WAVELET: EFFICIENT DNN TRAINING WITH TICK-TOCK SCHEDULING

Guanhua Wang [1] **Kehan Wang** [1] **Kenan Jiang** [1] **Xiangjun Li** [1] **Ion Stoica** [1]

ABSTRACT

DNNs have revolutionized across a wide range of applications, such as image classification, speech recognition and robotics control. As DNN models become more computationally expensive to train, parallel execution with multiple accelerators (e.g. GPUs) is widely-adopted. System efficiency is a big issue when scaling out. However, as computation power increases, GPUs are under-utilized mainly due to limited local memory size. To address this memory bound, we present Wavelet, an efficient and generic approach that can fully utilize all the available on-device memory among GPUs involved in the same distributed training job. Wavelet achieves near optimal on-device memory usage by adopting a simple but novel scheduling scheme called Tick-Tock, which interleaves waves of peak memory usage among the accelerators. Evaluations on a variety of DNN models and tasks show that, Wavelet trains models up to 6.7x faster than commonly used parallelism techniques.

1 INTRODUCTION

Deep Neural Networks (DNNs) enable machine to excel in a wide spectrum of applications, such as language translation (Devlin et al., 2018), image classification (He et al., 2016; Krizhevsky et al., 2012), game playing (Silver et al., 2017), etc. To pursue higher intelligence and model serving performance, both DNN model sizes (Real et al., 2019; Brown et al., 2020) and training datasets (Lin et al., 2015; Deng et al., 2009) grow drastically. Distributed model training across multiple accelerators is widely-adopted to boost up training speed (Goyal et al., 2017). The most popular distributed approaches are data parallelism (Li et al., 2014; Chen et al., 2015) and model parallelism (Dean et al., 2012; Lee et al., 2014). In data parallelism, each accelerator (e.g. a GPU) maintains a full copy of the whole model and conduct local training on a subset of the total dataset (Li et al., 2014; Wang et al., 2020). Model parallelism (Dean et al., 2012) adopts a different scheme, where each device only holds one partition of the whole model and conducts training on the same input data.

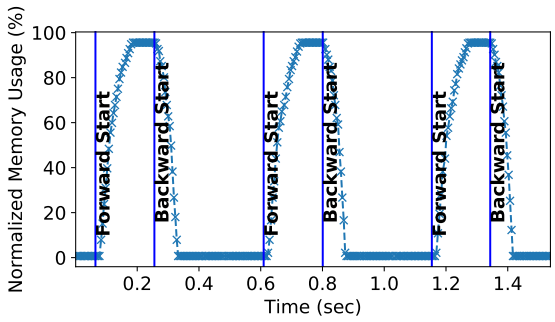
Distributed DNN training typically requires all the training tasks to be launched at the same time, i.e., gang-scheduled (Mahajan et al., 2020; Jeon et al., 2019). The main reason is two-fold: first, hyper-parameters (e.g. learning rate, batch size, etc.) need to be picked given a fixed number of accelerators (Goyal et al., 2017). Second, synchronization happens frequently among all the accelerators involved in the same job (Gu et al., 2019), which imposes the constraint that all the in-parallel training tasks should start and end at the same time in order to avoid stragglers (Or et al., 2020).

Despite the overwhelming popularity of gang scheduling policy in distributed DNN training, it may under-utilize system resources. For accelerators like GPUs, on the memory side, gang-scheduling forces all GPUs to reach the peak and valley of their on-device memory usage at the same time. For a single job, this memory valley period is completely wasted. On the computation side, with state-of-the-art GPUs (e.g. Nvidia A100 (Nvidia A100) and V100 (dgx1; dgx2)), the computation power grows tremendously, but on-device memory is often limited. Thus it makes a wide range of deep learning computation to be memory-bounded, and on-chip compute cores are often under-utilized (Ivanov et al., 2021). Figure 1 depicts a toy example of a gang-scheduled data parallel training job using 2 V100 GPUs on image classification tasks (He et al., 2016). We show the normalized GPU utilization of both memory and compute core usage. The on-device memory is repeatedly underutilized ($> 90\%$ in Figure 1(a)) during backward propagation in every training iteration. The compute core usage is consistently underused ($\sim 70\%$ in Figure 1(b)) during the whole training session due to its limited memory capacity.

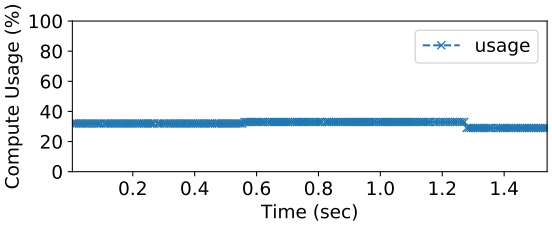
Although GPU sharing at fine-granularity by multiplexing multiple jobs on the same device can increase GPU utilization, it introduces extra overheads, such as frequent context switching and data loading from storage (Mohan et al., 2021), maintaining multiple models within the same GPU memory (Yu & Chowdhury, 2020; Huang et al., 2020), inter-job interference (Xiao et al., 2018), etc. More importantly,

[1]University of California, Berkeley. Correspondence to: Guanhua Wang <guanhua@cs.berkeley.edu>.

Proceedings of the 4^{th} MLSys Conference, San Jose, CA, USA, 2021.

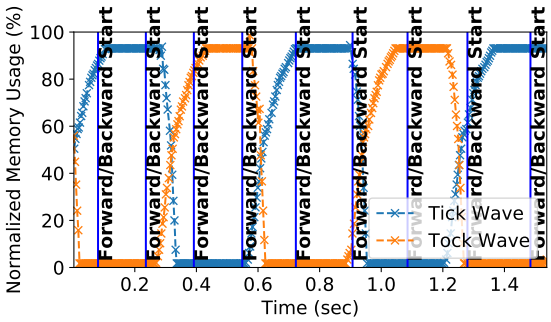


(a) Memory Usage

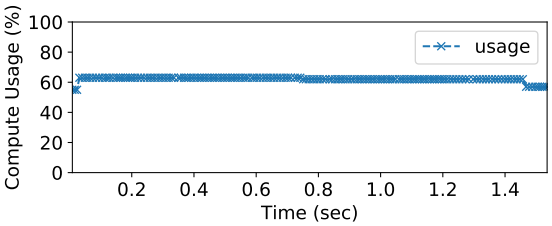


(b) Computation Usage

Figure 1. On-device memory and computation usage of data-parallel training job using 2 V100 GPUs with gang-scheduling.



(a) Memory Usage



(b) Computation Usage

Figure 2. GPU memory and computation utilization on 2 V100 running data-parallel training job with tick-tock scheduling.

job multiplexing does not contribute to the training progress of the original job or the single job case.

