Waterwave: A GPU Memory Flow Engine for Concurrent DNN Training

Xuanhua Shi, *Senior Member, IEEE,* Xuan Peng, Ligang He, Yunfei Zhao, and Hai Jin, *Fellow, IEEE*

Abstract—Training Deep Neural Networks (DNN) concurrently is becoming increasingly important for deep learning practitioners, e.g., *hyperparameter optimization (HPO)* and *neural architecture search (NAS)*. The GPU memory capacity is the impediment that prohibits multiple DNNs from being trained on the same GPU due to the large memory usage during training. In this paper, we propose *Waterwave*, a GPU memory flow engine for concurrent deep learning training. Firstly, to address the memory explosion brought by the long time lag between memory allocation and deallocation time, we develop an allocator tailored for multi-streams. By making the allocator aware of the stream information, a *prioritized allocation* is conducted based on the chunk's *synchronization* attributes, allowing us to provide useable memory after scheduling rather than waiting it to be really released after GPU computation. Secondly, *Waterwave* partitions the compute graph to a set of continuous *node groups* and then performs finer-grained scheduling: *NodeGroup pipeline execution*, to guarantee a proper memory requests order. *Waterwave* can accomplish up to 96.8% of the maximum batch size of solo training. Additionally, in scenarios with high memory demand, *Waterwave* can outperform existing spatial sharing and temporal sharing by up to 12x and 1.49x, respectively.

Index Terms—GPU, Memory management, Deep learning training, Scheduling.

1 Introduction

Deep neural networks (DNNs) have made huge strides forward in a variety of fields, including image classification, object identification, speech recognition, and natural language processing. Given the marvelous power of various computing hardware, including GPU, TPU [1] and ASIC, the deep learning practitioners are able to explore deeper and more intricate deep neural networks architectures. In recent years, more and more automatic approaches for deep learning have been proposed. Hyperparameter optimization [2], [3], [4] and Neural Architecture Search [5], [6] are two examples, which pursue finding a set of optimal/sub-optimal parameters and neural network architectures, respectively. These methods usually define a large search space and need to train a large number of homogeneous or heterogeneous neural networks. During the training, the next optimization direction is determined according to the previous results.

In prior works, a GPU device is shared by multiple deep learning (DL) jobs from *temporal* and *spatial* perspectives. *Temporal sharing*, which is adopted by the systems such as Gandiva [7] and Salus [8], seeks to multiplex a GPU device at the time slice level (e.g., a training step or several minutes). Through switching the contexts of co-located jobs, only one active job runs on the GPU at a time. Therefore, the GPU resources are still under-utilized for some models such as Tacotron2 [9] and GNMT [10]. This is because such models lack enough degree of parallelism to saturate the GPU and suffer from scheduling overhead due to tens of thousands of kernels being launched in a single iteration. On the other hand, *spatial sharing* runs multiple jobs concurrently on a GPU and usually exhibits the greater throughput than temporal sharing. However, this method is constrained by the total memory size required by the concurrently running jobs, which cannot exceed the GPU memory capacity. Although the Multi-Process Service (MPS) [11] and Unified Virtual Memory (UVM) provided by NVIDIA can serve multiple jobs in single GPU and leverage CPU DRAM as an external storage when GPU memory is being oversubscribed, they will stall the GPU computation due to the on-demand memory swap based on page fault and limited PCI-e bandwidth, which substantially affects the training performance.

Figure 1 shows the memory usage of three iterations' training in ResNet50 [12] and Bert-base [13], in which ResNet50 uses the SGD optimizer while Bert-base adopts the Adam optimizer. It can be seen that the memory usage increases as the feature maps accumulate in the forward propagation and decreases as the backward propagation progresses. Although the peak memory consumption of a DL job during its execution could approach the GPU memory capacity, the average memory footprint is actually small. This is because the majority of the memory footprint during the training are *feature maps* [14], [15], [16], which are generated at each layer in forward propagation and will be freed when the corresponding backward propagation has completed. Therefore, the memory footprint in a single training iteration of a DL job exhibits the trend of *"first increase then drop"*. And this pattern repeats across the iterations.

It is obvious that the co-located jobs should not be allowed to reach their maximum memory footprint at the same time, which could easily exceed the GPU memory

- *Xuanhua Shi, Xuan Peng, Yunfei Zhao, and Hai Jin are with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Huazhong University of Science and Technology, Wuhan, 430074, China. E-mail: {piecesix, xhshi, yfzhao, hjin}@hust.edu.cn*
- *Ligang He is with University of Warwick, UK. E-mail: ligang.he@warwick.ac.uk*

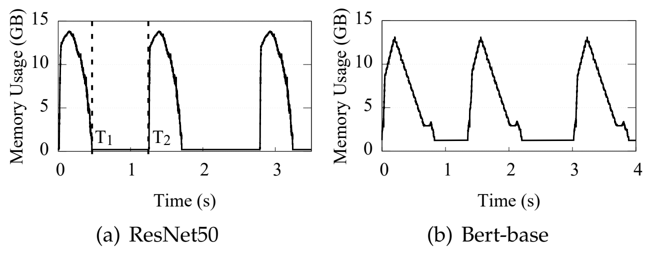


Fig. 1: Memory Usage over Time in Model Training

capacity. An intuitive idea is to schedule the *'memory drop'* phase of one job in parallel with the *'memory increase'* phase of another job. In that case, the memory freed by one job can be utilized by another job. Although this idea seems easy and straightforward, it is not applicable in the current DL frameworks like TensorFlow [17] or PyTorch [18], since they are both tailored in such a way that a single job occupies the entire GPU exclusively during its execution.

To enable efficient memory sharing in concurrent training with current DL frameworks, there are two key problems. Firstly, the DL frameworks usually allocate a large bulk of memory (e.g. the entire GPU memory) as a memory pool and perform the dynamic (de)allocations on it. This means that instead of actually releasing the physical memory when freeing memory, only the memory chunk in the memory pool is marked as available. This is to avoid the expensive overhead of frequent GPU (de)allocations, like `cudaMalloc` and `cudaFree` in NVIDIA GPU. Therefore, a job's freed memory cannot be utilized due to this caching approach. A natural method is to share a common memory pool across co-located jobs, but there could be memory corruption with concurrent training. To fully utilize GPU computing resources, the next ready kernels can be scheduled in CPU without waiting the preceding kernel to finish its computations in GPU, i.e., *the kernel scheduling in CPU is performed in parallel with the computations in GPU.* Memory allocation and deallocation occur during the scheduling without memory corruption, which is achieved by dispatching all computations to a single GPU stream, ensuring that they are processed in a strictly sequential manner. However, it is not the case anymore when multiple jobs run concurrently in different streams (it will lose the meaning of spatial sharing if multiple jobs are run in a single stream sequentially). There is no guarantee of the execution order when the computations are scheduled into different streams. The prior works [17], [19] opt to free the memory when a kernel's computations are finished in GPU. Nonetheless, the scheduling in CPU is much faster than the heavy GPU computations, which results in a big gap between the memory allocation time and deallocation time. Consequently, the memory footprint could explode very quickly. On the other hand, in order to avoid the memory bloat, if we also move the timing of memory allocation to when GPU computations are performed, the GPU computations can no longer overlap with the CPU scheduling, which leads to the decrease in training performance.

Secondly, when multiple jobs' computations are scheduled in parallel, we need to ensure that the order of the memory requests is the order we wanted. Namely, the memory of a job should be freed first and then used by other jobs. All prior works [19], [20] perform *mini-batch level scheduling*, where there could be thousands, even hundreds of thousands, of memory requests in an iteration. When a job's forward computations are scheduled before another job's backward computations, it will lead to memory oversubscription since many memory deallocation operations in backward propagation are not scheduled promptly before the memory is allocated in the forward propagation. To make the situation more complicated, there are also a lot of memory allocations in the backward phase. Furthermore, prior spatial sharing works do not naturally support more than two concurrent jobs except NVIDIA MPS. This is because they made the assumption that a job can only be scheduled into a single GPU stream. Zico [19] gives an advice that organizes the jobs into two group pairs and schedules them using the method of handling two jobs. In addition to losing a degree of parallelism, it may lead to the imbalanced distribution of workloads in two GPU streams.

