TENSILE: A Tensor Granularity Dynamic GPU Memory Scheduling Method Toward Multiple Dynamic Workloads System

Kaixin Zhang, Hongzhi Wang©, Han Hu, Songling Zou, Jiye Qiu, Tongxin Li, and Zhishun Wang

Abstract—Recently, deep learning has been an area of intense research. However, as a kind of computing-intensive task, deep learning highly relies on the scale of GPU memory, which is usually prohibitive and scarce. Although some extensive works have been proposed for dynamic GPU memory management, they are hard to apply to systems with multiple dynamic workloads, such as in-database machine learning systems. In this paper, we demonstrated TENSILE, a method of managing GPU memory in tensor granularity to reduce the GPU memory peak, considering the multiple dynamic workloads. TENSILE tackled the cold-starting and across-iteration scheduling problem existing in previous works. We implemented TENSILE on a deep learning framework built by ourselves and evaluated its performance. The experiment results show that TENSILE can save more GPU memory with less extra overhead than prior works in single and multiple dynamic workloads scenarios.

Index Terms—DB4AI, GPU memory schedule

1 INTRODUCTION

Deep learning is widely used in many areas of data analysis. With appropriate training, deep learning models can achieve much higher performance than traditional ones [1]. However, a significant problem of deep learning is that it is usually costly for GPU memory, especially in the communities of image recognition and natural language processing, whose models like ResNet [2], BERT [3], and GPT-3 [4] have as many as billions of parameters. These parameters and the corresponding feature maps cause massive GPU memory footprint [5], which impedes their application. For example, such deep neural networks(DNN) could hardly be performed in a database since building a large GPU cluster is prohibitive for the existing database systems. Thus, the increasing size of deep learning models acquires that the GPU memory must be managed more efficiently.

Additionally, many scenarios of deep learning have multiple dynamic workloads, such as cloud database services [6], in-database machine learning [7], [8], cloud deep learning services [9], [10], [11], and AI-driven DBMSes [12], [13], [14]. Although the memory occupied by a single load is limited, the total amount of running jobs is huge, so there needs to be enough GPU memory for each job. For example, prior work [11] mentioned that there are situations in that jobs cannot be co-located due to the GPU memory constraints in cluster scheduling problems. To tackle such problems under multiple dynamic workload scenarios, we propose TENSILE to reduce the total GPU memory peak.

The critical problem is how to reduce the GPU memory peak of each and entire computation process efficiently and update the scheduling plan in time to fit the dynamic workloads. Although there are already some efficient methods [15], [16], [17] to reduce the GPU memory cost by swapping and recomputation, the following three issues are still unsolved.

- **Multiple dynamic workloads.** To our best knowledge, most existing approaches are designed to schedule GPU memory for a single workload. When it comes to scenarios with multiple dynamic workloads, these methods can hardly manage the GPU memory as well as they were on a single job. Scheduling GPU memory with multiple dynamic workloads is challenging since the workloads are executed asynchronously, and such a mechanism will cause dynamic tensor access patterns. The core challenge is to update the scheduling plan with its fluctuation. Also, choosing the proper tensors that need to be scheduled is critical to scheduling efficiency. Furthermore, the challenge involves the design of the system architecture to support such kinds of scheduling algorithms.

- **Cold-starting.** The most advanced method, such as Capuchin [15] which schedules in tensor granularity, needs to observe the tensor access pattern first before generating the scheduling plan. As shown in Fig. 1b, such ‘Passive Mode’ will cause extra overhead by

---

© Kaixin Zhang, Hongzhi Wang, Han Hu, Songling Zou, Jiye Qiu, Tongxin Li, and Zhishun Wang are with the Computer Science and Technology, Harbin Institute of Technology, Harbin, Heilongjiang 150001, China. E-mail: {21b903037, 1180300713, 1182100101, 1180300717, 1180300221}@stu.hit.edu.cn, wangzh@hit.edu.cn.

Zhishun Wang is with the School of engineering, The Chinese University of Hong Kong, Hong Kong. E-mail: 1155164886@link.cuhk.edu.hk.

This work was supported by NSFC under Grant 62232005.

(Corresponding author: Hongzhi Wang.)

Recommended for acceptance by B. He.

Digital Object Identifier no. 10.1109/TKDE.2022.3221426
passively swapping tensors when the GPU memory is insufficient. Also, the multiple workload scenario causes the observation to be inaccurate. So the core challenge is to find a proper way to initialize the tensor access pattern.

- **Across-iteration Scheduling.** All prior methods only schedule within a single iteration. They ignore that scheduling across iterations is also necessary since some tensors evicted in the current iteration’s Opt phase are used as inputs in the next iteration’s F/B phase, as shown in Fig. 1, which leads to extra overhead. The core challenge is to consider the swapping between two phases.

In this paper, we tackle these challenges and propose TENSILE, a run-time dynamic GPU memory scheduling method of tensor granularity designed to reduce the GPU memory footprint peak towards multiple dynamic workloads.

For the first problem of multi-workloads, we develop a scheduling method that can keep tracking the tensor access latency for updating the scheduling plan to support the multiple dynamic workloads. By updating the scheduling plan in time, the dynamic tensor access patterns caused by multi-workloads scenarios can be well adapted.

To solve the second problem, we use a light neural network model to predict the latency of each operator under current GPU usage. With this model, TENSILE can avoid the extra overhead caused by the ‘Passive Mode’.

And for the third problem, our method could schedule tensors across iterations. TENSILE can swap out the parameters at the Opt phase and prefetch them before they are used in the following computation iteration. As the schematic Figs. 1b and 1c shows, such an approach can effectively reduce the total GPU memory peak with less extra overhead of passively swapping and supporting multiple dynamic workloads.

TENSILE is a tensor granularity scheduling approach for supporting deep learning tasks based on tensor computation, such as the training of CNN [18] and GCN [19]. Additionally, we can achieve higher performance with scheduling in tensor granularity since tensor is the most fundamental component of deep learning algorithms. Also, benefiting from the tensor granularity, TENSILE is not limited to the scope of application with forward/backward propagation like most prior works do. It can also handle any tensor computation tasks expressed with tensor operators in a static compute graph.

Our contributions can be concluded as:

- Inspired by existing GPU memory scheduling methods, we proposed a scheduling method toward multiple dynamic workloads. To support such scheduling, we designed a system to track the jobs’ running and manage their memory as a whole, rather than optimizing them separately as previous works did. This method can also schedule the tensors across iterations and reduce the entire process’s GPU memory peak to increase the system’s model capacity.
- We also proposed an innovative algorithm to generate the swapping and recomputation plan based on tensor access sequence, which can support the multiple dynamic workloads better than prior works. With this algorithm, we could initialize the scheduling plan without measuring first and update the scheduling plan in time with the fluctuation of GPU usage.
- According to the experiment results, TENSILE can achieve a higher GPU memory saving rate with less performance loss, especially in multiple dynamic workload scenarios.

2 RELATED WORK

In this section, we introduce some prior works in GPU memory management.

NVIDIA is the first organization to pay attention to the GPU management problem. It integrated the Unified Memory technology in CUDA6.\(^1\) This work allows programmers to use the GPU and Host Memory as a unified space. Unlike

---

\(^{1}\) https://developer.nvidia.com/blog/unified-memory-in-cuda-6/
TENSILE, this work is not specially designed to schedule deep learning processes, which implies programmers need to decide when and which tensor should be swapped with CUDA programming.