In this paper, we propose `wavelet`, an efficient and generic approach that can achieve both high computation and memory utilization to accelerate the training process of a single job. Wavelet interleaves waves of training tasks on the same group of GPUs, such that tasks belong to one wave can leverage on-device memory from tasks in another wave during their memory valley period, thus boost-up the training throughput. As shown in Figure 2, `wavelet` divides data-parallel training tasks into two waves, namely tick-wave and tock-wave. The task launching offset is achieved by delaying the launch time of tock-wave tasks for half of a whole forward-backward training cycle. Therefore, the tock-wave tasks can directly leverage GPU memory valley period of tick-wave tasks (e.g. 0.4s-0.6s in Figure 2(a)), since backward propagation of tick-wave tasks is compute-heavy but memory is often unused. Similarly, tick-wave tasks can leverage memory valley period of tock-wave tasks in the same way.

Besides reaching ideal GPU utilization in data parallel training, `wavelet` also achieves decent performance gain in model parallel training. Conventional gang-scheduled model-parallel training has severe GPU under-utilization issue (Shoeybi et al., 2019). The main reasons are: first, there is huge communication overhead of transferring intermediate results among GPUs holding different model partitions. Second, the sequential dependency of DNN layers held on differnet GPUs creates hard synchronization barriers (Lepikhin et al., 2021; Huang et al., 2019). Although recent literature advances model parallel training performance by incorporating pipeline parallelism (Huang et al., 2019; Narayanan et al., 2019), the issue of GPU under-utilization is still not eliminated (Ivanov et al., 2021). In model parallelism, due to its longer on-device memory valley period and lower computational usage characteristics (when comparing with data parallel schemes), `wavelet` inserts more training waves (e.g. in Figure 10 of 4-GPU case, we insert 3 additional training waves for training on batch 1-3) into the original model parallel training pipeline (i.e. batch 0 in Figure 10) and arrange training tasks in a round-robin fashion. We inject model synchronization accordingly during backward propagation, which is illustrated in Section 3.3.

We evaluate `wavelet`'s performance over a number of different DNN models, datasets, and hardware configurations. The results confirm the effectiveness and training time benefits of `wavelet`. Compared to data/model parallel training with gang-scheduling, `wavelet` achieves up to 1.88x time reduction for data parallel training workloads, up to 6.7x faster for model parallel training tasks.

2 Backgroud and Motivation

This section first overviews standard distributed DNN training approaches. Then it analyzes the GPU utilization pattern

when adopting different in-parallel training schemes on varied DNN workloads in each training iteration.

2.1 Distributed DNN Training Schemes

Standard in-parallel training methods include data parallelism (Chen et al., 2015; Li et al., 2014) and model parallelism (Dean et al., 2012; Lee et al., 2014). Recent literature also proposes pipeline parallelism which leverages inter-batch pipelining to improve system efficiency (Huang et al., 2019; Narayanan et al., 2019). Since pipeline parallelism is often implicitly involved in the above two alternatives (Xing et al., 2015), we mainly explain data parallelism and model parallelism in this section.

Data Parallelism: All the GPUs involved in the job maintain a full copy of the whole model, and conduct training iterations independently on a subset of the input dataset. Since model on each device is trained on different input data, the model parameter needs to be periodically synchronized. There are two ways for model synchronization: parameter-server (Li et al., 2014), and collective communication (e.g. All-Reduce (Wang et al., 2020)). Since parameter server needs human efforts to manually determine the roles of GPUs in use (i.e. either a parameter server or a worker) and is hard to determine server-to-worker ratio (Sergeev & Balso, 2018), collective communication gains more attention and is widely adopted. Therefore, we choose All-Reduce based data parallel training scheme as our baseline, which means all the GPUs are workers.

Model Parallelism: Model parallelism adopts a different approach, where each GPU only keeps a portion of the whole model and all the GPUs share the same input data for each training iteration. However, GPUs are often under-utilized in model parallel training, which is mainly due to huge communication overhead of transmitting intermediate results across GPUs holding different model partitions. Furthermore, the sequential dependency of underlying DNN naturally forms hard synchronization barriers among GPUs maintaining adjacent model partitions (Narayanan et al., 2019; Lepikhin et al., 2021). Therefore, to achieve better system efficiency, state-of-the-art model parallelism is incorporated with pipeline parallelism (Huang et al., 2019; Narayanan et al., 2019).

2.2 Jobs Characteristics of Distributed DNN Training

We monitor GPU resource utilization with different DNN training workloads. We measure GPU memory consumption, and computation utilization using occupancy rate metric (Nvidia-occupancy, 2020) (instead of NVIDIA-SMI (Nvidia-smi, 2020), more discussion in Appendix A) through all our measurements. We discuss our findings about system inefficiency in both data parallel training (Section 2.2.1) and model parallel training (Section 2.2.2) with gang scheduling.

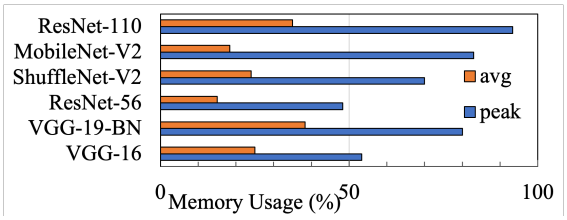


Figure 3. Normalized average and peak GPU memory consumption during in data parallel training.

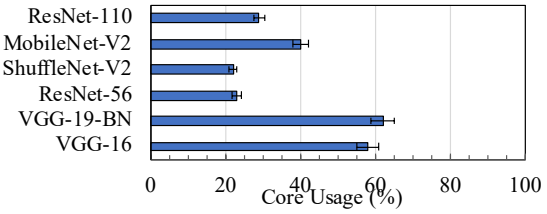
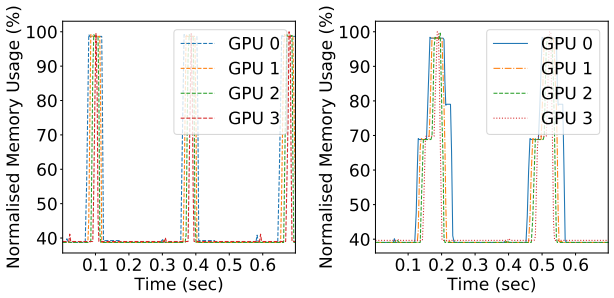


Figure 4. Average computation cost on V100 GPU during data parallel training among different CNNs.

2.2.1 *Zoom-in analysis on data parallel training*

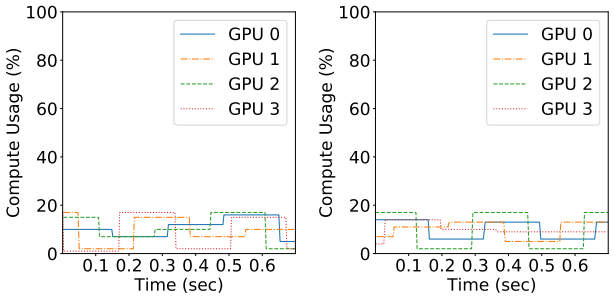
We argue that gang-scheduling, as a generally accepted design principle for distributed DNN training, may cause system under-utilization. We conduct data parallel training of several standard convolutional neuronal networks (CNNs) on ImageNet-1K dataset (Russakovsky et al., 2015). We monitor memory and GPU compute core usage of each training iteration, which consists of 1 forward propagation followed by 1 backward propagation.

