HOME: A Holistic GPU Memory Management Framework for Deep Learning

Shuibing He, Ping Chen, Shuaiben Chen, Zheng Li, Siling Yang, Weijian Chen, Lidan Shou

Abstract—We propose HOlistic MEmory management (HOME), a new framework for performing tensor placements in large DNN training when GPU memory space is not enough. HOME combines tensor swapping with tensor recomputation to reduce GPU memory footprint. Different from existing work that only considers partial DNN model information, HOME takes the holistic DNN model information into account in tensor placement decisions. More specifically, HOME uses a custom-designed particle swarm optimization algorithm to achieve the globally optimized placement for each tensor of the DNN model with a greatly reduced searching space. This holistic awareness of the whole model information enables HOME to obtain high performance under the given GPU memory constraint. We implement HOME in PyTorch and conduct our experiments using six popular DNN models. Experimental results show that HOME can outperform vDNN and Capuchin by up to 5.7× and 1.3× in throughput. Furthermore, HOME can improve the maximum batch size by up to 2.8× than the original PyTorch and up to 1.3× than Capuchin.

Index Terms—DNN, GPU, Tensor, Swapping, Recomputation

1 Introduction

Deep learning has gained great success in various domains, such as computer vision [1], natural language processing [2], recommendation systems [3], and speech recognition [4]. Larger deep neural networks (DNNs) are designed to deal with more complex tasks, such as InceptionV3 [5] and BERT [6]. GPU is a popular hardware accelerator for DNN training. However, due to the limited memory space, training large DNN models on GPUs may happen to out-of-memory errors [7]. For example, the latest BERT [6] with 768 hidden layers consumes 73 GB memory with a batch size of 64, which exceeds the maximal memory size of a powerful NVIDIA V100 GPU, i.e., 32 GB memory [8].

During the DNN training process, deep learning frameworks, such as TensorFlow [9], PyTorch [10], and Caffe [11], usually generate a large number of feature maps in the forward propagation. These feature maps retain in the GPU memory until they are reused in the back propagation. Consequently, the training process of large DNN models usually renders a high GPU memory demand.

There are three kinds of techniques commonly used to mitigate the GPU memory capacity issue. *Data compression* consists of lossy and lossless compression, both use fewer bits for data representation to save memory footprints [12]. However, lossy compression may affect the accuracy of DNNs and lossless compression may incur significant time overhead [13]. *Data swapping* treats the CPU DRAM as the temporary and external memory and transfers the data between the GPU memory and CPU [14], [15], [16]. *Data recomputation* regenerates the required intermediate feature maps by replaying the forward computation [17]. Swapping and recomputation will not affect the accuracy of the DNNs [18]. Therefore, both of them are widely used to mitigate the GPU out-of-memory issue.

To further reduce memory usage, recent studies also explored the idea of jointly utilizing swapping and recomputation techniques [18], [19]. Their strategies are to swap or recompute each feature map for back-propagation based on tensor characteristics and training dataflow. However, these approaches result in sub-optimal performance and could be improved because they only consider partial DNN model information (i.e., several layers instead all of the layers of the DNN model) when making the swapping or recomputation decision for each tensor.

In this paper, we propose HOlistic MEmory management (HOME), a new framework for performing tensor placements in large DNN training when GPU memory space is not enough. HOME allows tensor swapping and tensor recomputation to reduce GPU memory footprint. It fully considers the holistic DNN model information in making tensor placement decisions. This holistic awareness of the whole model information enables HOME to obtain high performance under the given GPU memory constraint. To this end, HOME first profiles the whole DNN model to grab the desired memory usage statistics. It then makes one of the following management strategies: swapping, recomputation, and retaining, for each tensor based on this information.

However, there are two challenges for HOME to make the tensor placement policies for the globally optimized performance. First, there may be a large number of tensors in the DNN and each tensor can be swapped or recomputed. Therefore, the search space is huge and a brute force search to find the optimal solution is time-unaffordable. To address this issue, HOME uses a custom-designed particle swarm optimization (PSO) algorithm to achieve the globally optimized tensor placement policy for the DNN model with a

- S. He, P. Chen, S. Chen, S. Yang, W. Chen, and L. Shou are with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China. E-mail: {heshuibing, zjuchenping, jsxnh, slingzjunet, weijianchen, should}@zju.edu.cn
- Z. Li is with the Computer Science Program, School of Business, Stockton University, USA. E-mail: zheng.li@stockton.edu

greatly reduced searching space.

Second, during the policy searching process, HOME needs to evaluate the efficiency of the potential policy in the PSO algorithm, i.e., the DNN training time with each possible policy. A naive method is to evaluate the DNN training time on a real hardware platform. However, as the whole search process space may involve a large number of policies, evaluating the training times with all policies on the real hardware platform is costly and unfeasible. To address this issue, HOME proposes a time-cost model to accurately predict the DNN training time instead of doing real tests.

In summary, this paper offers the following contributions:

- We propose HOME, a holistic GPU memory management framework, which fully considers the DNN model information to find the globally optimized tensor placement policy in GPUs with reduced searching space by leveraging the particle swarm optimization algorithm.
- To evaluate the efficiency of a given tensor placement policy, we propose a time cost model, which can accurately predict the DNN training time with tensor swapping and recomputation.
- We implement HOME in the deep learning framework PyTorch [10] and evaluate it using six popular DNN models. The experimental results show that HOME can outperform vDNN and Capuchin by up to 5.7× and 1.3× in throughput.

The remainder of this paper is organized as follows. Section 2 describes the background and motivation of HOME. Section 3 describes the design and implementation. Section 4 presents the evaluation results. Section 5 introduces the related work. Finally, we conclude the paper in section 6.

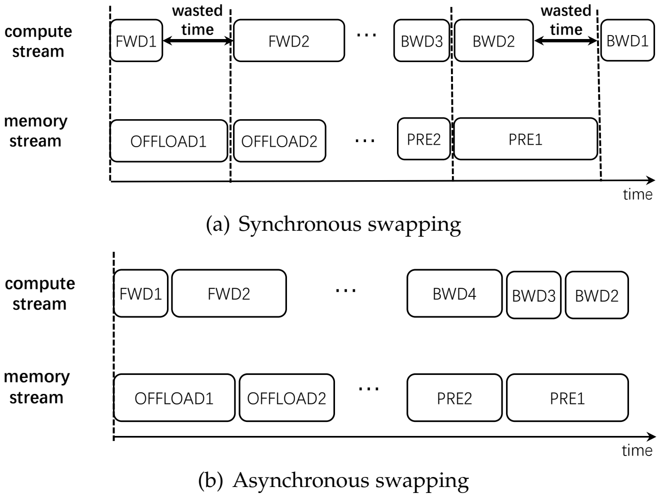
2 Background and Motivation

2.1 DNN Training

DNN models consist of many layers, and each layer contains many parameters to be trained. The training process usually consists of a large number of iterations in order to update a large number of model parameters iteratively until the accuracy converges. The dataflow of each iteration is the same and composed of two phases, i.e., forward propagation and backward propagation. In the forward propagation, feature maps produced by each layer will be used as the input of the next layer and retained in GPU memory. They are represented as tensors and usually dominate the memory footprint. The backward propagation can be treated as a reversed process of forward propagation in which the generated feature maps will be used to calculate gradients and then released from GPU memory.

2.2 Swapping and Recomputation

Swapping refers to the exchange of data between GPU memory and CPU memory. In particular, in the forward propagation of DNN training, tensors generated in GPU memory can be offloaded to CPU memory to save space. In the backward propagation, tensors offloaded to CPU memory should be prefetched back to GPU memory before the tensor is accessed. The swapping and computation process can be executed simultaneously using different streams. The overall DNN training time is determined by the longer time of swapping and recomputation stream.



(a) Synchronous swapping

(b) Asynchronous swapping

Fig. 1. Examples of layer-wise synchronous and asynchronous data swapping. FWD1 represents the forward propagation of the first layer in the DNN model, BWD1 represents the backward propagation of the first layer in the DNN model.

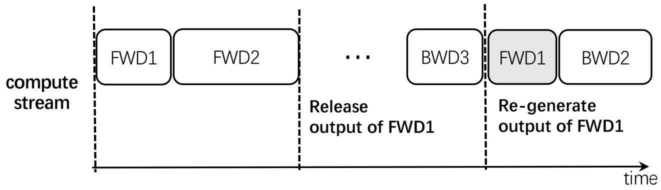


Fig. 2. An example of data recomputation. The shaded rounded rectangle represents the recomputation of a DNN model's first layer.