In the area of GPU memory scheduling for deep learning, Minsoo Rhu et al. proposed the vDNN [16], which is the first work focusing on scheduling the GPU memory footprint for deep learning processes. The central part of vDNN is an algorithm to decide which layer’s interim result should be released, evicted, or prefetched. Although it is a pioneering work, some problems remain, such as delayed computation start, high pinned memory requirements, and GPU memory fragmentation. Shriram S.B. et al. improved vDNN and solved the three problems mentioned above [20]. By allowing to overlap of the tensor eviction and prefetching process with the computation process, Shriram S.B. et al. significantly improved the vDNN’s performance. Linnan Wang et al. proposed SuperNeurons [17] which is the first dynamic GPU memory run-time management approach. It also introduced recomputation into the scheduling. The authors believe the convolution layer is the most valuable layer to be swapped and designed some approaches to schedule the convolution layers. Although this design helps it to achieve significant efficiency on convolutional networks, it also limits the compatibility for more advanced network structures as transformer [21]. Donglin Yang et al. proposed the first method that supports a unified memory pool for non-linear networks [5]. The method can construct the execution order for non-linear networks based on graph analysis so that it could support non-linear networks better than previous works.

Although all the methods above have significant advantages, they have a major shortcoming compared to TENSILE: all of them are designed to schedule GPU memory in layer granularity. Such a mechanism makes them lack compatibility and unable to save GPU memory as much as tensor granularity methods like TENSILE. This is the fundamental reason why they can not schedule the newest deep learning models that contain complex structures inside a ‘layer’ (or as known as ‘block’), such as transformer-based networks [21] and the Capsule Network [22]. And the tensors inside the ‘layer’ can not be scheduled by these methods, and neither do the interim tensors in the optimizer, such as the 1st moment vector and the 2nd moment vector in the Adam optimizer [23]. These works aim to improve the batch size in a single workload scenario, so only scheduling the tensors in the F/B phase is enough. However, we aim to reduce the memory footprint peak to make the system support more job training simultaneously when it comes to multiple workload scenarios. Moreover, such methods are not good enough since the GPU memory peak will be not only appear in the F/B phase but also in the Opt phase.

In 2019, another method was proposed [24], the first approach that can schedule GPU memory in tensor granularity. This method proposed two orthogonal approaches to reduce the memory cost from the system perspective. It reduces the GPU memory peak by managing the memory pool and scheduling the swapping with a heuristic method. Benefiting from its tensor granularity, this method can support any deep learning network. Xuan Peng et al. proposed Capuchin [15]. Capuchin can observe the tensor access pattern and decide which tensor to be swapped and which to be recomputed with a value-based approach. This method is more flexible in supporting various types of neural networks than those layer granularity methods. However, it highly depends on observing the tensor access pattern when the calculation is running for the first time since the tensor access pattern is needed to generate the scheduling plan. Such observation leads to passive swapping and PCIe channel occupancy when the GPU memory is insufficient.

These two advanced works schedule GPU memory in tensor granularity and solve compatibility problems. However, compared to TENSILE, neither of these prior works can solve the three problems we propose in Section I. We take Capuchin as an example to explain it.

In an environment with multiple workloads, influenced by multiple workloads, the GPU usage fluctuates irregularly, which causes the operators’ latency and tensor access pattern to keep changing, so the scheduling plan must be able to be kept updating iteratively. Capuchin makes a fixed scheduling plan once at the first epoch based on the observed tensor access pattern in the ‘Passive Mode,’ which is not feasible because the latency of the operator varies with the dynamic load of multiple jobs. Especially when a job launches, the GPU usage will change drastically and cause the initial scheduling plan to lose efficacy. Capuchin has not enough ability to adjust the plan to fit the fluctuation of the tensor access pattern in multiple dynamic workload scenarios. This can be proved by Fig. 2, the time cost of ‘wait for tensors ready’ increases with the number of workloads.

The ‘Passive Mode’ also causes the cold-starting discussed problem in Section I. According to the ‘first iteration cost’ of Fig. 2, this unexpected eviction causes the time cost of the first iteration increases since passive swap-out causes the computation to block.

Also, Capuchin and other works above only considered the tensor access in a single iteration. For example, the updated parameters’ accesses at the end of the iteration can not be swapped proactively. Thus, these parameters can only be swapped passively at the next iteration with an additional time overhead. As shown in Fig. 2, the passive eviction takes 32.1% of the overall time cost. Most of these evictions are caused by the memory peak caused by the unexpected swap-in across iterations.

### 3 Overview

This section defines the GPU memory scheduling problem and overviews our deep learning system built to support TENSILE.
For the convenience of scheduling, we use a graph model called static compute graph [25] to describe the tensor calculation jobs. Let \( V \) be the set of all basic operators to manipulate tensors, \( E \) be a set of tensors generated by operators in \( V \), and \( S \) be a set of scheduling events, whose elements are [Swap-out, Swap-in, Recompute, Release]. With these symbols, we define the dynamic memory workload scheduling problem:

**Job**: A deep learning job \( j \) is denoted by a static compute graph as \( G_j(V, E) \). Note that the graph is a DAG (Direct Acyclic Graph) just like the one in TensorFlow [25], and the operators of the optimizer are also included in \( G_j(V, E) \). An example of the graph is shown in Fig. 3a, a convolution operator is represented as a node in the graph. Such a node takes multiple tensors as the input and outputs other tensors. Such a model could effectively represent the computation in modern deep learning frameworks, such as TensorFlow [25].

**Tensor Access**: For a tensor \( x \) in \( V \), which is generated by the operator \( \mu \) and used by the operator \( \nu \), we define the execution of \( \mu \) as the Tensor Generating Access (TGA) of \( x \) and the execution of \( \nu \) is the Tensor Using Access (TUA) of \( x \). Both tensor accesses are denoted as \( a_{ij}^t \), where \( i \) is the access index. Usually, if a given operator \( \mu \) is not at the start or the end of a job, it corresponds to two kinds of tensor accesses, the TUA and the TGA. For example, as shown in Fig. 3b, a convolution operator takes tensors kernel, bias, and \( x \) generated by the previous operator as inputs and outputs the feature map tensor \( y \). The latency of a Tensor Access is defined as the corresponding operator’s execution time cost, which can be inferred from the compute graph or logged at runtime. The TUA is access of the input tensors of \( \mu \), and the TGA represents the accesses of the output tensors of \( \mu \). Additionally, since the operator placeholder generates the input tensors of the whole compute graph, we regard the placeholder as a TGA.

**Workload**: A workload \( W_j \) is denoted by a tensor access sequence \( \langle a_1^t, a_2^t, \ldots \rangle \), which is ordered by the topological order of job \( G_j(V, E) \). The topological order corresponds to the execution order of operators in \( V \). When a workload \( W_j \) is launched, the framework executes the operators of \( W_j \) in topological order. A dynamic workload has tensor accesses whose latency keeps changing at run time.

**Scheduling Plan**: We denote a scheduling plan for workload \( W_j \) as a sequence \( S_j = \langle s_{j1}^t, s_{j2}^t, \ldots \rangle \), such as \( s_{j1}^t, s_{j2}^t, \ldots > \) \( \vdots \). And for each event \( s_{j1}^t \), \( j \) represents that it belongs to the scheduling plan of workload \( W_j \), \( i \) is its trigger tensor access, which is the prior tensor access of the scheduled event, and \( t \) is the time interval between the execution end of the trigger tensor access and the start of the scheduling event, just as Fig. 3c shows.

**Memory Footprint**: Let function \( \text{Mem}(\cdot) \) be the GPU memory changing caused by tensor access \( a \) or scheduling event \( x \), the memory footprint of a workload \( W_j \) at the end of operator \( k \) is

\[
\text{MF}_j^k = \sum_{i=0}^{k} \text{Mem}(a_i^{'j}) + \text{Mem}(x_i^{'j}),
\]

and the memory footprint peak of workload \( W_j \) (with \( m \) operators) is

\[
\text{MP}_j = \max_{0 \leq k \leq m} \text{MF}_j^k = \max_{0 \leq k \leq m} \sum_{i=0}^{k} \text{Mem}(a_i^{'j}) + \text{Mem}(x_i^{'j}).
\]

The summation process in Equation (1) can be regarded as a discrete GPU memory occupancy change process as the workload’s tensor access sequence is executed. Assuming there are \( n \) workloads running asynchronously in the system, the global memory footprint peak is \( MP = \sum_{j=0}^{n} \text{MP}_j \).

