor add.

### D. Memory Requirements

Unlike the computational cost, the memory requirement for the block depends significantly on the optimizations used by the memory manager, such as layer fusion, changing memory layouts, and work space optimizations. Hence, simple aggregation of memory requirements per layer to estimate the memory requirements for a block could be highly inaccurate [31]. We use an empirical method that relies on measurements done on a specific target hardware. Note that the empirical results gathered through profiling is done offline. Empirically measuring actual memory used by blocks and layers is not a straightforward task. Most DL frameworks, including PyTorch (which we use in this paper), use a caching memory allocator or manager to speed up memory allocations. As a result, the values shown in CUDA profilers usually don't reflect the true memory usage. In addition, only differentiable training-related variables and tensors are retained per default, i.e. variables

that require gradients will persist. Major DL frameworks often provide low-level APIs to monitor their memory allocator at fine granularity. We conduct a set of experiments to gather information on the memory required per layer, and also fused layers commonly occurring together, using PyTorch’s `memory_stats()` API that monitors the memory occupied by tensors. We use Nvidia’s profiling tools to monitor the memory required for non-model data, e.g. contexts created on the GPU. Finally, after profiling once for each model, we break down and analyze memory use patterns per variable type, i.e. inputs, weights, weight gradients, activations, and activation gradients. This way we can project the memory requirements when we increase the mini-batch size without the need to repeat the offline profiling step.

### E. Proposed Capacity-based Strategy for Swapping Blocks

1) **Occupancy Analysis:** The goal of our model is to assure we maintain the highest possible occupancy<sup>3</sup> of the device. Occupancy in this context factors in the cost of stalling when overlapping layer swapping and prefetching. The occupancy  $\mathbb{O}$  in step  $j$  can be represented, using the device’s busy time  $T_j^{\text{busy}}$  and ideal time  $T_j^{\text{idle}}$ , as follows:

$$\mathbb{O}_j := \text{Occupancy}_j = \frac{T_j^{\text{busy}}}{T_j^{\text{busy}} + T_j^{\text{idle}}} \quad (1)$$

For sake of simplicity, we assume buffers of variable sizes, where a buffer size is set equal to the total size of the data arrays containing one or more layers. Hence, the number of available buffers  $\mathfrak{B}$ , capable of holding data arrays for active layers in GPU memory at the current time step  $j$ , can be a proxy for occupancy at step  $j$ , which means the occupancy, we seek to maximize, can also be formulated as:

$$\mathbb{O}_j \approx \begin{cases} \frac{\mathfrak{B}_j^{\text{avail}}}{\mathfrak{B}_j^{\text{requ}}} \\ 1, & \text{for } \mathfrak{B}_j^{\text{avail}} > \mathfrak{B}_j^{\text{requ}} \end{cases} \quad (2)$$

To get the number of available buffers  $\mathfrak{B}^{\text{avail}}$ , we have to know which buffers are left after processing at the previous step, and which layers have been swapped in during the same time period. So, for  $\mathfrak{B}_1^{\text{avail}} := \{\text{entire GPU memory}\}$  and  $j \in \mathbb{N}$ , we can calculate  $\mathfrak{B}^{\text{avail}}$  in step  $j$  as:

$$\mathfrak{B}_j^{\text{avail}} = \begin{cases} \mathfrak{B}_{j-1}^{\text{avail}} - \sum_{i=1}^{j-1} (\mathfrak{B}_i^{\text{swapped-in}} - \mathfrak{B}_i^{\text{processed}}) \\ 0, & \text{for } \mathfrak{B}_{j-1}^{\text{avail}} \leq \mathfrak{B}_{j-1}^{\text{swapped-in}} - \mathfrak{B}_{j-1}^{\text{processed}} \end{cases} \quad (3)$$

where  $\mathfrak{B}_i^{\text{swapped-in}}$  and  $\mathfrak{B}_i^{\text{processed}}$  are the swapped-in and processed buffers at step  $i$ , respectively. Note that the buffers are released and tagged as available after processing on them has finished. Obviously, if the rate of swap-in grows faster than processing, then the value of  $\mathfrak{B}^{\text{avail}}$  will approach 0. As  $\mathfrak{B}^{\text{avail}}$

keeps decreasing,  $\mathfrak{B}_{j-1}^{\text{avail}}$  could become less than the required  $\mathfrak{B}_{j-1}^{\text{requ}}$ , and hence occupancy  $\mathbb{O}$  would then fall towards 0.

If processing is faster, which is typically the case, the value will always be bounded by the overall block swap-in throughput  $\mathbb{T}^{\text{swap-in}}$  that can be inferred from the hardware parameters:

$$\mathbb{T}^{\text{swap-in}} = \min\{ \mathbb{T}^{\text{FM}}, \mathbb{T}^{\text{NM}}, \mathbb{T}^{\text{IC}} \} \quad (4)$$

Where  $\mathbb{T}^{\text{FM}}$ ,  $\mathbb{T}^{\text{NM}}$ , and  $\mathbb{T}^{\text{IC}}$  are the block-adjusted throughput for the far memory (host memory), near memory (device memory), and inter-connection (PCIe/NVlink), respectively. As discussed previously, if swap-in is faster than processing, the memory consumption will finally reach the GPU memory capacity limit, called  $\mathfrak{C}_{\text{GPU}}$  hereafter, and the capability of swap-in will be affected by the memory space left (since one would have to wait for buffers to clear). If there is not enough memory space for swap-in, the number of buffers swapped in at time step  $j$  will be limited to  $\mathfrak{B}_{j-1}^{\text{avail}}$ .

To control the granularity of the swapping, we divide the layers into disjoint blocks in both the forward and backward propagation phases. The start time of each block  $b$  is the start time of processing or swap-in, and for simplicity we assume they are equivalent. The processing time  $T_{\text{proc}}(b)$  defines that both swap-in and processing finished. Hence, blocks swapped-in depend on the available buffers from the previous time step:

$$\mathfrak{B}_j^{\text{swapped-in}} = \min\{ \mathbb{T}^{\text{swap-in}} \cdot T_{\text{proc}}(b), \mathfrak{B}_{j-1}^{\text{avail}} \} \quad (5)$$

Accordingly, for active blocks  $b$  in time step  $j$ , the occupancy can be approximated via:

$$\mathbb{O}_j \approx \max \left\{ \frac{\mathfrak{B}_j^{\text{avail}}}{\sum_b (\mathfrak{B}_j^{\text{processed}}(b) + \mathbb{T}^{\text{swap-in}} \cdot T_{\text{proc}}(b))}, 1 \right\}, \quad (6)$$

where  $\mathfrak{B}_j^{\text{avail}}$  is defined in Equation 3.

