SwapAdvisor: Push Deep Learning Beyond the GPU Memory Limit via Smart Swapping

Chien-Chin Huang
New York University

Gu Jin
New York University

Jinyang Li
New York University

Abstract

It is known that deeper and wider neural networks can achieve better accuracy. But it is difficult to continue the trend to increase model size due to limited GPU memory. One promising solution is to support swapping between GPU and CPU memory. However, existing work on swapping only handle certain models and do not achieve satisfactory performance.

Deep learning computation is commonly expressed as a dataflow graph which can be analyzed to improve swapping. We propose SwapAdvisor, which performs joint optimization along 3 dimensions based on a given dataflow graph: operator scheduling, memory allocation, and swap decisions. SwapAdvisor explores the vast search space using a custom-designed genetic algorithm. Evaluations using a variety of large models show that SwapAdvisor can train models up to 12 times the GPU memory limit while achieving 53-99% of the throughput of a hypothetical baseline with infinite GPU memory.

ACM Reference Format:
Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. SwapAdvisor: Push Deep Learning Beyond the GPU Memory Limit via Smart Swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3373376.3378530

1 Introduction

The deep learning community has been using larger deep neural network (DNN) models to achieve higher accuracy on more complex tasks over the past few years [16, 47, 55, 58, 60]. Empirical evidence shows that, since the 80s, the number of parameters in the state-of-the-art neural network has doubled roughly every 2.4 years [10], enabled by hardware improvements and the availability of large datasets. However, the size of a DNN model that can be explored today is constrained by the limited GPU memory (e.g. 16GB for NVIDIA's V100 GPU).

Existing work have sought to reduce memory consumption by using lower-precision floating points [12, 24], or compressing model parameters via quantization and sparsification [3, 9, 13–15, 20]. However, these techniques can affect model accuracy and require heavy hyper-parameter tuning. Some other solutions discard intermediate data and recompute them later when needed [2, 11, 29]. However, they cannot support large models since model parameters cannot be easily recomputed.

A promising approach to address the GPU memory limitation without affecting accuracy is to swap tensor data between GPU and CPU memory during DNN computation [26, 30, 40, 49]. Several technological trends make swapping attractive: 1) CPU memory is much larger and cheaper than GPU memory, 2) modern GPU hardware can effectively overlap communication with computation, 3) communication bandwidth between GPU and CPU is sufficiently good now and will grow significantly with the arrival of PCIe 5.0 [37] and the wide adoption of NVLink [26, 36].

Swapping for DNN computation differs from traditional swapping (between CPU memory and disk) in that the DNN computation structure is usually known prior to execution, e.g. in the form of a dataflow graph. Such knowledge unleashes tremendous opportunity to optimize swapping performance by maximally overlapping computation and communication. Unfortunately, existing work either do not utilize this information (e.g. TensorFlow's swap extension [57]) or only use it in a rudimentary way based on manual heuristics [26, 30, 49]. For example, TFLMS [26] and vDNN [40] swap only activation tensors according to their topological sort order in the graph. SuperNeurons [49] only swaps data for convolution operations. As a result, not only do these work support only limited types of DNNs, but they also fail to achieve the full performance potential of swapping.

In this paper, we propose SwapAdvisor, a general swapping system which can support various kinds of large model training and inference with limited GPU memory. For a given DNN computation, SwapAdvisor plans for what and when to swap *precisely* prior to execution in order to maximize computation and communication overlap.

A dataflow graph alone is not sufficient for such precise planning, which is also dependent on how operators are scheduled to execute and how the memory allocation is done. More importantly, memory allocation and operator

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378530

scheduling also critically affect the best achievable swapping performance. SwapAdvisor uses a custom-designed genetic algorithm to search the space of all memory allocation and operator schedules so that the final swapping plan represents the result of joint optimization over operator scheduling, memory allocation and swapping.

Our evaluation shows that SwapAdvisor can support the training of very large models up to 12× of the GPU memory limit. The throughput achieved by SwapAdvisor is 53%-99% of the hypothetical baseline with infinite GPU memory. SwapAdvisor can also be used for model inference. Inference has a smaller memory footprint than training. However, to save cost, one may have multiple models use a single GPU. In this setup, one can use SwapAdvisor to constrain each model to use only a fraction of the memory as opposed to time share the entire memory across models. Our experiments show that SwapAdvisor achieves up to 4× lower latency compared to the alternative time-sharing approach.

To the best of our knowledge, SwapAdvisor is the first general swapping system to support a variety of large DNN models. Though promising, SwapAdvisor has several limitations (Sec 8). In particular, it requires a static dataflow graph with no control-flow primitives and plans swapping for only a single GPU. Removing these limitations requires further research.

2 Background

DNN training and inference are usually done on a GPU, which is attached to a host CPU via a high-performance bus, e.g., PCIe and NVLink. GPU uses different memory technology with higher bandwidth but limited capacity, e.g. 16GB on the NVIDIA V100. By contrast, it is common for CPUs to be equipped with hundreds of gigabytes of memory. Therefore, it is attractive to swap data between GPU and CPU memory, in order to support training and inference that otherwise would have been impossible given the GPU memory constraint.

Modern DNNs have evolved to consist of up to hundreds of layers, which are usually composed together in a sophisticated non-linear topology. Programming frameworks such as TensorFlow/MXNet express DNN computation as a dataflow graph of tensor operators. DNN's memory consumption falls into 3 categories:

1. Model parameters. In DNN training, parameters are updated at the end of an iteration and used by the next iteration. Parameter tensors are proportional to a DNN model's "depth" (the number of layers) and "width" (the size of a layer). For large models, these dominate the memory use.
2. Intermediate results. These include activation, gradient and error tensors, of which the latter two are only present in training but not in inference.
3. Scratch space. Certain operator's implementation (e.g. convolution) requires scratch space, up to one gigabyte. Scratch space is a small fraction of total memory use.

Existing work use manual heuristics based on the memory usage patterns of different categories. For example, prior work do not swap parameters[1], but only swap activation to the CPU [26, 40]. Without parameter swapping, prior work cannot support DNNs whose parameters do not fit in the GPU memory. Furthermore, designs based on manual heuristics miss opportunities for performance improvements as modern DNN dataflow graphs are too complex for analysis by humans.

In this paper, we propose a general swapping mechanism in which any tensor can be swapped in/out under memory pressure. More importantly, we aim to move away from manual heuristics and to automatically optimize for the best swapping plan given an arbitrarily complex dataflow graph. We focus the discussion on swapping for a single GPU, but our design can be used in a multi-GPU training setup which replicates the model on different GPUs using data parallelism.

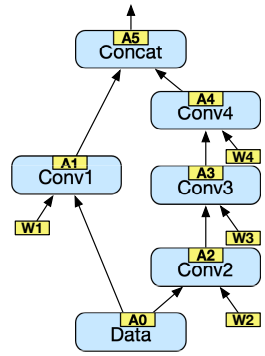
3 Challenges and Our Approach

A good swapping plan should overlap communication and computation as much as possible. The opportunities for overlapping come from swapping out a (temporarily) unused tensor to make room for swapping in an out-of-memory tensor before the latter is required for operator execution. We aim to maximize such overlapping by carefully planning for what and when to swap with the help of the dataflow graph.

Prior work attempt to find a good swapping plan heuristically based on the dataflow graph structure alone [26, 40, 49]. However, this is not enough. In particular, we argue that there are two critical factors affecting swap planning:

- *Memory allocation.* DNN computation uses a wide range of tensor sizes, from a few KB to hundreds of MB. To improve speed and reduce internal fragmentation, frameworks such as MXNet use a memory pool which pre-allocates a number of fixed-size tensor objects in various size classes. As a result, swapping happens not just when the GPU memory is full, but when there is no free object in a particular size class. Therefore, how to configure the memory pool for allocation can critically affect swapping performance.
- *Operator scheduling.* Modern DNNs tend to have complex dataflow graphs as the layers no longer form a chain, but contain branches, joins and unrolled loops. As a result, there are many different potential schedules for executing operators. The order of execution

[1]The only exception being SuperNeuron [49] which swaps convolution but not other types of parameters.



(a) The partial dataflow graph of a model. All tensors are 1MB, except for A2 and A5 which are 2MB. The GPU memory is 10MB.

Time	t1	t2	t3	t4	t5	t6	t7
Computation	Data	Conv1	Conv2	Conv3	Conv4	Concat	
Swapin	W_1	W_2	W_3	W_4	A_1		
Swapout			A_1	A_0	A_3	A_2	
Memory Usage	A_0W_1	$A_0A_1W_1W_2$	$A_0A_1A_2W_2W_3$	$A_0A_2A_3W_3W_4$	$A_1A_2A_3A_4W_4$	$A_1A_2A_4A_5$	

(b) Schedule the left branch first. Allocate five memory objects, each is 2MB.