In this paper, we propose a GPU memory flow engine named *Waterwave*, which aims to make the GPU memory *flow back and forth* among the concurrently running DL jobs. Firstly, we develop an *asynchronous multi-streams memory allocator* to tackle the first problem. By making the allocator aware of what GPU stream each memory chunk is allocated on, it can determine whether a chunk that's freed after scheduling is *'safe'* to be used in current allocation — this chunk has been freed in the same stream or before streams synchronization. Moreover, the allocator can proactively synchronize the GPU streams to leverage free chunks on other streams when there is no such memory chunk to satisfy the current allocation request. Since this stream synchronization is an asynchronous operation, the allocator can provide useable memory after scheduling rather than waiting for GPU computation to finish.

To solve the second problem, we propose *NodeGroup pipeline execution*, which schedules the jobs in a finer-granularity than existing mini-batch level scheduling. The compute graph of each job is partitioned into a set of continuous *node groups*, each with a similar total allocation or deallocation size. The *node group* is then used as the basic scheduling unit and node groups inside a job are processed sequentially. In this manner, a job's node group with positive allocation size can be scheduled in parallel with the node group with negative allocation size from another job. And such parallel node groups are referred as a node group *pair*. Before scheduling a new node group pair, the *streams synchronization* function is invoked to mark previous free chunks in all streams as *synchronized*. In that case, the released memory of previous node group pair is free to be used by current node group pair's allocation requests. Furthermore, from *Waterwave*'s point of view, a job's node groups can be dispatched to any stream as long as the order of the node groups in a job doesn't change. Therefore, there is actually no *job* concept in *Waterwave*'s scheduler but the ready node group, which give us much more flexibility to schedule more than two jobs.

We have prototyped *Waterwave*[1] on top of a popular deep learning framework, TensorFlow. By evaluating six models including CNN, RNN, and transformer on the V100 and P100 GPUs, the results reveal that *Waterwave* delivers effective memory sharing in concurrent training, which can achieve up to 96.8% maximum batch size of solo running. Moreover, *Waterwave* outperforms MPS and temporal sharing in terms of throughput by up to 12x and 1.49x, respectively.

2 Background

2.1 Deep Learning Training

Typically, the training process involves millions of iterations, aiming to find an adequate collection of the model parameters. In each iteration, the inputs are fed into the neural network, and then the training process starts from the input layer and proceeds until the *loss* is calculated, which is known as *forward propagation*. The *backward propagation* will then start from the output layer and progress reversely. There are various optimizers that can be used to update the model parameters, like stochastic gradient descent (SGD), Momentum [21], and Adam [22].

During the DL training, the memory allocation requests are primarily issued by four components: 1) *model parameters*; 2) *feature maps*, which are the output in forward propagation; 3) *gradient maps*, which are the output in backward propagation; 4) *temporary storage*, which is used to assist the computation (e.g., workspace in the convolutional layer). The model parameters are stored permanently in the GPU memory, which usually consumes little memory. The latter two types of memory can be released immediately upon the completion of the associated computations. Since the feature maps are required in both forward computation and backward computation, which results in a long lifespan, they consume the majority of the GPU memory.

2.2 Sharing the GPU Demand

Except the scenario in which multiple tenants commit the training of their models in a shared GPU cluster (it has been shown in prior works [7], [8], [23] that concurrent training is capable of improving GPU utilization and reducing the average job completion time), there are two other specific scenarios where concurrent training can speedup the training process too. We will give a brief introduction of these two cases in this section.

2.2.1 Hyperparameter Optimization

Apart from the ideal model parameters determined during the training, a number of parameters termed hyperparameters must be adjusted prior to the training, e.g., batch size and learning rate. Additionally, determining the optimal (or suboptimal) selection of hyperparameters is critical to the effectiveness of deep neural networks. Typically, the range of potential hyperparameter values is far too large to try all of them. There are some straightforward search strategies, such as *grid search* and *random search* [2]. In both methods, a grid of hyperparameters are established firstly. The grid search will then train the model on each of the possible combinations whereas the random search just randomly pick some combinations. To make the search more intelligent, several works have been proposed, such as Hyperdrive [3], Hyperband [4], and HyperOpt [24]. Hyperdrive supports the dynamic scheduling and the early termination mechanism to jointly optimize the model quality and searching cost. HyperOpt accepts a set of hyperparameters as an input to search and move within the set based on the results of previous trials. As the result, regardless of the search approach used, the ability of training as many combinations concurrently as feasible will benefit the search process. There are several works on deep learning cluster scheduling [7], [8], [23], which try to accelerate the process of hyperparameter optimization mainly through temporal sharing.

2.2.2 Neural Architecture Search

Handcrafting neural networks to identify the best performing structure has always been a painstaking and time consuming task. Neural Architecture Search (NAS) is developed to automate this process of identifying effective architectures for a given DL problem. Modern deep neural networks often have many (up to hundreds) layers of different types, with the varied connections between the layers. As a result, NAS has a vast design space for exploring the neural architectures. Apart from the fundamental grid search and random search that are similar as in hyperparameter optimization, there are numerous advanced techniques for optimizing the search strategy, such as evolutionary algorithm [5], Bayesian optimization [6], and reinforcement learning [25]. The reinforcement learning has been used successfully to drive the search process for better architectures. It samples from the search space using a controller network (usually a recurrent neural network) and then updates itself using the training results of the sampled networks. NAS requires concurrent training in the same way as hyperparameter optimization; the distinction is that the neural network architectures searched for in NAS are different while the network architecture is the same in hyperparameter optimization.

3 Opportunity and Challenges

3.1 Memory Usage Pattern in Deep Learning Training

Except for the memory usage pattern of *"first increase then drop"* during training, we have also noticed that there is a time gap between the last deallocation of an iteration and the start of the next iteration. For example, in Figure 1(a), there is no memory allocation or deallocation after T_1 in the first iteration and before the second iteration that started at T_2. We found that T_1 is actually the time when the scheduling has been completed in CPU whereas T_2 is the time when the computation has finished in GPU (ignore the time spent in post-processing of an iteration). This means that all memory allocations and deallocations occur at the CPU scheduling. In detail, when the scheduling of a kernel[2] is started, it will

1. Waterwave is available as an open-source software at http://github.com/HinPeng/tensorflow/tree/waterwave-dev.

2. A kernel is the basic computing unit in deep learning frameworks, which has the same meaning as an *operation* in this paper.

allocate all required memory from the GPU memory pool and return the memory that is no longer needed at the end of its scheduling. The purpose is to overlap the CPU scheduling with the GPU computation so as to avoid the scheduling overhead. Although it is obtained by running in TensorFlow, we also observe the similar results in other deep learning frameworks, such as PyTorch [18].

In summary, although the peak memory usage is enormous, it only lasts for a very short time, which indicates that there is much free memory during each iteration. However, it cannot utilize the memory freed by other jobs due to the caching strategy in the DL frameworks as mentioned before. A natural idea is to share a common memory pool across the co-located jobs. However, it may cause memory corruption when deallocating the memory at the end of the kernel's scheduling. We will clarify this point in more detail in next section.

3.2 Memory Explosion in Concurrent Training

There is no guarantee on the kernels' execution order when they are scheduled into multiple GPU streams, which will lead to memory corruption if the memory is freed at the end of scheduling. A straightforward solution to this problem is to release the memory after the kernel's computation in GPU has been completed, rather than when its scheduling in CPU is finished. This method is adopted in the work presented in [19]. However, the kernel's computation in GPU is significantly slower than its scheduling in the CPU as shown in Figure 1, which increases the time gap between the allocation and deallocation of memory chunks. This, in turn, causes a rapid increase in GPU memory usage.

To demonstrate how fast the memory growth is, we conduct a micro-benchmark, which runs a ResNet50 model solely in a P100 GPU with two configurations and compares two scenarios where the memory is deallocated after the kernel's scheduling (marked by "Sche." in Figure 2) or after its computation ("Comp."). The results are depicted in Figure 2. It can be observed from the figure that when the memory is released after the scheduling, the memory usage increases during the forward pass, reaching the peak usage of approximately 5.4 GB at 50 ms. The memory usage then decreases during the backward pass. However, the memory usage peaked with over 13 GB being observed at about 200 ms when the memory is released after the computations, due to the delayed memory deallocation. It is not until 180 ms that the backward pass starts its computations on the GPU and frees the feature map memory. This caused the memory sharing space to be considerably more constrained, as the peak and average memory usage become significantly larger.