Figure 3 summarizes the statistics about peak and average GPU memory usage during data parallel training across different CNNs (Simonyan & Zisserman, 2015; Sandler et al., 2018; Ma et al., 2018). As model size becomes larger, the corresponding memory consumption also increases. Although the peak memory usage may hit the on-device memory capacity, the average memory utilization is relatively low ($\sim$ 30% on average). Figure 1(a) shows the spatiotemporal memory snapshot during the time period of training 3 iterations with ResNet-56 (He et al., 2016), which indicates that the memory usage is highly predictable with well-defined peak and valley in each training iteration. Figure 1(b) visualizes the corresponding compute core usage. Due to the on-device memory wall, the core utilization is relatively low, which only around 30% and leaves 70% computation power under-utilized. Figure 4 summarizes computation usage of different CNNs and shows that the computation utilization is generally low across different workloads.



(a) no pipelining (b) with pipelining

Figure 5. GPU Memory spatiotemporal pattern of BERT model training using 4 V100 with gang-scheduled model parallelism.



(a) no pipelining (b) with pipelining

Figure 6. GPU computation usage of BERT model training using 4 V100 with gang-scheduled model parallelism.

2.2.2 *Sub-iteration analysis on model parallel training*

In model parallel training with gang scheduling, here we mainly focus on collecting system statistics on a popular language model, namely, BERT (Devlin et al., 2018). We monitor the detailed GPU computation and memory utilization in model parallel training with 4 V100 GPUs. The training workload is BERT-base model fine-tuning on SQuAD 2.0 dataset (Rajpurkar et al., 2018).

Figure 5 and Figure 6 visualize the memory and computation usage of 2∼3 model parallel training iterations on BERT, which includes per-GPU forward propagation (i.e. GPU0–>GPU1 –>GPU2 –>GPU3) followed by backward propagation in the reverse order (i.e. GPU3–>GPU2 –>GPU1–>GPU0). And we have two settings: Figure 5(a) and 6(a) on the left show memory and compute usage on each GPU with vanilla model parallel training. Figure 5(b) and 6(b) depict on-device compute and memory usage when combining pipeline parallelism (Huang et al., 2019) with model parallelism.

In Figure 5(a), similar as memory usage of data parallel training in Section 2.2.1, the memory usage of vanilla model parallel training also shows clear and well-defined peaks (forward propagations) and valleys (backward propagations). The main difference is here the memory valley period within each training iteration is longer than the data parallel counterpart. This is mainly due to the sequential blocking of generating gradients on the GPU chain. More specifically, the long memory valley period consists of head-of-line blocking back propagation from GPU3–>GPU2 –>GPU1–>GPU0. Thus the memory valley period here is much longer than the valley period in data parallel training, which only contains single GPU backward propagation. This low GPU memory usage leads to low computation usage in Figure 6(a). Similar inefficiency of core utilization has also been found even by using the optimized implementation such as Megatron (Ivanov et al., 2021). Results shown in Figure 5(b) and 6(b) indicates that, input pipelining mitigates the inefficiency issue on memory and compute usage in model parallel training, but does not completely solve it.

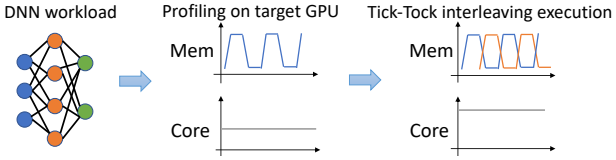


Figure 7. Wavelet workflow overview.

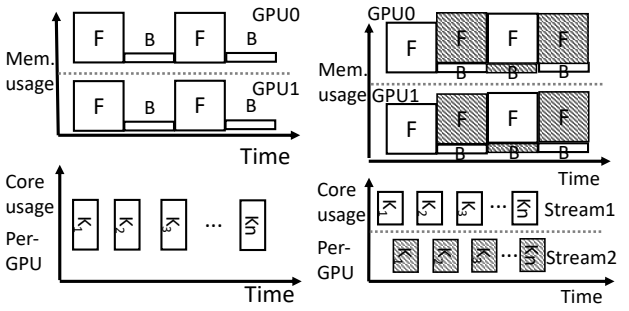
In summary, the system inefficiency introduced by gang-scheduling is repeated and highly-predictable across training iterations in both data and model parallelism, which leaves ample room for us to improve.

3 WAVELET DESIGN

We outline `wavelet` design in this section, and describe the detailed techniques to address the dual challenges in achieving high GPU memory and computation utilization in distributed DNN training. We first illustrate data parallel training with `wavelet`'s tick-tock scheduling. Then we describe how we extend our approach to model parallelism by sequentially launching multiple training waves to further increase GPU utilization and training throughput.

3.1 System overview

Based on the analysis we conduct in Section 2.2.1 and 2.2.2, the memory usage pattern is highly predictable and repeated across training iterations (Xiao et al., 2018; Yu & Chowdhury, 2020), which leaves plenty of space for improving system efficiency. We assume a distributed training job is already assigned to a set of GPUs with proper communication links in between. We also assume the training workload has been evenly partitioned into all the homogeneous GPUs. For the case of GPU heterogeneity, we can wisely balance workload to make sure that all GPUs can finish a training



(a) gang scheduling (b) tick-tock scheduling

Figure 8. Data parallel training with tick-tock scheduling.

iteration using the same amount of time (Narayanan et al., 2020) or conduct job migration to form homogeneous execution environments (Xiao et al., 2018). As our main goal is to improve system efficiency regarding GPU memory and computation usage, we directly borrow existing collective communication protocols (Sergeev & Balso, 2018; Jeaugey, 2017; Wang et al., 2020) for network scheduling and data transfer.

As depicted in Figure 7, at high level, `wavelet` works as follows: Given a DNN training job, we first collect resource usage metadata via a short profiling run of hundreds of mini-batches on a single target GPU. Given that a normal training job usually consists of millions of mini-batches training iterations (Narayanan et al., 2019; Yu & Chowdhury, 2020), this profiling overhead is negligible.

In the profiling, we record three quantities for each mini-batch training on the target GPU, which are total computation time of 1 forward and 1 backward pass for each mini-batch processing, peak and valley period in memory usage, computational core usage. If the training task is memory-bounded, which indicates that computation cores are under-utilized due to limited on-device memory capacity, we conduct our tick-tock scheduling policy for interleaving multiple waves of training tasks on each GPU, and periodically synchronize both intra-wave and inter-wave training tasks. By interleaving both compute core and memory usage, we can achieve near-optimal device utilization.

We illustrate `wavelet` approach in both data parallelism and model parallelism application scenarios. We call the original gang-scheduled training wave as *tick-wave* tasks. And *tock-wave* tasks refer to the training tasks `wavelet` injects into the training pipeline that leverage the unused resources when running the original *tick-wave* tasks.