As shown in Figures 1(a) and 1(b), swapping can be conducted synchronously and asynchronously. Synchronous swapping is to ensure the offloaded data is safely released before the starting of the next forward or backward propagation to maximize the memory saving [14], [19]. However, as shown in Figure 1(a), the computation stream is not fully utilized when the swapping takes longer than the layer's computation. To eliminate the idle time of computation stream, Capuchin [18] proposes an asynchronous swapping policy as shown in Figure 1(b). In this paper, we are also to utilize asynchronous swapping technique to improve the DNN model training throughput.

Although swapping can reduce GPU memory footprint, the time cost of swapping the produced tensors between GPU and CPU is usually more 2× than the computation time for most layers, which may slow down the training process [20]. Rather than swapping, recomputation is another effective method for reducing memory usage. Concretely, in the forward propagation, the feature maps will not be retained in the GPU memory and will be regenerated in the backward propagation before being accessed again [21]. For example, as shown in Figure 2, the output of the first layer is released after the forward propagation of the second layer, and the computation stream replays the forward propagation of the first layer to guarantee the execution of the backward propagation of the second layer.

Feature maps (FMs) produced in the latter layer will be regenerated earlier in the backward propagation due to the reverse execution order compared to the forward propagation. The recomputation of a feature map may trigger the feature maps in the precedent layers to be recomputed as well. For instance, we use FM1 -> FM2 -> FM3 to represent a data dependency between feature maps generated in layer 1, 2, and 3, and we decide FM2 and FM3 to be recomputed during backward propagation. When FM3 is to be regenerated, it will trigger the regeneration of FM2 because FM3 is dependent on FM2.

2.3 Motivation

Limitations of existing approaches. Our proposed research is to maximize DNN training throughput while satisfying the memory capacity constraint. To achieve this goal, we intend to manage all the feature maps generated in the forward propagation which usually dominate the memory footprint [22]. Some existing works like vDNN [14], moDNN [15], AutoTM [23], SwapAdvisor [16], and Checkmate [17] shown in Table 1 use either swapping (Swp) or recomputation (Recmp) solely to reduce GPU memory footprint, which can be further improved by jointly utilizing these two strategies. Therefore, we take both swapping and recomputation methods into consideration to manage feature maps for memory saving.

Although the state-of-the-art SuperNeurons [19] and Capuchin [18] combine swapping and recomputation techniques into their frameworks, there is still potential to improve DNN training throughput because they only consider partial DNN model information. Specifically, SuperNeurons only optimizes the tensor placements for a certain type of DNN layers and Capuchin determines the policy according to the DNN dataflow from the first DNN layer to the current layer. Unlike the existing works, we profile the swapping and recomputation overhead during model training, and dynamically decide each feature map's placement strategy, i.e., swapping, recomputation or retaining, in a global optimum manner to maximize DNN training throughput under the memory capacity constraint.

Analysis of different searching algorithms. By numbering the placement strategies swapping, recomputation, and retaining as 0, 1, and 2, respectively, our problem can be formulated as an assignment problem that assigns 0, 1, and 2 to each tensor in the DNN with the memory constraint. Instead of using sub-optimal algorithms to find the tensor placement policy, as in SuperNeurons and Capuchin, we adopt effective searching algorithms to solve the assignment problem in an optimized manner.

There are several popular optimization algorithms that can be used to solve the problem, such as integer linear programming (ILP) [23], mountain climbing algorithm (MC) [24], simulated annealing algorithm (SA) [25], Bayesian optimization (BO) [26], and reinforcement learning (RL) [27]. However, ILP is suitable for solving small-scale problems because of its high algorithm complexity [17]. The assignment problem of DNN memory management is a high-dimensional problem. Thus, it is difficult for ILP to converge quickly when training large DNN models [23]. Furthermore, the MC, SA, BO, and RL algorithms rely on the initial point and search the optimized strategy serially, incurring two drawbacks. First, they are likely to fall into the local optimization because of the random initial point. Second, they cannot search for the solution in parallel [16]. Thus, these algorithms are neither an ideal choice to solve our problem.

TABLE 1
Comparison with existing GPU memory management works.

	Recmp	Swp	Global Optimization	Decision Method
vDNN [14]		✓		Expert Knowledge
moDNN [15]		✓		Expert Knowledge
AutoTM [23]		✓	✓	ILP
SwapAdvisor [16]		✓	✓	Genetic Algorithm
Checkmate [17]	✓		✓	ILP
SuperNeurons [19]	✓	✓		Expert Knowledge
Capuchin [18]	✓	✓		Greedy Algorithm
HOME	✓	✓	✓	PSO

To avoid the above-mentioned issues, inspired by the effectiveness of particle swarm optimization (PSO) algorithm [28] in solving assignment problems, we use a custom-designed PSO algorithm to determine the tensor placement policies for improved system performance.

PSO is a random search algorithm based on group cooperation developed by simulating the foraging behavior of birds. It aims to optimize a problem by iteratively improving the candidate solution and returning the best-known solution. It is metaheuristic because it makes few or no assumptions or prerequisites about the problem. Rather than initializing a single candidate solution, it solves the problem by initializing a specified number of candidate solutions and gradually improving these solutions. The improving process for one particular solution is not only based on its local best-known solution but also the best solution tried in the searching space, which makes it more holistic and effective. Hence, PSO is more likely to make the search jump out of the local optimization. Besides, PSO can reduce the algorithm execution time because the searching operations can be executed on multicore CPUs in parallel. Therefore, PSO is fast and has been used for scheduling in parallel systems to accelerate the searching process.

3 Design

The design objective of HOME is to maximize DNN model training performance when the model's memory demand is larger than the GPU memory capacity. HOME allows tensor swapping and recomputing to reduce GPU memory footprint. Since existing approaches are biased because they only use partial DNN model information to make decisions, HOME leverages the PSO algorithm to holistically consider the model information in tensor placement. However, applying the PSO algorithm to the tensor placement problem is nontrivial because (1) the algorithm needs to guarantee a near-optimal solution in a limited searching space, and (2) evaluating a solution is difficult as the policy evaluation on a real hardware platform is time-consuming. In this

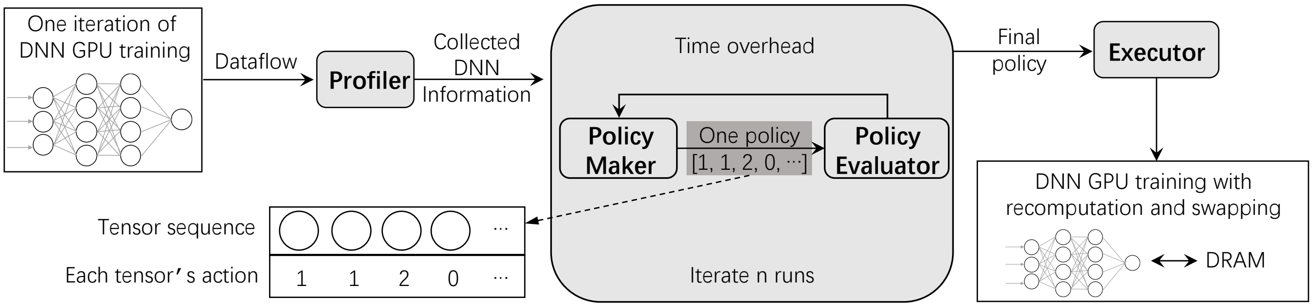


Fig. 3. The system overview of the HOME framework.

section, we illustrate the system architecture of HOME and elaborate its key components.

3.1 Overview of Architecture

We model the tensor placement problem as a n assignment problem and intend to solve it using a custom-designed particle swarm optimization algorithm. The goal of the PSO search is to decide placement sequence, i.e., 0 (retaining), 1 (swapping), and 2 (recomputation), according to the feature maps of all DNN layers, to minimize the time overhead of the whole DNN training process with the given GPU memory constraint.

Figure 3 shows the architecture of our system, which consists of the following four components. The **Profiler** is used to extract the required network and execution information. The **Policy Maker** iteratively decides the placement policy for all tensors with the information collected by the Profiler and the resource limitation of GPU memory. The **Policy Evaluator** estimates the overall DNN training time with the policy given by the Policy Maker. After multiple search iterations, the best tensor placement policy is fed into the Executor. Finally, the **Executor** conducts the actual tensor placement actions, i.e., swapping, recomputation, or retaining, according to the best policy in the subsequent iterations of the DNN training process.

3.2 Profiler

The Profiler is responsible for capturing the network information and the execution information for both the Policy Maker and the Policy Evaluator. Observing that DNN training consists of a large number of iterations and the tensor accesses have repeated access patterns across iterations [11], the Profiler is only executed online in the first iteration. The obtained information can be used to guide the tensor placements in the subsequent iterations.