2) **Occupancy in Capacity-based Strategy:** In this section we refine the occupancy analysis of the previous section to accommodate our capacity-based strategy. First we take a simple example from the related work, e.g. vDNN [32], and illustrate the shortcomings of their swap-in/out strategy (shown in Figure 2 (a)). During forward propagation, only swap-out happens. The simple eager strategy used is to swap-out whenever one block’s forward step is concluded (including the last block). This can cause performance inefficiency before the backward phase starts. More specifically, the GPU has to wait until the last block is swapped out and then swap it in again before the backward phase can start. During backward propagation, the buffer of block  $b-1$  starts swap-in at the same time as block  $b$  starts processing. Accordingly, block  $b$  will run right after both swap-in and  $b+1$  processing is finished. In this case, different layers have different processing times, while we can assume the attributes related to swap-in/out stay the same. If swap-in of  $b$  takes more time than processing  $b+1$  (which is generally expected), there will be idle times during backward propagation (i.e. stalls), yielding a longer training time.

<sup>3</sup>Occupancy in this context is not to be mixed with the occupancy metric of CUDA, but refers to the relative amount of compute time, see eq. (1).



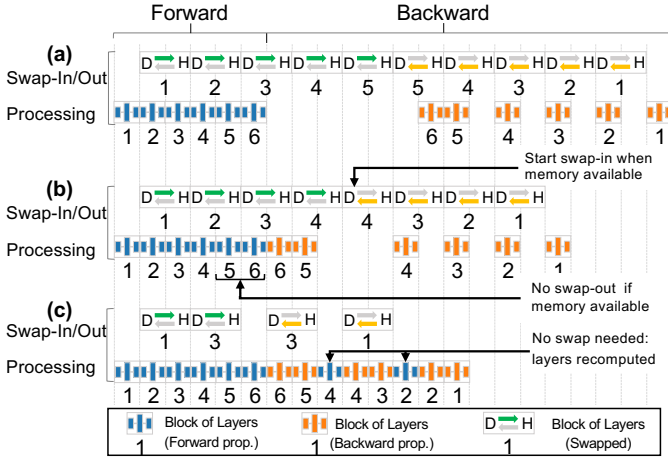


Fig. 2: Different swapping strategies (simplified illustration that assumes the swap-in/out of one block of layers at a time, with swap-in/out taking double the time of the computation). (a) vDNN [32] and ooc\_cuDNN [11] family of solutions may lead to inefficiency when switching from forward to backward propagation, (b) Proposed capacity-based schedule can reduce the idle time by swapping in ahead of time and avoiding swaps when device capacity allows, (c) Improving capacity-based schedule to further reduce the idle time: swap-in ahead of time interleaved with recomputing following layers

In this paper we propose an alternative capacity-based strategy for swapping that we explain in following paragraphs. Since the swap-in is only affected by the memory capacity of the GPU, regardless of which block is being processed, we should keep on swapping in as long as we have enough memory space for the buffer. This allows us to keep as much as possible data available for processing at any step (obviously any wait will cause a drop in the average swap-in throughput).

Figure 2 (b) shows the proposed capacity-based swap strategy. We get the estimate of the memory consumption from the parameters of the layers in the block by using the empirical method discussed in Section III-D. This means even before the forward pass starts, we can know when to stop the swap-out (from block 5 in this example). When the backward phase starts, the required data would still remain in the GPU memory, and therefore the process can start as soon as the forward stage ends. When one block is processed, the data will be swapped out immediately, in order to make new space for data to be swapped in (i.e. prefetched). In this example, block 4 can be swapped-in in a very early stage to avoid some GPU stalling caused by data dependency.

Now that we have a clear strategy for swapping data (i.e. capacity-based swap schedule), we can refine the performance model based on this. Let the number of active blocks that can be kept in GPU memory at time step  $j$  be  $\mathfrak{A}_j^{blks}$ . At the very beginning of the backward stage, the processing of the blocks will not be impacted by the swap-in. If swap-in is fast enough, all the blocks will be swapped in before the processing of the first block. But if swap-in is relatively slow, the processing may finally catch up with swap-in at a certain time step  $\theta$ ,

which satisfies the equation:

$$\sum_{b \in \mathfrak{A}_\theta^{blks}} T_{\text{proc}}(b) < \frac{\mathfrak{B}_{\theta+1}^{\text{swapped-in}}}{\mathbb{T}^{\text{swap-in}}} \quad (7)$$

Once  $\theta$  is reached, the situation becomes the same as discussed in the previous section. In addition, if at time step  $\theta$  the inequality in Eq. 7 is unsatisfied, that would mean that processing cannot catch-up with data transfer: the whole training process can be at 100% device occupancy. Now we can derive the refined occupancy:

$$\mathbb{O}_j = \begin{cases} \mathfrak{B}_j^{\text{avail}} / \sum_{b \in \mathfrak{A}_{blks}} (\mathfrak{B}_j^{\text{processed}}(b) + \mathbb{T}^{\text{swap-in}} \cdot T_{\text{proc}}(b)) \\ 1, \text{ for } T < \theta \text{ where } \theta \text{ occurs when Eq. 7 holds} \end{cases} \quad (8)$$

The occupancy is the objective function to minimize the training iteration total runtime when using our proposed out-of-core strategy. The cost function for compute is calculated analytically as discussed in Section III-C. The swapping time is calculated as  $\mathbb{T}^{\text{swap-in/out}}$  that maximises the occupancy  $\mathbb{O}$  in Eq. 8 for the buffers of each block to be transferred. The same compute and swap models are also used as cost functions for the recompute optimization that is proposed in the next section.

#### F. Interleaving Recompute with the Capacity-based Strategy

The capacity-based strategy discussed in the previous section reduces the swap-in/out overhead by keeping data in near memory as long as there is space, along with a FIFO swapping strategy. However, this could still lead to gaps in the pipeline at which the processor stalls when waiting for a swap-in to finish (e.g. backward phase of blocks 4 to 1 in Figure 2 (b)). The improvement we propose for that method is to interleave the recomputation of the activations of layers in a block with the swap-in of preceding block(s). Those redundantly recomputed layers would have been otherwise swapped-in at the backward stage. The objective of this interleaving is to make sure the computation pipeline remains full and one does not stall due to swapping. The interleaving enables us to overlap compute of the block before the recomputed block with the blocks being swapped-in. As shown in Figure 2 (c), recomputation of block 5 could be overlapped with the swap-in of block 3, and 3 is overlapped with block 1.