3.2 Wavelet in data parallelism

In data parallel training, gang-scheduling leaves ample space for GPU memory and computation resources sharing. Figure 8(a) depicts a gang-scheduled, 2-GPU data parallel training process, where F refers to forward propagation and B indicates backward propagation. $K_1 ... K_n$ are the sequence of computational kernels (K_i) launched on each GPU.

On the memory side, with gang-scheduling, all the GPUs in use reach their memory usage peak and valley at roughly the same time (shown as top figure in Figure 8(a)). This synchronized memory usage pattern make it impossible for one GPU to borrow the memory of other GPUs for its own forward computation, as other GPUs' memory also stay at the peak usage at this point. Thus, the on-device memory of all the GPUs in the same training job is under-utilized simultaneously during their backward propagation period.

On the compute side, with gang scheduled tasks showing at the bottom of Figure 8(a), due to the limited local memory capacity, the tensor size for parallel execution within each CUDA kernel is also limited. Thus the GPU compute cores are consistently under-utilized when launching sequence of CUDA kernels during both forward and backward propagation of each training iteration.

To improve system utilization, one straw man scheme is to preempt and switch to another training batch during the memory valley period (i.e. tick-wave tasks' backward propagation time shown as the blank B boxes in Figure 8(a)). However, it may interrupt and block the original process (i.e. tick-wave task) computing its own backward propagation, due to the high frequency of memory peak-valley changing cycles in DNN training jobs.

Different from the straw man approach above, `wavelet`'s design allows concurrently launching and interleaving multiple waves of training tasks on the same GPU without interfering each other. Below we describe how we enable this interleaving concurrent waves of task execution in three aspects: memory overlapping, computation overlapping, and model synchronization among different waves of tasks.

3.2.1 *Memory overlapping*

Different from gang scheduling, with tick-tock scheduling shown as Figure 8(b), we allow the tick-wave tasks (blank F, B box in the top figure of Figure 8(b)) to be launched first. Right after tick-wave tasks finishing their forward propagation, we inject tock-wave tasks (denoted as the shadow F, B boxes in the upper figure of Figure 8(b)) into the same set of GPUs. This injection starts at the time when the tick-wave tasks start to free the allocated memory in their backward propagation. At high level, by adding launching delay the same duration of one forward computation time, tick-tock scheduling can overlap tick-wave's backward computation with tock-wave's forward computation and vice-versa. This decent attribute guarantees the memory usage always stays near the on-device memory capacity (e.g. Figure 2(a)).

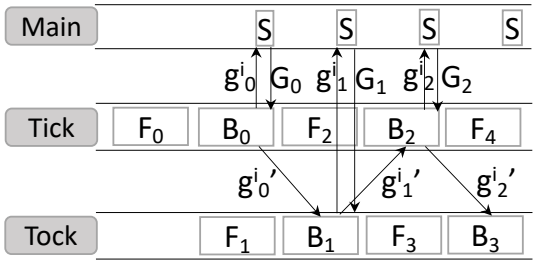


Figure 9. Wavelet model synchronization between tick and tock waves on GPU-i in data parallel training.

To concurrently run two training tasks (i.e. one in tick wave, the other from tock wave) on each GPU, we need to maintain two model replicas on the same GPU: one for tick wave and the other for the tock wave. The necessity of holding two model replicas is for version control of model parameters between two different training waves since they are training on different input data batches (Narayanan et al., 2019). Recent literature (Xiao et al., 2018; Yu & Chowdhury, 2020) shows that the model size is usually order-of-magnitude smaller than the generated intermediate results over input data.

3.2.2 *Computation overlapping*

As discussed in Section 3.2.1, we hold two model replicas on each GPU, one for tick wave training, the other for tock wave. Different from traditional GPU sharing via time-slicing (Xiao et al., 2018; Yu & Chowdhury, 2020), we actually launch two sequences of different kernels concurrently. One for forward propagation kernels of one wave, the other sequence for backward propagation kernels of the other wave. In this way, we improve computation usage in both *spatial* and *temporal* dimensions.

To avoid head-of-line-blocking of two concurrent kernel sequences, as shown in the bottom figure in Figure 8(b), we launch these two kernel sequences using separate CUDA streams (NVIDIA, 2020a), which guarantees kernel executions are ordered within a stream, but non-blocking across different streams. By concurrently launching multiple sequences of computation kernels, we can leverage computation resources more efficiently in *spatial* dimension.

Another benefit of concurrently launching two streams of computation kernels is to enable two sequences of kernels fill-in each other's execution bubble time. There is evidence that the latency of CPU sending instructions to GPU is amplified and introduces empty bubbles when launching sequences of computation kernels (Narayanan et al., 2019; CUDA MPS). As shown in the bottom figure of Figure 8(b), those empty space between adjacent compute kernels in one sequence is the bubble/idle time in *temporal* dimension that `wavelet` can potentially fill-in the gap by concurrently launching another group of kernel functions.

3.2.3 *Model synchronization between waves*

For model synchronization among the training tasks in the same wave, we directly borrow the model synchronization schemes (e.g. All-Reduce) inherited from deep learning frameworks (Abadi et al., 2016; Paszke et al., 2017; Li et al., 2020). Since tasks in the same wave finish each training iteration at the same time, we directly impose a normal model synchronization at the end of each in-parallel mini-batch training iteration. The main issue is, there is no model synchronization scheme available between our tick and tock waves' tasks.

Figure 9 summarizes `wavelet` model synchronization design between tick and tock waves' training tasks. *Main, Tick, Tock* are three threads inside the training process. $g_n^i{}'$ is generated gradients on GPU i on its current local batch input. g_n^i refers to the averaged gradients between the n-th and the $(n-1)$-th iteration on GPU i (defined as Equation 1). F_n, B_n indicates forward and backward propagation of global iteration n. S stands for model synchronization among all GPUs in use for training tasks belongs to 1 wave (either tick or tock wave). G_n is the global averaged gradients from the n-th iteration model synchronization (e.g. All-Reduce) among GPUs for the same wave tasks.

In Figure 9, on GPU i, B_0 of tick wave finishes generating its local gradients $g_0^i{}'$ ($g_0^i{}' = g_0^i$ since no need of averaging cross-wave in the first training iteration) and sends it to the consequent tock wave task during its backward propagation B_1. Then *Main* thread launches model synchronization among all the GPUs in use and gets global averaged gradient G_0 for this tick-wave training iteration. Tick wave task then updates model parameters with G_0, and starts its second training iteration F_2. Simultaneously, tock wave task B_1 computes its local gradients of $g_1^i{}'$ and sends it back to tick wave task in B_2 phase. Then tock wave B_1 averages $g_1^i{}'$ with received $g_0^i{}'$ from tick wave task as follows:

$$g_m^i = \frac{g_m^i{}' + g_{m-1}^i{}'}{2} \tag{1}$$

Then it uses g_1^i to synchronize with all the corresponding tock wave tasks from other GPUs to get back G_1 for this tock-wave training iteration. Then the tock wave starts its second training iteration as F_3, and so on and so forth.