To mitigate this explosive memory growth and usage, one way is to shift the timing of memory allocation from the start of the kernel scheduling in CPU to the start of the kernel's computation in GPU. Specifically, a kernel will not be scheduled until the previous kernel's computation has been completed in GPU[3]. However, this introduces another problem: *scheduling overhead*, since the scheduling and computation will become sequential. If the kernel scheduling and the kernel computation can overlap with each other, we call this scheduling "parallel scheduling". Otherwise, it is called "sequential scheduling". Figure 3 presents the normalized performance when performing parallel scheduling and sequential scheduling on three models respectively. We can see that the three models show 24.6%, 34.6%, and 70.7% performance degradation respectively. The performance loss is basically proportional to the number of kernels launched in an iteration. Bert-base will launch over 7000 kernels in an iteration while this number in Tacotron2 is close to 80000. Although there are fewer nodes in CNN models, it can still experience more than about 25% performance loss. Such performance overhead is unacceptable in DL training due to the expensive GPU resources. Moreover, the performance degradation will become more severe in concurrent training as one job needs to wait for others to release their memory, which can take longer time. Therefore, *the first challenge is to regulate the allocation and deallocation timings without increasing the scheduling overhead.*

3. Before launching a GPU kernel, the required GPU memory must be provisioned in advance. As a result, the timing of memory allocation can be adjusted only through regulating the timing of the kernel launch.

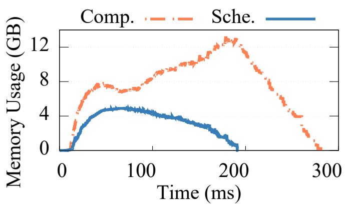


Fig. 2: Memory Usage of an Iteration with Different Memory Deallocation Timings

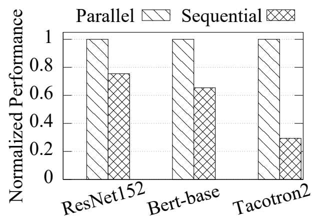


Fig. 3: Normalized Performance of Parallel and Sequential Scheduling

3.3 Scheduling Granularity in Concurrent Training

The memory sharing could not be accomplished if the forward computations of the co-located jobs (issuing many allocation requests) were scheduled simultaneously — the memory will oversubscribe before the deallocation starts. The order of memory requests from the jobs, or called the operation scheduling sequence, determines whether the requested memory will exceed the GPU memory capacity.

To facilitate the following description, we use *allocation part* to represent the training process that mainly produces the allocation requests, and *deallocation part* to represent the training process that frees the memory. Prior works all adopt the *mini-batch-level* scheduling approach to scheduling the allocation part of one job and the deallocation part of another in parallel. There are lots of operations (over thousands, even hundreds of thousands) in an iteration. Hence, the execution order of the operations in the concurrent training is determined totally by the process scheduling in CPU. It is almost impossible to obtain exactly the same operation scheduling order for so many operations in two scheduling solutions. Besides, lots of allocations also occur (up to tens of GB) in the deallocation part. Let us assume a situation where the memory footprint is close to the memory limit and the free memory is only capable of scheduling one operation. If we schedule the operation that will free the memory first, the released memory can serve

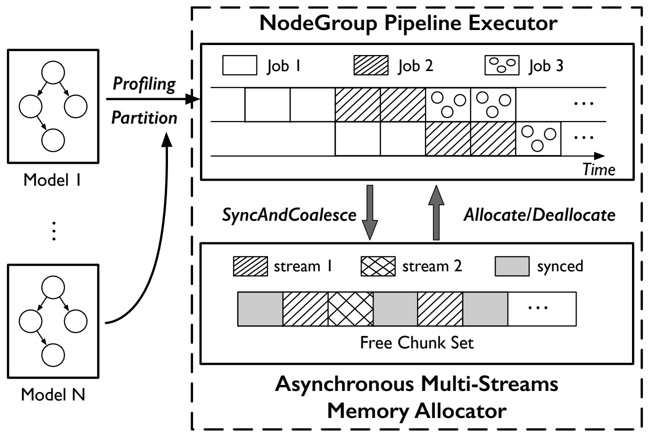


Fig. 4: *Waterwave* System Architecture

the next operation that needs the memory allocation. But if the operation that needs the allocation is scheduled first, there is not enough memory for scheduling the operations that will free the memory. Therefore, such coarse-grained scheduling is much easier to oversubscribe the memory in a scenario of high memory usage, which leaves the *"fate"* to the sequence in CPU process scheduling. Moreover, it is obvious that too fine-grained scheduling (like operation-level) in a centralized way will introduce the scheduling overhead. Therefore, *the second challenge is to determine the appropriate level of granularity at which multiple jobs' computations are scheduled and the memory freed by one job can exactly be used by other jobs.*

4 Design of *Waterwave*

4.1 Design Overview

The overall architecture of *Waterwave* is shown in Figure 4. When multiple jobs attempt to share the GPU, the iterations of these jobs are first scheduled sequentially, which operates in a similar way as the temporal sharing with all model parameters being kept in GPU memory. The purpose is to profile the execution characteristics of each job in its solo run and decide the memory sharing strategy accordingly. Waterwave re-designs two modules in the current DL frameworks: *allocator* and *executor*, aiming to address the two aforementioned difficulties.

Firstly, we develop an *asynchronous multi-streams memory allocator*, which frees the memory when the kernel's scheduling is completed rather than wait until its computation is finished in GPU. The intuition is to make use of the fact that allocations and deallocations in the same stream are sequential, which makes it safe to allocate a free chunk that is asynchronously deallocated in the same stream. Furthermore, if the synchronization function of the streams has been invoked (e.g., streams synchronization) prior to a specific memory allocation, any chunks that have been deallocated before this allocation are accessible regardless of which streams they are in. Only when trying to allocate a chunk that is deallocated in different streams does the explicit synchronization becomes necessary.

Although the accessible memory can be obtained from other streams by synchronizing the streams explicitly, it harms the parallel execution of multiple jobs. Moreover, the increase in the accessible memory after the synchronization depends on the sequence in which the memory is allocated and freed. To address this issue, we propose the *NodeGroup pipeline executor*, which divides the computation graph of each job into a set of continuous *node groups* with the similar total (de)allocation size. Then the node group with the positive allocation size is scheduled in conjunction with the node group with the negative allocation size.

Next, we will dive into the details of the *asynchronous multi-streams memory allocator*, *NodeGroup pipeline executor*, and the graph partition and scheduling algorithms.

4.2 Asynchronous Multi-streams Memory Allocator

In DL frameworks, the typical memory allocator is oblivious of any additional information about a memory chunk. This implies that when an allocation request comes in, we have no idea if the allocator's returned chunk has been deallocated (and if so by which stream) and if it is a *"clean"* chunk that has never been used. To ensure the memory correctness, current works opt to release the memory until the relevant GPU computations are completed. Since all memory allocations occur in the CPU time view, the time gap between CPU scheduling and GPU computation rapidly cause the memory footprint to increase rapidly.

A basic idea is to presume that the memory is released when the scheduling in CPU finishes. Then when an allocation request is received, we can still attempt to search for a 'clean' chunk or the free chunks that are deallocated in the same stream. The latter chunks can be used safely without synchronization since all operations in the same stream are executed sequentially. Based on the above discussions, we design a memory allocator tailored for multiple streams. Note that two streams are sufficient for parallel allocation and deallocation, and often contain enough computations to saturate the GPU compute resources [26], [27]. Thus, we only use two compute streams in the current implementation. Nonetheless, the multi-stream memory allocator can support any number of streams.

```
class Allocator{
  void* Allocate(int64 bytes, int stream_id);
  void Deallocate(void* addr);
  void SyncAndCoalesce();
  vector<FreeChunkSet> no_synced_free_chunks;
  FreeChunkSet synced_free_chunks;
}
```