The collected information includes the size of feature maps, the execution time of each kernel, the dependencies of feature maps, and the data transfer bandwidth between GPU and CPU. The size of each feature map is calculated according to the dimension of the feature map. The dependencies of feature maps are obtained according to the network structure. We add the timestamps in the original DNN framework to collect the execution time of each kernel. The data transfer bandwidth between GPU and CPU is evaluated with the performance test in the real system.

The profiling process incurs additional time overhead. However, it is executed only once in the first iteration. Such time overhead is acceptable since it can be amortized by the whole training process, which may be composed of millions of iterations. We will discuss this in § 4.7.

3.3 Policy Search with PSO

We leverage the PSO algorithm to determine tensor placement policy. In order to utilize the PSO algorithm, the first step is to formulate our tensor management problem as a mathematical assignment problem. As mentioned above, we use 0, 1, and 2 to denote three different tensor placement actions, i.e., retaining, swapping, and recomputation, respectively. In typical DNN frameworks, each layer of the DNN model has one feature map and the feature map is represented by one tensor. Then, placement assigned to all feature maps composes a vector, which we called the tensor placement policy of the DNN. As shown in Figure 3, the tensor placement policy [1, 1, 2, 0] represents swapping on the first two feature maps, recomputation on the third feature map, and retaining on the fourth feature map. Such a vector can be seen as a particle in a high-dimension searching space, whose dimension is equal to the number of total feature maps. Therefore, finding a tensor placement policy can be treated as finding the position of the particle, and PSO can be applied to get a near-optimal policy in the large search space.

Algorithm 1 shows the detailed process of solving the tensor management problem using PSO. First, we specify the number of search iterations (denoted as $iters$) for PSO and the number of particles (denoted as m) in one iteration. One particle represents one possible placement policy of all feature map tensors (shown as Figure 3) as defined before. The PSO algorithm will randomly initialize a specified number of particles (line #1-3) at the beginning. Then, PSO runs $iters$ iterations for searching for a better policy. In each iteration, the Policy Evaluator predicts each particle's time overhead (§ 4.6) and records the ever best particle which occurs the lowest overhead (line #6-12). After that, each particle will be updated by integrating the current and historical information of all the particles (line #13-15), which is a crucial step and will be discussed in detail in Section 3.5. Finally, the PSO algorithm returns the best-ever solution.

The complexity of Algorithm 1 is O(iters × m × n^2), where $iters$ is the number of iterations, m is the number of particles, and n is the number of DNN layers. While the n is a fixed number depending on the given DNN model, the $iters$ and m can be specified by users. Both the $iters$ and m can impact the algorithm convergence and searching time.

Algorithm 1 PSO search algorithm

```
Require: $iters$: the number of iterations for searching; $m$: the number of particles in one iteration;
Ensure: $best\_particle$: the best tensor placement policy;
for $i$=1, 2, ..., $m$ do
    $randomly\ initialize\ particle_i$
end for
$best\_particle \leftarrow [\,]$, $lowest\_overhead \leftarrow MAX$
for $i$=1, 2, ..., $iters$ do
    for $j$=1, 2, ..., $m$ do
        $overhead \leftarrow evaluate(particle_j)$          ▷ call Alg. 2
        if $overhead < lowest\_overhead$ then
            $best\_particle \leftarrow particle_j$
            $lowest\_overhead \leftarrow overhead$
        end if
    end for
    $particles\_update()$          ▷ call Alg. 3
end for
return $best\_particle$
```

Specifically, a smaller value for both $iters$ and m reduce the searching time but may not guarantee the algorithm convergence, which may degrade the DNN training performance; a larger value results in more searching time but brings better algorithm convergence, which can improve the overall DNN training throughput. The setting of the two values makes a trade-off between algorithm execution time and the DNN model training time.

To obtain the optimized training throughput with a limited algorithm searching time, we empirically set both the number of iterations (i.e., $iters$) and the number of particles (i.e., m) to 500 in our current design. Our evaluation shows that these settings are sufficient to serve our purpose (§ 4). Note that these empirical values can change with different DNN models and training parameters. However, these values have ensured that the near-optimal results can be achieved for all DNN models in our experiments within an acceptable searching time. For convenience and efficiency, we set both of them as 500 in our current design. However, using more refined parameters could be more beneficial for different DNN models. Therefore, one can adjust the hyperparameters to run Algorithm 1 based on the current settings. For example, if the training time of the DNN model with the current output policy of Algorithm 1 shows a declining trend, one can increase the number of the particles and the number of the iterations, until the DNN training time converges and the algorithm execution time is acceptable. We leave such exploration for future work.

3.4 Policy Cost Evaluation

All the tensor placement policies are fed into the Policy Evaluator to evaluate the time cost. To save time for a large number of real evaluations, the Policy Evaluator estimates the execution time of a policy using a time cost model instead of doing the real execution. Because the training iterations are repeatable, the Policy Evaluator uses the execution time of kernels to compute the overall execution time and evaluates the memory consumption by considering the size of the feature maps. However, the evaluation is difficult since the tensor placement policy is executed asynchronously in the DNN model.

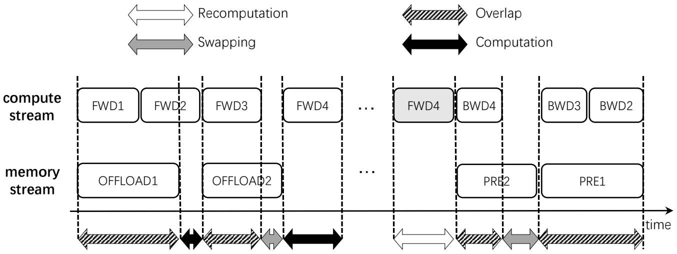


Fig. 4. The time overhead of recomputation and swapping.

To address this issue, we propose a cost model fully considering the complex data computing and memory access process to accurately predict the training time with a given tensor policy. Figure 4 represents the time overhead of recomputation and swapping in our cost model. To ease the following presentation, we define the following abbreviations: FWDi: the forward propagation of i-th layer; BWDi: the backward propagation of i-th layer; OFFLOADi: swap the tensor of i-th layer from GPU memory to CPU memory; PREi: pre-fetch the tensor of i-th layer from CPU memory to GPU memory; t(ACTi): the time cost of executing the action for tensor i.

As shown in Figure 4, OFFLOAD1 is overlapped with FWD1 and FWD2. In addition, $t(OFFLOAD1) < t(FWD1) + t(FWD2)$. OFFLOAD1 will not result in any extra time cost as the memory swapping and the regular forward computing are executed in parallel. However, the time of swapping out tensor2 ($OFFLOAD2$) is longer than the forwarding computation of the third layer ($FWD3$), thus the time overhead of swapping out tensor2 is $OFFLOAD2 - FWD3$. The backward propagation is counted from the last layer to the first layer and BWDi needs to access the tensor of the i-1 layer. As a result, the tensor2 is swapped in before executing BWD3 and the time overhead is $PRE2 - BWD4$. Therefore, the total time overhead of swapping tensor2 equals $(OFFLOAD2 - FWD3) + (PRE2 - BWD4)$.

Algorithm 2 shows the calculation of the time cost for a given tensor placement policy in our model. For a neural network composed of n layers, n placements will be determined. The policy (e.g., a list of actions of each feature map, denoted as $policy[n]$) determined by the policy Maker, and the data dependency info in the forward and the backward process (denoted as $Inputsforward[n], Inputsbackward[n]$ respectively) collected by the Profiler, are inputs to the Policy Evaluator for Algorithm 2. Specifically, HOME conducts the estimation in forward propagations and backward propagations (line #2-21). First, when the memory demand exceeds the capacity, HOME swaps the tensors from GPU memory to the CPU memory (line #3-9). Then, HOME executes the computation propagation and conduct actions based on input policy ($policy[n]$) and the data dependencies ($Inputsforward[n]$) (line #11-18). After that, the Policy Evaluator updates the profiled execution time and the memory consumption (line #19). As the result in $\S$ 4 shows, the Policy Evaluator can accurately predict the real training time of a DNN model.