Lemma 1: For any globally synchronized gradients $G_m (m > 0)$ among N GPUs, it is also globally synchronized between the overlapped 1 tick wave and 1 tock wave training tasks (i.e. F_{m-1}, B_{m-1} and F_m, B_m).

$$G_m = \frac{\sum_{i=1}^{N} (g_{m-1}^i{}' + g_m^i{}')}{2 \times N} \tag{2}$$

$$\text{where } m \in \{1, 2, ...n\} \tag{3}$$

Proof: Given Equation 1 as averaging gradient between each tick-tock tasks pairs, we can plug-in it to Equation 2 and calculate the corresponding global synchronization result G_m as follows:

$$\begin{aligned} G_m &= \frac{\sum_{i=1}^{N} g_m^i}{N} \\ &= \frac{\sum_{i=1}^{N} (g_m^i\prime + g_{m-1}^i\prime)}{2 \times N} \end{aligned} \tag{4}$$

Therefore, with our design for cross tick-tock waves model synchronization, it is the same as synchronizing over 2*N data parallel training tasks on 2*N GPUs, which guarantees its convergence.

3.3 Wavelet in Model Parallelism

Larger models show remarkable serving performance improvements (Real et al., 2019), especially in the natural language processing area (Devlin et al., 2018; Rosset, 2020). State-of-the-art performance on tasks like question answering or next sentence prediction is achieved by transformer-based models, which usually consist of millions or even billions of parameters (Brown et al., 2020).

Recent literature (Huang et al., 2019; Narayanan et al., 2019) leverages inter-batch pipelining to improve GPU utilization in model parallel training. At high level, after each GPU finishes forward computation on current batch, instead of waiting for its turns to do backward propagation on the same batch of data, the GPU loads in another batch for forward processing to fully utilizes the idle time between 1 batch's forward and backward time gap.

3.3.1 Launching multiple tock-wave tasks

As we discussed in Section 2.2.2, model parallelism naturally forms longer memory valley period on each GPU (shown in Figure 5). Thus, we can leverage the memory valley and compute underutilized period on each GPU more aggressively. `Wavelet` in model parallelism injects more tock-waves of training tasks, which enables us to concurrently train the same model partitions on different input batches.

Figure 10 depicts a model parallel training job in 4-GPU case. F and B still refer to forward and backward propagation. $F_{m,n}^i$ denotes forward propagation of model partition m with batch n on GPU-i, and $B_{m,n}^i$ follows the same naming rule. The baseline (i.e. the original tick-wave training) is shown as the blank boxes (e.g. $F_{0,0}^0, F_{1,0}^1, F_{2,0}^2 ... B_{0,0}^0$). And during each training cycle, instead of only one GPU is computing while others remain idle, we directly force all the GPUs to load the first model partition during first compute cycle and conduct forward propagation on different input batches (e.g. $F_{0,0}^0, F_{0,1}^1, F_{0,2}^2, F_{0,3}^3$). And in the second compute cycle, all GPUs swap to the second model partition and process on different input batches, so on and so forth. Thus, it can be regarded as we sequentially launch 3 tock-wave tasks (i.e. tasks processing batch 1,2,3 in Figure 10) on top of the original tick-wave baseline (processing batch 0) in this 4-GPU example. In theory, we can maximally inject $(N-1)$ tock waves in N-GPU case to fully utilize GPU memory and computational resources.

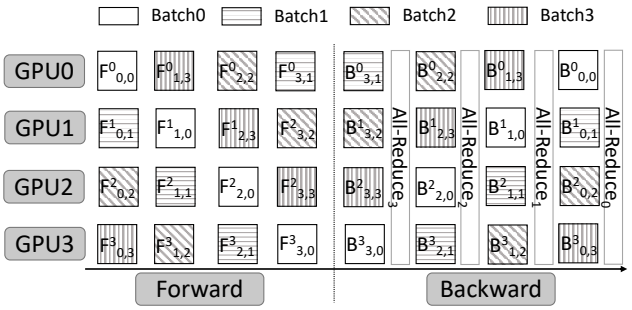


Figure 10. Model parallelism with wavelet in 4-GPU case.

3.3.2 Model partition switching

Different from traditional model parallel training or hybrid of model and pipeline parallelism that each GPU always holding the same model partition, `wavelet` imposes every GPU to maintain different model partitions and process different input batches in different training cycles. Thus, context switching is needed in both model and input dimensions. We design it in a *round-robin* fashion. As shown in Figure 10, during the first forward propagation cycle, all the GPUs load the first model partition, while feeding in different input batches. After this cycle finishes, we down-shift the batch id and up-shift the model partition on each GPU. For example, GPU 1 first processes on batch 1 with model partition 0 during the first computation cycle ($F_{0,1}^1$). In the second compute phase, GPU 1 processes batch (1-1)%4=0 with model partition (0+1)%4=1 ($F_{1,0}^1$). All the GPUs involved follow the same rule. And backward propagation is just symmetric to forward propagation in the inverse order.

Frequently switching content of both input and model partitions definitely introduces overheads. Regarding the overheads introduced by `wavelet`'s model parallelism, different from state-of-the-art hybrid of model and pipeline par-

allelism (Huang et al., 2019; Narayanan et al., 2019), each GPU in our `wavelet` approach needs additional switching on model partitions in different training cycles (as shown in Figure 10). We empirically evaluate our context switching overheads through our end-to-end tests in Section 4.2. To further reduce the memory overheads for holding intermediate results, we can also leverage techniques like tensor re-materialization (Kirisame et al., 2020; Jain et al., 2020) to recompute intermediate results on demand (Huang et al., 2019). And we leave this as a future research direction.

3.3.3 *Inter-batch synchronization*

As shown in Figure 10, Our round-robin approach can impose $(N-1)$ tock waves of training tasks on top of 1 tick wave training when using N GPUs. Given this round-robin assignment, for a single model partition, all the batches finish generating gradients (i.e. backward propagation) at the same time (e.g. in Figure 10, $B^x_{3,y}$, $x, y \in \{0, 1, 2, 3\}$ are finished at the same time). This inherit alignment of backward propagation on the same model partition simplifies our synchronization scheme. We simply add model synchronization (i.e. All-Reduce) right after the backward propagation of the same model partition across different batches, and update the parameters of each model partition accordingly (e.g. All-Reduce$_3$ for $B^x_{3,y}$).

4 EVALUATION

We evaluate `wavelet`'s effectiveness with 5 different DNNs on two different datasets. All the experiments are conducted on DGX-1 machines, which consist of V100 GPUs (dgx1) and point-to-point NVLink (NVLink) in between. Cross machine communication can be built over a 25Gb/s Ethernet interface. The models we test for data parallel training are VGG-16, VGG-19-BN (with batch normalization) (Simonyan & Zisserman, 2015), ResNet-50 and ResNet-101 (He et al., 2016). We train these CNNs using ImageNet-1K dataset (Deng et al., 2009). We also evaluate `wavelet` performance in model parallelism. We fine-tune BERT (Devlin et al., 2018) model on Squad 2.0 dataset (Rajpurkar et al., 2018) and conduct model parallel training of VGG-19-BN on Imagenet-1K dataset. We also discuss some implementation tricks we use as Appendix B.