Listing 1: Multi-streams Memory Allocator

The *Allocator* structure is partially illustrated in Listing 1. We will mainly discuss the distinctions between the multi-stream allocator and a traditional allocator. The `stream_id` specifies the stream that the allocator should perform, which is the same as the corresponding operation. In a traditional allocator, there is only one `FreeChunkSet` to store the freed chunks. However, in a multi-stream environment, we divide the free chunks into two categories according to their synchronization property. The first one is `no_synced_free_chunks`, which is used to hold the free chunks that are deallocated without synchronization. Each stream is configured with its own `no_synced_free_chunks` set. That is, if an allocation in stream `A` wishes to utilize the `no_synced_free_chunks` in stream `B`, a memory error may occur as the result of undefined concurrent activities on the same chunk. The second is `synced_free_chunks`, which indicates that all free chunks in this set have been synchronized. For example, all chunks associated with the job will become synchronized at the end of an iteration. Prior to the execution, all chunks are initialized as *synced free chunk*. While the free chunks are summarized in two categories, there are actually three distinct types of free chunk for a given allocation: 1). *no synced free chunk* with the same stream id; 2). *no synced free chunk* with a different stream id; 3). *synced free chunk*.

a) Allocate: When an allocation request is received, a *prioritized search* will be performed across the three kinds of chunks. Firstly, we search for free chunks with the same stream id in the `no_synced_free_chunks`. It behaves just like the case where a single job performs the computation in a single stream. Thus the synchronization is not required. Secondly, we consider the case of `synced_free_chunks`. While these two types of chunks can both be safely used without synchronization for the current allocation, the former has the more restricted usage: the allocations that occur in other streams cannot use these chunks for free (synchronization is needed). Therefore, we begin with searching for the former type of chunk. If neither of the above two searches succeeds, we have to consider the `no_synced_free_chunks` in other streams. Here, a stream's synchronization function is invoked to convert the *no synced free chunk* to the *synced free chunk*.

b) Deallocate: When a chunk is ready to be deallocated, it is placed in the associated `no_synced_free_chunks` with its allocation stream id as the index. To avoid the memory fragmentation, a coalescing function is often performed to coalesce the free chunks. The difference between this and the multi-stream case is that this free chunk can only be coalesced with other free chunks with the same stream id in `no_synced_free_chunks`. It is self-evident that it cannot be coalesced with other streams' freed chunks. While coalescing with *synced free chunks* is safe and does not need synchronization, this synced free chunk will become a `no_synced_free_chunks`, which reduces the available memory size of the synced free chunks. This is detrimental to the subsequent allocations. Therefore, we defer such coalescing when we have to select *no synced free chunks* in distinct streams in `Allocate`.

c) SyncAndCoalesce: The traditional allocator for a single stream does not adopt this approach. It is obvious that when an allocated *synced free chunk* is deallocated, the *synced free chunk* will be converted to *no synced free chunk*. Conversely, the approach in a traditional allocator aims to mark all chunks in `no_synced_free_chunks` as *synced* for all streams and try to coalesce all the free chunks. To ensure that this operation can change the *synchronization* property of the chunks correctly, a stream's synchronization function needs to be invoked explicitly. It may be triggered by the allocator, for example, when selecting the chunks from `no_synced_free_chunks` in other streams, or by the executor, e.g., at the end of each training iteration.

4.3 NodeGroup Pipeline Execution

Although the *multi-stream allocator* attempts to first allocate *no synced free chunk* in the current stream, this kind of memory is extremely rare when the memory demand is large. Thus, the critical issue for ensuring the *high availability* and *high-performance* of *multi-streams allocator* is how we can always reserve enough *synced* free chunks for subsequent allocations. On the one hand, the synchronization at the end of each iteration can only yield the synchronized free chunks when the computations of the iteration are completed in GPU, despite the fact that a significant amount of memory has already been deallocated during the scheduling of the iteration. On the other hand, relying on the active synchronization in *allocator* cannot guarantee there are enough *synced* memory available after the synchronization. For instance, if two jobs' allocation parts are both scheduled first, the number of free chunks will be depleted rapidly. Additionally, excessively frequent synchronization will degrade the training performance. Briefly speaking, we aim to enable the deallocated memory from one job to be reused by other jobs *immediately* and *at a low cost*. To tackle this problem, we develop a *NodeGroup pipeline executor* that allows for fine-grained scheduling of computations.

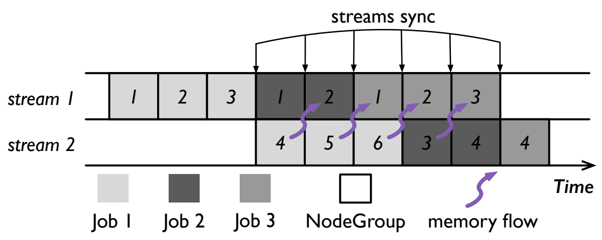


Fig. 5: An Example of NodeGroup Pipeline Execution on Three Jobs (a purple arrow represents that the freed memory from a DNG can be used by the very next ANG, i.e., symbolizing the *memory flow*.)

Let us first define the concept of *NodeGroup* ($\mathbb{NG}$). Given a compute graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ of a job, it will be partitioned into a set of continuous *NodeGroups*, which is a subgraph of $\mathbb{G}$. The node groups of a job are processed in strict sequence regardless of which stream they are dispatched to. For a *node group*, the *root nodes* are the nodes that will be scheduled first while the *leaf nodes* are the last to be scheduled. The completion of the leaf nodes' scheduling indicates that the scheduling of next node group can start. Notice that the set of leaf nodes can be empty if all nodes in the node group have no dependency to each other. In such circumstances, the root nodes are also the leaf nodes.

When a node group is determined, the total (de)allocation memory size can be calculated using the allocations information of each node inside this *node group*. When this value is positive, it indicates that the specified amount of memory will be allocated, while the memory will be released when the value is negative. They are called *allocation node group* (ANG) and *deallocation node group* (DNG) respectively. Considering the memory usage pattern during an iteration, a job's node groups can be divided into a series of continuous ANGs followed by a series of continuous DNGs. Figure 5 presents an example of three jobs' pipeline execution. These three jobs are partitioned into six, four, and four node groups respectively with the first half being ANGs. For the sake of convenience, we denote the first node

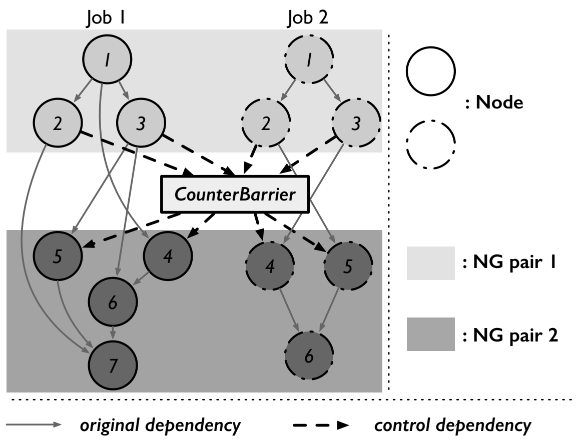


Fig. 6: The Implementation of *CounterBarrier* that Ensures the Sequential Execution Order of NodeGroup Pair; the *original dependency* is represented by the *solid arrows* in the compute graph whereas the *control dependency* is represented through the *dotted arrows* added by the *CounterBarrier*.

group of `Job 1` by J^1NG_1. The node groups aligned from different streams are processed in parallel, e.g., J^1NG_4 and J^2NG_1, J^2NG_3 and J^3NG_2. Such parallel node groups are referred to as a *node group pair*. The next node group pair is not scheduled until the scheduling of the previous node group pair is completed. Moreover, each time when a new node group pair is scheduled, a streams synchronization function is invoked, which also calls `SyncAndCoalesce API` in the allocator to transform all previous *no synced free chunks* to *synced free chunks*. In this way, the allocations in the current node group pair can use the freed memory of the previous node group pair directly. This direction of memory flow is depicted by the purple arrow in Figure 5.