Algorithm 2 Evaluate the overhead of one iteration

```
Require: $n$: the numbers of layers; $policy[n]$: a list of actions given to each tensor; $Inputs[2n]$: the data dependencies for each stages; $M$: total GPU memory capacity;
Ensure: $executime$: overall training time under such policy; $overhead[n]$: overhead of each training stage;
$executime \leftarrow 0$, $overhead[n] \leftarrow 0$, $mem \leftarrow 0$
for $i$ = 1, 2, ..., $2n$ do    ▷ 2n stages in forward propagations and backward propagations
    while $mem > M$ do
        if $swapout\_lists.empty()$ then
            return $error$
        end if
        $x = swapout\_lists.front()$
        $overhead[x]{+}{=}syncoverhead(x)$
        $updatemem(mem)$
    end while
    $inputs = Inputs(i)$
    for $input\ in\ inputs$ do
        if $policy[input] = swap$ then
            $overhead[input]+ = swap(input)$
        end if
        if $policy[input] = recompute$ then
            $overhead[input]+ = recompute(input)$
        end if
    end for
    $updatetime(executime)$, $updatemem(mem)$
end for
return $executime, overhead[n]$
```

Algorithm 2 has the polynomial time complexity O(n), where n is the number of layers in the DNN model. As current DNN models only have a limited number of layers, the time overhead of Algorithm 2 is acceptable. We will discuss this with experiments in Section 4.6.

3.5 Policy Upadate with PSO

PSO is initialized as a group of random particles (random solution), and then iteratively finds the optimal solution. In each iteration, the particle updates itself by tracking two extremums. In the PSO algorithm, each particle represents a tensor management policy.

Algorithm 3 shows the update process of the PSO algorithm. The goal of PSO is to use the historical placement information to guide the current placement policy for each tensor in all particles in the current iteration. To this end, the algorithm maintains a global last iteration policy, which records the best placement policy among all particles in the last iteration, and multiple best historical placement policies, which record the best placement policy for each particle in the past iterations. For a given particle, the algorithm first gets the best last iteration policy and the best historical policy (i.e., P_1 in line #1, and P_2 in line #3), respectively. Then, for each tensor in the current particle, the algorithm tries to replace the tensor's placement action with the historical action in P_1 and P_2 respectively, and evaluates their corresponding training overhead (O_1 and O_2 in Line #5-8). Finally, the algorithm updates the tensor's action with the historical action that causes the smaller overhead (Line #9-12). Multiple new actions for all the tensors make a new placement policy for the current particle. The algorithm will repeat this process until the policies for all particles are found.

Algorithm 3 Particles update

```
Require: $m$: the number of particles in one iteration; $n$: the numbers of layers; $policy[m][n]$: a list of actions given to each tensor;
 1: $P_1[n]$ = $getLastIterBest()$    ▷ get the best policy in the last interation
 2: for $i$ = 1, 2, ..., $m$ do    ▷ m particles
 3:     $P_2[n]$ = $getHistoricalBest(i)$    ▷ get the best historical policy of particle i
 4:     for $j$ = 1, 2, ..., $n$ do    ▷ n layers (tensors)
 5:         $policy[i][j]$ = $P_1[j]$    ▷ set the action for tensor j
 6:         $O_1 \leftarrow$ evaluate($particle_i$)    ▷ call Alg. 2
 7:         $policy[i][j]$ = $P_2[j]$
 8:         $O_2 \leftarrow$ evaluate($particle_i$)
 9:         if $O_1 < O_2$ then
10:             $policy[i][j]$ = $P_1[j]$    ▷ update policy
11:         else
12:             $policy[i][j]$ = $P_2[j]$
13:         end if
14:     end for
15: end for
```

Figure 5 shows an example of updating the placement policy of the first particle in the third iteration. In this example, three particles are assumed and the best placement policy in the last iteration among all particles is found in the second iteration of particle 2, i.e., [2, 2, 2, 0, . . .]. For particle 1, we assume its best historical policy is in the first iteration, i.e., [1, 1, 2, 0, . . .]. When updating the placement policy for the particle 1 in iteration 3, we first use the policy in the last iteration for particle 1, i.e., [0, 1, 2, 0, . . .] as the initial policy. Then, we try to update this policy by replacing each tensor's action. For the first tensor, since the action in the best last iteration policy and the historical policy are 2 and 1 (the underlined number in the policy) respectively, we will update this tensor's action (i.e., cur) with the historical actions accordingly and generate two candidate policies: [2, 1, 2, 0, . . .] and [0, 1, 2, 0, . . .]. Next, we evaluate the corresponding overheads with these two policies and choose the policy with the lower cost. For simplicity we only illustrate the action update of the first tensor and the procedures for other tensors are similar.

The time complexity of Algorithm 3 is O($m \times n^2$), where m is the number of particles in one iteration and n is the number of layers. m is a specified parameter and n depends on the DNN model. As these two parameters are a limited integer, the time overhead is acceptable ($\S$ 4.7). Algorithm 3 also incurs extra space overhead to store the historical placement policies. Each tensor has three actions, which can be encoded with two bits. In our current design, there are 500 particles. Hence, for a DNN model with even 10,000 layers, the space overhead is about 1.25MB, which is negligible.

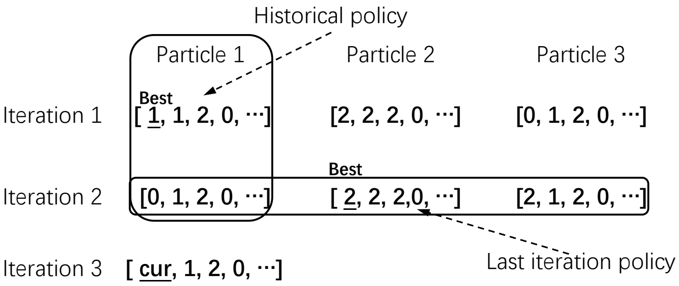


Fig. 5. An example of tensor placement policy update. The historical policy set: the best historical policies for all particles (each particle has its corresponding policy). The last iteration policy: the best placement policy among all particle policies in the last iteration (only one).

3.6 Policy Executor

The Executor performs the actual tensor management according to the final tensor placement policy in the real DNN training. The final tensor placement policy generated by the Policy Maker is stored as a policy table (PT) in a given location and records the action of each tensor. During the DNN training, the Executor will look up the PT from the given location and find the corresponding action for the current tensor. Then the Executor manages the memory of the current tensor (i.e., swapping, recomputation, or retaining) according to the guide of the action. Specifically, if the action is swapping, the Executor will transfer the tensor to CPU DRAM when the tensor is no longer needed in the forward propagation. The tensor will be prefetched into the GPU memory when its corresponding backward propagation begins. If the action is recomputation, the Executor will release the tensor immediately when the tensor is no longer needed in the forward propagation. The tensor will be recomputed in GPU when its corresponding backward propagation begins. Otherwise, the Executor will keep the tensor in the GPU memory.

3.7 Implementation

To demonstrate the performance of HOME, we implement its prototype in PyTorch 1.5.1. Overall, HOME is composed of four main components: Profiler, Policy Maker, Policy Evaluator, and Executor.

We develop the Profiler, named *begin_profile()*, to profile the DNN execution information, including the size of each tensor and the execution time of each layer during the first iteration. In the profiling period, we swap all tensors except the one in the tested layer from GPU memory to CPU memory, so that the GPU memory is large enough to hold the tensor for DNN training. This method can minimize the memory footprint because only one tensor resides in GPU memory.

For the Policy Maker, we use 16 threads to perform parallel searching to accelerate the policy-making process.

For the Policy Evaluator, to reduce the swapping overhead, we create a new asynchronous cuda stream using *cudaStreamCreateWithFlags()* and use *cudaMemcpyAsync()* to hide swapping into normal DNN training. To ensure the correct order of tensor access during backward, we use a flag queen, *back_flag*, to store the status of each tensor indicating whether it has been swapped or not. If the tensor has not yet swapped, we must use *CudaDeviceSynchronize()* to wait for the finish of swapping. As for recomputation, we create *recompute()* to generate the needed tensors. However, swapping and recomputation need frequent GPU/CPU memory allocations/frees which decrease the performance severely because of the inefficient default cuda *cudaMalloc()* and *cudaMallocHost()* functions. To mitigate such overhead, we use PyTorch self memory pool functions,*getCUDADeviceAllocator()* and *getPinnedMemoryAllocator()* to reduce the overhead of memory allocation and free.

After that, we develop the swapping and recomputation in the Executor according to the final optimized tensor placement policy.

4 Evaluation

In our evaluation we aim to answer the following questions:

- How does HOME perform across several DNN models? (§4.2)
- Why does HOME achieve the improved throughput with the limited GPU memory space? (§4.3)
- What is the maximum batch size of HOME? (§4.4)
- How about the convergence of the PSO algorithm?(§4.5)
- How about the accuracy of the evaluator? (§4.6)
- How about the overhead incurred by HOME? (§4.7)

4.1 Experimental Setup

Experimental platforms. We set up our experimental platform using a CPU-GPU hybrid server equipped with two 2.10 GHz Intel(R) Xeon(R) Gold 5218R CPUs, 128 GB main memory, and an RTX 2080Ti GPU with 11 GB GPU memory. The CPU and GPU are connected via the PCIe 3.0×16 bus. In addition, Ubuntu-18.04, CUDA 10.0.13 [29], CuDNN 7 [30], and PyTorch 1.5.1 [31] software packages are installed on the server.