Our experimental results support a number of important findings: first, `wavelet` achieves up to 1.88x speedup in data parallel training settings. Second, `wavelet` outperforms state-of-the-art model parallel training with pipeline parallelism on popular language model by up to 4.15x, and up to 6.7x faster when compared with vanilla model-parallel training baseline.

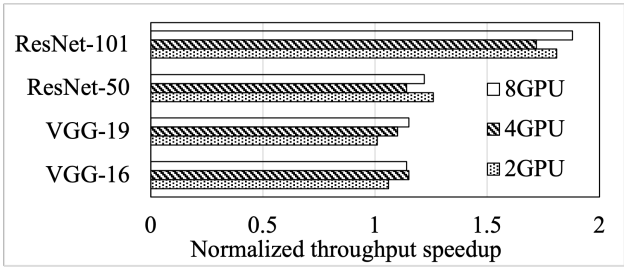


Figure 11. single-machine data parallelism throughput speedup with wavelet.

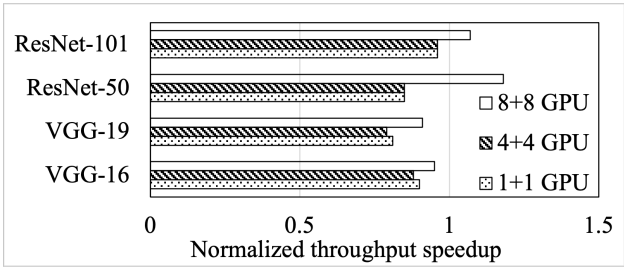


Figure 12. cross-machine data parallelism throughput speedup with wavelet.

4.1 Data parallelism

We use the largest mini-batch size that can fit into the GPU memory without introducing out-of-memory exceptions. This ensures the best throughput performance of the baseline. It also guarantees the most efficient GPU memory utilization `wavelet` can achieve with our tick-tock interleaving. We use 64 as the per-GPU batch size for data parallel training with both VGGNet and ResNet.

4.1.1 *Single machine multi-GPU*

In single machine case, we test `wavelet`'s data parallel training performance in three hardware settings: 2*V100, 4*V100, and 8*V100. All the V100 GPUs are interconnected with point-to-point NVLink (NVLink) in a hypercube topology, with each NVlink line speed around 25GB/s.

To get clear results on the performance gain, we normalize the throughput of data parallel training baseline as 1, and report our speedup numbers over the baseline. As shown in Figure 11, we achieve up to 1.88x speed up compared with data parallel baseline. And on average we achieve 1.4x speed-up across all the tested CNNs. Note that in VGGNets training, our speedup is less than 1.2x, It is mainly due to that, each training iteration is very short on VGGNets. And our injected tock-wave tasks actually lengthen the duration of each training iteration (i.e. 1 forward propagation + 1 backward propagation) of the original tick-wave training tasks. The longer training iteration time decreases our overall throughput speed-up, especially when the original training iteration time is very short.

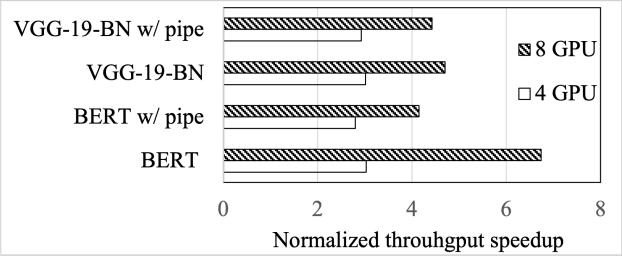


Figure 13. Single machine model parallelism throughput speedup with wavelet.

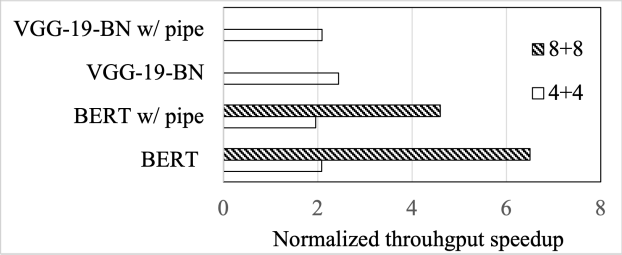


Figure 14. Cross-machine model parallelism throughput speedup with wavelet.

4.1.2 *Multi-machine multi-GPU*

In mutli-machine settings, we conduct experiments with two DGX-1 machines (dgx1) which are connected with 25Gb/s Ethernet. We test three different cases: $1+1, 4+4$ and $8+8$. As shown in Figure 12, $1+1$ meaning 1-GPU on 1 machine and another GPU is located on the other machine. Similar cases like $4+4$, $8+8$ follow the same rule.

Compared with single machine performance, we only achieve up to 1.18x speedup over multi-machine data parallel training baseline. In addition, `wavelet` even performs worse than the baseline in many cases in Figure 12. The main reason for this bad performance is that, the limited cross-machine bandwidth indeed affects `wavelet` more than the baseline, since we need an extra All-Reduce operation for the tock-wave tasks we inject in each training iteration. Although we also interleave the All-Reduce operations between tick and tock wave tasks (as G_0, G_1 in Fig. 9) to mitigate the burst of communication, the limited cross-machine bandwidth is still the bottleneck.

4.2 Model parallelism

As suggested by recent model parallel literature (Narayanan et al., 2019; Huang et al., 2019), we evenly split the model (for both VGG-19-BN and BERT) among the GPUs to balance work. Similar as data parallel evaluations, we feed in the largest batch size without introducing out-of-memory exceptions.

4.2.1 *Single machine multi-GPU*

For model parallel training over BERT and VGG-19-BN on a single machine, we tested 4-GPU and 8-GPU cases. For each model we tested, we have two settings: one is model parallelism with pipeline parallelism (i.e. w/ pipe in Figure 13), the other is vanilla model parallelism. As shown in Figure 13, we achieve higher throughput speed-up in 8-GPU case than 4-GPU cases. More specifically, comparing with vanilla model parallelism in 8-GPU case, we achieve 4.7x speed-up on VGG-19-BN, 6.7x on BERT, separately. Compared with model parallelism with input pipeline, we achieve 4.4x speed up on VGG-19-BN and 4.15x speed up on BERT in 8-GPU cases.