Time	t1	t2	t3	t4	t5	t6	t7
Computation	Data	Conv2	Conv3	Conv4		Conv1	Concat
Swapin	W_2	W_3	W_4		W_1		
Swapout				A_2	A_2	A_3	A_0
Memory Usage	A_0W_2	$A_0A_2W_2W_3$	$A_0A_2A_3W_3W_4$	$A_0A_2A_3A_4W_4$	$A_0A_2A_3A_4W_1$	$A_0A_1A_3A_4W_1$	$A_0A_1A_4A_5$

(c) Schedule the right branch first. Allocate five memory objects, each is 2MB.

Time	t1	t2	t3	t4	t5	t6	t7
Computation	Data	Conv2	Conv3	Conv4	Conv1	Concat	
Swapin	W_2	W_3	W_4	W_1			
Swapout				A_2	A_2	A_3	
Memory Usage	A_0W_2	$A_0A_2W_2W_3$	$A_0A_2A_3W_3W_4$	$A_0A_2A_3A_4W_1W_4$	$A_0A_1A_2A_3A_4W_1$	$A_0A_1A_3A_4$	

(d) Schedule the right branch first. Allocate nine memoryobjects, one is 2MB and others are 1MB each.

Time	t1	t2	t3	t4	t5	t6	t7
Computation	Data	Conv1	Conv2	Conv3	Conv4		Concat
Swapin	W_1	W_2	W_3	W_4			
Swapout				A_0	A_2	A_2	A_3
Memory Usage	A_0W_1	$A_0A_1W_1W_2$	$A_0A_1A_2W_2W_3$	$A_0A_2A_3W_3W_4$	$A_1A_2A_3A_4W_4$	$A_1A_2A_3A_4$	$A_1A_3A_4A_5$

(e) Schedule the left branch first. Allocate nine memory objects, one is 2MB and others are 1MB each.

Figure 1. Different schedule and memory allocation for a dataflow graph.

can profoundly affects the memory usage and thus the performance of swapping.

Example. We use an example to show how memory allocation and scheduling affect swapping. The example is based on a portion of the dataflow graph of a toy neural network, as illustrated in Figure 1(a). For simplicity, the dataflow graph only shows the forward propagation and omits the backward part. This branching structure is common in modern CNNs [47, 60].

In Figure 1(a), blue rounded rectangles represent operators and small yellow rectangles represent tensors. A tensor is labelled as A_x (activation tensor) or W_x (parameter tensor). Suppose all tensors are 1MB, except for A_2 and A_5, which are 2MB (because A_2, A_5 are used to join two paths). Thus, the memory consumption is 12MB[2]. Suppose the GPU's memory capacity is 10MB, and it takes one unit of time to execute an operator or to transmit 1MB data between GPU and CPU.

A parameter tensor is initially in the CPU memory and must be swapped into GPU memory before being used. We can swap out a parameter tensor without copying it to CPU memory as it is not changed during the forward pass. By contrast, there is no need to swap in an activation tensor (because it's created by operators) but it must be copied to CPU memory upon swap-out because it is needed in the backward pass.

There are many ways to allocate memory and schedule execution for Figure 1(a). We show 2 example schedules: *left-first* executes operators on the left branch first, and *right-first* executes the right branch first. We show 2 example memory allocations: *coarse-grained* allocates 5 memory objects of 2MB each, and *fine-grained* allocates 8 memory objects of 1MB each and 1 object of 2MB. Together, there are 4 combinations of schedule/allocation and we show the best swapping plan under each combination, in Figures 1(b) to (e). As GPU-CPU communication is duplex and concurrent with GPU execution, each table's top 3 rows give the timeline of actions for GPU computation, swap-in (from CPU to GPU), and swap-out (from GPU to CPU) respectively. The last table row shows the tensor objects that are currently resident in the GPU memory.

Let's contrast Figure 1(c) and (d) to see why memory allocation affects swap planning. Both (c) and (d) have the same right-first scheduling. However the total execution time of (c) is one unit time longer than that of (d). Specifically, in Figure 1(c), GPU sits idle in time slot t_5 while operator $Conv_1$ waits for its parameter W_1 to be swapped in. It's not possible to swap in W_1 earlier because the coarse-grained memory pool of five 2MB objects is full at time t_4. One cannot swap out any of the 5 GPU-resident objects earlier: A_3, W_4 and A_4 are input/output tensors needed by the currently running operator $Conv_4$, while A_0 is needed as input for the next operator $Conv_1$. A_2 is being swapped out but the communication takes two units of time due to its larger size. Figure 1 uses a fine-grained memory pool with eight 1MB objects and one 2MB object. As a result, it can swap in W_1 needed by operator $Conv_1$ one unit time earlier, at t_4, because there is still space in the memory pool.

We contrast Figure 1(d) and (e) to see why scheduling affects swap planning. Both (d) and (e) use the same fine-grained memory pool. However, Figure 1(e) takes one unit of time longer than (d) because of its left-first schedule. In Figure 1(e), GPU is idle for the time slot t_6 as operator *Concat* waits for 2MB tensor A_2 to complete swapping in order to make room for its 2MB output A_5. It is not possible to swap out A_2 any time earlier as it is the input of operator $Conv_3$ which executes at time t_4. By contrast, Figure 1(d)'s right-first schedule is able to execute $Conv_3$ earlier at t_3, thereby allowing A_2 to be swapped out earlier.

[2]We do not consider memory reuse in the example as the partial dataflow graph does not include the backward pass which forbids many reuse cases

Our approach. Since memory allocation and operator scheduling critically affect swapping performance, we derive a swapping plan (aka which tensors to swap in/out and when) assuming a given dataflow graph as well as a corresponding memory allocation scheme and an operator schedule (Sec 4). Specifically, the swap plan optimizes computation and communication overlapping by swapping out tensors not needed for the longest time in the future and prefetching previously-swapped out tensor as early as possible.

We search the space of possible memory allocations and operator schedules to find a combination with the best swapping performance. Instead of using manual heuristics to constraint and guide the search, we adopt Genetic Algorithm (GA for short) [5, 8, 18] to search for a good combination of memory allocation and operator scheduling. GA has been used for NP-hard combinatorial problems [28, 39] and scheduling in parallel systems [43]. We chose GA among other search heuristics (e.g. simulated annealing) because it is fast: GA can be parallelized and computed efficiently on multi-core CPUs.

To enable effective exploration of a vast search space, we must be able to quickly evaluate the overall performance (i.e. end-to-end execution time) of a swap plan under any combination of memory allocation/scheduling. We found it too slow to perform the actual execution on real frameworks. Therefore, we estimate the performance by running the swap plan under a dataflow engine simulator. The simulator uses measured computation time for each operator as well as GPU-CPU communication bandwidth so that it can estimate the execution time of a dataflow graph under a given scheduling, memory allocation, and swap plan. The running time of our simulator on a CPU core is orders of magnitude faster than that of actual execution, reducing the search time for a model to less an hour. The simulator enables SwapAdvisor's GA to directly optimize the end-to-end execution time.

4 SwapAdvisor Design

Overview. Figure 2 gives the architecture of SwapAdvisor, which is integrated with an existing DNN framework (MXNet in our implementation). Given a dataflow graph, SwapAdvisor picks any legitimate schedule and memory allocation based on the graph as initial values, and passes them to the swap planner to determine what tensors to swap in/out and when. The result of the swap planner is an augmented dataflow graph which includes extra swap-in and swap-out operators and additional control flow edges. The additional edges are there to ensure the final execution order adheres to the given schedule and the planner's timing of swaps.

For optimization, the augmented graph is passed to SwapAdvisor's dataflow simulator to estimate the overall execution time. SwapAdvisor's GA-based search measures the performance of many memory allocation/schedule combinations, and proposes new allocation/schedule candidates for the swap planner. Once a swap plan has been sufficiently optimized, the final augmented dataflow graph is given to the framework for actual execution.

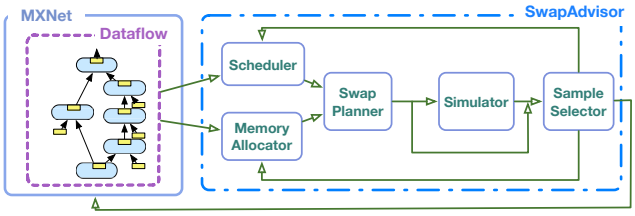


Figure 2. System overview of SwapAdvisor

This section describes swap planner's inputs (Sec 4.1) and explain how the planner maximizes performance given a specific schedule and memory allocation (Sec 4.2). A later section (Sec 5) discusses GA-based optimization.

4.1 Operator schedule and memory allocation

In addition to the dataflow graph, the swap planner takes as input an operator schedule and memory allocation.

Operator schedule. Given an acyclic dataflow graph G, an operator schedule is any topological sort ordering of nodes in G. When using a single GPU, the framework can issue operators to the GPU according to the schedule to keep the GPU busy. Indeed, frameworks such as MXNet commonly perform topological sort to schedule operators.

NVIDIA's recent GPUs support multiple "streams". SwapAdvisor uses 3 streams: one for performing GPU execution, one for swapping out tensors to the CPU, and one for swapping in tensors from the CPU. Since GPU-CPU communication is duplex, all three streams can proceed concurrently when used in this manner. By contrast, if one is to use multiple streams for computation, those streams cannot execute simultaneously if there is not enough GPU compute resource for parallel execution. We have observed no performance benefits in using more than one stream for computation for all the DNN models that we've tested. This observation is also shared by others [40].