Notice that we have made an assumption that the node group pairs can be scheduled sequentially. However, this cannot be guaranteed by the scheduler in a current DL framework since it is possible that there is no dependency between the leaf nodes of previous node group pair and the root nodes of and the current node group pair. In Figure 6, for instance, *Node* 4 in `Job 1` is ready to be scheduled once *Node* 1 has finished the scheduling. However, we want it to be scheduled when Node 1-3 of two jobs (i.e., NG_1) have both finished the scheduling. Hence, we need a mechanism to implement such scheduling semantic. Although it is easy to implement such control dependency by adding the *edges* in the original compute graph, it is not that flexible as the jobs are *"free to come and go"*. Figure 6 depicts how we ensure the scheduling dependency between the adjacent node group pairs. We use a *CounterBarrier (CB)* that is initialized with two `int` values: the leaf nodes number of the previous node group pair (*prev_count*) and the root nodes number of the current node group pair (*succ_count*). A *CB* is shared by the adjacent node group pairs and can be added or removed on-the-fly. During the scheduling, the leaf nodes will decrease *prev_count* whereas the root nodes decrease *succ_count* of the same *CB*. Each time before scheduling a root node, it will check whether *prev_count* of the corresponding *CB* is zero, if it is, this

Algorithm 1: The Scheduling Algorithm

```
Input  : jobs, mem_limit
Output: two vectors: av and dv
1  $Sort(jobs)$;
2  $f\_numANG \leftarrow jobs.front().numANG$;
3  $f\_peakMem \leftarrow jobs.front().peakMem$;
4  $p \leftarrow (mem\_limit - f\_peakMem)/split\_size - 1$;
5  $offset \leftarrow f\_numANG - p$;
6  $idx1 \leftarrow 0$, $idx2 \leftarrow f\_numANG$;
7  foreach job in jobs do
8      $s \leftarrow isSecondJob(job)\ ?\ p\ :\ 0$;
9      /* Schedule job's ANGs */
10     if isSecondJob(job) then
11         for $i = 0$ to $s$ do
12             $dv[offset++] = \{i++, job.job\_id\}$
13         end
14     end
15     for $i = s$ to $(job.numANG - 1)$ do
16         $av[idx1++] = \{i++, job.job\_id\}$;
17     end
18     /* Schedule job's DNGs */
19     while $i < job.numNG$ do
20         if $isLastJob(job)$ and $idx2 > idx1$ then
21             $av[idx1++] = \{i++, job.job\_id\}$;
22         end
23         else
24             $dv[idx2++] = \{i++, job.job\_id\}$;
25         end
26     end
27 end
```

node will be scheduled. Otherwise, it will *wait*. When the value of *succ_count* becomes zero, the *CB*'s state will be reset to be used for the next iteration. These two decrement operators are implemented by `atomic` and `lock` operations respectively, which are efficient and introduce nearly no scheduling overhead.

4.4 Graph Partition and Scheduling

In this section, we describe how to partition the graph into a series of continuous node groups and assign them to different GPU streams.

The partition goal is to ensure that each node group has a comparable size of overall (de)allocation memory, which is called *split_size*. In the profiling phase, we have obtained the allocation information of each node in a computation graph, as well as the graph topology. With this information, we first perform a Breadth First Search (BFS) on the graph to obtain the depth of each node[4]. The nodes with the same depth are arranged in the ascending order of *node id* (a number to identify a node). Then, starting at the first depth, we traverse the graph, adding nodes to the current node group until the total (de)allocation size meets the predefined *split_size*. The scheduling aims to schedule as many parallel node groups as possible subject to the GPU memory limit. For instance, if two co-located jobs have such peak memory usages that

4. There are the cycles in RNNs. We remove the cycles through DFS and then perform BFS.

their total does not exceed the GPU memory limit, both jobs can be scheduled simultaneously. However, if their peak memory usage is too high, we must delay the scheduling of the second job until the sufficient memory is available (i.e., wait for the DNGs of the first job to free the memory). The challenge then becomes determining an appropriate value for the delay, in other words, determining the point at which enough node groups of the first job have been scheduled to allow the scheduling of the second job to commence. By carefully controlling the scheduling of ANGs and DNGs to different GPU streams, we can ensure high utilization of memory and computing resources.

The scheduling algorithm is shown as Algorithm 1. We use the pair $\{$*node group id, job_id*$\}$ to denote a unique node group. The output of the algorithm is two vectors that represent two streams, in which each element is a unique node group. The node groups that have the same index in two vectors are a pair of node groups that will be scheduled in parallel. At the beginning, we arrange the jobs in descending order of their node groups' sizes, which is to minimize the pipeline bubbles caused by node group dependency. We decide the number of ANGs of the second job (denoted by p) that can be scheduled in parallel with ANGs of the first job according to the total GPU memory, peak memory of the first job and split size (line 4 in Algorithm 1). The reason why we decrease by one is because we want a DNG to finish first so that the memory is freed for next ANG. Except for the first p ANGs in the second job being scheduled to the second stream, the rest ANGs are all scheduled to the first GPU stream, i.e., *av*. For the DNGs of the last job, we will schedule them to *av* first if there are still empty place (line 20-21 in the algorithm), which can increase parallelism. The rest DNGs will be scheduled to the second stream. We can then add the *CounterBarrier* between the adjacent node group pairs in the output vectors.

In practice, it can be difficult to determine an optimal *split_size* that ensures each node group has a comparable size of overall (de)allocation memory, particularly when there are significant architecture differences between two different DNN models. In such situation, we may relax the criterion by allowing for greater variance in the memory size of the node groups. We also prefer to guarantee that the memory sizes of DNGs are greater than that of ANGs to ensure that there is sufficient memory for the next ANG when the scheduling of the current DNG is completed, although this may come at the cost of modest memory sharing efficiency. Additionally, the selection of appropriate jobs for co-running is another research topic known as deep learning cluster scheduling, and is beyond the scope of this work.

Note that we aim to propose a simple but effective graph partition and scheduling algorithm. There can be more delicate and intricate algorithms. For example, it is unnecessary to guarantee all node groups' (de)allocation memory size being equal, because it is good enough that in the adjacent node group pairs, the allocation memory size of ANGs is comparable to the deallocation memory size of DNG. So the compute graph can be partitioned with a variable *split_size*.

5 Evaluation

5.1 Methodology

5.1.1 Experimental setup

Our experiments were conducted on two servers. The first server is equipped with NVIDIA Tesla V100 GPU with 32 GB GPU memory, dual 2.00Ghz Intel Xeon Gold 5117 CPU and 256 GB RAM. We refer to it as V100-S for simplicity. The second server is equipped with NVIDIA Tesla P100 GPU with 16 GB GPU memory, dual 2.60GHz Intel Xeon CPU E5-2680 v4 processors and 256 GB RAM. We refer to it as P100-S. Both servers are installed with Ubuntu 16.04, the CUDA Toolkit 10.0 and cuDNN 7.6.5. TensorFlow of version 1.15.2 (based on which *Waterwave* is modified).

5.1.2 Workloads

We evaluate *Waterwave* on 7 state-of-the-art deep learning workloads including 1) CNN models: ResNet50 [12], ResNet152 [12], InceptionV3 [28], and NasNet [29]; 2) RNN models: Tacotron2 [9]; 3) Transformer model: Bert [13]. All the CNN models are very popular in the CV field. NasNet is a neural network that is computed by the NAS (Neural Architecture Search) algorithm. All CNN models use the stochastic gradient descent (SGD) optimizer. BERT is proposed by Google AI Language and has achieved the state-of-the-art results in a wide variety of NLP tasks. The base version of BERT includes 768 hidden layers of 110 million parameters. Tacotron2 is a neural network for speech synthesis directly from text, which includes 29.016 million parameters. BERT and Tacotron2 use the default *Adam* optimizer.

5.1.3 Baselines

We have set three baselines to evaluate the memory sharing efficiency and training throughput, which are listed below.

- NVIDIA MPS: Serve multiple GPU computing jobs by intercepting the specific CUDA `API` and re-schedule them. UVM is also enabled with MPS to meet larger memory requirement.
- Temporal sharing: We simulate the temporal sharing in prior works by disabling the profiling phase in *Waterwave*.
- *Waterwave*-s: In order to demonstrate the memory sharing efficiency of fine-grained scheduling, we implement other two coarser-grained graph partition algorithms. One partitions the graph from the *peak memory node* into two *node groups*. The *peak memory node* means that the memory footprint reaches the maximum size when scheduling this node, which is obtained in the profiling phase. The other algorithm partitions the graph into two *node groups*: forward propagation part and backward propagation part, which is similar to the method adopted in Wavelet [20]. In the following experiments, we select the better algorithm and denote it as *Waterwave*-s.