**1) Dividing Layers to Blocks and Interleaving Recompute: An Optimization Problem:** In this section we formulate blocking and interleaving as an optimization problem. The execution schedule we define includes a serial sequence of *stages*. Each stage includes a set of blocks, and independent operations on those blocks, i.e. operations on a stage can be overlapped. Deciding on the stages with minimum makespan necessitates the identification of which layers need to be contained in each block, with consideration of the trade off between occupancy and the cost of processing/moving the blocks. In addition, identifying which blocks can be recomputed, rather than swap-in/out to ideally reduce the stalls

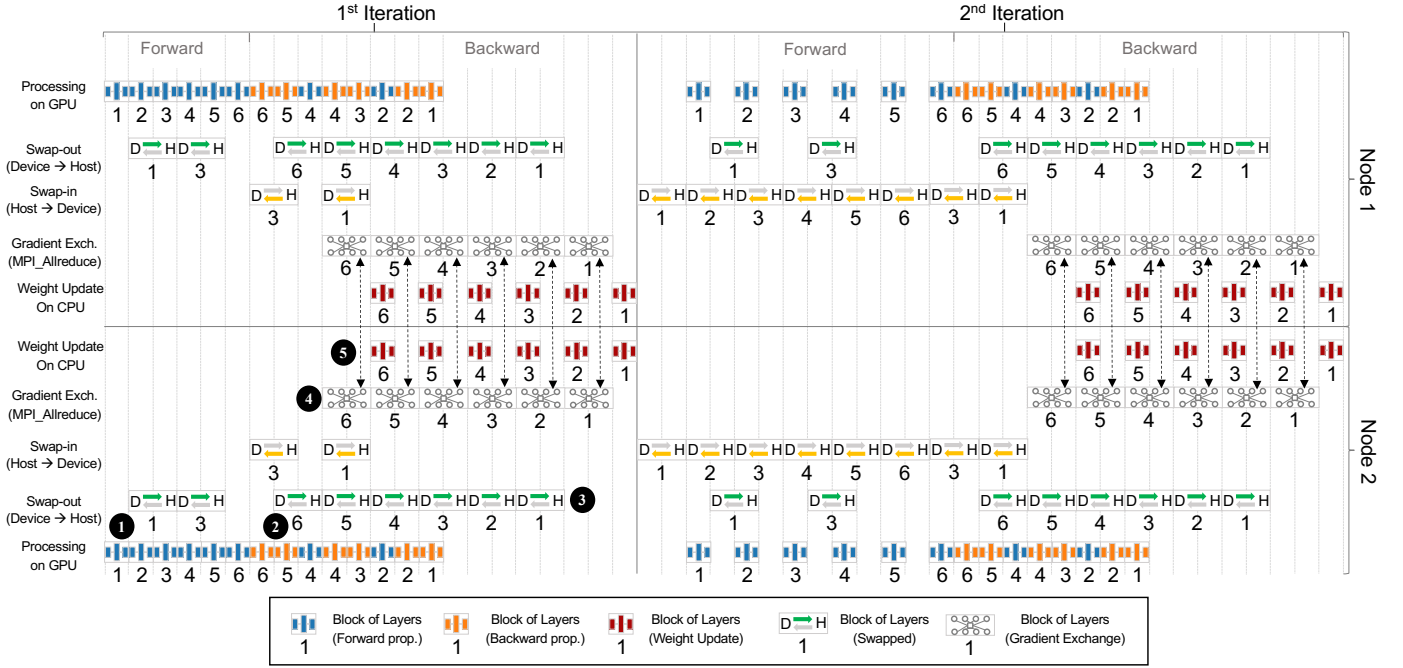


Fig. 3: Using KARMA for data parallel multi-GPU training (an illustrative example of training for two iterations using two nodes). The capacity-based strategy for layer swapping is interleaved with block recomputation (① and ②). We overlap the gradient computation with swap-out of blocks to the CPU (③). We then overlap the gradients exchange (④) in the backward stage with the weights update (⑤) that is done on the CPU side in a heterogeneous manner, before the blocks are swapped back again to the GPU for starting the following iteration. Iterations after the 2<sup>nd</sup> iteration are similar to the 2<sup>nd</sup> iteration

in the pipeline, is another point to consider. This two-stage optimization problem is a variation of the dynamic clustering problem [33] (NP-hard), given that the split that defines the first layer in each block is not independent from the split that defines the last layer in the same block (which itself defines the first layer of the following block). We formulate this two-stage optimization as an Integer Linear Programming (ILP) problem to find the boundaries of the blocks that would be swapped, and from that we extract the stages. Next, we further refine the stages to reduce their makespan by identifying the schedule for interleaving recomputation that would minimize the stalls in the pipeline while satisfying the constraints of device memory capacity and dependencies in the pipeline.

Different alternatives for the optimization were considered, including a multi-objective formulation. However, An important point to note is that: a) pure recompute (i.e. gradient checkpointing [16]) and, b) interleaving recompute with swap-in optimize for different objectives. The aim of recompute in gradient checkpointing methods is to relax the memory limitations (with minimal overhead of recomputation). On the other hand, the goal of recompute in this paper is to reduce the makespan by efficiently reducing the stalls in the pipeline. Therefore, doing a two-step optimization allows us to reduce the search space of interleaving permutations by moving from the granularity of layers to the granularity of blocks. Moreover, we assure that the recomputation interleave optimization is independent of the memory constraint by using a separate optimization step for the recompute interleave and moving the memory capacity constraint to the first optimization problem.

**2) Problem Formulation:** For a layer set  $L := \{L_1, \dots, L_l\}$  split to a set of blocks  $B := \{B_1, \dots, B_b\}$ , finding the execution schedule results in set of stages  $S := \{S_1, \dots, S_s\}$  having occupancies of  $\mathbb{O}(S_i)$ ,  $1 \leq i \leq s$ , where each stage includes operations on the blocks to forward compute, backward compute, swap-in, or swap-out (in short: *fw*, *bw*, *in*, and *out*). We have two optimization problems that are satisfied one after the other. First, grouping the layers into blocks that would yield stages of maximum occupancy. Next, refine the stages by identifying blocks to recompute (rather than swap in/out) to reduce the stalls in the pipeline.

First, the target is to find the  $b$ -partition  $B_1 \cup \dots \cup B_b$  such that: a) the split of the layers over the blocks is pairwise disjoint and complete, b) operations in a stage are independent and can be overlapped, c) no stage should exceed the device memory capacity, and d) the occupancy of each of the stages  $\mathbb{O}(S_i)$  is maximized. Note that for all  $1 \leq i \leq s$ ,  $\mathbb{O}(S_j)$  is a dynamic value, i.e., the value is calculated depending on both the make up of the blocks in  $S_i$  and the efficiency of the overlap of the operations in the stage. Next, the stages  $S$  are refined to new stages  $S'$  such that a block is recomputed if the recompute time until the next checkpoint is less than the swap-in time of the previous block in the path to the checkpoint.