Memory allocation. We need to configure the memory pool and specify memory allocation for a given dataflow graph. The memory pool consists of a number of different size-classes each of which is assigned a certain number of fixed-size tensor objects. Given a dataflow graph G which contains the sizes of all input/output tensors needed by each operator, a memory allocation scheme can be defined by specifying two things: 1) the mapping from each tensor size in G to some size class supported by the memory pool. 2) the set of supported size classes as well as the number of tensor objects assigned to each class. As an example, the coarse-grained allocation scheme in Figure 1(b)(c) has only one size-class (2MB) with 5 objects, and maps each 1MB or

2MB tensor to the 2MB size-class. The fine-grained scheme in Figure 1(d)(e) has two size-classes (1MB and 2MB) with 8 and 1 objects respectively, and maps each 1MB tensor to the 1MB size-class and each 2MB tensor to the 2MB size-class.

4.2 Swap planning

The swap planner is given the dataflow graph as well as a valid operator schedule and memory allocation scheme. Its job is to find the swap plan with the best performance under the given schedule/allocation combination. In particular the swap planner decides: 1) which memory-resident tensors to swap out under memory pressure, 2) when to perform swap-in or swap-out.

Which tensors to swap out? At the high level, the swap planner uses Belady's strategy to pick the tensor that will not be needed for the longest time in the future to swap out. Seeing into the future is possible as the planner is given the schedule. Belady's strategy is optimal for cache replacement and also works well in our context as it gives the planner sufficient time to swap the tensor back before its next use. Concretely, the planner scans each operator according to the order in the schedule and keeps track of the set of input/output tensor objects that become resident in memory as a result of executing the sequence of operators. Upon encountering memory pressure when adding a tensor of size s (i.e. there is no free object in size-class of s), the planner chooses the tensor from the same size-class as s to swap out. If there are multiple candidates, the planner chooses one that will be used in the furthest future.

There is a caveat when using Belady's strategy in our setting. Suppose tensor T_i–which is last used by operator op_i–is chosen to make room for tensor T_j, an input tensor to the current operator op_j. Thus, the earliest time T_i can be swapped out is when op_i finishes. If operators op_i and op_j are too close in time in the schedule, there is little time to swap in T_j before it's needed by operator op_j as its memory space is not available until after T_i is swapped out. As a remedy, when choosing a candidate tensor to swap out, the planner picks among those who are most recently used at least a threshold of time ago.

DNN training is iterative, but the swap planner is given the dataflow graph for a single iteration only. At the end of each iteration, all tensors other than the parameter tensors can be discarded. However, to ensure that the same swap plan can be used across many iterations, we must ensure that the set of parameter tensors in the GPU memory at the end of an iteration is the same as in the beginning. To achieve this, we perform a double-pass, i.e. scan the schedule to plan for swapping twice. In the first pass, we assume no parameter tensors are in the GPU memory, and must be swapped in before their first usage. At the end of the first pass, a subset of parameter tensors become residents in the memory, which we refer to as the initial resident parameters. We then do

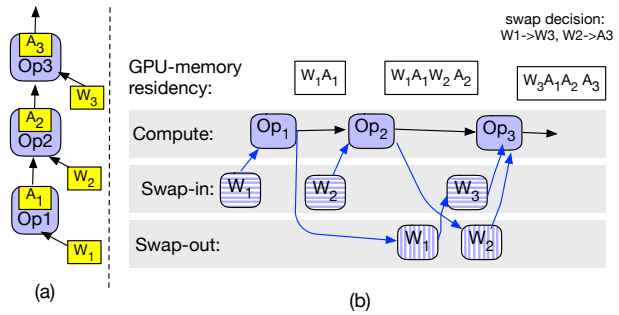


Figure 3. Example swap planning for a simple dataflow graph. All tensors have unit size and total GPU memory is 4 units.

a second pass assuming the initial resident parameters are present in memory in the beginning of the schedule. In the second pass, if there is additional memory pressure that did not happen in the first pass, we remove a parameter tensor from the set of initial residents to resolve the pressure. The final swap plan's initial GPU-resident parameters do not include those removed in the second pass.

When to swap in and out? Our previously discussed selection strategy has determined pairs of tensors to swap-out and swap-in if necessary in order to execute each operator according to the schedule. To maximize computation and communication overlap, we want to complete a pair of swap-out and swap-in as early as possible in order not to block the execution of the corresponding operator in the schedule. However, we must also ensure that the timing of swap-in/out is safe.

We illustrate how the planner controls swap timing using an example 3-node dataflow graph (Figure 3(a)) and the schedule op_1, op_2, op_3 (Figure 3(b)). For simplicity, we assume all tensors in the example are 1 unit in size and the total GPU memory size is 4 units. In order to execute op_1, we must swap in the parameter tensor W_1, thus the planner adds a new dataflow node for swapping in W_1 which is to be run on the GPU stream dedicated for swap-ins. Similarly a swap-in node for W_2 is added. We note that there is sufficient memory to hold the input/output tensors of both op_1 and op_2. However, in order to run op_3, we need room to swap in W_3 and to allocate space for $A3$. The planner chooses W_1 to make room for W_3 (referred to as $W_1 \rightarrow W_3$) and chooses W_2 to make room for A_3 (referred to as $W_2 \rightarrow A_3$). Let's consider the case of $W_1 \rightarrow W_3$ first. The planner adds two dataflow nodes W_1(swap-out) and W_3(swap-in). A control flow edge from W_3(swap-in) to op_3 is added to ensure that operator execution starts only after W_3 is in GPU memory. An edge from W_1(swap-out) to W_3(swap-in) is added to ensure that swap-in starts only when the memory becomes available upon the completion of the corresponding swap-out. Additionally, an edge from op_1 to W_1(swap-out) is included as W_1 cannot be removed from the memory until op_1 has finished using it.

The case of $W_2 \rightarrow A_3$ is similar, except that the planner does not need to add a swap-in node for A_3 because it is created by the operator. The resulting augmented dataflow graph can be passed to the framework's dataflow engine for execution.

5 Optimization via Genetic Algorithm

5.1 Algorithm overview

Genetic algorithm (GA) aims to evolve and improve an entire population of individuals via the nature-inspired mechanisms such as crossover, mutation, and selection [5, 8, 18]. In SwapAdvisor, an individual's chromosome consists of two parts: an operator schedule and a memory allocation. The first generation of individuals are created randomly and the size of the population is decided by a hyper-parameter, N_p.

To create a new generation of individuals, we perform crossover and mutation on the chromosomes of the current generation. A crossover takes a pair of parent chromosomes and produces new individuals by combining the features from parents so that children can inherit "good" characteristics (probabilistically) from the parents. In SwapAdvisor, each crossover generates two new schedules and two memory allocation, thereby resulting in 4 children. We then perform mutation on the children which is essential for GA to escape the local minimum and avoid premature convergence [5, 8, 18]. The resulting mutated children are given to the swap planner to generate the augmented dataflow graph with swapping nodes. We use a custom-built dataflow simulator to execute the augmented graph and obtain the execution time, which is used to measure the quality of an individual. Finally, the GA selects N_p individuals among the current population to survive to the next generation.

Selection methodology. How to select individuals to survive is crucial in GA. If we choose only the best individuals to survive, the population can lose diversity and converges prematurely.

SwapAdvisor's selection takes into account the quality of an individual to determine the probability of its survival. Suppose an individual's execution time is t, we define its normalized execution time as $t_{norm} = (T_{Best} - t)/T_{Best}$, where T_{Best} is the best time among all individuals seen so far. The survival probability of an individual is decided by the $softmax$ function.

$$Prob_i = \frac{e^{t_{norm_i}}}{\sum_{j=1}^{S} e^{t_{norm_j}}} \text{ for } i = 1 \ldots S \quad (1)$$

In the equation, S is the population size before selection (usually larger than N_p). We use the softmax-based selection because our experiments show that it reaches more stable results compared to the popular tournament selection [5, 8, 18].

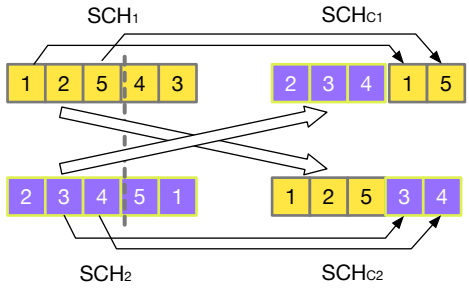


Figure 4. Cross over SCH_1 and SCH_2. There are five nodes in the dataflow graph.

5.2 Creating new schedules

Encoding. As a schedule is a topological ordering of the dataflow graph (G), it is natural to encode a schedule as a list, SCH, where each element in SCH is a node id in G.