**3.2 Problem Definition**

With these symbols above, we define the problem here. Let \( M \) be the GPU memory size, our multiple dynamic workloads scheduling problem is how to generate a scheduling plan \( X_j \) for each workload \( W_j \) to minimize the global memory footprint peak \( MP \) with as little extra overhead. So that the system with GPU memory \( M \) can run more neural networks simultaneously or run these given workloads with larger batch sizes for higher efficiency. The optimization goal is formalized as Equation (3) shows.

\[
J(X) = \arg \min_{\{X_0, \ldots, X_j\}} \sum_{j=0}^{n} \max_{0 \leq k \leq m} \sum_{i=0}^{k} (\text{Mem}(a_i^{'j}) + \text{Mem}(x_i^{'j})).
\]
3.3 Concepts Explanation

To facilitate understanding, we explain the basic concepts of TENSILE.

The Tensor Access Sequence is the alias of workload. For each tensor, its Tensor Access Sequence begins with a TGA which generates the tensor and followed by a series of TUA which represents the tensor used by operators as input, as shown in Fig. 1.

A Swap Event transfers data between the GPU and the Host Memory. It has two kinds of types, Swap-Out Event and Swap-In Event. As Fig. 3c shows, Swap-Out Event of a tensor will copy the tensor from the GPU memory to the Host Memory and then releases it from the GPU memory as soon as it is not occupied, which can be regarded as eviction. A Swap-In Event will copy a tensor from the Host Memory to the GPU memory, which is used to prefetch the tensor. It is necessary to emphasize that swap-in does not mean the tensor in the Host Memory will be released. On the contrary, it will remain in the Host Memory until the last access of the tensor is finished, which could save much cost of swap-out.

A Recomputation Event regenerates the tensors which have already been released from the GPU memory. For a Tensor Access $T_i$, we can release it after accessing and recompute it before the Tensor Access $T_j$ that takes the tensor $r$ as input.

A Release Event releases the corresponding tensor from the GPU memory as soon as it has not been used.

With these concepts, we can offload tensors when not used and then prefetch or regenerate them by swapping or recomputing them before their TUA.

3.4 System Architecture

Since existing approaches cannot schedule among multiple dynamic workloads, we developed a system to support scheduling under such scenarios. The system can collect the necessary information on all the jobs before and during running to support the scheduling algorithm. The information includes the compute graph of jobs, the latency of each operator, and other run-time information such as the GPU utilization. The system will trigger the scheduling algorithm multiple times during startup and running. When receiving the newest scheduling plan, the system will apply it at the next round of iteration. To achieve the functions above, our system contains four components: Global Controller, Memory Scheduler, Executor, and Swap Executor, as shown in Fig. 4.

Global Controller. The Global Controller is responsible for launching new jobs’ processes and delivering information between the Executor and the Memory Scheduler. When the user launches a new job, the Global Controller creates a new sub-process of the job’s Executor, and the threads for Swap Executor. It is also responsible for communicating with the Memory Scheduler, and each job’s information is organized and sent to the Memory Scheduler. Also, the scheduling plan of each job is distributed to the corresponding Executor. With the global controller, our system can support the scheduling algorithm described in Section IV.

Memory Scheduler. The Memory Scheduler takes the jobs’ information as inputs, such as the compute graph and the operators’ latency tracked by the Executor. Such information is used to generate the Tensor Access Sequence and the scheduling plan, which consists of multiple swapping, releasing, and recomputation instructions. The detailed algorithm will be described in Section IV. Each instruction is described by a tuple $(\text{trigger}, \Delta t)$, the trigger is the previous operator’s end time, and the $\Delta t$ is the time interval after the trigger. Therefore, the corresponding Swap Event will be triggered in $\Delta t$ after the trigger, just as the Fig. 3c shows. These instructions are grouped by job id and will be distributed to the corresponding Executor by the Global Controller.

Swap Executor. As Fig. 5 shows, the Swap Executor contains an Execution Queue and an Execution Buffer. The Execution Queue is a thread with an event queue in it. It receives the swap events triggered by a certain operator (trigger tensor) and pushes these swap events into the queue in chronological order. The Execution Queue pops up the swap events to the Execution Buffer for implementing swapping according to their time intervals.

Executor. The Executor executes each operator node in the graph by topological order. For each node, it checks whether all input tensors are in the GPU memory before the computation since the inaccuracy of the estimated Tensor Access Sequence may lead to delays in swapping. If that happens, the Executor will pause the computation and swap the tensors into the GPU memory. When the computation is finished, certain tensors will be released if the Memory Scheduler has set the operator as the release trigger for these tensors. Before releasing, the Executor checks the Swap Executor. The release and computation processes will be blocked if some Swap-Out Events have not been executed. By such a synchronous method, the execution order of swapping, releasing, and
computation can be kept. Otherwise, if a given tensor’s Swap-Out Event is executed later than its release, then the data of the tensor will be lost and would cause exceptions.

The Executor also tracks every operator’s latency periodically and sends the information to the Memory Scheduler through the Global Controller for updating the scheduling plan. The Executor will check whether the scheduling plan has been updated each time before a new round of calculation. During the execution of the compute graph, if the scheduling plan requires the current tensor access to trigger some Swap Events, the Executor will send the Swap Event to the Execution Queue of the Swap Executor waiting for execution. At the end of an iteration, the Executor pauses to wait for the unfinished swap events of this iteration.

The system can collect information from each job and execute the corresponding scheduling plan with these components. The whole scheduling procedure has four steps:

1) Collecting the new jobs information through the Global Controller.
2) The Memory Scheduler generates the initial scheduling plan and distributes it to the corresponding job’s Executor through the Global Controller.
3) The Swap Executor executes the scheduling plan during computing. Also, the Executor performs the recomputation events at running.
4) The Executor collects the time cost of each operator and reports to the Global Controller. When the current operators’ latency and the latency used to generate the previous scheduling plan deviate more than the preset threshold, the Global Controller will call the Memory Scheduler to update the scheduling plan based on the newest estimated Tensor Access Sequence.

4 METHODS

In this section, we detail our algorithms of TENSILE in the order of actual execution, i.e., Tensor Access Sequence Generation (Section 4.1), GPU Memory Peak Analysis (Section 4.2), Swap Event Scheduling Section 4.3), Recomputation Scheduling (Section 4.4), and Scheduling Plan Updating (Section 4.5).

4.1 Tensor Access Sequence Generation

We first introduce our tensor access sequence generation method since our method is based on GPU memory peak analysis, which depends on tensor access sequence generation. We use a learn-based latency prediction method to solve the cold-starting problem mentioned in Section 1. In consideration that there are two pieces of information required to infer the Tensor Access Sequence, which are the Tensor Accesses’ order and the latency of each Tensor Access. We can deal with this problem by diving into two sub-problems of inferring the Tensor Access order and getting each Tensor Access’s latency.

As mentioned above, we execute the operators in the topological order with the compute graph. Therefore, we can solve the first sub-problem and get the Tensor Accesses’ order statically based on each operator’s inputs and outputs. It only solves the first sub-problem, which is inferring the access order. The execution order is unrelated to the latency.

However, accurately predicting the CUDA kernel’s latency is a challenging problem, especially with multiple dynamic workloads.

The most straightforward way to solve the second sub-problem is to run the job with ‘Passive Mode’. However, it will cause extra overhead since calculation and swapping can not overlap, and the computation must wait for swapping. To make the initial scheduling plan more efficient, our solution uses pre-trained machine learning models to predict each GPU operator’s latency with the input data size and the current GPU utilization rates. When the system initializes, it will measure each GPU operator’s latency with different input data and GPU usage to generate the training data.