The canonical form of the problem is shown in Figure 4. Constraints ensure the feasibility. Constraints 9.1, 9.2, and 9.3 ensure that the dependencies in the model are not violated. Constraint 9.4 enforces the memory capacity limit, and constraint 10.1 marks blocks for recompute rather than swapping, if that reduces the stalls in the execution pipeline.

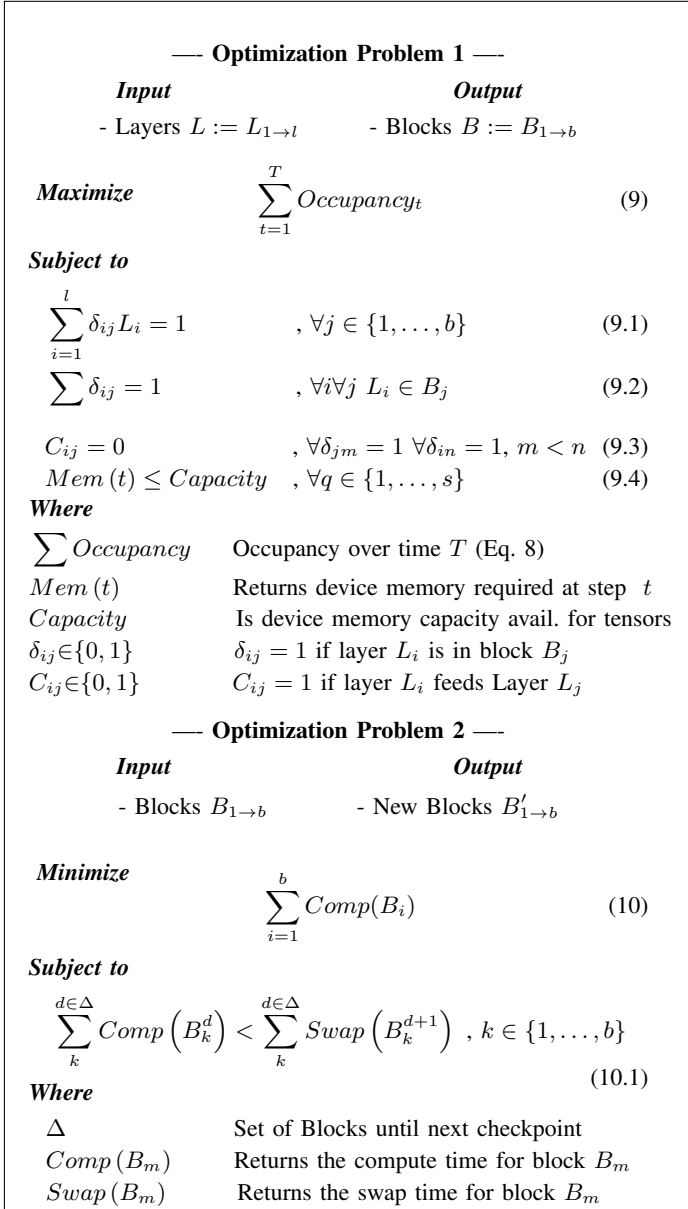


Fig. 4: Block generation formulated as an optimization problem. A schedule of stages defines both the blocks and the sequence of swaps and recomputations to execute. Each stage includes an overlap of independent operations on different blocks in the stage. Output of optimization problem 1 is input of optimization problem 2

3) **Execution Plan Generation:** The best feasible solution to the two-step optimization problem in Figure 4 is used to generate the execution plan. Algorithm 1 shows how we generate the schedule of the stages to execute. The following is an example of the execution plan for the illustrative model in Figure 2 (c). For a layer  $l$ ,  $F_l$  denotes forward computation,  $B_l$  denotes backward computation,  $S_l^{in}$  denotes swap-in, and  $S_l^{out}$  denotes swap-out. The “ $\rightarrow$ ” symbol denotes the start of the next stage, and “ $||$ ” denotes operations done in parallel:  $F_1 \rightarrow F_2 || S_1^{out} \rightarrow F_3 \rightarrow F_4 || S_3^{out} \rightarrow F_5 \rightarrow F_6 \rightarrow B_6 || S_3^{in} \rightarrow B_5 \rightarrow F_4 \rightarrow B_4 || S_1^{in} \rightarrow B_3 \rightarrow F_2 \rightarrow B_2 \rightarrow B_1$

#### Algorithm 1 Schedule Generation of Stages

---

**Require:** Layers:  $L$   
**Ensure:** Stages:  $S' = (*_n \rightarrow \dots \rightarrow *_m), * \in \{F, B, S^{in}, S^{out}\}$

```

1:  $S' \leftarrow 0, TStep \leftarrow 0$  ▷ Initialization
2:  $B \leftarrow Opt1()$  ▷ Blocks defined by Opt1 in Figure 4
3: while  $B \neq \emptyset$  do
4:    $B' \leftarrow getNext(TStep)$ 
5:    $newStage \leftarrow B'$ 
6:    $S' \leftarrow add(newStage)$ 
7:    $B \leftarrow B - B'$ 
8:    $TStep \leftarrow TStep + 1$ 
9: end while
10:  $B' \leftarrow Opt2()$  ▷ Blocks inc. recomp. defined by Opt2 in Figure 4
11:  $S' \leftarrow S$ 
12: for  $b \in B'$  do
13:   if  $recomp(b) = TRUE$  then
14:      $s \leftarrow getStage(b^{out})$ 
15:      $S' \leftarrow remove(b^{out})$ 
16:      $S' \leftarrow advance(b^{out})$  ▷ Advance subsequent swap-out perpetually
17:      $s \leftarrow getStage(b^{in})$ 
18:      $S' \leftarrow remove(b^{in})$ 
19:      $S' \leftarrow advance(b^{in})$  ▷ Advance subsequent swap-ins perpetually
20:      $newStage \leftarrow b^{fw}$ 
21:      $S' \leftarrow insert(newStage)$  ▷ Insert recomputed block
22:   end if
23: end for

```

---

4) **Support of Non-linear Models in KARMA:** KARMA supports non-linear models at which there are connections between non-consecutive layers: residual networks (e.g. ResNet), Transformer-family (e.g. Turing-NLG), and fully convolutional (e.g. U-Net). We inspected the schedules generated by KARMA for different models. For models with affine residual connections, the objective function of the first optimization problem geared the ILP solver towards picking execution plans at which all connections to the layers in a block come from the immediate previous block. Candidate execution plans that have connections that jump over a block would cause a delay in the swap-out and hence be non-optimal solutions.