Crossover. We borrow the idea from [43] to create two child schedules. by crossing over two parent schedules, SCH_1 and SCH_2. We explain via an example, shown in Figure 4. First, a crossover point, CR, is chosen randomly. In the example, $CR = 3$. To create child SCH_{C1}, the crossover takes a slice of SCH_2 ($SCH_2[1 \ldots CR] = [2, 3, 4]$) to be the first part of SCH_{C1}. The nodes not in the SCH_{C1} are filled in according to their order in SCH_1. In the example, nodes 1 and 5 are not in $SCH_{C1} = [2, 3, 4]$, thus we fill them in the remaining slots of SCH_1, in that order. SCH_{C2} can be created via the same approach but with different parts from SCH_1 and SCH_2, as shown in the bottom of Figure 4. The algorithm guarantees SCH_{C1} and SCH_{C2} are the topological ordering of G [43].

Mutation. A simple way to mutate a schedule is to change a node's position in the list randomly as long as the result remains a topological ordering [43]. However, we have empirically observed that GA works better if we mutate multiple nodes in one mutation (e.g., more than 2x performance improvement for RNNs).

SwapAdvisor's mutation algorithm mimics a dataflow scheduler. It maintains a ready set, containing all the nodes which are ready to run (all the predecessor nodes are executed). The core function of the mutation algorithm is to choose a node from the ready set based on following two conditions. First, with a probability $\mathcal{P}$, a mutation happens. In such a case, the algorithm randomly chooses a node from the ready set. Otherwise, the algorithm selects the node from the ready set which is executed earliest in the original schedule (not mutated). A chosen node is viewed as "executed". The algorithm terminates when all the nodes are "executed". The mutation algorithm generates a new schedule which mostly follows the input schedule but with some nodes scheduled differently.

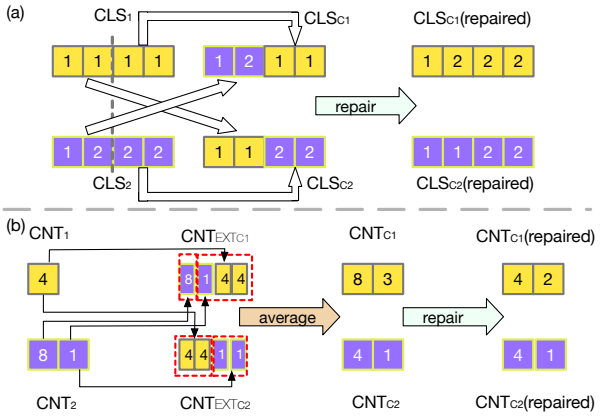


Figure 5. Cross over (CLS_1, CNT_1) and (CLS_2, CNT_2). There is 12MB memory and there are four different sizes of tensors, 1MB, 2MB, 3MB, and 4MB.

5.3 Creating new memory allocation

Encoding. The memory allocation controls how to map each size to a size class and how many objects to assign to each size class. Although it's natural to use a hash map to map tensor sizes to size classes, doing so loses the relative size information between different tensor sizes, making it more difficult to do efficient crossover. We use two lists, *CLS* and *CNT*, to represent the tensor size-class mapping.

Let *TS* be the sorted list of unique tensor sizes observed in the dataflow graph. *CLS* is a list with the same length as *TS*, and the i_{th} item in *CLS* ($CLS[i]$) is a positive integer representing the size-class for tensors with size $TS[i]$. Thus, the number of the size-classes is $Max(CLS)$. *CNT* is a list representing the number of tensor objects allocated for each size-class. Consequently, the length of *CNT* is $Max(CLS)$. As an example, the dataflow graph of Figure 1 has only two different tensor sizes, thus $TS = [1MB, 2MB]$. The coarse-grained allocation with five 2MB objects corresponds to $CLS = [0, 0]$, indicating that both 1MB and 2MB sizes are mapped to the same size-class with id 1 and object size is 2MB. $CNT = [5]$ contains the number of objects assigned to each size class from $id = 0...Max(CLS)$.

The number of potential *CLS* lists is $O(N^N)$ where N is the number of unique tensor sizes [52]. Such a gigantic search space can seriously cripple the performance of GA. We prune the search space by imposing the restriction that *CLS* must be a monotonically increasing sequence and each $CLS[i]$ is equal to or one greater than $CLS[i-1]$. The intuition is that the allocation is more efficient when consecutive sizes are mapped to the same or adjacent size classes. This restriction cuts down the search space from $O(N^N)$ to $O(2^N)$.

Crossover. We first explain how to cross over two *CLS* using an example, shown in Figure 5. There are two *CLS* before crossover (CLS_1 and CLS_2) and four different tensor sizes: 1MB, 2MB, 3MB and 4MB. $CLS_1 = [1, 1, 1, 1]$ means that all four tensor sizes belong to the same size-class, 4MB, and $CNT_1 = [4]$ indicates that there are four 4MB tensor objects allocated. $CLS_2 = [1, 2, 2, 2]$ means that there are two size-classes, 1MB and 4MB. There are eight 1MB tensor objects and one 4MB tensor object as CNT_2 is $[8, 1]$.

The crossover randomly picks a crossover point, *CR* to partition the parent lists, CLS_1 and CLS_2. The first child size-class mapping CLS_{C1} is made by concatenating $CLS_2[1 \ldots CR]$ and $CLS_1[CR+1 \ldots N]$. The second child size-classes mapping CLS_{C2} is made by concatenating $CLS_1[1 \ldots CR]$ and $CLS_2[CR+1 \ldots N]$. Figure 5(a) shows the crossover for *CLS*. In the figure, *CR* is 2 and the results, are CLS_{C1} ($[1, 2, 1, 1]$) and CLS_{C2} ($[1, 1, 2, 2]$). Note that CLS_{C1} is not monotonically increasing. Thus, we repair it to ensure the new size-class mapping is still valid. Our repair increases the elements in a problematic *CLS* by the minimal amount so that the resulting sequence becomes valid. In Figure 5(a), we would increase the 3rd and 4th elements of CLS_{C1} by 1, so that the repaired CLS_{C1} becomes $[1, 2, 2, 2]$.

The same crossover scheme cannot be directly used for *CNT* as its length depends on the content of the corresponding *CLS*. As a result, we extend a *CNT* to an extended CNT_{EXT} which has the same length as *CLS* (and *TS*). *CNT* captures how many tensor objects are allocated for each size-class, while CNT_{EXT} indicates how many tensor objects can be used for each tensor size. For example, in Figure 5, CLS_2 is $[1, 2, 2, 2]$ which means 1MB belongs 1MB size-class and 2MB, 3MB, and 4MB belong to 4MB size-class. CNT_2 is $[8, 1]$ and CNT_{EXT_2} can then be viewed as $[8, 1, 1, 1]$. CNT_{EXT_1} is $[4, 4, 4, 4]$. We apply the same technique to cross over the extended *CNT*. Figure 5(b) shows the resulting extended *CNT*s for child1 and child2. For example, $CNT_{EXT_{C1}}$ is $[8, 1, 4, 4]$ and the last three element belong to the same size-class (according to CLS_{C1}). We average all the elements in the same size class to get the count for that size-class. Thus the resulting CNT_{C1} is $[8, 3]$.

Similar to *CLS*, a new *CNT* may need to be repaired. For example, CNT_{C1} is invalid, as the memory consumption is 20MB ($8 * 1 + 3 * 4$), exceeding the 12MB memory capacity. We repair a *CNT* by decreasing each element inverse-proportionally to the element's corresponding size-class. For example, the original CNT_{C1} is $[8, 3]$ and the repaired CNT_{C1} is $[4, 2]$, as the first size-class is 1MB and the second size-class is 4MB.

Mutation. Similar to the scheduling mutation, we mutate more than one element in *CLS* (and *CNT*). An element is mutated with the probability of $\mathcal{P}$

To mutate the i_{th} element in *CLS*, we can either increase it by 1 or decrease it by 1. If $CLS[i]$ equals to $CLS[i-1]$, we can increase $CLS[i]$ by 1 as decreasing it breaks the monotonic increasing feature. If $CLS[i]$ equals to $CLS[i-1]+1$, we decrease $CLS[i]$ by 1. Note that, all the elements after the i_{th} element also need to be increased or decreased to maintain the monotonic increasing feature.

To mutate an element in *CNT*, we use a Gaussian distribution with the original value as the mean. With Gaussian distribution, the mutated value is close to the original value most of the time but can have a large variation with a small chance. The mutated *CNT* and *CLS* can exceed the memory limit. We use the same methodology as crossover to repair them.

6 Evaluation

In this section, we evaluate the performance of SwapAdvisor. The followings are the highlights of our results:

- SwapAdvisor can achieve 53% - 99% of the training throughput of the ideal baseline with infinite GPU memory when training various large DNNs. SwapAdvisor outperforms the online swapping baseline up to 80× for RNNs and 2.5× for CNNs.
- When being used for model inference, SwapAdvisor reduces the serving latency up to 4× compared to the baseline which time-shares the GPU memory.
- SwapAdvisor's joint optimization improves the training throughput with swapping from 20% to 1100% compared to only searching memory allocation or only searching scheduling.

6.1 Experimental setup

Prototype implementation. SwapAdvisor is based on MXNet 1.2. The GA and simulator are written in Python (4.5K LoC). We implement a parallel GA – each process performs crossover, mutation, and simulation on a part of the samples on a CPU core.