Since the latency of a given operator depends on its inputs’ shape, the parameters such as strides of convolution operator, and the current GPU usage, the inputs of our prediction model are denoted as inputs \( \equiv [\text{dim}_{x_1}^0, \ldots, \text{dim}_{x_1}^{m_1}, \ldots, \text{dim}_{x_j}^k, \text{para}_1, \ldots, \text{para}_s, \text{gpu_usage} ] \), where the \( x_1 \) to \( x_n \) denotes the input tensors of the operator, the \( \text{dim}_{x_j}^k \) denotes the i-th dimension’s size of input \( x_j \), the \( \text{para}_j \) denotes the i-th parameter of the operator, and the \( \text{gpu_usage} \) denotes the current GPU’s utilization rate. We used a four layers MLP network as the prediction model since the input is not complex, and a simple MLP model is enough to give a relatively precise prediction. According to our experiment, the average MSE of every operator’s prediction result is 4.269, which means that our method can give a relatively precise prediction to most operators. A detailed evaluation of this method’s influence on the scheduling process will be introduced in Section 5.2.

Although the GPU’s utilization rate is a general representation of the GPU usage condition, and the prediction results will not be precisely accurate, we can keep tracking the operators’ latency and dynamically correct the predicted result with the exponential moving weighted average method [26] during the job’s run-time. The details will be introduced in Section 4.5.

4.2 GPU Memory Peak Analysis

To reduce the entire system’s GPU memory peak, the scheduling algorithm has to know when the GPU memory peak appears and caused by which tensors so that it can try to schedule the Swap Event of these tensors. Thus, with the information inferred from the compute graph, we develop an algorithm to analyze the GPU memory peak statistically.

The algorithm is a discrete simulation process of the Equation (2). According to the Equation (2), the GPU memory footprint is changed only when these five situations happen.

1. **Iteration Beginning:** At the beginning of a training iteration, these tensors will be in the GPU memory:
   - Input data of the model
   - Parameters which are not swapped out from the last iteration
   - Initial space used by CUDA, cuDNN, and cuBLAS.

   These tensors will be used to initialize the GPU memory footprint.
2) **Tensor Generating Access**: When a tensor is generated in the GPU memory via **TGA**, GPU memory footprint increases. Note that when updating a parameter, we logically treat the updated parameter as a new tensor, but the new parameter will not cause an increase in GPU memory footprint since it uses the memory address of the old parameter.

3) **Swap-In Event**: A Swap-In Event causes GPU memory consumption increase. Since we only concern with the peak of GPU memory footprint, the Swap-In Event’s finishing time point can be regarded as the time point of the GPU memory footprint increases.

4) **Swap-Out Event**: When a tensor is copied to Host Memory and released, the GPU memory footprint is reduced at the end of the swap-out event (or at the end of the corresponding **TUA**, if the **TUA** ends later).

5) **Tensor Release**: For a Tensor Access i which needs to recompute or swap in, when the last Tensor Access i-1 finished, the tensor can be released from GPU memory, so that the GPU memory footprint will decrease.

Based on the above discussions, we just need to sort the Tensor Accesses and Swap Events by the time they trigger the GPU memory footprint change. Then we can traverse the job’s timeline to determine the GPU memory peak. We then introduce the analysis algorithm in detail as shown in Algorithm 1.

### 4.3 Swap Event Scheduling

The primary approach in our system to reduce the GPU memory peak is by swapping tensors out to the Host Memory when they are not in use. It is obvious that swapping every tensor out of the GPU memory after their **TGA**, and swapping them back right before their **TUA** can save the most GPU memory. However, due to the data transfer occupying the PCIe channels exclusively [15], there can only be one tensor being swapped simultaneously. Therefore, swapping tensors within and between tasks must be scheduled appropriately.

The scheduling process is a deformation of the task scheduling problem with the next two difficulties:

- A **Swap Event** cannot be executed without prerequisites. For example, a **Swap-In Event** can be executed if the corresponding **Swap-Out Event** can be performed. Otherwise, no data will be swapped into the GPU memory. Also, to avoid interrupting task execution, the **Swap-Out Event** can be executed only when the tensor’s **Swap-In Event** can be scheduled successfully before the next Tensor Access (we do not allow passively swapping in for performance reasons).
- The scheduling process must be performed considering global jobs’ information, and it needs to be updated frequently due to the dynamic workloads. This fact determines that we can not use a very complex algorithm to generate the scheduling plan. Otherwise, the generation of the scheduling plan will cause significantly extra overheads and makes the scheduling plan can not be updated in time under the fluctuation of GPU utilization.

### Algorithm 1. GPU Memory Peak Analysis

**Input**: TAS: tensor access sequence; **SE**: swap events  
**Output**: **MP**: memory peak; **MPTensor**: tensors caused memory peak; **LUA**: last input access for each job before memory peak; **MPTime**: time of the memory peak

1. initialize the **memory_used** and the **in_gpu_tensors** with the tensors which will be updated in the optimizing phase and the inputs/outputs of the deep learning model;

2. for each event in time_axis do
   1. time ← event.time;
   2. Release the tensor in tensors.to_be_release if the present time is after its end time;
   3. if event.type = output_access and event.tensor is not used to initialize the **memory_used** and the **in_gpu_tensors** then
      1. **memory_used** ← **memory_used** + event.size;
      2. **in_gpu_tensors**.add(event.tensor);
   4. else if event.type = input_access then
      1. if event.release = True then
         1. tensors.to_be_released.append(event);
      2. else
         1. last_input_access ← event;
   5. else if event.type = swap.out or event.type = swap.in then
      1. last_access ← get_last_access(event, time_axis);
      2. event.execute.ref ← last_access;
      3. event.execute_time ← last_access.time;
      4. if event.type = swap.in then
         1. **memory_used** ← **memory_used** + event.size;
         2. **in_gpu_tensors**.add(event.tensor);
      5. else
         1. **memory_used** ← **memory_used** − event.size;
         2. **in_gpu_tensors**.remove(event.tensor);
   6. if **memory_used** > **memory_peak** then
      1. **memory_peak** ← **memory_used**;
      2. **MPTensor** ← copy(in_gpu_tensors);
      3. **LUA** ← last_input_access;
      4. **MPTime** ← time;
   7. return **memory_peak**, **MPTensor**, **LUA**, **MPTime**;

Facing the difficulties above, we design a greedy algorithm to balance the performance and the time overhead. The main idea is to swap the largest available tensor whose access will cause the GPU memory footprint peak. Since the scheduling for each tensor can change the GPU memory peak, we must analyze the GPU memory peak and generate the swap events iteratively. In particular, the GPU memory peak will be analyzed first based on the Tensor Access Sequence, and choosing the largest tensor that caused the peak to swap. This process is iterated several times until there are no tensors to be scheduled.

With such a greedy algorithm, each scheduled tensor can significantly reduce the GPU memory peak. Now we will detail our algorithm.

We firstly choose the biggest **MPT** among all jobs as the most valuable tensor to swap (candidate tensor). Note that each job has a max swapping rate limit. Considering the swapping among all jobs, the PCIe channel will be jammed if we swap the tensors in each job as many as possible. Also, we can not control the order of swap events among all jobs.
because jobs in our system are running asynchronously for performance reasons, and each job's time cost of one training step is significantly different. If we use a synchronous way to execute these jobs, then there must be extra synchronization overheads. From this perspective, to reduce the conflict opportunity, we define the max swapping rate limit for each job's tensor swapping as the rate of tensors swapped in a job to tensors swapped in the whole system. It can be regarded as a hyper-parameter. The max swapping rate can be considered the job's priority. Because the tensors' swapping cannot always overlap with computation with 100 percent odds due to the conflict on PCIe and the changing GPU utilization, swapping implies the job may run slower than usual. In this perspective, a lower swapping rate means higher priority.

After choosing the candidate tensor, each Swap Event's feasible time intervals will be inferred, as shown in Algorithm 2 Line 4 and Lines 23-24, which is defined as its earliest time and latest time. There are three bases for inferring the feasible time interval:

- The latest time of the Swap-Out Event cannot be later than the time the GPU memory peak appears since we want to swap it to reduce the GPU memory peak. Furthermore, a Swap-Out Event must start after its TGA.
- A Swap-In Event cannot be earlier than the corresponding Swap-Out Event or later than the TUA that uses it as input. Therefore, our algorithm needs to decide the start time of each event under the constraint of its feasible time intervals.
- The Swap Event can not overlap with the corresponding tensor's access.