There are models with non-affine connections, e.g. U-Net has connections from layers in the contracting path to layers in the expansive path. Inspecting the execution plan revealed that the second optimization problem geared the ILP solver towards picking execution plans that changes blocks in the contracting path to recompute, when a layer in the block has an outgoing connection to a layer in the expansive path. Candidate execution plans that would not recompute would be non-optimal solutions since the swapped out blocks in the contracting path would have to be swapped-in prematurely, i.e. before the backward pass.

#### G. Data Parallel KARMA: An Alternative to Hybrid Data/Model Parallelism

The rapid increase in model and dataset sizes makes it imperative for KARMA, and out-of-core solutions in general, to support multi-GPU and multi-node training. However, distributed training is a challenge for all out-of-core solutions, as made evident by the absence of multi-GPU support in all of the previous out-of-core solutions. In single GPU training, the weight update is combined with the backward phase. In typical multi-node or multi-GPU training, where the samples



are divided among the GPUs (i.e. data parallelism), the weight update requires a separate step. This separate weight update step follows after computing the gradients and exchanging them among nodes (or GPUs). However, in out-of-core methods, the layers (including their weights) do not entirely reside on the GPU after the end of the backward phase. Since the layers would not entirely fit on the device memory, a trivial workaround is to move the layers entirely to the CPU after the backward phase to apply the weight update there, but this yields an unacceptable performance penalty.

Our alternative solution for enabling multi-GPU training requires the expansion of the pipeline of the single GPU (Figure 2) from a 2-stage to a 5-stage pipeline. As shown in Figure 3, the three extra stages in the pipeline are as follows: *First* (3), each block is swapped out to the host after computing the gradients for the layers in the block. Note that inter-connect (either PCIe and NVLink) is bidirectional, and therefore swapping out blocks can be overlapped with the swap-in of early blocks in the model to minimize the and stalls introduced by this pipeline stage. *Second* (4), rather than exchanging the gradients all at once, we do the AllReduce exchange of the gradients in phases, i.e. finished blocks from the end of the model do the exchange for their gradients without waiting for the other unfinished blocks. The breakdown of the gradient exchange was recently explored as a way to overlap communication and backward computation in distributed synchronous SGD algorithms [34]–[36]. In this context, however, we do the exchange to the CPU (rather than GPU) since the blocks were swapped out to the CPU before the exchange. In this work we specifically adopt the layer grouping gradient exchange model by Shi et al. [36], since it requires only minimal modification to be applied to our approach of grouping layers in blocks. *Third* (5), after the gradients of a block are exchanged, the weights are updated on the CPU. The computational cost for it is larger than on GPU (plus it requires implementing the CPU side update). However, this heterogeneous approach allows us to have a better overlap with negligible overhead. Note that this also alters the stages starting from the second iteration (as Figure 3 shows).

#### H. Integration of KARMA in a Deep Learning Framework

The model is interpreted during training, since we implement KARMA in PyTorch. For frameworks that use a define-and-run scheme (e.g. TensorFlow), the execution schedule can be encoded statically as a new computational graph, but this is outside of the scope of our work. We extract the forward and backward phases, then after offline profiling we construct and solve the two-stage ILP optimization problem using the MIDACO solver [37], [38]. KARMA then composes the execution schedule from the solution (i.e. stages) and generates a new training script. In the new training script, we use CUDA Unified Memory (UM) technology in addition to a prefetcher. In our implementation, we used `cudaMemPrefetchAsync()` as a data transfer method. Even though we don’t swap data out explicitly, this method allows us to implicitly apply the capacity-based strategy described in Section III-E. Since we

TABLE II: Environment Information (ABCI Supercomputer)

Compute nodes (w/ GPU)	1,088 (total 4,352 GPUs)		
GPU	Nvidia Tesla V100 (SMX2)	x4	(16 GiB)
CPU	Intel(R) Xeon(R) Gold 6148	x2	(32 GiB×6)
CPU-GPU Interconnect	PCI-Express Gen3 x16		(16 GB/s)
GPU-GPU Interconnect	NVLink		(50 GB/s)
System Interconnect	100 Gbps EDR InfiniBand	x2	(12.5 GB/s)
CUDA	v10.1		
cuDNN	v7.6.1		
PyTorch	v1.2		

TABLE III: Overview of Models and Datasets Used in Experiments

Model	Dataset	# Samples	Parameters	Layers
ResNet-50 [2]	ImageNet [6]	1,280,000	> 25M	50
VGG16 [39]			> 169M	38
ResNet-200 [2]			> 64M	200
WRN-28-10 [40]	CIFAR-10 [41]	60,000	> 36M	28
ResNet-1001 [2]			> 10M	1001
U-Net [42]	ssTEM [43]	30	> 31M	27
Megatron-LM [3]	OpenWT [44]	7,200,000	> 8.3B	72
Turing-NLG [4]			> 17B	78

are trying to overlap data transfer as much as possible, we synchronize before the prefetch to make sure the prefetch will not start too early. Note that PyTorch typically calls the cuDNN library for processing only after all buffers for the layer have been prepared (i.e. swapped in) during backward propagation. We also synchronize after the prefetch to make sure the data is ready to be processed, or we would risk a significant penalty from page faulting.

In an ideal prefetch, the prefetch instruction for block  $b$  will be launched to the CUDA stream of the block at the same time that the backward pass on the layers in block  $b + 1$  start running. In this case, although the backward pass of block  $b$  will not even be launched before the data is prefetched properly, it will start much later, so the performance will not be affected by the synchronization.

For multi-node KARMA, we initially relied on Nvidia’s NCCL communication library to implement the phased gradient exchanges. Yet, due to NCCL’s stability issues that appeared at scale ( $> 1,000$  GPUs), we instead used the MPI back-end of the PyTorch communicator (`torch.distributed`). Finally, we implemented a stand-alone direct CPU kernel to update the weights of individual blocks.

## IV. EVALUATION

### A. Hardware Platform, Models, and Datasets

The following experiments are all performed on ABCI supercomputer: 8<sup>th</sup> in the Top500 list (November 2019). The specifications are shown in Table II.

Table III lists the models and datasets we use in our experiments and summarizes the relevant features. We use the same pre-processing configurations and hyper-parameter values reported by the cited models, unless mentioned otherwise. The parallel version of MIDACO library [37], [38] we use to solve the ILP problems converged in under four minutes for all of our inputs, and is thus negligible.