Note that even with pipeline parallelism added as Gpipe (Huang et al., 2019), the reported speed-up over vanilla baseline is only $<$ 2x with 4 GPUs and $\sim$ 3x with 8 GPUs (Huang et al., 2019). This is mainly due to higher frequency of both CUDA kernel launching and intermediate results transfer between GPUs that holding different model partitions. More specifically, for each mini-batch (e.g. with size M) of training data, Gpipe (Huang et al., 2019) or PipeDream (Narayanan et al., 2019) further breaks it into smaller micro-batches (e.g. with size N) and do data pipelining. Thus for training on 1 mini-batch data, the communication frequency between adjacent GPUs increases from 2 (1 during forward the other during backward) to $2 \times \frac{M}{N}$ (the first $\frac{M}{N}$ in forward propagation, the second $\frac{M}{N}$ in backward propagation). And this high-frequency but small-size data chunks may not fully saturate link bandwidth of NVLink (NVLink) or NVSwitch (NVSwitch), which leads to longer communication latency. Similarly, the number of CUDA kernel calls also increased by $\frac{M}{N}$ times. Thus it also introduces higher control overheads. In contrast, `wavelet` still keep intermediate result transfer frequency to be 2 per mini-batch input with same number of CUDA kernel calls per-GPU. In theory, by incorporating `wavelet`, we can achieve linear scalability, which means given A number of GPUs, we can achieve throughput speed up by Ax over the baseline.

4.2.2 *Multi-machine multi-GPU*

In Figure 14, similar as cross-machine data parallel results in Section 4.1.2, in 4+4 GPU settings, we achieve moderate ($\sim$2x) speed-up over baseline with/without input pipelining on both VGG-19-BN and BERT models. For 8+8 setting, we only test BERT, since it is not reasonable to decouple a CNN model over 16 GPUs. In this case, we achieve 4.5x speed-up over BERT with pipeline parallelism, and 6.5x faster over vanilla BERT model parallelism.

Given that our pipeline only happens inside each single GPU, we believe increasing the number of GPUs to scale out with `wavelet` may not cause any significant issues.

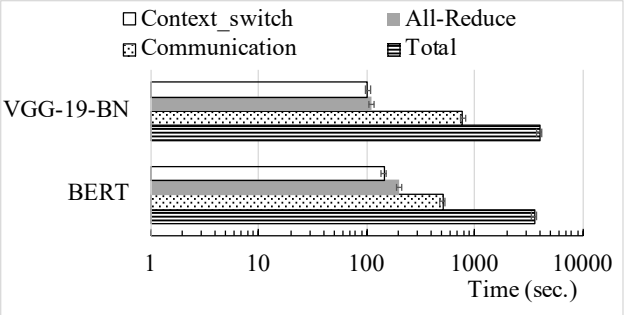


Figure 15. Wavelet overhead breakdown in 4+4 cross-machine setting.

4.2.3 *Overhead analysis*

Here we analyse the overhead introduced by incorporating `wavelet` into model parallelism. We only test 1 hardware setting of $4 + 4$, which is to use 4 V100 GPUs on one machine, and 4 V100 GPUs on the other machine.

As shown in Figure 15, we record the total time of training 1 epoch in both VGG-19-BN and BERT. We also collect the time spending on 3 main aspects: context_switch, communication and All-Reduce. Context_switch overhead refers to the time we spend on switching model partition on each GPU in different training cycles. Communication time means the time spending on transferring intermediate result across GPUs in use. And All-Reduce overhead is our model synchronization during backward propagation phase described in Section 3.3.3.

As depicted in Figure 15, cross machine communication is the dominant overhead, which is 14% of BERT total training time, and 19.5% of VGG-19-BN's total training time. Other overhead like All-Reduce and context-switch contribute to around 5% on BERT training and 2% on VGG-19-BN training time, respectively.

5 Related Work

Related literature falls into the following two categories.

5.1 Resource allocation for distributed DNN training

Multiple resource scheduling policies have been tailor-made for distributed DNN training jobs recently (Jeon et al., 2019), with focus on fairness (Mahajan et al., 2020), better locality (Xiao et al., 2018), minimizing job completion time (Gu et al., 2019), auto-scaling (Or et al., 2020), etc. Gang-scheduling, or all-or-nothing scheduling policy is generally-accepted and adopted as the *only* option for launching in-parallel training tasks of a single DNN training job. Different from gang-scheduling policy, `wavelet` proposes tick-tock scheduling policy which achieves higher GPU utilization in both memory and computation resources, and faster single job training speed.

5.2 GPU sharing

GPU sharing has been targeted by a number of recent works such as NVIDIA MPS (CUDA MPS), Salus (Yu & Chowdhury, 2020), Gandiva (Xiao et al., 2018), etc. NVIDIA Multi-Process Service (CUDA MPS) allows multiple processes to share GPU resources with static partitioning. Static resource partitioning introduces large and hard to predict inter-job inferences when multiplexing different DNN jobs, which decreases the overall performance. Gandiva (Xiao et al., 2018) co-locates deep learning jobs by trail-and-error with job migration, and finally reaches non-sharing mode. Salus (Yu & Chowdhury, 2020) aims to achieve fine-grained GPU sharing among different DNN tasks by spatial memory sharing and temporal sharing on compute resources. NVIDIA Tensor-RT (NVIDIA, 2020c) achieves concurrent deep learning inference on single GPU but lack of training supports. Different from all above works that targeting on multi-job multiplexing, `wavelet` focuses on GPU sharing for speeding up a single DNN training job. In addition, `wavelet` achieves spatiotemporal sharing in both GPU memory and computation dimensions.

Earlier GPU sharing works (Yeh et al., 2017; Pai et al., 2013; Zhang et al., 2018) are designed for workloads with a few GPU kernel functions, which are not applicable and scalable to deep learning applications with hundreds of unique kernel functions.

6 Conclusion

We present `wavelet`, an efficient and generic approach that interleaves waves DNN training tasks in a single job and achieves ideal GPU computation and memory utilization. Different from generally accepted gang-scheduling policy, `wavelet` proposes a novel yet simple policy called tick-tock scheduling. By launching different waves of training tasks with different delay offsets, `wavelet` improves GPU utilization in both on-device memory and computation resources. With proper modification on the synchronization stage, `wavelet` achieves near-optimal GPU utilization and up to 6.7x training time reduction for single job. `Wavelet` is generally applicable to both data parallel, model parallel and hybrid parallel training of a single job.

Acknowledgements

Guanhua Wang and Ion Stoica are supported by NSF CISE Expeditions Award CCF-1730628. This research is also supported by gifts from Amazon Web Services, Ant Group, CapitalOne, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk and VMware.

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *USENIX OSDI*, 2016.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *arXiv preprint arXiv:1512.01274*, 2015.

CUDA MPS. CUDA multi-process Service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, 2020.

Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Le, Q. V., Mao, M. Z., Ranzato, M., Senior, A., Tucker, P., Yang, K., and Ng, A. Y. Large scale distributed deep networks. In *NeurIPS*, 2012.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009.

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

dgx1. NVIDIA DGX-1. https://www.nvidia.com/en-us/data-center/dgx-1/, 2017.

dgx2. NVIDIA DGX-2. https://www.nvidia.com/en-us/data-center/dgx-2/, 2018.

Goyal, P., Dollar, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv preprint arXiv:1706.02677*, 2017.

Gu, J., Chowdhury, M., Shin, K. G., Zhu, Y., Jeon, M., Qian, J., Liu, H., and Guo, C. Tiresias: A gpu cluster manager for distributed deep learning. In *USENIX NSDI*, 2019.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *CVPR*, 2016.