Based on the feasible time intervals, we choose the available intervals greedily. For a Swap-Out Event, it will start as early as possible, and for a Swap-In Event, it will start as late as possible so that we can save the GPU memory occupied by the tensor for a longer time. If there are no available intervals, then we will break the outer loop in Line 20 and try to swap the following tensor.

Next, we have to decide when to execute the Swap-Out Event and the Swap-In Event. Since the tensors in the forward/backward propagation phase and the optimizing phase have different constraints, we will introduce them separately.

The tensors in the forward/backward propagation phase must be swapped into the GPU before the first TUA after the Swap-Out Event. Otherwise, an exception will be triggered due to missing necessary input tensors, and the system has to pause the computation to wait for their swapping. As shown in Algorithm 2 Lines 10-14, we look for the first TUA after the Swap-Out Event and generate the Swap-In Event and its feasible intervals to see whether any of these intervals can cover the Swap-In Event. If they cannot cover the event, we will update the Swap-Out Event's earliest start time and regenerate the feasible intervals(Lines 18-19). Otherwise, for the rest TUA of the tensor, we try to swap them in as much as possible(Line 34). If a Swap Event is scheduled successfully, the corresponding tensor can be released after the previous TUA is finished. TENSILE will also analyze the compute graph and release the tensors after their last access(Activity Analysis). The detailed algorithm of swapping scheduling is shown in Algorithm 2.

Most prior works ignore the tensors which need to be updated in the Opt phase. This causes the 'Across-iteration Scheduling' problem. TENSILE solves this problem with the following rules: if these tensors are swapped out at the optimizing phase, the corresponding tensors must be swapped in at the beginning of the next iteration. As shown in Algorithm 2 Lines 8-9 and Line 27, it tries to swap out the updated tensors and then swaps in the corresponding tensor before the first TUA. Fig. 1c shows this process more vividly.

Algorithm 2. Swap Scheduling Algorithm

Input: TAS: tensor access sequence; SE: swapped events; MPT: the tensors that cause the memory peak; MSR: the max swapping rate limit for each job, SON: swapped out tensors number for each job; TAT: TAS grouped by tensor and sorted by time; all the outputs of Algorithm 1

Result: Add new Swap Events to SE

1 Function: scheduling_swap(SE: list, tensor):
2   succeed, have_first_access —— False;
3   Generate swap_out_event and feasible_intervals;
4   for each intervals in feasible_intervals do
5     if intervals covers swap_out_event then
6       SE.add(swap_out_event);
7       succeed_swap_out—— True;
8     if tensor is an updated parameter then
9       first_access—— the first TUA of the corresponding non-updated parameter;
10     else
11       first_access—— the first TUA after access;
12       have_first_access—— first access! = None;
13       Generate swap_in_event from the first_access;
14       if try_schedule_swap_in(first_access) success then
15         SE.add(swap_in_event);
16         succeed—— True;
17       else
18         SE.remove(swap_out_event);
19         swap_out_event_latest_time—— first_access.end_time;
20         break;
21       return succeed, succeed_swap_out, have_first_access;
22   for each tensor in MPT do
23     Infer the latest time from the peak analysis result;
24     Infer the earliest_time from the TGA of the swap-out event;
25     if tensor has not been swapped out then
26       if tensor is an updated parameter in optimizing phase then
27         scheduling_swap(SE, tensor);
28       else if SON[tensor.job_id] ≤ MSR[tensor.job_id] and
29         TAT[tensor].length > 1 then
30         succeed—— False;
31         accesses—— TAT[tensor];
32         succeed_swap_out, have_first_access —— True;
33       while not succeed and
34         latest_time > earliest_time and
35         succeed_swap_out and have_first_access do
36         succeed, succeed_swap_out, have_first_access ——
37         scheduling_swap(SE, tensor);
38         // Ascending sort by end time
39         Try to swap-in the rest of accesses greedily if the above scheduling succeeds;

With this algorithm, we can select the most valuable tensors to be appropriately scheduled among all jobs, thus reducing the entire computation job’s GPU memory peak.

### 4.4 Recomputation Scheduling

If the predicted GPU memory peak is still larger than the GPU memory after swapping scheduling, we need to use recomputation to save more GPU memory. However, a major shortcoming of recomputation is that it will add extra TUA and change the Tensor Access Sequence. If the inputs of a recomputation have been swapped out or released, then it needs to be swapped in or recomputed, which will disrupt the swapping plan just generated and cause much more time overhead. Therefore, we will only apply recomputation on those tensor accesses which have never been released. To achieve the best GPU memory saving effect, we will select those accesses that conform to the above conditions with the largest recomputation value until the GPU memory is enough to run these jobs.

#### Algorithm 3. Complete Scheduling Algorithm

**Input:** TAS: tensor access sequence

1. **initialization** swap succeeds, recomputation succeeds $\leftarrow$ True, iter $\leftarrow 0$;
2. run Activity Analysis;
3. while swap succeeds or recomputation succeeds $\leftarrow$ True do
   - if the average $MP_j$ of the jobs corresponding to the selected tensors in the past 3 iterations reduces less than 0.05% and iter $> 100$ then break;
   - if iter $= 0$ then
     - Run the $GPU_Memory_{Peak\_Analysis}$ algorithm for each job and merge their GPU memory footprint as $MP$;
   - if swap succeeds $\leftarrow$ False;
8. Run Swap Scheduling to generate swap events;
10. if no swap events generated then
11. swap succeeds $\leftarrow$ False;
12. else if $MP \geq Free\_GPU\_Memory$ then
13. Run Recomputation Scheduling;
14. if no recomputation events generated then
15. recomputation succeeds $\leftarrow$ False;
16. iter $\leftarrow iter + 1$;

To measure a recomputation’s value, we used the metric proposed in Capuchin [15], known as MSPS. This metric can measure the memory saving and extra time cost at the same time.

$$MSPS = \frac{\text{Memory Saving}}{\text{Recomputation Time}}.$$  

### 4.5 Scheduling Plan Updating

As discussed in Section 1, each operator’s latency changes along with the jobs’ running in the multiple dynamic workloads scenarios. This leads to a changing Tensor Access Sequence, so the initial scheduling plan based on the outdated Tensor Access Sequence will be less efficient. For example, the hit rate of Swap-In Event will decrease and causes an increasing extra overhead. To adapt to such kinds of dynamic workloads, the Memory Scheduler needs to be able to update the scheduling plan based on the operators’ latency collected by the Executor.

Two major problems should be solved.

1. When should the scheduling plan needs to be updated?
2. How to update the scheduling plan?

Since the reason for updating the scheduling plan is that the latency of operators has been significantly changed compared to the time used to generate the last Tensor Access Sequence, we only need to propose a metric to measure this change. We use the average latency rate of change, which is defined as $\frac{\sqrt{r^2-s^2}}{s}$, where $s$ is the newest sum of each operation’s latency, and $s$ is the last sum of each operation’s latency. If the changing rate is larger than the updating threshold set before, then the updating of the scheduling plan will be triggered.

To correct the Tensor Access Sequence and make the scheduling plan more accurate, we use the exponential weighted moving average algorithm [26] to update the latency of operators. By predicting and correcting the latency, TENSILE can achieve higher performance than Capuchin’s ‘Passive Mode’ since passively swapping tensors requires extra overhead.