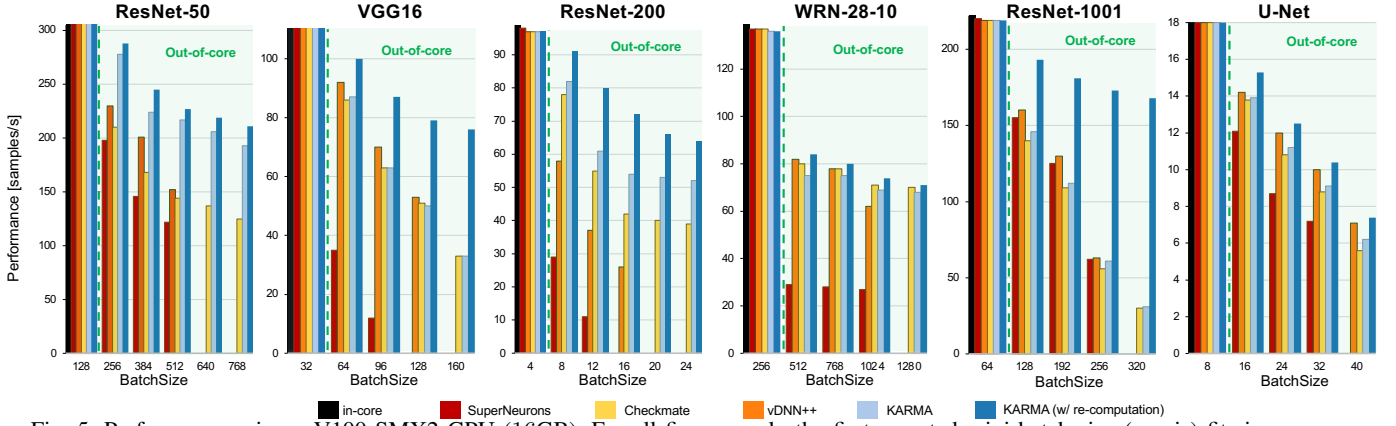


Fig. 5: Performance using a V100 SMX2 GPU (16GB). For all figures, only the first reported mini-batch size (x-axis) fits in memory

### B. KARMA for Single GPU

1) **Performance:** Figure 5 shows the training performance (samples/second) of different batch sizes<sup>4</sup>, for different models. As the batch size grows, the out-of-core effects start to appear. The performance begins to drop after the memory footprint exceeds the GPU memory capacity (starting from the second data point on each x-axis). As discussed earlier, the data movement time should typically be larger than compute time when out-of-core is required, yet the performance does not drop suddenly. Due to our capacity-based strategy, there is a stage where utilization can still be at 100% in the beginning of the backward phase. However, as the input size grows, that duration will become shorter. Finally, the performance converges to be limited by the data swapping throughput.

In addition to out-of-core methods, we compare with two recompute methods: Checkmate [20] which reports state-of-the-art performance and formulates the problem as a constrained optimization problem and solves it using ILP, and b) SuperNeuron [12] (which mixes recompute with out-of-core). KARMA’s capacity-based strategy interleaved with recompute outperforms both in all tested models.

2) **Efficiency:** Figure 6 gives insight into the effectiveness of combining the capacity-based method with recomputing. The large spikes appearing towards the end are stalls resulting from waiting for the swap-in of the convolution layers which appear early in the model. SuperNeurons’s stalls are spread out across the layers due to the inaccuracy of the static cost functions and the designation of swapping vs. recompute based on the layer type, and not a cost function. KARMA and KARMA w/recompute, and vDNN++ exhibit more spread out stalls. However, vDNN++ suffers from an early large spike. KARMA’s capacity-based approach avoids the swapping of late layers in the model, which reduces stalls at the start of the backward phase. Furthermore, vDNN++ includes a significant number of smaller spikes just before the large spikes. KARMA w/recompute is more flat between the large spikes since the interleaved recomputation fills the gaps in the pipeline and reduces the stalls to only a few unavoidable large spikes.

<sup>4</sup>In this paper, *batch size* is synonymous to *mini-batch size*

TABLE IV: Data Parallel KARMA Configurations and Performance for Megatron-LM; Label Explanation:  $H$  = Hidden Size,  $A$  = Attention Heads,  $L$  = Layers,  $P$  = Parameters,  $MP$  = Model Parallel,  $DP$  = Data Parallel,  $OCC$  = Out-of-core (KARMA),  $Perf$  = Iter./sec, and  $PPL$  = Zero-shot Perplexity for OpenWebText [44]

H	A	L	P	Megatron-LM ( $\ddagger$ Num. GPUs)			DP KARMA		
				$MP_{\ddagger}^{\dagger}$	$MP+DP_{\ddagger}^{\dagger}$	Perf	PPL	GPUs	PPL
1152	12	18	0.7B	1	64	5.8	13.66	32	2.2
1536	16	40	1.2B	2	128	1.6	10.47	64	0.73
1920	20	54	2.5B	4	256	2.9	8.21	128	1.94
2304	24	64	4.2B	8	512	5.0	N/A	256	3.11
3072	32	72	8.3B	16	1024	8.4	N/A	512	6.3

Figure 7 shows the best blocking found by KARMA for ResNet-50 on a V100 GPUs. The blocking of layers by KARMA results in a well-balanced overlap of data movement and workload that would reduce the stalling to minimum. Compared to other methods, the execution plan resulting from this blocking reduces the stalling by 43% and 37% over SuperNeurons and vDNN++, respectively.

### C. Data Parallel KARMA (Multi-GPU)

In this section we discuss the results of data parallel KARMA. Table IV shows different configurations tested for the state-of-the-art Megatron-LM model (based on GPT-2 [45]). Megatron-LM configurations generate large models that do not fit in device memory and therefore it was previously implemented using a model/data-parallel hybrid [3]. This can be a major limitation for practitioners. For instance, the 8.3 billion parameter configuration requires 8 GPUs with 32 GiB memory each for 8-way model parallelism which is prohibitive. We use data parallel KARMA to run Megatron-LM entirely using data parallelism and avoid the complexities of model parallelism. The performance is listed in Table IV. The table also shows comparable perplexity in comparison with Megatron-LM on the OpenWebText dataset [44] (excluding the last two configurations that are not feasible to train to completion, given the cost of training).