For MXNet, we modify the scheduler to ensure swap-in and swap-out operations are run on two separated GPU streams other than the computation stream. We also implemented a new memory allocator which follow the result from SwapAdvisor. The modification of MXNet is 1.5K LoC.

Testbeds. We run SwapAdvisor on an EC2 c5d.18xlarge instance with 72 virtual CPU cores and 144GB memory. The results of SwapAdvisor are executed on an EC2 p3.2xlarge GPU instance, which has one NVIDIA V100 GPU with 16GB GPU memory and 8 virtual CPU cores with 61GB CPU memory. The PCIe bandwidth between the CPU and GPU is 12GB/s unidirectional and 20GB/s bidirectional. For experiments with more than 61GB memory consumption, we use a p3.8xlarge instance with 244 GB CPU memory but utilize only a GPU.

Genetic algorithm parameters. All parameters of the GA are determined empirically using grid-search. The sample size is set to 144 to allow evenly distributing search tasks to the 72 CPU cores. The effectiveness of the mutation probability varies with different models. However, 10% is a good start search point for our evaluation. We set the search time for the GA to be 30 minutes.

Evaluated DNN models. ResNet [16] is one of the most popular CNNs. A ResNet contains several residual blocks; a residual block has several convolution operators. The activation of a residual block is combined with activation from the previous block to form the final output. We use ResNet-152, a 152 layers ResNet, for the inference experiments. Wide ResNet [58] is a widened version of ResNet. The channel size of convolution operators are multiplied by a wide scale. Due to the large memory consumption, the original work applies wide ResNet on small images (32x32) dataset CIFAR-10 instead of ImageNet (224x224) which is used by ResNet. In our training experiments, the input images are the same size as ImageNet. We denote a wide ResNet model as WResNet-152-X, a 152-layers wide ResNet with X wide scale.

Inception-V4 [47] is another popular CNN model. An Inception-V4 model contains several types of inception cells; an inception cell has many branches of convolution and the activation tensors of all the branches are concatenated to form the final output. We enlarge the model by adding a wide scale parameter similar to WResNet. We use the notation Inception-X to denote an Inception-V4 model with wide scale X.

Unlike manually designed ResNet and Inception-V4, NasNet [60] is crafted by a deep reinforcement learning search. Thus, the model structure of NasNet is more irregular. A NasNet model consists of a chain of Reduction and Normal cells, with residual connection (same as ResNet) between consecutive cells. A Reduction or Normal cell is like an inception cell but with different branch structures. The Normal cells are repeated by $3R$ times. In the original design, the max R is 7. In our experiments, we train NasNet-25, a NasNet with $R = 25$.

RNN [17] is a DNN for training sequence input (e.g., text). A layer of RNN consists of a list LSTM cells where the input of a cell is the corresponding element in the sequence (e.g., character). Bidirectional-RNN (BRNN) [41] is a variation of RNN. A hidden layer in BRNN contains two sub-layers. The input for the first sub-layer is the original sequence, and the input for the second sub-layer is the reversed sequence. The activation tensors of the two layers are concatenated to form the final activation. Each sub-layer has its own parameters. We use the notations RNN-L-XK (BRNN-L-XK) to denote a RNN with L layers and the parameter size of a layer is XK.

Baselines. We compare SwapAdvisor with two baselines. The first one is the ideal baseline, denoted as *ideal*. For the ideal baseline, we assume the GPU memory is infinite. We implement the ideal baseline by directly reusing the GPU memory (and thus compromising computation correctness). The ideal baseline gives the strict upper-bound performance achievable by a swapping system.

The second baseline is an online on-demand swapping system, denoted as *ODSwap*, which incorporates heuristics including LRU-based swapping and prefetching used by

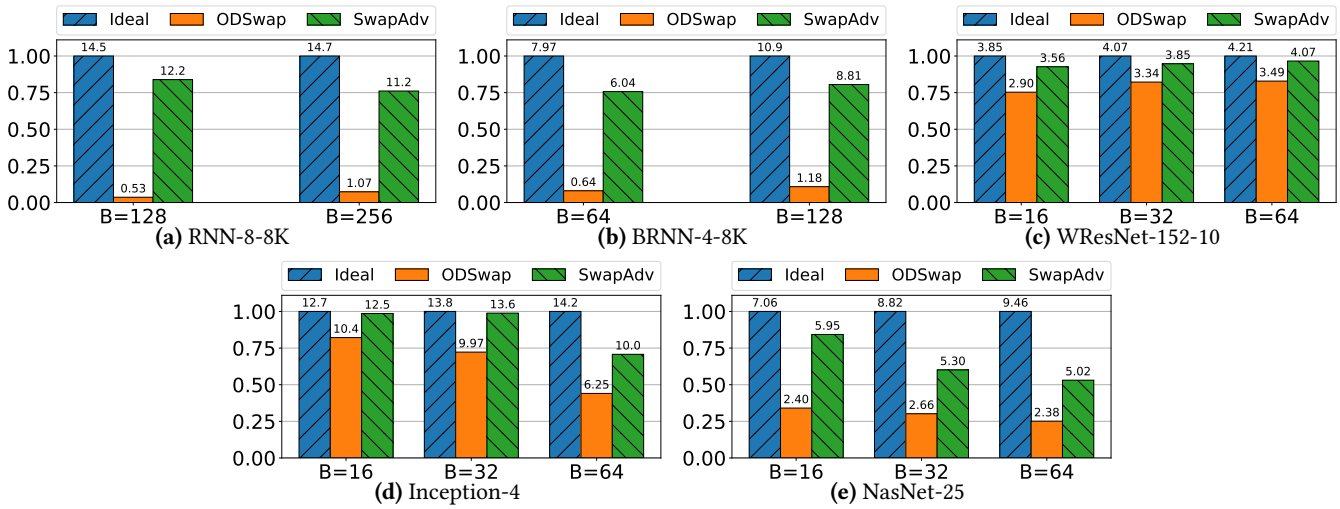


Figure 6. Normalized throughput relative to the ideal performance. Each group of bar represents one batch size. The number on each bar shows the absolute throughput in samples/sec. X axis shows the different batch sizes.

vDNN [40] and SuperNeuron [49]. Specifically, in the first training iteration, ODSwap swaps out a tensor when the GPU memory is insufficient to run the next node based on LRU. In subsequent iterations, ODSwap performs prefetching based on the scheduling decisions seen in the first iteration to best overlap communication and computation. Our reported performance numbers for *ODSwap* ignore the first iteration.

6.2 Wider and deeper DNN model training

Table 1 shows the statistics of the models evaluated in this section. Each row shows the memory usage, number of operators, and number of different tensor sizes. The batch size for a model is the largest one in Figure 6.

RNN performance. Figure 6a and 6b show the throughput for RNN-8-8K and BRNN-4-8K. SwapAdvisor achieves 70-80% of the ideal performance for RNN and BRNN, while the throughput of ODSwap is only less than 1% of ideal. For RNN and BRNN, the parameter tensors are shared by the LSTM cells in the same layer. Thus, a schedule which executes LSTM cells from different layers results in terrible swapping performance as the system has to prepare memory for different large parameters. Unfortunately, randomly generating a topological ordering almost always results in such a schedule, as is MXNet's default schedule. Thus ODSwap has poor performance. SwapAdvisor is able to find a swap-friendly schedule through GA.

CNN performance. Figure 6c, 6d, and 6e demonstrate the throughput for WResNet-152-10, Inception-4, and NasNet-25. Table 1 shows that WResNet-152-10 uses astonishingly 180GB memory, but both SwapAdvisor and ODSwap perform well; SwapAdvisor achieves 95% of the ideal performance, and ODSwap achieves 80% of ideal. WResNet-152-10 has only 26 different tensor sizes, making it less difficult to do the memory allocation. More importantly, unlike RNN/BRNN, the topology of the dataflow graph of WResNet more resembles to a line – only a jump link for a residual block. Thus, the scheduling choice may affect little to the final results.

Model	MemUsage	OPs	TensorSizes
WResNet-152-10	180GB	882	26
Inception-4	71GB	830	64
NasNet-25	193GB	5533	65
RNN-8-8K	118GB	8594	7
BRNN-4-8K	99GB	9034	9

Table 1. Statistics of DNN models. The batch size for CNN models is 64, is 256 for RNN, and is 128 for BRNN.

On the other hand, Inception-4 and NasNet-25 have more than 60 different tensor sizes, making it harder to do memory management. The topology of the dataflow graph for Inception-V4 and NasNet is also more complicated as discussed in Sec 6.1. SwapAdvisor achieves 20% - 150% performance improvement compared to ODSwap.

Note that, for Inception-4 and NasNet-25, SwapAdvisor can achieve 80% performance of the ideal baseline, when the batch size is 16. However, SwapAdvisor cannot achieve more than 65% of the ideal performance when the batch size is 64. Both Inception-4 and NasNet-25 have many large activation tensors (>500MB) when the batch size is 64. Together with large number of tensor sizes (> 60), it can be difficult for SwapAdvisor to search a good pool configuration to minimize swapping overhead. NasNet-25 also has more than 9000 nodes in the graph, making it hard to schedule. As a result, SwapAdvisor achieves only 53% of the ideal baseline for NasNet-25 with batch size 64.