5.2 Memory Sharing Efficiency

In this section, we evaluate the memory sharing efficiency of *Waterwave* on both V100-S and P100-S. We use the

TABLE 1: Maximum Batch Size of Running Two Identical Models

	V100-S					P100-S				
Models	Sole	MPS	*Waterwave*-s	*Waterwave*	*Waterwave*/*Sole*	Sole	MPS	*Waterwave*-s	*Waterwave*	*Waterwave*/*Sole*
ResNet50	406	384	266	370	91.61%	190	190	120	176	92.63%
ResNet152	186	176	118	160	86.02%	86	86	62	78	90.70%
NasNet	560	556	316	544	96.80%	280	265	192	260	92.86%
InceptionV3	332	328	204	310	93.34%	160	160	106	150	93.75%
BERT-base	138	136	88	124	89.86%	60	56	38	48	80.00%
Tacotron2	198	198	96	152	76.77%	98	94	48	72	73.47%

maximum batch size to quantify the efficiency of memory sharing. To demonstrate that, we also test the largest batch size that a model can achieve when being run solely. Due to the fact that temporal sharing does not share the GPU memory between jobs, the temporal sharing is excluded from this comparison. Nonetheless, the maximum batch size for temporal sharing is predictable, which is equal to the maximum batch size for the solo run when the context switching is utilized, and is somewhat smaller (depending on the quantity of model parameters) when the context switching is not employed. Besides, there is actually no memory sharing in MPS, in which the batch size increment results from UVM leveraging extra CPU memory to swap the GPU memory.

We first conduct the experiment with two identical models running on the same GPU. The results are shown in Table 1, which include the maximum batch sizes of the six models that sole run, MPS, *Waterwave*-s, and *Waterwave* can achieve. MPS exhibits the maximum batch size among all models, thanks to the power of unified virtual memory that automatically swaps the memory between CPU host memory when GPU memory is oversubscribed. Nonetheless, the maximum batch sizes that *Waterwave* can achieve are also comparable to those in sole run. Specifically, *Waterwave* achieves the best memory efficiency when training NasNet on V100-S, which is 96.8% of maximum batch size in sole run, and 92.2% on average on the CNN models. This ratio on P100-S is usually smaller than that on V100-S, especially for the Bert-base. This is because Bert-base has the most model parameters among these six models. As a result, twice size of the model parameters has a greater impact on the P100-S, which has only half the GPU memory capacity of the V100-S.

In comparison to the CNN models and Bert-base, Tacotron2's memory sharing efficiency is smaller, which is less than 80%. There are two reasons for this. On the one hand, the memory requirement of a *node* increases with the batch size, which implies that we cannot divide the compute graph in a too fine granularity. On the other hand, the number of kernels launched in each iteration of Tacotron2 is far bigger than those of the other models. Thus, there are still lots of operations in each node group after partitioning the compute graph, which makes it difficult to schedule memory requests in the desired order. By the way, we partitioned Tacotron2's compute graph into eight node groups to achieve the maximum batch size.

Compared to *Waterwave*-s, *Waterwave* outperforms by around 1.5x in all six models. Notably, the maximum batch size of *Waterwave*-s is obtained by running a batch size six times. This batch size is recorded as long as one of the runs succeeds. It is possible that *Waterwave*-s failed with this batch size in other five times. This is because such coarse-grained scheduling cannot organize the memory requests in the desired order, resulting in unstable results.

TABLE 2: Maximum Batch Size When Running Two Non-identical Models

Maximum Batch Size (ModelA + ModelB)	Model A	Model B
ResNet50 + Bert-base	358	116
ResNet50 + Tacotron2	370	150
Bert-base + Tacotron2	120	150

Additionally, we conduct concurrent training on the different types of models. We choose three combinations according to the model type: 1) ResNet50 + Bert (CNN+Transformer); 2) ResNet50 + Tacotron2 (CNN+RNN); 3) Bert + Tacotron2 (Transformer+RNN). The results are shown in Table 2. As can be seen, the maximum batch size of each model is basically equivalent to the maximum batch size for running two identical models.

5.3 Concurrent Training Throughput

In this section, we evaluate the concurrent training throughput (the total training speeds of the co-located jobs) of *Waterwave* against MPS and temporal sharing on V100-S. We exclude *Waterwave*-s for the throughput comparison since it is mainly set for demonstrating the memory sharing efficiency of fine-grained partition and scheduling. The throughput of *Waterwave*-s is almost the same as that of *Waterwave* as long as the batch size does not exceed the GPU memory limit. Note that we also conduct the experiments on P100-S. The conclusions are identical to those on V100-S. Hence, we only present the results on V100-S due to space limitations.

Figure 7 shows the throughput of the six models under different batch sizes. When the batch sizes do not exceed the memory limit, *Waterwave* and MPS show the comparable throughput, with MPS often outperforming *Waterwave* by a very small margin. This is due to the fact that *Waterwave* runs on a single framework, while MPS runs different jobs on different framework instances and processes, which result in the increased scheduling latency when accessing shared components in *Waterwave* such as the memory allocator and operator library. Compared to temporal sharing, the throughput improvement is limited in the CNN models and Bert-base. *Waterwave* obtains the highest promotion rate of 14.27% in ResNet152 with the batch size of 32. This is because CNN and Transformer-based models show the great degree of parallelism, which can occupy the most GPU compute resources even with a relatively small batch size. But in Tacotron2, which cannot fully utilize GPU, *Waterwave*

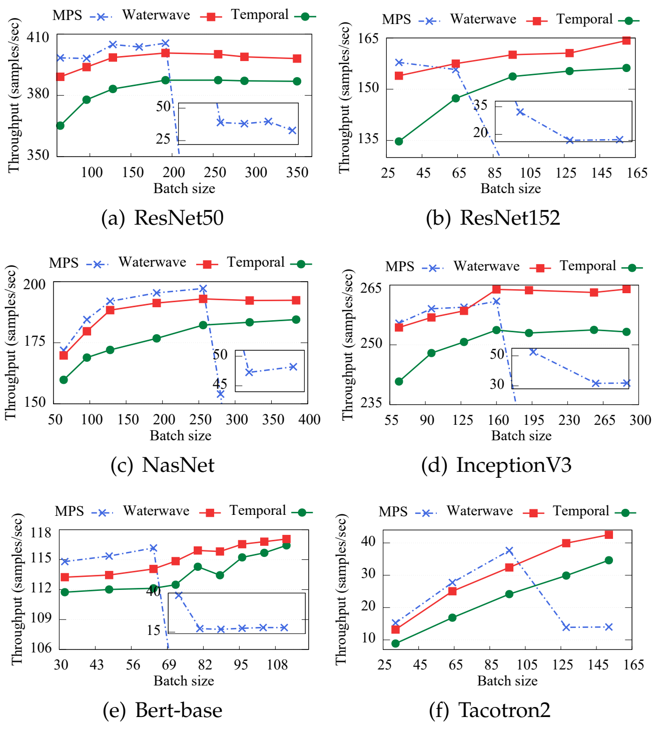


Fig. 7: Throughput of Training the Same Models

outperforms temporal sharing by up to 1.49x. It is worth noting that the result of temporal sharing in our configuration does not include the context switching overhead.

As the batch size grows much larger, the throughput of MPS begins to decline substantially (by up to more than 90%). This is due to the fact that when the GPU memory is oversubscribed, UVM will swap memory pages between GPU and CPU, which cause the GPU computation to stall. As a result, such frequent memory swapping significantly reduces the overall training throughput. Because of this, compared to MPS, *Waterwave* is over 12 times faster in ResNet50, 6.8 times faster in Bert-base, and 3.1 times faster in Tacotron2.

5.4 Effect of Split Size

The split size has the impact on the number of node groups created after the graph partition, which determines the number of times we need to synchronize the streams in an iteration. In this section, we will evaluate the maximum batch size and the training throughput on V100-S with varied split sizes. We select split sizes of 2, 4, 6, 8, and 12 GB for ResNet50, NasNet, InceptionV3, and Bert-base. An extra split size of 1 GB is evaluated on ResNet152. Due to the fact that we need to guarantee the dependency between node group pairs, we cannot cut the compute graph from a node that is in a cyclic path in the RNN models, because such nodes will be executed for multiple times (related to input) in an iteration. This constraint leads to an imbalanced total memory size of each node group in Tacotron2. Thus we use "Number of NodeGroups" instead.

The results are depicted in Figure 8. For the first four models, we can observe that the maximum batch size and throughput are both rather tiny at the split size of 2 GB.