Workloads and datasets. To validate the effectiveness of our approach, we evaluate HOME with three *linear* DNN models (i.e., VGG16 [32], Plain20 [33], and MobileNet [34]) and three non-linear models (i.e., ResNet [35], SqueezeNet [36], and InceptionV3 [5]). We set up our experimental platform using tuned model parameters (e.g., learning rate and optimizer) as [31]. In addition, we use CIFAR10 [37] as the dataset for evaluation. The CIFAR10 dataset is a collection of 60, 000 labeled color images (32 × 32 pixels each). The size of CIFAR10 is around 162MB.

Compared Baselines. We compare HOME with the following four baselines for tensor management in GPUs.

- **PyTorch**: This is the original PyTorch framework without any memory management optimization, such as swapping and recomputation. We choose PyTorch to verify the efficiency of other state-of-the-art tensor management methods with data swapping or recomputation.
- **vDNN**: This is one of the state-of-the-art tensor management approaches only using data swapping [14]. It offloads the convolution input tensors from GPU memory to CPU memory during forwarding training and prefetches them back in backward propagation

through overlapping data swapping with computation.

- **SuperNeurons**: SuperNeurons utilizes both swapping and recomputation for tensor management [19]. However, it applies a specific policy to a specific layer. Specifically, it only swaps the tensors of convolution layers and only recomputes the tensors in pool, activation, local response normalization, and batch normalization layers. We reimplement SuperNeurons as the description in the paper. Before DNN training, SuperNeurons will scan the architecture of the model and record the types of all layers in the DNN model. During training, SuperNeurons triggers different policies for tensors based on the pre-defined rules. For convolution layers, it recalls CUDA asynchronous memory copy API (i.e., cudaMemcpyAsync) to swap tensors. For pool, activation, local response normalization, and batch normalization layers, SuperNeurons releases the tensor immediately once the forward propagation in the given layer is finished and recomputes them back when needed in the backward propagation.
- **Capuchin**: It also applies tensor swapping and recomputation to reduce memory footprint [18]. However, it finds the tensor placement policy in a local optimum way as only part of the tensor information is utilized in the decision making. Specifically, it iteratively searches the candidate policy to achieve the best performance only for the tensor(s) from the first layer to the current layer while ignoring the tensors in the latter layers. Hence, it can't achieve globally optimized performance for all layers. We reimplement the Capuchin as the description in the original paper. Capuchin assigns a group of action pairs for each DNN layer, such as $\{tensor\_id : action\_id\}$, where $tensor\_id$ denotes the tensor number and $action\_id$ represents this tensor's policy. For each DNN layer, Capuchin scans the actions and performs the related functions to swap (i.e., *cudaMemcpyAsync*), release (i.e., *cudaFree*), or recompute tensors.

4.2 Experimental Results

We first evaluate the training throughputs of different tensor management frameworks with varying batch sizes. Figure 6 shows the results for different DNN models and we have the following observations.

First, compared to the original PyTorch, HOME allows a larger batch size and maintains a comparable or even higher training throughput. Taking MobileNet as an example, Figure 6(a) shows that PyTorch can only run a maximum batch size of 2100 while HOME can increase the batch size to 3500 and achieves the throughput by up to 2×, compared to PyTorch. HOME supports a larger batch size training because it can release some memory space occupied by tensors at runtime via data swapping or recomputation. Similarly, vDNN, SuperNeurons, and Capuchin also allow larger batch sizes than PyTorch. HOME achieves higher throughput than PyTorch because when more GPU memory is freed by swapping or recomputation, HOME can accommodate a larger batch size. A larger batch size means that more samples are fed into GPU to train. Since GPUs usually assign one core to train one sample, more samples will require more GPU cores to run, which will improve the utilization of the GPU and accelerate the DNN training process.

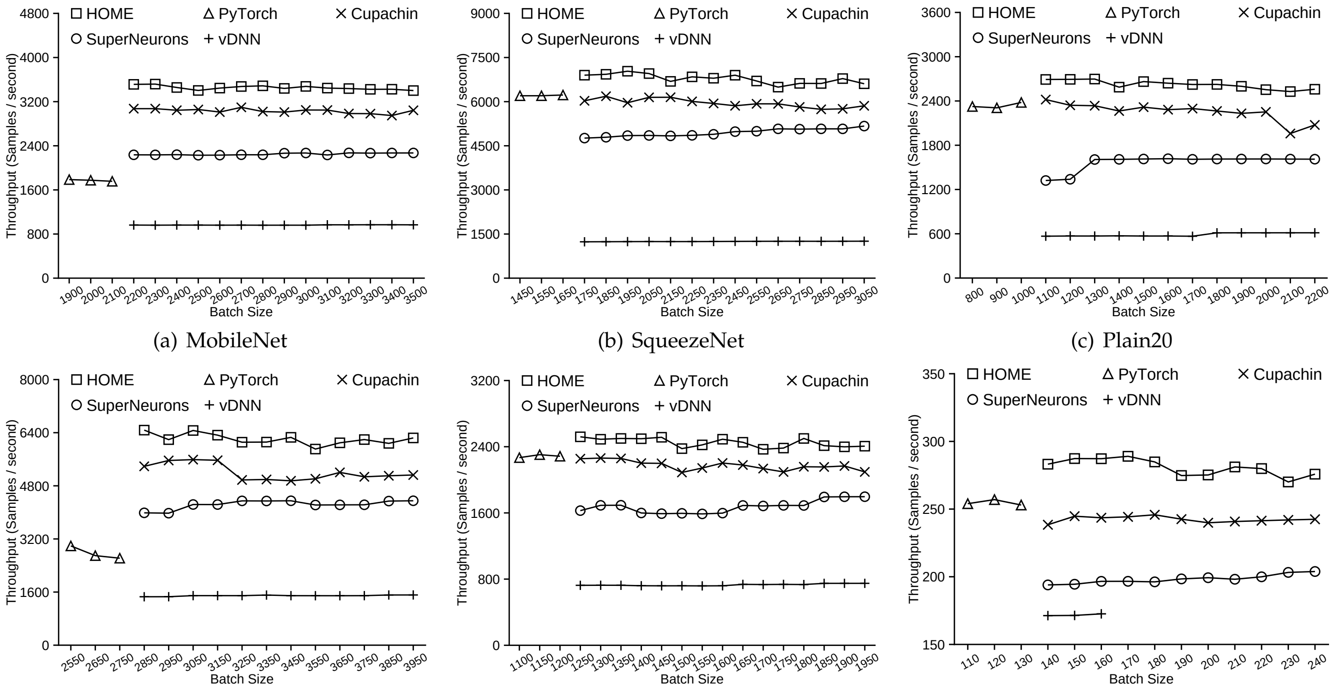
Second, among all the other state-of-the-art methods, HOME achieves the highest throughput under the limited memory space in GPU. More specially, under the same batch size, HOME achieves up to 5.7×, 2×, and 1.3× training throughput compared to vDNN, SuperNeurons, and Capuchin, respectively. HOME outperforms vDNN due to the following two facts. First, HOME adopts an asynchronous swapping policy while vDNN uses a synchronous policy, leading to more swapping time overhead. Second, HOME uses both data swapping and recomputation method while vDNN only considers the data swapping, losing the potential of data recomputation. Although SuperNeurons also applies both swapping and recomputation to reduce memory footprint, it specifies only one policy for a specific type of layer rather than all of them, which makes the decision simple but may hurt the training throughput. HOME also has a higher throughput than Capuchin because Capuchin finds the tensor placement policy in a local optimum fashion (it makes decisions only considering the tensors from the first tensor to the current one without considering the subsequent layers) while HOME searches for a policy from a global perspective.

Third, HOME brings different performance improvements for various DNN models. For example, HOME improves throughput by up to 2× over PyTorch for MobileNet (Figure 6(a)) and 2.46× for VGG16 (Figure 6(d)) while the improvement is averaged 1.11× for SqueezeNet (Figure 6(b)), Plain20 (Figure 6(c)), ResNet(Figure 6(e)), and InceptionV3 (Figure 6(f)). The improvements change because the performance benefits and time overheads from HOME differ for various models. For MobleNet and VGG16, the performance benefit brought by HOME owing to larger batch size obviously outweighs the time overhead introduced by swapping and recomputation. However, for the other four models, the benefit is comparable to the time overhead.