Comparing with TFLMS. We also compare SwapAdvisor and ODSwap with TFLMS [26], an swapping extension to TensorFlow. Unfortunately, TFLMS cannot support models in Figure 6. We evaluate WResNet-152-4 as this is the largest

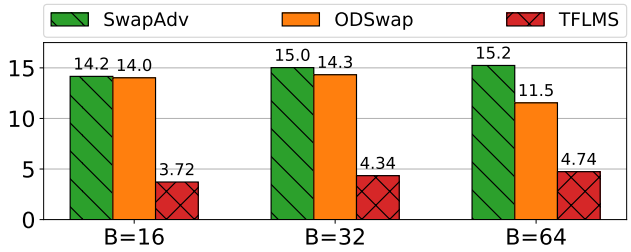


Figure 7. WResNet-152-4 throughput comparison

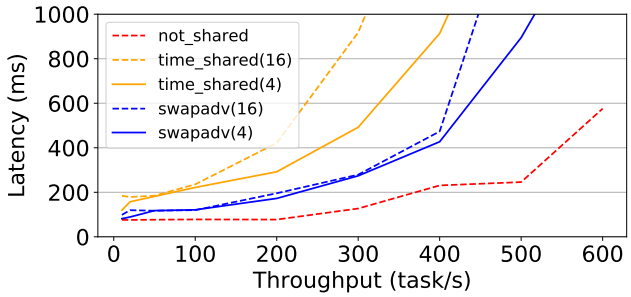


Figure 8. 99 percentile latency versus throughput for serving multiple ResNet-152 models.

executable one for TFLMS. Figure 7 shows that for all three batch sizes, SwapAdvisor is at least 3X better than TFLMS, while ODSwap is at least 2X better than TFLMS. TFLMS performs poorly due to not swapping out parameter tensors (which are large in WResNet-152-4). The design reduces the GPU memory capacity to store activation tensors and causes more swapping of activation tensors.

6.3 DNN models inference evaluation

MemSize/Batch	1	16	64
64MB	**0.024s**	N/A	N/A
128MB	**0.022s**	**0.077s**	N/A
192MB	**0.018s**	**0.044s**	N/A
512MB	*0.017s*	***0.040s***	**0.238s**
640MB	*0.017s*	***0.040s***	***0.123s***

Table 2. ResNet-152 inference time with different batch sizes and GPU memory sizes.

Model inference (serving) uses far less GPU memory than training a model as there is no back-propagation. However, SwapAdvisor is still useful for serving in several cases.

Table 2 shows how SwapAdvisor can reduce the memory requirement for ResNet-152 with different batch sizes. In the table, each cell is the running time of one inference iteration with the corresponding available GPU memory size. "N/A" means SwapAdvisor cannot run the inference job with such a small memory capacity. The running time with a bold font means swapping is required to run the job (memory is not enough). The running time with an italic font means that the running time is close to the performance using full 16GB memory capacity (< 1% performance difference).

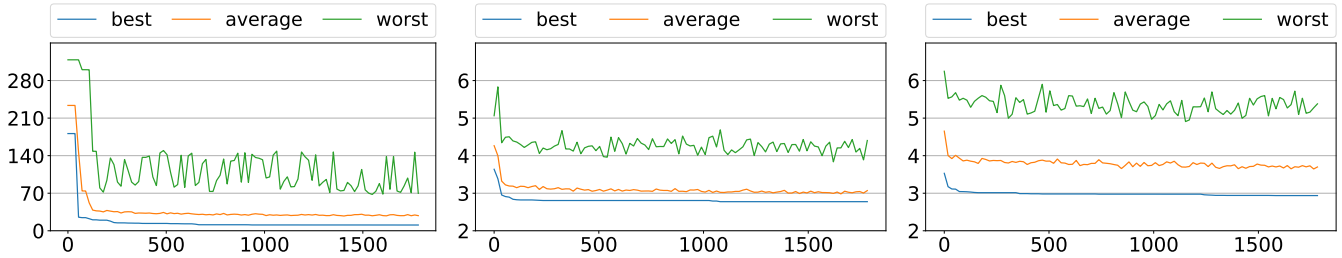
An interesting experiment is when the batch size is 1. Batch size 1 is rarely used for training or inference on a cluster, but it is not uncommon to be used for inference on a mobile device. Table 2 demonstrates SwapAdvisor can reduce the memory usage for batch size 1 to as few as 64MB with 40% running time overhead or to 192MB with only 6% overhead. SwapAdvisor can help to fit a DNN model to a resource-limited GPU. Though the communication speed between GPU and CPU on a mobile device is slower than that on an EC2 GPU instance, V100 GPU is also much faster than a mobile GPU. Consequently, the actual swapping overhead on a mobile device can be different from what Table 2 shows. Nevertheless, the result still poses a potential use case for SwapAdvisor.

Another possible serving use case for SwapAdvisor is to time-share the GPU resource among different DNN inferences. In the setting, a GPU machine time-shares the computation and memory among models. What if we only time-share GPU computation but partition the GPU memory and distribute the GPU memory to the models? Since the split GPU memory capacity may be too small for a DNN model, we apply SwapAdvisor so that all the models can fit on the partitioned memory.

We consider time-sharing GPU memory as the baseline and compare the latency of clients. In the evaluation, there are multiple ResNet-152 on the GPU, each has its own trained parameters. We assume that the client arrival rate follows a Poisson distribution and randomly assign batches of clients to different models. Figure 8 shows the 99 percentile latency versus the throughput of the GPU machine. The number after a legend denotes how many models are run on the GPU. The figure also shows the latency for serving only one model on the GPU, denoted as "not_shared".

We can see that the 99 percentile latency of SwapAdvisor is at most 2× slower than "not_shared" when the throughput is less than 400 for both 4 and 16 models. On the other hand, the latency of "time_shared" is 8x slower than "not_shared" with 16 models when the throughput is 300. It may not be a wise decision to serve several ResNet-152 on a GPU when the throughput is larger 400 as the latency dramatically increases for both SwapAdvisor and "time_shared".

The main benefit of SwapAdvisor is that it overlaps the memory copy with the computation. On the other hand, the baseline has to swap in the parameter tensors for the next model after the current model execution. It is possible for "time_shared" to prefetch the parameters for the next model if the system can predict which model to execute next. With such a task scheduler, the baseline may outperforms SwapAdvisor. However, SwapAdvisor can still be used to mitigate the potential overhead when the task scheduler predicts incorrectly.



(a) RNN-8-8K **(b)** Inception-4 **(c)** Inception-4 (w/o restricted size-class mapping)

Figure 9. The search result versus search time. X axis is the search time and Y axis is the running time (seconds/iteration).

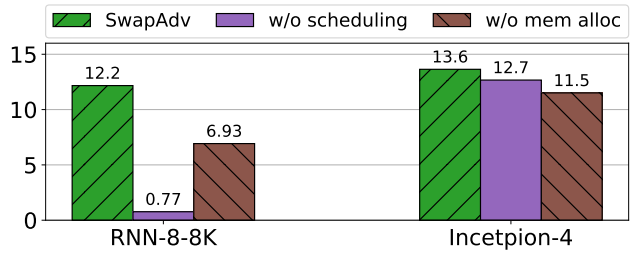


Figure 10. Throughput for SwapAdvisor with different search settings for RNN and Inception-V4.

6.4 The effectiveness of SwapAdvisor's design choices

The effectiveness of scheduling and memory allocation. We would like to see the importance to optimize both scheduling and memory allocation. Figure 9 shows that it is very important for a RNN model to search a swap-friendly schedule. Without searching a good schedule, SwapAdvisor can only achieve 7% of the performance with the full search. On the other hand, Figure 10 shows that the memory allocation affects the performance of Inception-V4 more than the scheduling. Without searching schedules, SwapAdvisor can still achieve 93% performance of the full search. Figure 10 demonstrates that it is important to optimize both scheduling and memory allocation for swapping as the effectiveness of the two components vary from model to model.

The performance of the genetic algorithm. Figure 9 shows the search performance of the GA. In the figures, there are three lines, representing the best, average, and worst simulated time of all the alive samples (144 samples) at the moment. The first generation of sample is randomly generated. The GA can generally find a good solution within 100 seconds, as both the average and the best simulated time converge quickly within the first 100 seconds. However, the difference between the worst (or the average) result and the best result shows that the population still maintains the diversity, allowing the GA to gradually optimizes the sample in the remaining time.

Both Figure 9b and Figure 9c show the search for Inception-4, but Figure 9c assumes that if the tensor size mapping is unrestricted. We can see that all of the best, average, and worst result in Figure 9c are worse than Figure 9b, proving that the restricted search space helps SwapAdvisor to find better results. The effectiveness of the restricted search space is universal to all the evaluated models.