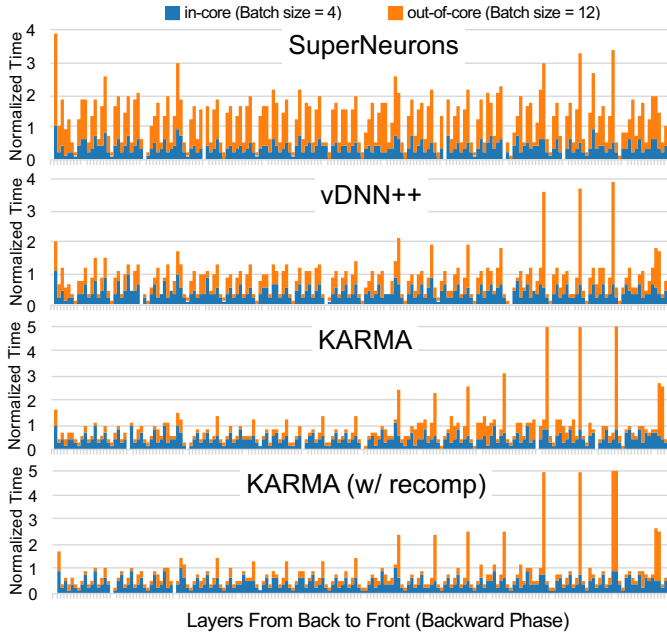


Fig. 6: Normalized runtime in the backward phase of ResNet-200: out-of-core run stacked on an in-core run. The orange bars include the runtime normalized to the same batch size in addition to all the stalls from layer swapping and recompute pipeline

We further conduct an experiment at which we use the same number of GPUs (parity comparison<sup>5</sup>) for the original implementation vs. data parallel KARMA (Figure 8). To get a fair comparison, we also compare to an optimized version of the original implementation for which we added the phased gradient exchange. Surprisingly, in the parity comparison, the pure data parallel KARMA outperforms the model-/data-parallel hybrid on 2,048 GPUs. Upon inspection it became clear that increasing the numbers of GPUs also increases the communication cost for the original version. Note that KARMA has fewer iterations (i.e. communication rounds) since it has a larger mini-batch size. Figure 8 also shows results for Turing-NLG [4], a 17B parameters model with state-of-the-art results in NLP. The Turing-NLG model has 78 Transformer layers with a hidden size of 4256 and 28 attention heads. Turing-NLG is implemented using ZeRO, a memory optimizer that reduces the memory footprint by splitting the model parameters, gradients, and optimizer states among GPUs. Despite the reduction in the memory footprint, large models, such as Turing-NLG, still require a hybrid of model and data parallelism. For the same number of GPUs, KARMA shows less performance than ZeRO, which is expected given the aggressive memory optimizations in ZeRO. However, we also tested KARMA on top of ZeRO, i.e. KARMA enabling the use of the same number of GPUs all in data parallel mode, rather than the model/data parallel hybrid of ZeRO. KARMA+ZeRO gives a speedup of  $1.35\times$  over ZeRO for Turing-NLG with up to 2,048 GPUs.

<sup>5</sup>We use the term *parity* to refer to using a number of GPUs with data parallel KARMA that is equal to the number of GPUs used in the hybrid of model and data parallel

TABLE V: Cost/Performance Normalized to  $\$/P$  of First Row. Number of Samples/GPU Fixed for Data Parallel (at Max of Memory Capacity). Number of GPUs for Data Parallel KARMA Fixed while Samples/GPU Increase; Label Explanation:  $DP$  = Data Parallel, and  $\$/P$  = Cost/Performance (Number of GPUs/training throughput)

ResNet-50					ResNet-200				
Batch	DP		DP KARMA		Batch	DP		DP KARMA	
	GPUs	$\$/P$	GPUs	$\$/P$		GPUs	$\$/P$	GPUs	$\$/P$
12.8K	100	1	100	1	400	100	1	100	1
25.6K	200	1.040	100	<b>1.026</b>	800K	200	1.088	100	<b>1.06</b>
38.4K	300	1.051	100	<b>1.037</b>	1.2K	300	1.093	100	<b>1.066</b>
51.2K	400	<b>1.066</b>	100	1.378	1.6K	400	<b>1.101</b>	100	1.263
64K	500	<b>1.089</b>	100	1.429	2K	500	<b>1.136</b>	100	1.371
76.8K	600	<b>1.092</b>	100	1.483	2.4K	600	<b>1.171</b>	100	1.412

It is important to mention that limitation of the system scheduler prevented us from running full epochs of Megatron-LM and Turing-NLG at large scale. As a mitigation strategy, we split the epoch into separate runs at which we checkpoint/restart the model state, and add up the individual runtime parts for an entire epoch, ignoring the C/R overhead.

Table V shows the cost/performance of two data parallel models vs. data parallel KARMA (where KARMA is using fewer GPUs). Interestingly, KARMA can be more cost effective than data parallelism when initially increasing the number of GPUs. This is due to the positive effect of the capacity-based strategy: only an initial small drop in performance when using KARMA to increase the mini-batch size. However, data parallelism becomes more cost effective when further increasing the number of GPUs as the slowdown due to out-of-core start to magnify.

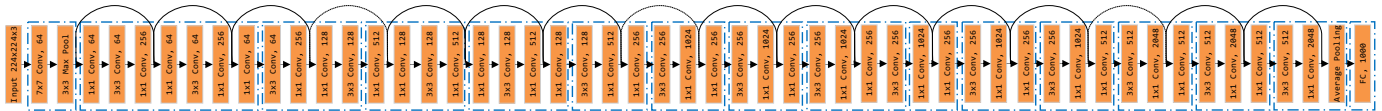
#### D. Accuracy

The proposed out-of-core strategy has no impact on the accuracy of the model since neither the shape nor the hyper-parameters of the model change. Nonetheless, to assure the correctness of our implementation, we ran selected configurations to convergence and compared with the reported accuracy (same number of epochs and hyper-parameters). ResNet-50/ImageNet with 256 samples/GPU on a single GPU using KARMA and 16 GPUs data parallel KARMA achieved 75.9% and 75.8% accuracy, respectively (vs. 75.9% reported top-1 accuracy for single crop on the validation dataset [2]). VGG16/ImageNet with 64 samples/GPU on a single GPU using KARMA and 32 GPU data parallel KARMA achieved 72% and 72%, respectively (vs. 71.9% reported top-1 accuracy for single crop on the validation dataset [39]). We also show perplexity values for the experiments on language models in Table IV

#### E. Discussion of Experimental Outcomes

The results and findings of our experiments, using our novel KARMA methodology, can be summarized as follows:

- KARMA enables a general increase of the batch size, 2x–6x above the memory limit, with only an average performance degradation of 9%–37%, outperforming other approaches.
- Our capacity-based strategy and interleaving recomputation significantly reduces the stalls in the swapping pipeline over state-of-the-art methods.