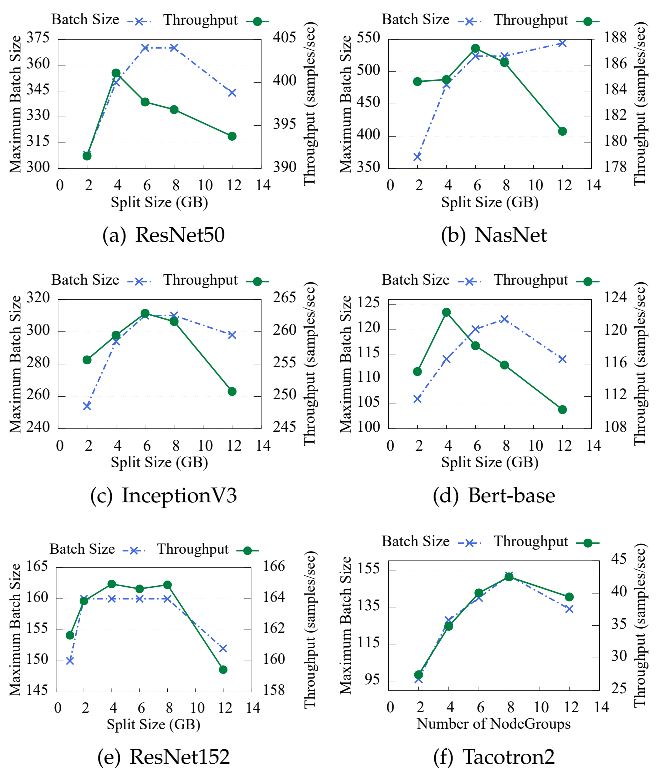


Fig. 8: Maximum Batch Size and Throughput under Different Split Sizes

The reason is that small split size introduces more stream synchronization and also makes it difficult to handle the nodes with large size of allocation requests. The batch size is maximized when the split size is set to 6 GB across these models. The corresponding throughput is either optimal or suboptimal. NasNet, on the other hand, achieves its maximum batch size with a split size of 12 GB. This is because NasNet has the greatest batch size, which results in the largest allocation size for a single node and requires a bigger split size to schedule. As the split size continues to increase, both the maximum batch size and throughput decrease, since it becomes more difficult to maintain the order of memory requests. This will subsequently cause the allocator to begin synchronizing the streams aggressively, which results in performance degradation. This conclusion can also be drawn from Tacotron2's result. In ResNet152, the maximum batch size can be achieved at the split size of 2 GB and remains the same at the split sizes of 4, 6, and 8 GB. This is because the large memory footprint of ResNet152 is caused by the biggest model depth rather than the large batch size compared to other CNN models. This means the maximum value of a single memory allocation in ResNet152 is smaller than other models. Therefore, a smaller split size is capable of achieving the best memory sharing efficiency.

5.5 Training More than Two Jobs

This section demonstrates *Waterwave*'s capability in enabling concurrent training for more than two jobs. We configure to run three and four ResNet50 models concurrently on V100-S and evaluate the corresponding maximum batch size and throughput. The maximum batch sizes are 336 and 310. The throughput is 402.76 and 401.58, which is slightly higher than those when two models are run concurrently. This result is because of the following two reasons: 1) we only use two streams for the computations currently as stated in Section 4, more jobs will not increase the degree of parallelism. 2) ResNet50 is a model with high GPU utilization. The GPU compute resource is already saturated even with two jobs running concurrently. However, we believe *Waterwave* is becoming increasingly valuable as GPU computing power continues to grow rapidly, while the GPU memory capacity increases at a relatively slower speed.

6 Discussion

a) Distributed training: Distributed training is common to speedup the overall training process and support massive models, such as GPT-serial models [30], [31]. It includes data parallelism, model parallelism [32], and pipeline model parallelism [33], [34], [35]. The distributed training is complicated than single-GPU training as it introduces communication and parameters aggregation across GPUs. Nonetheless, the memory usage of an iteration inside a single GPU still follows *"first increase then drop"* pattern. Therefore, the idea in *Waterwave* is still applicable in distributed training. But the selection of jobs to co-locate with distributed training job needs to be more careful, since the speed slowdown on one GPU lead to the performance degradation of overall distributed training. We leave the extension of *Waterwave* to distributed training as a future work.

b) Job priority: There are usually two kinds of jobs in DL GPU cluster: *high-priority job* and *low-priority job*. When co-locating jobs with different priorities, it's an important and challenging problem to provide performance guarantee on jobs with high-priority. AntMan [27] made an attempt to achieve it through limiting the operator launch rate of low-priority jobs. The current scheduling in *Waterwave* does not take the job priority into account. Nonetheless, it's easy to integrate the batch scheduling policy or operator scheduling policy into *Waterwave* to achieve prioritized scheduling, since all jobs' batches are submitted to *Waterwave* and the operator scheduling can also be achieved via NodeGroup abstract in *Waterwave*. And it's an interesting topic that how to achieve memory sharing efficiency while guaranteeing the job priority. We leave the consideration of job priority and computation time as a future work.

c) Balancing computation in graph partition and scheduling: In graph partition and scheduling of *Waterwave*, we merely consider balancing the memory footprint of each partition, ignoring the balance of computation. In theory, if both jobs have high GPU computation demand so that can fully utilize the GPU, then imbalanced computation of node groups will have little impact on performance. However, if one of the models has lower computational demands, such imbalanced computation will degrade the performance to some extent. A significant challenge in considering the computation time is that the parallel kernel scheduling in NVIDIA GPU is opaque. Therefore, it is difficult to predict whether the kernels in question can be parallelized and also difficult to assess the performance of the parallelized kernels. As a result, it is challenging to determine if a particular partition and scheduling policy can achieve a higher training throughput.

d) Security: Security is crucial for DL training as this process could last long for months, an accident exiting due to unpredictable faults will waste expensive GPU resources. When jobs are running in their own CUDA context on the same GPU, the fault only affects themselves, not to other jobs. However, in order to share the GPU memory resources across jobs, they need to run in the same CUDA context, which will break the *fault isolation*: the failure in a job doesn't impact other co-located jobs. This security issue exists in the frameworks that merge numerous jobs' contexts into one, such as NVIDIA MPS [11], Salus [8], and also *Waterwave*. A good aspect is that DL training job can be recovered at an arbitrary step as long as the training state (parameters) at that step has been saved. And this recovery is lightweight since the time to run a single training step is small (seconds level). But too frequent checkpointing will also influence the overall training performance. The users need to balance the frequency of checkpointing to make a tradeoff. On the other side, another method to support concurrent running of multiple jobs while guaranteeing fault isolation is virtualizing a GPU to numerous virtualized GPUs (vGPU) [36]. The latest technology is Multi-Instance GPU (MIG) [37] which is provided by NVIDIA and supported on the NVIDIA Ampere architecture and after (such as A100 and H100 GPU). MIG partitions a single GPU into numerous separate GPU instances where each one owns the dedicated compute, memory, and memory bandwidth resources. But it has limitations for supporting concurrent training. First, the compute and memory resource configurations of a GPU instance are pre-defined by NVIDIA that cannot be changed at will. Besides, when the GPU instances have been configured, the resources cannot be adjusted flexible on-fly. This collides with the diverse and dynamic resources requirement of DL training jobs. Second, the GPU memory resources are isolated across instances, thus cannot be shared.

7 Related Work

7.1 GPU Sharing in Training

Gandiva [7] and Salus [8] both propose a *time slicing* method on GPU which runs multiple models alternatively. The main difference between them is that before running a new model, Gandiva needs to swap the context of the current model out of GPU and swap the context of the new model into GPU. This *context switching* overhead is non-trivial for large models (200+ ms). Conversely, Salus opts to store all model's parameters in GPU memory, and hence does not introduce the context switching overhead.

Zico [19] adopts spatial sharing that manages the memory as a set of chunks with identical sizes and releases them when the computations have finished in GPU. It is hard to choose the chunk size since too small chunk size requires more CPU resources, which degrades the performance further for the models that are already bounded to preprocessing in CPU [38], [39], while a too big size limits the memory sharing space, which in turn limits the number of in-flight kernels and make the computation speed of GPU to be comparable to the CPU scheduling. But a fixed number may not work well with the training of the whole model. For the scheduling part, Zico performs the *mini-batch* scheduling while *Waterwave* schedules the computation in a finer-granularity to achieve precise memory sharing. Besides, Zico cannot support more than two jobs currently.