### 4.6 Discussion of Supporting Dynamic Workload and the Range of Application

With Algorithm 1, TENSILE can give a more effective initial scheduling plan than the ‘Passive Mode.’ The system will keep tracing recent tensor accesses time to correct the Tensor Access Sequence and update the scheduling plan. When a job is launched, the Tensor Access Sequences of other jobs are most likely to fluctuate greatly. In this situation, TENSILE updates the Tensor Access Sequence based on current GPU usage for all jobs. To avoid extra overhead caused by scheduling plan changes, the system will apply the new plan before computing the next batch of data. The complete scheduling algorithm is shown in Algorithm 3. Firstly, we run an activity analysis at the beginning of the scheduling to save more memory. Every tensor will be released when its last TUA is finished. Then, the Memory Scheduler will iteratively run the process below until no swapping or recomputation is scheduled. At each iteration, the algorithm will start with running the $GPU_Memory_{Peak\_Analysis}$ function for each job and merge the result as shown in Lines 6-7. Then, if at least one swap event is scheduled successfully in the previous iteration, it will keep trying to swap, as shown in Lines 8-12. Otherwise, if the GPU memory is insufficient, it will try to schedule the recomputation events as shown in Lines 13-15.

Although our method is designed to run on a single GPU, it can also support data-parallel models for multiple GPUs in the cluster. In this situation, the scheduling processes are the same with a single GPU situation on each node, with the premise that its CPU has enough PCIe channels for the GPUs to use in a single node.

### 5 Experiments

To verify the effectiveness of the proposed approaches, we conduct extensive experiments in this section.
5.1 Methodology

Experiment Platform. To apply our scheduling algorithm, we implement a naive deep learning framework with CUDA as our experiment platform. The framework is modified based on the tinyflow open-source project. We also reproduce vDNN [16], and Capuchin [15] coupling with our framework based on their papers.

Baselines. We choose two methods as baselines from layer-granularity and tensor-granularity methods to compare with our method. The first one is vDNN, which swaps out the appropriate feature maps and uses a static swap-in strategy for prefetching. There are two types vDNNs, vDNN_all and vDNN_conv, the former tries to swap every feature map, and the latter only swaps the feature maps of convolution layers. We take the vDNN_conv as one of our baselines since the authors of vDNN proved that vDNN_conv has much higher performance than vDNN_all [16]. The second baseline is Capuchin, the newest and the most representative dynamic scheduling method in tensor granularity for a single job. Although these baselines are designed for single workload scenarios, we can also create an instance for each workload in our experiments and schedule each workload independently. And both the baselines and our method overlap data transfer and computation with cudaMemcpyAsync.

Metrics. We introduce the metrics used to evaluate the performance of TENSILE. We choose Memory Saving Rate (MSR), Extra Overhead Rate (EOR), and Cost-Benefit Rate (CBR) to measure the TENSILE’s performance and efficiency compared to the baselines. Their definitions are shown as follows: \(MSR = \frac{VMP - EMP}{VMP} \), \(EOR = \frac{ETC - VTC}{VTC} \), \(CBR = \frac{MSR}{EOR}\).

Where \(VMP\) and \(EMP\) are the memory footprint peak of the vanilla group (no scheduling) and the experimental group (TENSILE or baselines), \(VTC\) and \(ETC\) are the time cost of the vanilla group and the experimental group.

The three metrics we choose are the key metrics to evaluate a GPU memory scheduling method. The Memory Saving Rate (MSR) represents how much GPU memory a scheduling method can save compared to the vanilla method (no scheduling), the higher MSR implies that the corresponding scheduling method can reduce more GPU memory footprint compared to the vanilla method. The EOR indicates how many times the time cost will be caused by saving these proportions of GPU memory. The lower EOR implies that the scheduling method has less extra overhead. The CBR is the ratio of MSR and EOR, which indicates how much extra overhead it takes to save a piece of GPU memory. The higher CBR implies that the corresponding scheduling method is more space-time efficient than other methods. These metrics are common and fair for TENSILE and both baselines since we analyzed and compared the MSR, EOR, and CBR of the whole computing process of TENSILE and both baselines in detail. Since TENSILE is designed for multiple dynamic workload scenarios, it is expected to have higher MSR and CBR and less EOR compared to other baseline methods.

Statement of Our Reproduction. It is necessary to emphasize that the reproduction for Capuchin may not be exactly the same as its authors’ implementation in the paper [15] in the aspect of the system framework since there is no source code to refer to, and it is implemented in TensorFlow. For example, in Capuchin, they measure the GPU kernel latency through CUBI. However, in our platform, every computation will be synchronized between CPU and GPU, so we can measure the operators’ latency in the python codes. We want to compare the scheduling algorithm that runs on the same platform rather than whose platform implementation is more computationally efficient. Therefore, the experiments will be fair since all the methods’ scheduling algorithms are implemented exactly as their paper described and run on the same platform for comparison.

Experiment Platform. Our experiment is conducted on a server that has Intel(R) Xeon(R) Silver 4210R CPU, 504 GB Host Memory, and one RTX A6000 with 48 GB memory. The system version is Ubuntu 18.04 with GCC 7.5.0, the CUDA version is 11.3, the cuDNN is 8.2.0, and Python 3.10.

Workloads. We choose the five neural networks in the evaluation of the Capuchin [15] to evaluate TENSILE, VGG-16 [27], InceptionV3 [28], InceptionV4 [29], ResNet-50 [2], and DenseNet [30], covered both linear network and nonlinear network. We will increase the workload number on one GPU and measure the GPU memory footprint and the extra time overhead for each method.

Extra Setting. The Capuchin is designed to schedule only when the GPU memory is insufficient. It will only save the GPU memory to the threshold that is just able to run the workloads, which makes it difficult to compare its performance with TENSILE since TENSILE is a more active scheduling method that will save GPU memory as much as possible. Therefore, we set Capuchin’s GPU memory budget as the same as TENSILE. Then we can compare these two methods’ time cost and efficiency fairly.

Also, our method includes two phases, the cold start phase, and the dynamic updating phase. The former uses predicted operations’ latency to generate the schedule plan before the job launching, and the latter updates the scheduling plan with the running information collected from the framework. To analyze these two phases separately, we denote the cold start phase by TENSILEcs and the updating phase by TENSILEup. The TENSILEcs’s performance is measured once the job is launched, and the TENSILEup’s performance is measured once the scheduling plan is updated. We also use TENSILE to denote the whole process of TENSILE.

5.2 Performance of TENSILE’s Each Phase

We first evaluate the performance of the cold start phase and the update phase of TENSILE by measuring the memory footprint and time cost separately. The results are shown in Table I.

The MSR of TENSILE equals the metric of TENSILEcs, which proves that the cold start phase is the bottleneck of the whole process of TENSILE. Also, compared to Capuchin’s performance in Table 3, TENSILEcs can achieve a similar performance as Capuchin with the prediction method only. This result proves the effectiveness of our prediction method. Note that the time cost of the cold start phase includes the overhead of the prediction of the MLP models, and this result proves the effectiveness of the scheduling.
5.4 Scalability

To evaluate our method’s performance on multiple dynamic workloads and test its scalability, we choose one to four workloads respectively and launch them simultaneously to simulate a multiple dynamic workloads scenario. We repeat each test three times and report the average metric in Fig. 6.

As the Fig. 6 shows, in most of the workloads, TENSILE has achieved the best MSR and CBR, which is solid proof of the performance of TENSILE in multiple dynamic workloads situations. The EOR and CBR of TENSILE outperform Capuchin in VGG-16 since the VGG-16 has a massive tensor generated by a wide MLP layer, which causes it much harder to schedule without the overhead. TENSILE tackles this problem by the updating method, and the whole TENSILE, including the updating phase, outperforms Capuchin. With the increased workload, TENSILE’s MSR of InceptionV3, InceptionV4, ResNet-50, and DenseNet drops slowly. And in all experiments, TENSILE saves the most GPU memory.