Finally, we also notice that vDNN fails to improve the batch size larger than 160, as shown in Figure 6(f). This is because vDNN only swaps CONV layers that occupy a small portion of GPU memory in Model InceptionV3. When the batch size is larger than 160, vDNN will also face the out-of-memory issue. For other methods, the batch size can be further increased.

4.3 Performance Breakdown Analysis

To further understand the impact of swapping and recomputing techniques in HOME and the three baselines (vDNN, SuperNeurons, and Capuchin), we collect the execution time details in different frameworks. We break down the whole training time into four parts according to Figure 4. The *Recomputation* time means the overhead for recomputation, the *Swapping* time denotes the time cost caused by swapping synchronization, the *Overlap* time means the overlapped time between swapping and normal



(a) MobileNet

(b) SqueezeNet

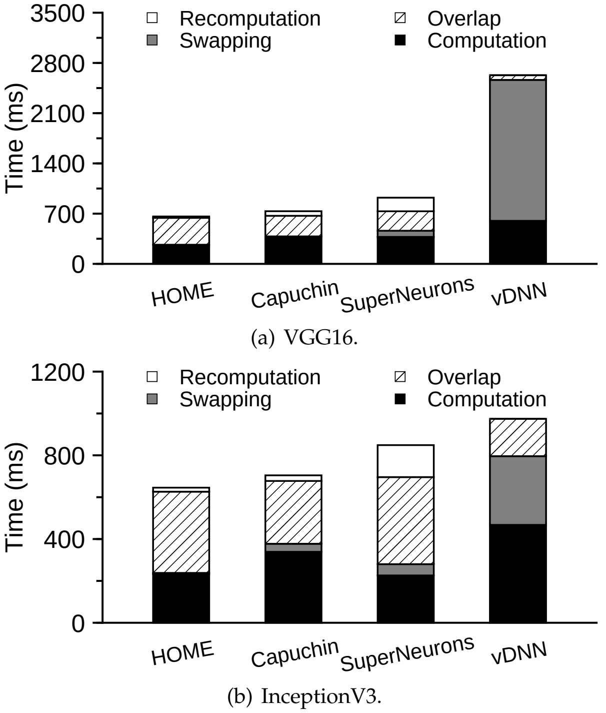
(c) Plain20

(d) VGG16

(e) ResNet

(f) InceptionV3

Fig. 6. Throughputs of different tensor management frameworks with varying batch sizes for different DNN models on CIFAR10.



(a) VGG16.

(b) InceptionV3.

Fig. 7. Performance breakdown analysis of different frameworks for VGG16 and InceptionV3.

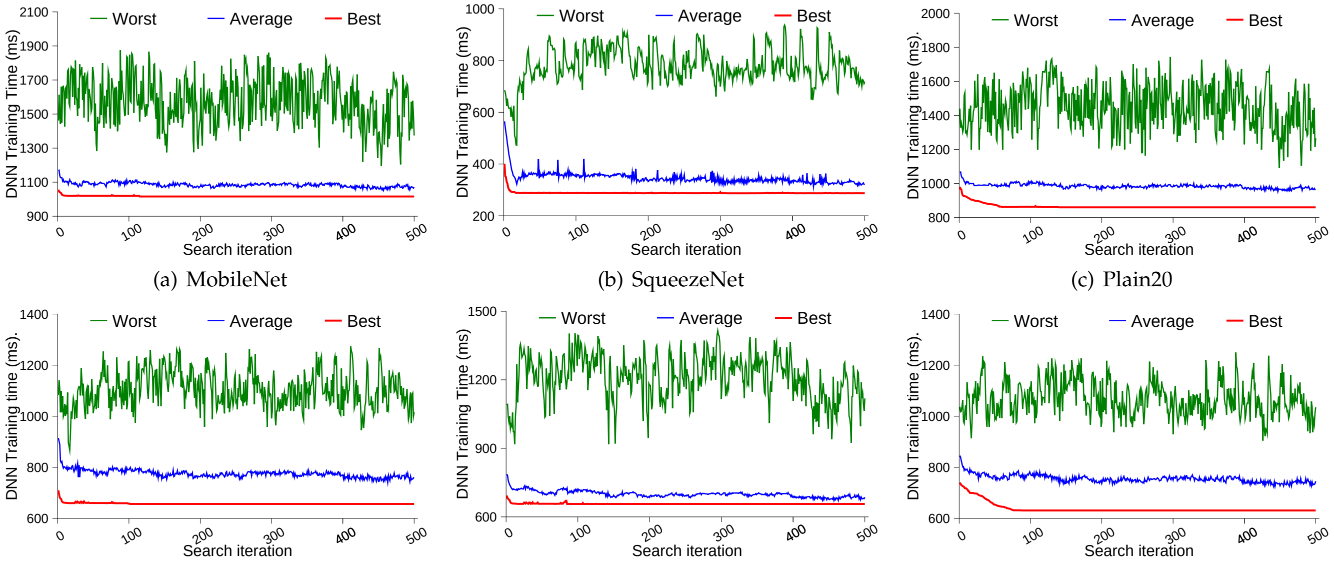
DNN computation, and the *Computation* time represents the original DNN training time without overlapping.

Figure7(a) shows the time breakdown for VGG16 with a batch size of 3950. We can observe that HOME achieves the lowest overall time mainly because of its relatively high overlap time, low swapping time, and small recomputation time. For example, the overlap time of HOME accounts for 57% and the swapping time of HOME is nearly zero in the whole training process, which means all the swapping operations are overlapped with the model computation and the swapping overhead is very low. While Capuchin has a close breakdown time distribution to HOME, it achieves worse overall performance than HOME because HOME has more optimized scheduling of tensor swapping and recomputation. Compared to HOME and Capuchin, SuperNeurons performs worse because it has relatively high recomputation overhead (20.5%) and swapping overhead (9%). For vDNN, the swapping overhead is high (i.e., 75%), which means vDNN spends a lot of time waiting for data synchronization, thus it has the lowest performance.

We also experiment on a larger DNN model, InceptionV3, with a batch size of 170. The result is shown in Figure 7(b). The results are similar to that of VGG16 except for the two new findings. First, different from the zero swapping time for VGG16 in Figure 7(a), HOME and Capuchin have 1% and 5% swapping time respectively for InceptionV3. This is because that InceptionV3 has a larger model size, thus it is difficult to overlap all swapping operations with the normal computation. Second, the swapping overhead of vDNN decreases from 75% (i.e., in VGG16) to 34%. This is because InceptionV3 has relatively smaller convolution layer inputs than VGG16. Thus, InceptionV3 needs less swapping time.

4.4 Maximum Batch Size of HOME

To analyze the effectiveness of HOME in reducing memory footprint, we evaluate the maximum training batch sizes allowed by different tensor management frameworks across the six DNN models. As Capuchin has shown its superiority



(a) MobileNet (b) SqueezeNet (c) Plain20

(d) VGG16 (e) ResNet (f) InceptionV3

Fig. 8. The legend ”Worst”, ”Average”, and ”Best” denote the training time with the worst policy, the average training time with all candidate policies, and the training time with the best policy selected by PSO.

TABLE 2
Comparison of the Maximum Batch Size (samples) and Training Throughput (samples/second) among PyTorch (PyT.), Capuchin(Cap.) and HOME.

Models	Maximum Batch Size			Training Throughput		
	PyT.	Cap.	HOME	PyT.	Cap.	HOME
MobileNet	2012	3954	**4200**	1706	2999	**3426**
Plain20	1040	2843	**2926**	2368	2012	**2554**
SqueezeNet	1667	4056	**4347**	6273	5719	**6613**
VGG16	2759	4250	**5125**	2786	5120	**6211**
ResNet	1255	2014	**2576**	2285	2179	**2519**
InceptionV3	125	345	**365**	257	240	**280**

over vDNN and SuperNeurons in previous experiments, we only compare HOME to the original PyTorch and Capuchin in this section.

Table 2 shows the allowed maximum batch size and corresponding training throughput of different frameworks under the same GPU memory capacity (i.e., 11GB in NVIDIA 2080Ti). We observe that both HOME and Capuchin can support larger batch size training than the original PyTorch framework. Specifically, HOME and Capuchin achieves averaged 2.2× and 1.97× batch size increments than the original PyTorch. This is because both HOME and Capuchin apply tensor swapping and recomputation, which can free GPU memory and allow larger batch size training. We also find that HOME always achieves the largest maximum batch size while maintaining the highest training throughput. In particular, HOME achieves a 1.12× larger batch size and 1.2× higher throughput than Capuchin on average for all the six models with the largest batch size. The throughput improvement comes from the efficient policy and higher resource utilization because the large batch size saturates the GPU.