7.2 Memory Optimization in Single Model's Training

There are many prior works aiming to optimize memory usage when a single job is trained, which can be categorized into three kinds according to the techniques: 1) *swap*, which leverages the CPU DRAM as an external storage to swap the GPU memory; 2) *recomputation*, which drops the feature maps in the forward pass and recompute them in the backward pass; 3) *compression*, which compresses the data in a more memory-saving encoding.

vDNN [14] is the leading work in *swap*, which chooses the inputs of the convolutional layers as the target since they tend to overlap the swapping overhead with the heavy comptuation of the convolutional layer. SwapAdvisor [40] optimizes the swapping decisions by taking both memory allocation and operator scheduling into account. SuperNeurons [15] and Capuchin [16] optimize memory through both swapping and recomputation. The swapping and recomputation tend to optimize memory usage according to the characteristics in the single-job training, while we aim to enable the efficient memory sharing in concurrent training. By observing the feature of particular model architecture, Gist [41] and CDMA [42] compress the tensor to reduce the memory footprint. The compression methodology is orthogonal to our work.

8 Conclusions

This paper proposes *Waterwave*, aiming to make the GPU memory *flow back and forth* smoothly among the concurrently running DL training jobs. *Waterwave* accomplished the goal through the coordinated allocation and scheduling. By making the allocator aware of the stream information associated with the chunks, the allocator can perform an asynchronous and prioritized de/allocation in the multi-stream scenario. Through partitioning the compute graph into a series of fine-grained node groups, we gain the ability and flexibility to order the memory requests as desired. The experimental results demonstrate that *Waterwave* delivers the efficient memory sharing and throughput compared to the existing works.

References

[1] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*, 2017, pp. 1–12.

[2] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization." *Journal of machine learning research*, vol. 13, no. 2, 2012.

[3] J. Rasley, Y. He, F. Yan, O. Ruwase, and R. Fonseca, "Hyperdrive: Exploring hyperparameters with pop scheduling," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, 2017, pp. 1–13.

[4] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6765–6816, 2017.

[5] K. Maziarz, M. Tan, A. Khorlin, M. Georgiev, and A. Gesmundo, "Evolutionary-neural hybrid agents for architecture search," *arXiv preprint arXiv:1811.09828*, 2018.

[6] K. Kandasamy, W. Neiswanger, J. Schneider, B. Poczos, and E. P. Xing, "Neural architecture search with bayesian optimisation and optimal transport," *Advances in neural information processing systems*, vol. 31, 2018.

[7] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou, "Gandiva: Introspective cluster scheduling for deep learning," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 595–610.

[8] P. Yu and M. Chowdhury, "Fine-grained GPU sharing primitives for deep learning applications," in *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, I. S. Dhillon, D. S. Papailiopoulos, and V. Sze, Eds. mlsys.org, 2020. [Online]. Available: https://proceedings.mlsys.org/book/294.pdf

[9] J. Shen, R. Pang, R. J. Weiss, M. Schuster, N. Jaitly, Z. Yang, Z. Chen, Y. Zhang, Y. Wang, R. Skerrv-Ryan *et al.*, "Natural tts synthesis by conditioning wavenet on mel spectrogram predictions," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2018, pp. 4779–4783.

[10] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[11] "NVIDIA MPS," https://docs.nvidia.com/deploy/mps/index.html/.

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

[13] 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.

[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 *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2016, p. 18.

[15] 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 *ACM SIGPLAN Notices*, vol. 53, no. 1. ACM, 2018, pp. 41–53.

[16] 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 Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 891–905.

[17] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A System for Large-Scale Machine Learning," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, vol. 16. USENIX Association, 2016, pp. 265–283.

[18] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," 2017.

[19] G. Lim, J. Ahn, W. Xiao, Y. Kwon, and M. Jeon, "Zico: Efficient gpu memory sharing for concurrent dnn training," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 161–175.

[20] G. Wang, K. Wang, K. Jiang, X. Li, and I. Stoica, "Wavelet: Efficient dnn training with tick-tock scheduling," *Proceedings of Machine Learning and Systems*, vol. 3, 2021.

[21] X. Yu, N. K. Loh, and W. Miller, "A new acceleration technique for the backpropagation algorithm," in *IEEE International Conference on Neural Networks*. IEEE, 1993, pp. 1157–1161.

[22] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[23] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla, "Themis: Fair and efficient gpu cluster scheduling," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 289–304.

[24] J. Bergstra, D. Yamins, and D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in *International conference on machine learning*. PMLR, 2013, pp. 115–123.

[25] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *arXiv preprint arXiv:1611.01578*, 2016.

[26] H. Dai, X. Peng, X. Shi, L. He, Q. Xiong, and H. Jin, "Reveal training performance mystery between tensorflow and pytorch in the single gpu environment," *Science China Information Sciences*, vol. 65, no. 1, pp. 1–17, 2022.

[27] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia, "Antman: Dynamic scaling on gpu clusters for deep learning," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 533–548.

[28] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2818–2826.

[29] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," *arXiv preprint arXiv:1707.07012*, 2017.

[30] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[31] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[32] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[33] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in neural information processing systems*, vol. 32, 2019.

[34] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 1–15.

[35] S. Eliad, I. Hakimi, A. De Jagger, M. Silberstein, and A. Schuster, "Fine-tuning giant neural networks on commodity hardware with automatic pipeline model parallelism," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 381–396.

[36] "NVIDIA vGPU," https://docs.nvidia.com/grid/10.0/grid-vgpu-user-guide/index.html.

[37] "NVIDIA Multi-Instance GPU," https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

[38] J. Mohan, A. Phanishayee, A. Raniwala, and V. Chidambaram, "Analyzing and mitigating data stalls in dnn training," *arXiv preprint arXiv:2007.06775*, 2020.

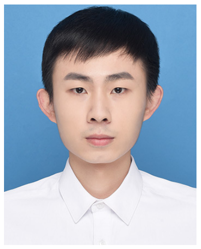
[39] W. Xiao, Z. Han, H. Zhao, X. Peng, Q. Zhang, F. Yang, and L. Zhou, "Scheduling cpu for gpu-based deep learning jobs," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2018, pp. 503–503.

[40] C.-C. Huang, G. Jin, and J. Li, "Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1341–1355.

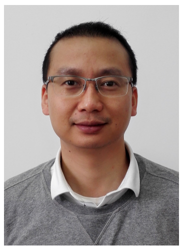
[41] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, "Gist: Efficient data encoding for deep neural network training," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 776–789.

[42] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler, "Compressing DMA engine: Leveraging activation sparsity for training deep neural networks," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 78–91.

Xuanhua Shi (Senior Member, IEEE) received the PhD degree in computer engineering from the Huazhong University of Science and Technology, Wuhan, China, in 2005. He is currently a professor with the National Engineering Research Center for Big Data Technology and System Services Computing Technology and System/Services Computing Technology and System Lab, Huazhong University of Science and Technology (China). From 2006, he worked as an INRIA postdoctoral in PARIS team at Rennes for one year. His research interests cloud computing and big data processing. He published over more than 100 peer-reviewed publications, received research support from a variety of governmental and industrial organizations, such as National Science Foundation of China, Ministry of Science and Technology, Ministry of Education, European Union, Alibaba, ByteDance, Intel and so on. He is a senior member of CCF.



Xuan Peng is currently working toward the PhD degree with the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China. His research focuses on intelligent computing, memory management for AI systems.



Ligang He is now a Reader in the Department of Computer Science at the University of Warwick, UK. His research area is mainly parallel and distributed computing. He has published more than 190 papers in the research area.

Yunfei Zhao received the bachelor's degree in computer science and technology from the Huazhong University of Science and Technology, Wuhan, China, in 2022. Her research interests include intelligent computing, AI systems optimization.

Hai Jin (Fellow, IEEE) received the Ph.D. degree in computer engineering from the Huazhong University of Science and Technology in 1994. He received the German Academic Exchange Service Fellowship to visit the Technical University of Chemnitz, Germany, in 1996. He worked at The University of Hong Kong from 1998 to 2000 and as a Visiting Scholar at the University of Southern California from 1999 to 2000. He received the Excellent Youth Award from the National Science Foundation of China in 2001. He is a Cheung Kung Scholars Chair Professor of computer science and engineering of the Huazhong University of Science and Technology. He has coauthored 22 books and published over 800 research articles. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security. He is a fellow of the CCF and a member of the ACM.