Furthermore, according to the EOR metric, Capuchin’s overhead dramatically increases as we analyze in the last paragraph of Section 2. The VDNN’s EOR reduces with the increasing workload numbers since the VDNN is a layer-

### TABLE 1

| Workloads | Phase         | MSR    | EOR    | CBR    |
|-----------|---------------|--------|--------|--------|
| VGG-16    | TENSSILE      | 0.2670 | 1.6287 | 0.1639 |
|           | TENSSILEcs    | 0.2670 | 6.0128 | 0.0444 |
|           | TENSSILEup    | 0.3192 | 1.2475 | 0.2559 |
| InceptionV3| TENSSILE     | 0.4361 | 1.6468 | 0.2648 |
|           | TENSSILEcs    | 0.4361 | 3.1128 | 0.1401 |
|           | TENSSILEup    | 0.4788 | 1.4839 | 0.3227 |
| InceptionV4| TENSSILE    | 0.5154 | 1.4281 | 0.3609 |
|           | TENSSILEcs    | 0.5154 | 2.6727 | 0.1929 |
|           | TENSSILEup    | 0.5558 | 1.2898 | 0.4309 |
| ResNet-50 | TENSSILE      | 0.4258 | 1.5540 | 0.2740 |
|           | TENSSILEcs    | 0.4258 | 3.5935 | 0.1203 |
|           | TENSSILEup    | 0.4647 | 1.3813 | 0.3564 |
| DenseNet  | TENSSILE      | 0.5135 | 1.6787 | 0.4397 |
|           | TENSSILEcs    | 0.5135 | 2.2532 | 0.2279 |
|           | TENSSILEup    | 0.5236 | 1.0664 | 0.4910 |

5.3 Single Workload Performance

We compare TENSILE’s performance with vDNN and Capuchin on a single workload. The batch size is set to 16. And for TENSILE, the max swapping rate limit is set to 100% for each task since there is only one task in this experiment. The result is shown in Table 3.

And the result also shows that our method can save 26.70%-51.54% memory than the vanilla method, which is much higher than vDNN. This is foreseeable since TENSILE is a tensor granularity scheduling method. And the cost-benefit rate of TENSILE is also much higher than vDNN and Capuchin, which proves the efficiency of TENSILE.

The high efficiency of TENSILE is due to the fact we put the extra time cost into priority consideration, which is embodied in that we do not allow the non-overlapping between swapping and computation logically. On the contrary, the other baseline methods allow the computation to wait for swapping when scheduling. Also, compared to Capuchin, TENSILE can schedule tensors across iterations. This feature helps to avoid some passive swap-in.

This result proves that although TENSILE is designed for multiple jobs and dynamic workloads, it can still achieve the best efficiency and memory saving rate in a single workload situation.

### Table 2

| Workloads | Method    | EOR    | Prediction Cost(s) | Prediction EOR |
|-----------|-----------|--------|--------------------|----------------|
| VGG-16    | vDNN      | 0.1109 | 1.7835             | 0.0622         |
|           | Capuchin  | 0.2258 | 1.8304             | 0.1398         |
|           | TENSILE   | 0.2670 | 1.6287             | 0.1639         |
| InceptionV3| vDNN      | 0.1633 | 2.5184             | 0.0648         |
|           | Capuchin  | 0.4331 | 2.7745             | 0.1561         |
|           | TENSILE   | 0.4361 | 1.6468             | 0.2648         |
| InceptionV4| vDNN      | 0.1903 | 2.5028             | 0.0760         |
|           | Capuchin  | 0.5129 | 3.2362             | 0.1585         |
|           | TENSILE   | 0.5154 | 1.4281             | 0.3609         |
| ResNet-50 | vDNN      | 0.1733 | 2.2981             | 0.0754         |
|           | Capuchin  | 0.3859 | 2.2320             | 0.1729         |
|           | TENSILE   | 0.4258 | 1.5540             | 0.2740         |
| DenseNet  | vDNN      | 0.1381 | 2.6148             | 0.0528         |
|           | Capuchin  | 0.4860 | 3.4400             | 0.1413         |
|           | TENSILE   | 0.5135 | 1.1678             | 0.4397         |

This result proves that although TENSILE is designed for multiple jobs and dynamic workloads, it can still achieve the best efficiency and memory saving rate in a single workload situation.
granularity method, which cannot schedule all tensors so it has many free time intervals for swapping. This makes its extra time cost less than the vanilla method’s time cost. And with increasing workload numbers, the vanilla time cost rises faster than the extra time cost. So the EOR gets lower. Although VDNN has a lower overhead in multiple workload scenarios, its MSR is meager compared to TENSILE and Capuchin.

In any case, the efficiency of TENSILE is still much higher than both baselines. Moreover, the ratios of TENSILE’s CBR to VDNN/Capuchin’s CBR under the multi-workload environment are higher than in the single-workload environment. The conclusion is that TENSILE can run these workloads with less (or at least the same) GPU memory and time cost than baselines. Especially compared to Capuchin, whose EOR is significantly increasing when the number of workloads increases, which implies most of their swapping transfer cannot overlap with the computation, TENSILE’s EOR is much more stable. This experiment strongly proves that the performance of TENSILE is much better than VDNN and Capuchin in multiple dynamic workloads scenarios. To evaluate our method’s performance on multiple dynamic workloads and test its scalability, we choose one to four workloads respectively and launch them simultaneously to simulate a multiple dynamic workloads scenario. We repeat each test three times and report the average metric in Fig. 6.

5.5 Mixed Neural Architectures

To measure the TENSILE’s performance in a mixed neural architectures workloads scenario, we randomly launch the five above workloads and measure the total GPU memory footprint and execution overhead. The five workloads are launched one by one in random order, and this procedure is repeated three times. We set the budget of each workload of Capuchin with the MSR of TENSILE in Table 4.

As Table 4 shows, TENSILE is comprehensively leading in EOR and CBR under different neural architectures. It can save much more GPU memory with less extra cost in scenarios with complex non-linear workloads, making it more suitable for environments such as in-database machine learning.

5.6 Overhead Analysis

Although TENSILE outperforms two baselines, the overhead during GPU memory scheduling can hardly be avoided completely.

For single workloads scenarios, the overheads are caused by two reasons. The first reason is that when the memory is extremely insufficient, the overhead of passive swap-in and recomputation cannot be avoided completely. However, using a more efficient scheduling algorithm, TENSILE achieves less overhead than Capuchin. The second reason is the failed tensor prefetch caused by the fluctuation of GPU operator latency and the error of estimated operator latency.

For multiple workloads scenarios, the GPU utilization rate and the PCIe channel are all influenced by the asynchronous workloads, which makes the prefetch much harder and makes more passive swap-in and recomputation. TENSILE alleviates this problem by choosing the most valuable tensors among all workloads to schedule to achieve higher efficiency. However, it is not a perfect solution, and we will continue to find better solutions to this problem in the future.

5.7 Influence of Batch Size

We also analyzed the batch size’s influence on our method by running those five workloads, respectively, with batch sizes 4, 8, 16, 32, and 64. The results are shown in Fig. 7. The MSR increases with the batch size for all workloads, and so does their CBR in general. This result shows that the TENSILE can save more GPU memory with less overhead when the batch size increases. This result proves that TENSILE is more effective in scheduling the interim results than the parameters since the former is the only one influenced by the batch size. The DenseNet with batch size 32 and 64 is an exception. We believe it is because there are too many tensors to swap. And when the batch size increases, the swap events can more easily conflict with each other. And for VGG-16, it is the only workload whose EOR drops with the increase in batch size. This is because that VGG-16 has a wide MLP layer that outputs a huge tensor. This tensor’s computation cost increases faster than its scheduling cost, so the EOR gets lower.
6 Conclusions & Future Work

We propose TENSILE, a tensor granularity GPU memory scheduling method toward multiple dynamic workload scenarios. TENSILE has solved three major problems of prior works, multiple dynamic workloads, cold-starting, and across-iteration scheduling by designing a scheduling method and corresponding system, a latency prediction method, and a scheduling algorithm. Furthermore, TENSILE is a generic scheduling method for computing tasks described as DAGs, especially deep learning tasks. The experiment results show that TENSILE can achieve much higher performance in such scenarios than vDNN and Capuchin. Even in single workload scenarios, TENSILE can still outperform baselines. These features can help systems such as in-database machine learning run more deep learning tasks with tiny extra overhead.