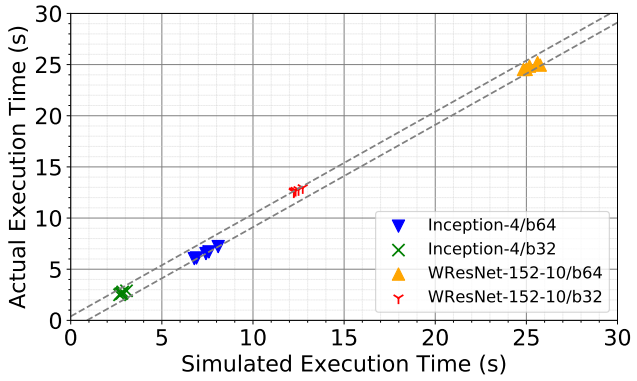


Figure 11. Simulated execution time versus actual execution time for the simulation results of two different models with batch sizes, 32 and 64.

6.5 Simulator accuracy

Figure 11 shows 20 randomly selected simulation results and their corresponding actual execution time. There are two models, Inception-4 and WResNet-152-10, with two different batch sizes, 32 and 64. Thus, there are totally 4 different experiments, and each has 5 data points. The two dashed lines depict the bounds of simulation inaccuracy. The upper dashed line means the actual execution is 4% slower than the simulation, and the lower one means the actual execution is 12% faster than the simulator. For example, all the data points WResNet-152-10 with batch size 32 (WResNet-152-10/b32) are closer to the upper dashed line. Thus the simulation of WResNet-152-10/b32 is around 4% faster than the actual execution.

Among the 20 data points in Figure 11, 19 data points maintain the same relative performance according to simulation and actual execution. The only exception is the rightmost data point (belonging to WResNet-152-10/b64). Whenever two data points have different relative performance according to simulation and actual execution, their actual execution time difference is less than 10% when evaluated on a larger

set of data points than shown in Figure 11 (In Figure 11, the max difference is 2%, corresponding to the two rightmost data points). These results indicate that using simulated performance is an effective way to accelerate GA-based search.

7 Related Work

Swapping for DNN. Existing swapping systems rely on manual insights to determine what to swap. vDNN [40] swaps out all activation tensors or swap all convolution tensors only. TFLMS [26] also only swaps activation. [30] uses the length of the critical path for an activation tensor and its loss-function node as the heuristic to decide what activation tensors to swap out. [57] is an on-demand swapping mechanism for TensorFlow. Its heuristic is to swap out tensors from the previous iterations to the host memory, a strategy that only works for RNNs.

None of the work above swaps parameters, hence cannot support a large model. SuperNeurons [49] adopts a different approach; it combines swapping with recomputation. However, SuperNeurons restricts the swapping to convolution operators. The decision forbids SuperNeurons from supporting an RNN model with large parameters. By contrast, SwapAdvisor can support various kinds of deeper and wider DNN models.

Alternative approaches to GPU memory limit. There exist approaches that do not rely on swapping to reduce memory consumption. The first direction includes computing with lower-precision floating-point numbers [12, 24], quantization, and parameters compression [3, 9, 13–15, 20]. [33] observes the similarities among the activation tensors for CNN inferences and proposes to reuse the activation tensors to speedup the performance and to reduce memory consumption. These techniques either affect the model accuracy or require heavily hyper-parameter tuning while swapping does affect the results.

Another approach is recomputation. Recomputation utilizes the fact that an activation tensor can be recomputed. As a result, [2, 11, 29] deallocate activation tensors after their last usage in the forward-propagation and later recompute the activation tensors when they are needed. Although recomputation can be used for deeper models and large input data, it fails to support wider models where large parameter tensors occupy the memory and cannot be recomputed.

Finally, training DNN models with multiple GPUs is an active research. The most popular way to parallelize a DNN model is data parallelism [4, 27, 51]. With data parallelism, each GPU gets a portion of the input data and full parameters of the model. Thus, the input tensor and activation tensors are sliced and distributed to GPUs, effectively reducing the memory consumption of each GPU. While easy to use, data parallelism duplicates the full parameters on each GPU, limiting the largest parameter size the model can have. Contrary to data parallelism, model parallelism partitions both activation tensors and parameter tensors [6, 25]. However, applying model parallelism for a model requires significant engineering work. [22, 23, 50] propose to automate the model parallelism and reduce the communication with dataflow graph analysis.

Scheduling to reduce memory utilization. Some work have considered the trade-off between the throughput of a task graph schedule and its resource consumption [44, 45, 53]. These work target scenarios where the buffer (memory) is expensive, and the application has a throughput constraint (e.g. video frame rates). Consequently, the optimization goal is not to maximize the throughput but to minimize the memory consumption subject to a throughput constraint. SwapAdvisor, on the other hand, aims to minimize execution time (maximize throughput) with unconstrained CPU-memory and constrained GPU-memory consumption. The key design factor for SwapAdvisor is communication and computation overlapping, which is not considered in the related work.

Overlapping GPU communication and computation. There are existing work to overlap communication and computation [1, 59] for other types of computation not related to DNNs. They adopt two patterns: 1) divide a coarse-grained kernel to fine-grained sub-kernels so some sub-kernels can start computation while others wait for communication, 2) use queues to communicate fine-grained computation results between producers/consumers who process fine-grained tasks. Kernel subdivision or the queue-based approach enable overlapping within a single coarse-grained kernel or a producer/consumer pair. However, our past experience has shown non-trivial performance overhead when dividing coarse-grained DNN kernels into fine-grained sub-kernels. Thus, SwapAdvisor relies on pre-fetching to enable overlapping within a dataflow graph of many coarse-grained kernels.

Multiple DNN inferences on a GPU. TensorRT [34] leverages GPU streams to run multiple model inferences concurrently. NVIDIA MPS [35] also supports concurrent GPU tasks, but the tasks are not limited to DNN inference. Both TensorRT and MPS requires users to partition the GPU memory for tasks. SwapAdvisor can help both systems to alleviate the memory pressure. Salus [56] aims to support fine-grained GPU sharing among DNN tasks. Salus allocates a shared memory space to store activation tensors and scratch space for all the models as these memory consumption can be dropped directly after the last usage in an iteration. It assumes parameter tensors are in the GPU memory, and thus can borrow SwapAdvisor's technique to support even more(and larger) models on a GPU.

Genetic algorithm for computer systems. Genetic algorithm has been used to schedule tasks on parallel or distributed systems [19, 42, 43, 46, 48, 54]. SwapAdvisor borrows several ideas from the existing work, e.g., how to cross over schedules. However, the setting of SwapAdvisor is different. The existing work is designed for a multi-cores or multi-machines system where a task can be scheduled on a different core or machine. On the other hand, all of the computation tasks in SwapAdvisor are executed on the same GPU. Consequently, only the execution order matters for SwapAdvisor, resulting in a different crossover and mutation.

Some work use genetic algorithm to allocate data objects in a heterogeneous memory system (e.g. SRAM vs. DRAM) [7, 38]. The memory allocation of SwapAdvisor also decides how many memory pools before allocating memory objects for a pool.

8 Discussion, limitations, and future work

Dynamic dataflow graph. The design of SwapAdvisor requires a static dataflow graph. To work with PyTorch or TensorFlow's imperative execution mode, one could extract a dataflow graph using techniques described in [21]. However, SwapAdvisor currently cannot handle any control-flow primitives [57] in the dataflow graph, which are needed to support a wider range of DNNs.

One potential solution is to adopt a hybrid swapping design. We can view a dynamic dataflow graph as consisting of three types of sub-graphs: The first one is a static sub-graph, which corresponds to the static part of the computation and is always executed. The second one is a repetitive sub-graph (e.g., containing a "while" loop), which can executed more than once. And the final one is a sub-graph with conditional nodes (e.g., containing "if/else"). SwapAdvisor can be directly applied to static sub-graphs, as well as repetitive sub-graphs using the same technique for handling multiple training iterations (Sec 4.2). For sub-graphs with conditional nodes, we can instead use an online on-demand swapping strategy (e.g., ODSwap). There remain open challenges (e.g., how to properly switch from SwapAdvisor to ODSwap) for the hybrid design, which require further investigation.

Multi-GPU support. SwapAdvisor currently swaps for a single-GPU, and it's desirable to extend it to work with multiple GPUs. There are two popular methods for multi-GPU training, data parallelism and model parallelism. With data parallelism, each GPU runs the same dataflow graph using a different sub-batch of training data. In this case, we can run SwapAdvisor for one GPU while taking into account that the available GPU-CPU bandwidth is 1/n with n-GPUs. Unfortunately, no straightforward adaptation of SwapAdvisor exists for model parallelism. This is because, unlike data parallelism, model parallelism incurs inter-GPU communication during both forward/backward propagation. Since SwapAdvisor does not consider communication across GPUs, the resulting swapping strategy is unlikely to be optimal. It remains an open research problem how to generalize SwapAdvisor for multi-GPU training under model parallelism.

Alternative search methods. We chose GA because we find that it works well empirically and is much faster than some alternatives (e.g. simulated annealing). However, we have yet to explore certain other promising search methods such as reinforcement learning [31, 32]. The use of GA also forces us to simplify certain design choices, e.g. we adopt a memory-pool based allocation because it is difficult to incorporate a dynamic allocator in GA's search. Whether it is beneficial to remove these limitations using better search methods requires further research.