4.5 Convergence of the PSO Algorithm

The PSO algorithm is an iterative process. In this section, we conduct experiments to evaluate its convergence for different models. We set the maximal search iteration as 500 and the number of particles to 500 in the PSO algorithm. This parameter setting is good enough for PSO to find an optimized tensor placement policy within several minutes (the overhead details of PSO are shown in §4.7).

Figure 8 shows the convergence results for the six popular DNN models as mentioned in Section 4.1. We conduct the experiments multiple times and show the average training time. From the figure, we have the following three observations.

First, it only takes dozens of iterations for HOME to find a modest policy. For example, 16 iterations (i.e., 16 × 20 ms = 320 ms) and 30 iterations (i.e., 30 × 18 ms = 540 ms) are enough for HOME to be close to the final optimization solution for ResNet and SqueezeNet. This is because the algorithm converges quickly with many initiated particles. Second, although the efficient policy can be found within the beginning dozens of iterations, HOME still optimizes the policy in the subsequent iterations. For example, in Figure 8(b), the training time difference between the average line and the best line is decreased from 30% at the beginning to 13% at the end of the training. This shows the PSO algorithm has good convergence. Third, the large DNN models need more searching iterations to converge. For example, InceptionV3 needs 89 iterations to find the optimized policy, as shown in Figure 8(f), while other models only need 42 iterations on average. This is because InceptionV3 has a larger searching space since it has more layers than other models, thus it needs more trials to find the near-optimal policy.

4.6 Evaluator Accuracy

The Policy Evaluator estimates the overall training time of a given tensor policy generated by the Policy Maker. In this

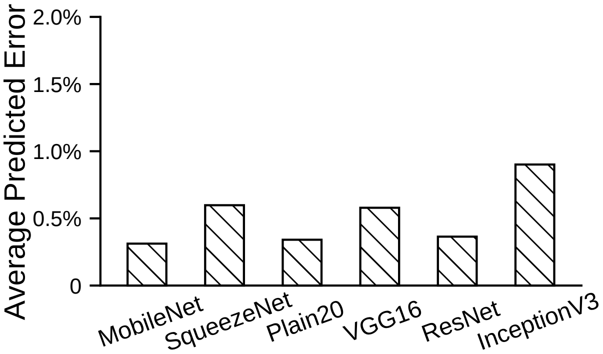


Fig. 9. Prediction accuracy of the evaluator for different DNN models.

section, we evaluate the prediction accuracy of the policy evaluator. We use the predicted error between the measured training time and predicted training time as the metric to indicate the prediction accuracy. We train each model 50 times on CIFAR10 with different batch size and calculate the average predicted error by using $\frac{\sum_{i=1}^{N}\frac{|\hat{y_i}-y_i|}{y_i}}{N}$, where N is the number of evaluations, $\hat{y}_i$ and y_i are the predicted and measured values, respectively.

Figure 9 shows the accuracy results for all the six DNN models. We can find the prediction errors for all the models are less than 1% (0.5% on average). For example, the prediction time on VGG16 is 546 ms and the actual training time is 543 ms, meaning only a 3 ms deviation. For Inception, the prediction error is 0.9%, which is the maximum among all the six models.

4.7 Overhead Discussion

While boosting the DNN training throughput, HOME also introduces the following overheads.

Profiling overhead. In the first iteration of the DNN training, the profiler needs to profile some required information, including the size of each tensor, the execution time of each layer, and the dependencies of feature maps, which causes time overhead. However, the overhead is negligible as the profiling is only conducted once, which can be amortized over the long training process. For example, the profiling for the InceptionV3 model takes about 10 ms and the whole training process takes several hours.

Search overhead. The PSO algorithm iteratively finds the optimal tensor placement solution in an offline way. This search process incurs time overhead. In our current design, we set the number of iterations to 500. The second column of Table 3 lists the time overhead of the entire search process for the six DNN models. We find that all models finish the whole search process within 6 minutes. Compared to the whole DNN training which takes several hours or even days, the search overhead is acceptable and can be amortized over the training.

Evaluation overhead. The third column of Table 3 lists the execution time of the Policy Evaluator for different DNN models. We find that all models only need 4.7 ms on average to estimate the DNN training time in one iteration while spending nearly a hundred milliseconds for one iteration training on real systems (e.g., 1.4 ms versus 700 ms on VGG16). Therefore, such overhead is acceptable.

TABLE 3
The time overhead that HOME incurs during DNN training.

Models	Search Time (s)	Execution Time of Policy Evaluator (ms)
MobileNet	28	1.7
Plain20	41	2.2
SqueezeNet	9	0.5
VGG16	25	1.4
ResNet	10	0.6
InceptionV3	350	21.8

5 Related Work

5.1 Data Parallelism and Model Parallelism

Due to the limited GPU memory capacity, extensive studies have been conducted to utilize parallelism techniques for training large-scale neural networks. Typical techniques include data parallelism, model parallelism, and both. With data parallelism, deep learning systems distribute a large batch to multiple GPUs to reduce memory footprint on one GPU [31], [9], [38]. Different from data parallelism, model parallelism spreads out the total neural network and parameters to multiple GPUs. [39], [40] choose to leverage both data and model parallelism to resolve the GPU memory shortage issue. However, both data parallelism and model parallelism require additional devices to reduce the memory footprint. Therefore, HOME is orthogonal to these approaches.

5.2 DNN Model Compression

Because DNN models are over-parameterized [41], several works utilize precision reduction to remove the redundancy of parameters in neural networks [26]. For example, [42] assesses the impact of low-precision of three distinct data formats and finds that very low precision is sufficient for DNN training. Gist [12] exploits existing value redundancy and proposes a lossy encoding scheme to reduce memory footprint. To reduce network redundancy, quantization and pruning techniques are also widely used in the existing literature. [43] apply network quantization by reducing the number of bits required to represent model parameters. [27] leverage filter pruning and weight pruning to remove redundant network connections. These approaches save memory footprint but may lead to lower model accuracy.

5.3 Swapping and Recomputation

Existing work has also exploited memory optimization via data swapping and recomputation. vDNN [14] analyzes the features of different layers and chooses to swap convolution tensors for saving GPU memory. moDNN [15] introduces heuristics to schedule data transfers and uses profiling information to overlap computation and communication. These works ignore the potential of recomputation in reducing memory footprint. SuperNeurons [19] and Layup [44] combine data swapping with data recomputation to reduce GPU memory footprint. However, they only apply a fixed placement policy to a specific layer. This fixed rule may ignore other possible solutions for the specific layer, which may degrade system performance. Capuchin [18] also utilizes tensor swapping and recomputation but decides the tensor placement policy only according to the tensors from the first layer to the current layer, without considering the subsequent tensors, leading to sub-optimal performance. AutoTM [23] uses Integer Linear Programming (ILP) to make data transfer schedules to reduce CPU memory footprint during DNN training. However, as a brute-force approach, the time cost is unaffordable when the models become complex. In addition, AutoTM only considers swapping without taking the advantage of recomputation.

6 Conclusion

With the increasing sizes of DNN models, deep learning training usually faces the memory-capacity-wall issue on GPUs. This problem is exacerbated by emerging DNN applications with large inputs. There are numerous memory management approaches to address this issue through data swapping and/or recomputation. However, these approaches mainly focus on reducing memory footprint but may suffer sub-optimal training performance because they only consider partial model information in making tensor placement decisions. In this paper, we propose a novel GPU memory management framework, HOME, which holistically considers the DNN model information to optimize the tensor placements. HOME explores the vast search space through the particle swarm optimization algorithm and chooses the best policy for DNN training. HOME intelligently selects swapping or recomputation, to maximize the training throughput within the memory capacity budget. Our experiments show that HOME can outperform vDNN and Capuchin by up to 5.7× and 1.3× in throughput. Furthermore, HOME can promote the maximum batch size by up to 2.8× than the original PyTorch and up to 1.3× than Capuchin. As deep learning applications become more and more popular, HOME will facilitate the successful running of some large DNN models on memory-constrained devices. This will inspire the next generation of memory management in deep learning frameworks and accelerate the development of deep learning in various fields.

As HOME relies on the profiler and heuristic accuracy, it is difficult for HOME to work well in multi-tenancy execution, transfer learning, and continuous learning. In the future, we will devise more efficient approaches to improve the performance of deep learning with memory constraints in such environments.

Acknowledgments

This work was supported in part by the National Key Research and Development Program of China No.2021ZD0110700, the National Science Foundation of China No. 62172361, the Zhejiang Lab Research Project No. 2020KC0AC01, and the Alibaba Innovative Research Project. Ping Chen is the corresponding author.