References

[1] S. Dagan, M. Kumar, M. R. Ayyagari, and G. Kumar, “A survey of deep learning and its applications: A new paradigm to machine learning,” Arch. Comput. Methods Eng., vol. 7, pp. 1071–1092, 2019.
[2] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in Proc. IEEE Conf. Comput. Vis. Pattern Recognit., 2016, pp. 770–778.
[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” in Proc. Annu. Conf. North Amer. Chapter Assoc. Comput. Linguistics, 2019, pp. 4171–4186.
[4] T. Brown et al., “Language models are few-shot learners,” 2020.
[5] D. Yang and D. Cheng, “Efficient GPU memory management for nonlinear DNNs,” in Proc. 29th Int. Symp. High-Perform. Parallel Comput., 2020, pp. 185–196.
[6] V. Narasayya, I. Menache, M. Singh, F. Li, M. Syamala, and S. Chaudhuri, “Sharing buffer pool memory in multi-tenant relational database-as-a-service,” Proc. VLDB Endow., vol. 8, pp. 726–737, 2015.
[7] T. Li, J. Zhong, J. Liu, W. Wu, and C. Zhang, “Ease.ml: Towards multi-tenant resource sharing for machine learning workloads,” Proc. VLDB Endow., vol. 11, pp. 607–620, 2018.
[8] B. Karlas, J. Liu, W. Wu, and C. Zhang, “Ease.ml in action: Towards multi-tenant declarative learning services,” Proc. VLDB Endow., vol. 11, pp. 2054–2057, 2018.
[9] S. Boag et al., “Dependability in a multi-tenant multi-framework deep learning as-a-service platform,” in Proc. 25th Int. Conf. Architectural Support Program. Lang. Oper. Syst., 2020, pp. 489–504.
[10] W. Xiao et al., “Gandiva: Intrusive cluster scheduling for deep learning,” in Proc. 13th USENIX Conf. Oper. Syst. Des. Implementation, 2020, pp. 481–498.
[11] R. Marcus et al., “Neon: A learned query optimizer,” Proc. VLDB Endowment, vol. 12, pp. 1705–1718, 2019.
[12] R. Marcus et al., “BAO: Making learned query optimization practical,” in Proc. Int. Conf. Manage. Data, 2021, pp. 1275–1288.
[13] T. Kraska, A. Buelth, E. Chi, J. Dean, and N. Polyzotis, “The case for learned index structures,” in Proc. Int. Conf. Manage. Data, 2018, pp. 489–504.
[14] X. Peng et al., “Capuchin: Tensor-based GPU memory management for deep learning,” in Proc. 25th Int. Conf. Architectural Support Program. Lang. Oper. Syst., 2020, pp. 891–905.
[15] M. Rhu, N. Gmelshel, J. Clemonds, A. Zulfiqar, and S. W. Keckler, “vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design,” in Proc. IEEE/ACM 49th Annul. Int. Symp. Microarchit., 2016, pp. 1–10.
[16] L. Wang et al., “Superneurons: Dynamic GPU memory management for training deep neural networks,” ACM SIGPLAN Notices, vol. 53, pp. 41–53, Feb. 2018.
[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” Commun. ACM, vol. 60, no. 6, pp. 84–90, May 2017, doi: 10.1145/3065386.
[18] T. N. Kipt and M. Welling, “Semi-supervised classification with graph convolutional networks,” 2016, arXiv:1609.02907.
[19] B. ShriramS, A. Garg, and P. Kulkarni, “Dynamic memory management for GPU-based training of deep neural networks,” in Proc. IEEE Int. Parallel Distrib. Process. Symp., 2019, pp. 200–209.
[20] A. Vasswani et al., “Attention is all you need,” in Proc. 31st Int. Conf. Neural Inf. Process. Syst., 2017, pp. 5999–6009.
[21] S. Sabour, N. Frosst, and G. E. Hinton, “Dynamic routing between capsules,” in Proc. 31st Int. Conf. Neural Inf. Process. Syst., 2017, pp. 3859–3869.
[22] D. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in Proc. 3rd Int. Conf. Learn. Representations, 2014.
[23] J. Zhang, S. H. Yeung, Y. Shu, B. He, and W. Wang, “Efficient memory management for GPU-based deep learning systems,” 2019, arXiv:1903.06631.
[24] M. Abadi et al., “Tensorflow: A system for large-scale machine learning,” in Proc. 12th USENIX Symp. Oper. Syst. Des. Implementation, 2016, pp. 265–283.
[25] M. B. Perry, “The exponentially weighted moving average,” Wiley Encyclopedia Operations Res. Manag. Sci., 2010.
[26] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” 2015, arXiv:1409.1556.
[27] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the inception architecture for computer vision,” in Proc. IEEE Conf. Comput. Vis. Pattern Recognit., 2016, pp. 2818–2826.
[28] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, “Inception-v4, inception-resnet and the impact of residual connections on learning,” in Proc. AAAI Conf. Artif. Intell., 2017, pp. 4278–4284.
[29] G. Huang, Z. Liu, and K. Q. Weinberger, “Densely connected convolutional networks,” 2016, arXiv:1608.06993.

Kaixin Zhang received the BS degree in computer science from the Harbin Institute of Technology, Harbin, China, in 2021. He is currently working toward the PhD degree in computer science with the massive data computing center of Harbin Institute of Technology, Harbin, China. His research interests include database systems, Auto-ML, and artificial intelligence. He has participated and served as the team leader with the CCF-Huawei Database Innovation Research Program.

Hongzhi Wang, professor, PhD supervisor, the head of massive data computing center and the vice dean of the Honors School of Harbin Institute of Technology, the secretary general of ACM SIGMOD China, outstanding CCF member, a standing committee member CCF databases and a member of CCF Big Data committee. Research Fields include Big Data management and analysis, database systems, knowledge engineering and data quality. He was “starring track” visiting professor with MSRA and postdoctoral fellow with the University of California, Irvine. He has been PI for more than 10 national or international projects including NSFC key project, NSFC projects and National Technical support project, and co-PI for more than 10 national projects includes 973 project, 863 project and NSFC key projects. He also serves as a member of ACM Data Science Task Force. He has won First natural science prize of Heilongjiang Province, MOE technological First award, Microsoft Fellowship, IBM PhD Fellowship and Chinese excellent database engineer. His publications include more than 200 papers in the journals and conferences such as VLDB Journal, IEEE Transactions on Knowledge and Data Engineering, VLDB, SIGMOD, ICDE and SIGIR, 6 books and 6 book chapters. His PhD thesis was elected to be outstanding PhD dissertation of CCF and Harbin Institute of Technology. He severs as the reviewer of more than 20 international journal including VLDB Journal, IEEE Transactions on Knowledge and Data Engineering, and PC members of more than 50 international conferences including SIGMOD, VLDB, KDD, ICML, NeurIPS, ICDE, etc. His papers were cited more than 1000 times. His personal website is http://homepage.hit.edu.cn/wang.
Han Hu is currently working toward the graduate degree with the Department of Computer Science, Harbin Institute of technology, Harbin, China. His research interests include database and machine learning.

Songling Zou is currently working toward the graduate degree with the Department of Computer Science, Harbin Institute of technology, Harbin, China. His research interests include the interdisciplinary technologies of database and machine learning.

Jiye Qiu is currently working toward the graduate degree with the Department of Computer Science, Harbin Institute of technology, Harbin, China. His research interests include the interdisciplinary technologies of database and machine learning.

Tongxin Li is currently working toward the bachelor’s degree with the Faculty of Computing, Harbin Institute of Technology, Harbin, China. His research interests include time-series databases, deep learning frameworks, and GPU memory management. Also, he is a good engineer and has participated in the design and implementation of several complex systems.

Zhishun Wang received the BS degree from the University of Electronic Science and Technology of China, in 2021, where he is currently working toward the master’s degree with the Chinese University of Hong Kong. His research interests include artificial intelligence and quantitative finance.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.