9 Conclusion

We present SwapAdvisor to enable training and serving DNN models with limited GPU memory size. SwapAdvisor achieves the good performances via optimizing three dimensions, scheduling, memory allocation, and swap planning. To simultaneously optimize scheduling and memory allocation, SwapAdvisor adopts the genetic algorithm to search for a good combination. For a given schedule and memory allocation, SwapAdvisor's swap planner is able to determine what and when tensors to swap to maximize the overlap computation and communication.

Acknowledgments

This work is supported in part by the National Science Foundation under award CNS-1816717, NVIDIA AI Lab (NVAIL) at NYU, AMD research grant, and AWS cloud credits for research. We thank anonymous reviewers for their helpful feedback on the paper, and Minjie Wang and Chengchen Li for insightful discussion throughout the project.

References

[1] B. Bastem, D. Unat, W. Zhang, A. Almgren, and J. Shalf. 2017. Overlapping Data Transfers with Computation on GPU with Tiles. In *2017 46th International Conference on Parallel Processing (ICPP)*. 171–180.

[2] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[3] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. BinaryConnect: Training Deep Neural Networks with Binary Weights during Propagations. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'15)*. MIT Press, 3123–3131.

[4] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, and et al. 2014. Exploiting Bounded Staleness to Speed up Big Data Analytics. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, 37–48.

[5] Lawrence Davis. 1991. Handbook of genetic algorithms. (1991).

[6] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, and et al. 2012. Large Scale Distributed Deep Networks. In

Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12). Curran Associates Inc., 1223–1231.

[7] Keke Gai, Meikang Qiu, and Hui Zhao. 2016. Cost-Aware Multimedia Data Allocation for Heterogeneous Memory Using Genetic Algorithm in Cloud Computing. *IEEE Transactions on Cloud Computing* (2016), 1–1.

[8] David E Goldberg and John H Holland. 1988. Genetic algorithms and machine learning. *Machine learning* 3, 2 (1988).

[9] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. 2014. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115* (2014).

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[11] Audrundefinednas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. 2016. Memory-Efficient Backpropagation through Time. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'16)*. Curran Associates Inc., 4132–4140.

[12] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep Learning with Limited Numerical Precision. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15)*. JMLR.org, 1737–1746.

[13] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. *SIGARCH Comput. Archit. News* 44, 3 (June 2016), 243–254.

[14] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[15] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning Both Weights and Connections for Efficient Neural Networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'15)*. MIT Press, 1135–1143.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778.

[17] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780.

[18] John Henry Holland et al. 1992. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press.

[19] Edwin SH Hou, Nirwan Ansari, and Hong Ren. 1994. A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems* 5, 2 (Feb 1994), 113–120.

[20] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized Neural Networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'16)*. Curran Associates Inc., 4114–4122.

[21] Eunji Jeong, Sungwoo Cho, Gyeong-In Yu, Joo Seong Jeong, Dong-Jin Shin, and Byung-Gon Chun. 2019. JANUS: Fast and Flexible Deep Learning via Symbolic Graph Execution of Imperative Programs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, 453–468.

[22] Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. 2018. Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks. In *International Conference on Machine Learning*.

[23] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2018. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358* (2018).

[24] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M. Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Proteus: Exploiting Numerical Precision Variability in Deep Neural Networks. In *Proceedings of the 2016 International Conference on Supercomputing (ICS'16)*. Association for Computing Machinery, Article 23.

[25] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997* (2014).

[26] Tung D Le, Haruki Imai, Yasushi Negishi, and Kiyokuni Kawachiya. 2018. Tflms: Large model support in tensorflow by graph rewriting. *arXiv preprint arXiv:1807.02037* (2018).

[27] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, 583–598.

[28] Kim-Fung Man, Kit Sang Tang, and Sam Kwong. 2001. *Genetic algorithms: concepts and designs*. Springer Science & Business Media.

[29] James Martens and Ilya Sutskever. 2012. Training deep and recurrent networks with hessian-free optimization. In *Neural Networks: Tricks of the Trade*. Springer.

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

[31] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V Le, and Jeff Dean. 2018. A hierarchical model for device placement. (2018).

[32] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device Placement Optimization with Reinforcement Learning. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70 (ICML'17)*. JMLR.org, 2430–2439.

[33] Lin Ning and Xipeng Shen. 2019. Deep Reuse: Streamline CNN Inference on the Fly via Coarse-Grained Computation Reuse. In *Proceedings of the ACM International Conference on Supercomputing (ICS'19)*. Association for Computing Machinery, 438–448.

[34] NVIDIA. 2018. NVIDIA TensorRT. (2018). https://developer.nvidia.com/tensorrt

[35] NVIDIA. 2019. CUDA Multi-Process Service. (2019). https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf

[36] NVIDIA. 2019. NVIDIA NVLINK High-Speed Interconnect. (2019). https://www.nvidia.com/en-us/data-center/nvlink/

[37] PCI-SIG. 2019. PCI Express Base Specification Revision 5.0. (2019). https://pcisig.com/specifications

[38] M. Qiu, Z. Chen, J. Niu, Z. Zong, G. Quan, X. Qin, and L. T. Yang. 2015. Data Allocation for Hybrid Memory With Genetic Algorithm. *IEEE Transactions on Emerging Topics in Computing* 3, 4 (Dec 2015), 544–555.

[39] Colin Reeves and Jonathan E Rowe. 2002. *Genetic algorithms: principles and perspectives: a guide to GA theory*. Vol. 20. Springer Science & Business Media.

[40] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. 2016. vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'49)*. IEEE Press, Article 18.

[41] M. Schuster and K. K. Paliwal. 1997. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing* 45, 11 (Nov 1997), 2673–2681.

[42] Harmel Singh and Abdou Youssef. 1996. Mapping and scheduling heterogeneous task graphs using genetic algorithms. In *5th IEEE Heterogeneous Computing Workshop (HCW'96)*.

[43] Oliver Sinnen. 2007. *Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, USA.

[44] Sander Stuijk, Marc Geilen, and Twan Basten. 2008. Throughput-Buffering Trade-Off Exploration for Cyclo-Static and Synchronous

Dataflow Graphs. *IEEE Trans. Comput.* 57, 10 (Oct 2008), 1331–1345.

[45] Sander Stuijk, Marc Geilen, Bart Theelen, and Twan Basten. 2011. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. 404–411.

[46] Sung-Ho Woo, Sung-Bong Yang, Shin-Dug Kim, and Tack-Don Han. 1997. Task scheduling in distributed computing systems with a genetic algorithm. In *Proceedings High Performance Computing on the Information Superhighway. HPC Asia'97*. 301–305.

[47] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A. Alemi. 2017. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI'17)*. AAAI Press, 4278–4284.

[48] Lee Wang, Howard Jay Siegel, Vwani P Roychowdhury, and Anthony A Maciejewski. 1997. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *J. Parallel and Distrib. Comput.* 47, 1 (1997).

[49] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU Memory Management for Training Deep Neural Networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'18)*.

[50] Minjie Wang, Chien-Chin Huang, and Jinyang Li. 2019. Supporting Very Large Models Using Automatic Dataflow Graph Partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys'19)*. Association for Computing Machinery, Article 26.

[51] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. 2015. Managed Communication and Consistency for Fast Data-Parallel Iterative Analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC'15)*. Association for Computing Machinery, 381–394.

[52] Eric W. Weisstein. 2010. Bell Number. From MathWorld – A Wolfram Web Resource. (2010). http://mathworld.wolfram.com/BellNumber.html

[53] Maarten H. Wiggers, Marco J. G. Bekooij, and Gerard J. M. Smit. 2007. Efficient Computation of Buffer Capacities for Cyclo-Static Dataflow Graphs. In *Proceedings of the 44th Annual Design Automation Conference (DAC'07)*. Association for Computing Machinery, 658–663.

[54] Annie S. Wu, Han Yu, Shiyuan Jin, Kuo-Chi Lin, and Guy Schiavone. 2004. An Incremental Genetic Algorithm Approach to Multiprocessor Scheduling. *IEEE Trans. Parallel Distrib. Syst.* 15, 9 (Sept. 2004), 824–834.

[55] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, and Mohammad Norouzi. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. In *arxiv.org:1609.08144*.

[56] Peifeng Yu and Mosharaf Chowdhury. 2019. Salus: Fine-Grained GPU Sharing Primitives for Deep Learning Applications. *arXiv preprint arXiv:1902.04610* (2019).

[57] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, and et al. 2018. Dynamic Control Flow in Large-Scale Machine Learning. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys'18)*. Association for Computing Machinery, Article 18.

[58] Sergey Zagoruyko and Nikos Komodakis. 2016. Wide residual networks. *arXiv preprint arXiv:1605.07146* (2016).

[59] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, and Wenguang Chen. 2019. HiWayLib: A Software Framework for Enabling High Performance Communications for Heterogeneous Pipeline Computations. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. Association for Computing Machinery.

[60] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. 2018. Learning Transferable Architectures for Scalable Image Recognition. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 8697–8710.