References

[1] C. Seifert, A. Aamir, A. Balagopalan, D. Jain, A. Sharma, S. Grottel, and S. Gumhold, "Visualizations of deep neural networks in computer vision: A survey," *Springer International Publishing*, 2017.

[2] X. Liu, P. He, W. Chen, and J. Gao, "Multi-task deep neural networks for natural language understanding," *arXiv preprint arXiv:1901.11504*, 2019.

[3] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *Proceedings of the 10th ACM conference on recommender systems*, 2016, pp. 191–198.

[4] U. Gupta, C.-J. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia *et al.*, "The architectural implications of facebook's dnn-based personalized recommendation," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, 2020, pp. 488–501.

[5] X. Xia, C. Xu, and B. Nan, "Inception-v3 for flower classification," in *Proceedings of the International Conference on Image, Vision and Computing*, 2017, pp. 783–787.

[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[7] Y. Gao, Y. Liu, H. Zhang, Z. Li, Y. Zhu, H. Lin, and M. Yang, *Estimating GPU Memory Consumption of Deep Learning Models*, 2020, p. 1342–1352.

[8] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter, "Nvidia tensor core programmability, performance & precision," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops*, 2018, pp. 522–531.

[9] G. B. Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vi, "TensorFlow: A system for large-scale machine learning," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2016, pp. 265–283.

[10] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *arXiv preprint arXiv:1912.01703*, 2019.

[11] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," in *Proceedings of the 22nd ACM International Conference on Multimedia*, 2014, pp. 675–678.

[12] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, "GIST: Efficient Data Encoding for Deep Neural Network Training," in *Proceedings of the International Symposium on Computer Architecture*, 2018, pp. 776–789.

[13] S. Jin, S. Di, X. Liang, J. Tian, D. Tao, and F. Cappello, "DeepSZ: A novel framework to compress deep neural networks by using error-bounded lossy compression," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, 2019, pp. 159–170.

[14] 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 *Proceedings of the Annual International Symposium on Microarchitecture*, 2016, pp. 1–13.

[15] X. Chen, D. Z. Chen, and X. S. Hu, "MoDNN: Memory Optimal DNN Training on GPUs," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2018, pp. 13–18.

[16] C. C. Huang, G. Jin, and J. Li, "SwapAdvisor: Push Deep Learning Beyond the GPU Memory Limit via Smart Swapping," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1341–1355.

[17] P. Jain, A. Jain, A. Nrusimha, A. Gholami, P. Abbeel, K. Keutzer, I. Stoica, and J. E. Gonzalez, "Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization," *arXiv preprint arXiv:1910.02653*, 2019.

[18] X. Peng, X. Shi, H. Dai, H. Jin, W. Ma, Q. Xiong, F. Yang, and X. Qian, "Capuchin: Tensor-based GPU memory management for deep learning," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 891–905.

[19] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska, "SuperNeurons: Dynamic GPU Memory Management for Training Deep Neural Networks," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2018, pp. 41–53.

[20] S. B. Shriram, A. Garg, and P. Kulkarni, "Dynamic memory management for GPU-based training of deep neural networks,"

in *Proceedings - 2019 IEEE 33rd International Parallel and Distributed Processing Symposium, IPDPS 2019*, 2019, pp. 200–209.

[21] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training Deep Nets with Sublinear Memory Cost," *arXiv preprint arXiv:1604.06174*, pp. 397–398, 2016.

[22] D. Yang and D. Cheng, "Efficient GPU Memory Management for Nonlinear DNNs," in *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, 2020, pp. 185–196.

[23] M. Hildebrand, J. Khan, S. Trika, J. Lowe-Power, and V. Akella, "AutOTM: Automatic tensor movement in heterogeneous memory systems using integer linear programming," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 875–890.

[24] R. Masadeh, A. Sharieh, S. Jamal, M. H. Qasem, and B. Alsaaidah, "Best path in mountain environment based on parallel hill climbing algorithm," *Int J Adv Comput Sci Appl*, no. 9, 2020.

[25] K. Amine, "Multiobjective simulated annealing: Principles and algorithm variants," *Advances in Operations Research*, 2019.

[26] P. Chen, S. He, X. Zhang, S. Chen, P. Hong, Y. Yin, X.-H. Sun, and G. Chen, "CSWAP: A Self-Tuning Compression Framework for Accelerating Tensor Swapping in GPUs," in *Proceedings of the 2021 IEEE International Conference on Cluster Computing*, 2021, pp. 271–282.

[27] S. Yang, W. Chen, X. Zhang, S. He, Y. Yin, and X.-H. Sun, "Autoprune: automated dnn pruning and mapping for reram-based accelerator," in *Proceedings of the ACM International Conference on Supercomputing*, 2021, pp. 304–315.

[28] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of the International Conference on Neural Networks*, 1995, pp. 1942–1948.

[29] CUDA, "Cuda toolkit 10.0 archive," 2018. [Online]. Available: https://developer.nvidia.com/cuda-10.0-download-archive

[30] CUDNN, "cudnn release 7.0," 2020. [Online]. Available: https://docs.nvidia.com/deeplearning/cudnn/release-notes/rel_7xx.html#rel_765

[31] PyTorch, "Pytorch/vision," 2020. [Online]. Available: https://github.com/pytorch/vision/tree/master/torchvision

[32] Karen Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arxiv: 1409.1556*, 2014.

[33] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, "Amc: Automl for model compression and acceleration on mobile devices," in *Proceedings of the European Conference on Computer Vision*, 2018, pp. 248–255.

[34] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv: 1704.04861*, 2017.

[35] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[36] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size," *arXiv: 1602.07360*, 2016.

[37] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," Tech. Rep., 2009.

[38] G. Lim, J. Ahn, W. Xiao, Y. Kwon, and M. Jeon, "Zico: Efficient gpu memory sharing for concurrent dnn training," in *Proceedings of the 2021 USENIX Annual Technical Conference*, 2021, pp. 161–175.

[39] Z. Jia, S. Lin, C. R. Qi, and A. Aiken, "Exploring hidden dimensions in parallelizing convolutional neural networks," *arXiv preprint arXiv:1802.04924*, 2018.

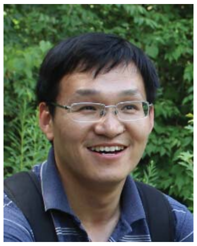
[40] M. Wang, C. chin Huang, and J. Li, "Supporting very large models using automatic dataflow graph partitioning," *arXiv preprint arXiv:1807.08887*, 2019.

[41] Y. Cheng, F. X. Yu, R. S. Feris, S. Kumar, A. Choudhary, and S. F. Chang, "An Exploration of Parameter Redundancy in Deep Networks with Circulant Projections," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 2857–2865.

[42] M. Courbariaux, Y. Bengio, and J.-P. David, "Training Deep Neural Networks with Low Precision Multiplications," *arXiv preprint arXiv:1412.7024*, 2014.

[43] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," in *Proceedings of the International Conference on Learning Representations*, 2016, pp. 91–104.

[44] W. Jiang, Y. Ma, B. Liu, H. Liu, B. B. Zhou, J. Zhu, S. Wu, and H. Jin, "Layup: layer-adaptive and multi-type intermediate-oriented memory optimization for gpu-based cnns," *ACM Transactions on Architecture and Code Optimization (TACO)*, pp. 1–23, 2019.



Shuibing He received the PhD degree in computer science and technology from Huazhong University of Science and Technology, in 2009. He is now a ZJU100 Young Professor with the College of Computer Science and Technology, Zhejiang University, China. His research areas include Intelligent computing, high-performance computing, memory and storage systems. He is a member of the IEEE and ACM.

Ping Chen is currently working toward the PhD degree with the College of Computer Science and Technology, Zhejiang University, China. His research focuses on intelligent computing, memory management for AI systems.

Shuaiben Chen received his MS degree in the College of Computer Science and Technology, Zhejiang University, China. His research areas include intelligent computing, systems for AI.

Zheng Li received his PhD degree in Computer Science from Illinois Institute of Technology, in USA. He is currently an assistant professor in the Computer Science Program, School of Business, Stockton University. His research interests include distributed computing, real-time computing, many-core computing, and reconfigurable computing.

Siling Yang is currently working toward the Phd degree with the College of Computer Science and Technology, Zhejiang University, China. Her research focuses on intelligent computing, processing-in-memory for AI.

Weijian Chen is currently working toward the Phd degree with the College of Computer Science and Technology, Zhejiang University, China. His research focuses on intelligent computing, storage and memory systems for AI.

Lidan Shou received the PhD degree in computer science from the National University of Singapore. He is a professor with the College of Computer Science and Technology, Zhejiang University, China. His research interests include spatial databases, data access methods, and big data analytics.