- F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: <http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>
- [16] T. Chen, B. Xu, C. Zhang, and C. Guestrin, “Training deep nets with sublinear memory cost,” *ArXiv*, vol. abs/1604.06174, 2016.
  - [17] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, “Device placement optimization with reinforcement learning,” in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML’17. JMLR.org, 2017, p. 2430–2439.
  - [18] Z. Jia, M. Zaharia, and A. Aiken, “Beyond Data and Model Parallelism for Deep Neural Networks,” *CoRR*, vol. abs/1807.05358, 2018. [Online]. Available: <http://arxiv.org/abs/1807.05358>
  - [19] M. Kusumoto, T. Inoue, G. Watanabe, T. Akiba, and M. Koyama, “A graph theoretic framework of recomputation algorithms for memory-efficient backpropagation,” in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, 2019, pp. 1161–1170. [Online]. Available: <http://papers.nips.cc/paper/8400-a-graph-theoretic-framework-of-recomputation-algorithms-for-memory-efficient-backpropagation>
  - [20] P. Jain, A. Jain, A. Nrusimha, A. Gholami, P. Abbeel, J. Gonzalez, K. Keutzer, and I. Stoica, “Breaking the memory wall with optimal tensor rematerialization,” in *Proceedings of Machine Learning and Systems 2020*, 2020, pp. 497–511.
  - [21] A. Gholami, A. Azad, P. Jin, K. Keutzer, and A. Buluc, “Integrated model, batch and domain parallelism in training neural networks,” *arXiv preprint arXiv:1712.04432*, 2017.
  - [22] Z. Jia, S. Lin, C. R. Qi, and A. Aiken, “Exploring hidden dimensions in parallelizing convolutional neural networks,” *arXiv preprint arXiv:1802.04924*, 2018.
  - [23] M. Wang, C.-c. Huang, and J. Li, “Supporting very large models using automatic dataflow graph partitioning,” in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–17.
  - [24] R. Mayer, C. Mayer, and L. Laich, “The tensorflow partitioning and scheduling problem: It’s the critical path!” in *Proceedings of the 1st Workshop on Distributed Infrastructures for Deep Learning*, ser. DIDL’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1–6. [Online]. Available: <https://doi.org/10.1145/3154842.3154843>
  - [25] A. Qiao, B. Aragam, B. Zhang, and E. Xing, “Fault tolerance in iterative-convergent machine learning,” in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. Long Beach, California, USA: PMLR, 09–15 Jun 2019, pp. 5220–5230.
  - [26] V. Amatyia, A. Vishnu, C. Siegel, and J. Daily, “What does fault tolerant deep learning need from mpi?” in *Proceedings of the 24th European MPI Users’ Group Meeting*, ser. EuroMPI’17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: <https://doi.org/10.1145/3127024.3127037>
  - [27] S. Mittal and S. Vaishay, “A survey of techniques for optimizing deep learning on gpus,” *J. Syst. Archit.*, vol. 99, 2019. [Online]. Available: <https://doi.org/10.1016/j.sysarc.2019.101635>
  - [28] A. Kasagi, T. Tabaru, and H. Tamura, “Fast algorithm using summed area tables with unified layer performing convolution and average pooling,” in *2017 IEEE 27th International Workshop on Machine Learning for Signal Processing (MLSP)*, 2017, pp. 1–6.
  - [29] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Comput.*, vol. 9, no. 8, p. 1735–1780, Nov. 1997. [Online]. Available: <https://doi.org/10.1162/neco.1997.9.8.1735>
  - [30] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. [Online]. Available: <http://arxiv.org/abs/1409.0473>
  - [31] C. Li, A. Dakkak, J. Xiong, and W. mei W. Hwu, “Benanza: Automatic  $\mu$ benchmark generation to compute “lower-bound” latency and inform optimizations of deep learning models on gpus,” *ArXiv*, vol. abs/1911.06922, 2019.
  - [32] 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 *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2016, p. 18.
  - [33] H. Masoud, S. Jalili, and S. M. H. Hasheminejad, “Dynamic clustering using combinatorial particle swarm optimization,” *Applied Intelligence*, vol. 38, 04 2013.
  - [34] M. Yamazaki, A. Kasagi, A. Tabuchi, T. Honda, M. Miwa, N. Fukumoto, T. Tabaru, A. Ike, and K. Nakashima, “Yet another accelerated SGD: Resnet-50 training on imagenet in 74.7 seconds,” *ArXiv*, vol. abs/1903.12650, 2019.
  - [35] N. Dryden, N. Maruyama, T. Moon, T. Benson, A. Yoo, M. Snir, and B. Van Essen, “Aluminum: An asynchronous, gpu-aware communication library optimized for large-scale training of deep neural networks on hpc systems,” in *2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC)*, Nov 2018, pp. 1–13.
  - [36] S. Shi and X. Chu, “Mg-wfbb: Efficient data communication for distributed synchronous SGD algorithms,” *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pp. 172–180, 2018.
  - [37] M. Schlüter, J. A. Egea, and J. R. Banga, “Extended ant colony optimization for non-convex mixed integer nonlinear programming,” *Computers & Operations Research*, vol. 36, no. 7, pp. 2217–2229, 2009.
  - [38] M. Schlueter, “Midaco 6.0: Mixed integer distributed ant colony optimization,” <http://www.midaco-solver.com>, 2020, accessed 03-15-2020.
  - [39] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. [Online]. Available: <http://arxiv.org/abs/1409.1556>
  - [40] S. Zagoruyko and N. Komodakis, “Wide residual networks,” in *Proceedings of the British Machine Vision Conference (BMVC)*, September 2016, pp. 87.1–87.12.
  - [41] A. Krizhevsky and G. Hinton, “Learning multiple layers of features from tiny images,” *Master’s thesis, Department of Computer Science, University of Toronto*, 2009.
  - [42] O. Ronneberger, P. Fischer, and T. Brox, “U-net: Convolutional networks for biomedical image segmentation,” in *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, Eds. Cham: Springer International Publishing, 2015, pp. 234–241.
  - [43] I. Arganda-Carreras, S. C. Turaga, D. R. Berger, D. Cireşan, A. Giusti, L. M. Gambardella, J. Schmidhuber, D. Laptev, S. Dwivedi, J. M. Buhmann, T. Liu, M. Seyedhosseini, T. Tasdizen, L. Kamensky, R. Burget, V. Uher, X. Tan, C. Sun, T. D. Pham, E. Bas, M. G. Uzunbas, A. Cardona, J. Schindelin, and H. S. Seung, ““serial section transmission electron microscopy (sstem) dataset”,” [http://brainiac2.mit.edu/isbi\\_challenge/](http://brainiac2.mit.edu/isbi_challenge/), retrieved 24 May 2020.
  - [44] J. Peterson, S. Meylan, and D. Bourgin, ““openwebtext dataset”,” <https://github.com/jcpeterson/openwebtext>, retrieved 22 February 2020.
  - [45] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language models are unsupervised multitask learners,” *OpenAI Technical Report*, 2019.