Huang, C.-C., Jin, G., and Li, J. Swapadvisor: Push deep learning beyond the gpu memory limit via smart swapping. In *ACM ASPLOS*, 2020.

Huang, G., Liu, Z., van der Maaten, L., and Weinberger, K. Q. Densely connected convolutional networks. In *CVPR*, 2017.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M. X., Chen, D., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., and Chen, Z. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *NeurIPS*, 2019.

Ivanov, A., Dryden, N., Ben-Nun, T., Li, S., and Hoefler, T. Data Movement Is All You Need: A Case Study on Optimizing Transformers. In *Fourth Conference on Machine Learning and Systems (MLSys 2021)*, 2021.

Jain, P., Jain, A., Nrusimha, A., Gholami, A., Abbeel, P., Keutzer, K., Stoica, I., and Gonzalez, J. E. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In *Third Conference on Machine Learning and Systems (MLSys)*, 2020.

Jeaugey, S. Optimized inter-GPU collective operations with NCCL 2. https://developer.nvidia.com/nccl, 2017.

Jeon, M., Venkataraman, S., Phanishayee, A., Qian, J., Xiao, W., , and Yang, F. Analysis of large-scale multi-tenant gpu clusters for dnn training workloads. In *USENIX ATC*, 2019.

Kirisame, M., Lyubomirsky, S., Haan, A., Brennan, J., He, M., Roesch, J., Chen, T., and Tatlock, Z. Dynamic tensor rematerialization. *arXiv preprint arXiv:2006.09616*, 2020.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *NeurIPS*, 2012.

Lee, S., Kim, J. K., Zheng, X., Ho, Q., Gibson, G. A., and Xing, E. P. On model parallelization and scheduling strategies for distributed machine learning. In *NeurIPS*, 2014.

Lepikhin, D., Lee, H., Xu, Y., Chen, D., Firat, O., Huang, Y., Krikun, M., Shazeer, N., and Chen, Z. Gshard: Scaling giant models with conditional computation and automatic sharding. In *ICLR*, 2021.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server. In *USENIX OSDI 2014*, 2014.

Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., and Chintala, S. Pytorch distributed: Experiences on accelerating data parallel training. In *VLDB*, 2020.

Lin, T.-Y., Maire, M., Belongie, S., Bourdev, L., Girshick, R., Hays, J., Perona, P., Ramanan, D., Zitnick, C. L., and Dollar, P. Microsoft COCO: Common Objects in Context. *arXiv preprint arXiv:1405.0312*, 2015.

Ma, N., Zhang, X., Zheng, H.-T., and Sun, J. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *CVPR*, 2018.

Mahajan, K., Balasubramanian, A., Singhvi, A., Venkataraman, S., Akella, A., Phanishayee, A., and Chawla, S. Themis: Fair and efficient gpu cluster scheduling. In *USENIX NSDI*, 2020.

Mohan, J., Phanishayee, A., Raniwala, A., and Chidambaram, V. Analyzing and mitigating data stalls in dnn training. In *VLDB*, 2021.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. Pipedream: Generalized pipeline parallelism for dnn training. In *ACM SOSP*, 2019.

Narayanan, D., Santhanam, K., Kazhamiaka, F., Phanishayee, A., and Zaharia, M. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *USENIX OSDI*, 2020.

NVIDIA. CUDA programming manual. https://bit.ly/30K2BEE, 2020a.

NVIDIA. NVIDIA Visual Profiler. https://developer.nvidia.com/nvidia-visual-profiler, 2020b.

NVIDIA. NVIDIA TensorRT. https://developer.nvidia.com/tensorrt, 2020c.

Nvidia A100. NVIDIA NVSWITCH A100. https://bit.ly/3izVeFF, 2020.

Nvidia-occupancy. Achieved Occupancy. https://bit.ly/36S3bnG, 2020.

Nvidia-smi. System Management Interface. https://developer.nvidia.com/nvidia-system-management-interface, 2020.

NVLink. NVIDIA NVLINK. http://www.nvidia.com/object/nvlink.html, 2017.

NVSwitch. NVIDIA NVSWITCH. http://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf, 2018.

Or, A., Zhang, H., and Freedman, M. J. Resource elasticity in distributed deep learning. In *Third Conference on Machine Learning and Systems (MLSys)*, 2020.

Pai, S., Thazhuthaveetil, M. J., and Govindarajan, R. Improving gpgpu concurrency with elastic kernels. In *ACM ASPLOS*, 2013.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in PyTorch. In *NeurIPS*, 2017.

Rajpurkar, P., Jia, R., and Liang, P. Know what you don't know: Unanswerable questions for squad. In *Annual Meeting of the Association for Computational Linguistics(ACL)*, 2018.

Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. Regularized Evolution for Image Classifier Architecture Search. In *AAAI*, 2019.

Rosset, C. Turing-NLG: A 17-billion-parameter language model by Microsoft. https://bit.ly/36z0G9p, 2020.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 2015.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018.

Sergeev, A. and Balso, M. D. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.

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

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., and Hassabis, D. Mastering the game of go without human knowledge. *Nature*, 550(7676), 2017.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

Wang, G., Venkataraman, S., Phanishayee, A., Thelin, J., Devanur, N., and Stoica, I. Blink: Fast and Generic Collectives for Distributed ML. In *MLSys*, 2020.

Xiao, W., Bhardwaj, R., Ramjee, R., Sivathanu, M., Kwatra, N., Han, Z., Patel, P., Peng, X., Zhao, H., Zhang, Q., Yang, F., and Zhou, L. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

Xing, E. P., Ho, Q., Dai, W., Kim, J. K., Wei, J., Lee, S., Zheng, X., Xie, P., Kumar, A., and Yu, Y. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data*, 1(2):49–67, 2015.

Yeh, T. T., Sabne, A., Sakdhnagool, P., Eigenmann, R., and Rogers, T. G. Pagoda: Fine-grained gpu resource virtualization for narrow tasks. In *Proceedings of Principles and Practice of Parallel Programming (ACM PPoPP)*, 2017.

Yu, P. and Chowdhury, M. Salus: Fine-grained gpu sharing primitives for deep learning applications. In *Third Conference on Machine Learning and Systems (MLSys)*, 2020.

Zhang, K., He, B., Hu, J., Wang, Z., Hua, B., Meng, J., and Yang, L. G-net: Effective gpu sharing in nfv systems. In *USENIX NSDI*, 2018.

A COMPUTATION MONITORING USING OCCUPANCY RATE INSTEAD OF NVIDIA-SMI

We believe occupancy rate (Nvidia-occupancy, 2020) that monitoring real active cores involved in each computation computation kernel is a better metric than directly parsing GPU utility percentage (i.e. *Volatile-GPU-Util*) from NVIDIA-SMI [1].

Volatile-GPU-Util provided by NVIDIA System Management Interface (Nvidia-smi, 2020) is inaccurate. When we use NVIDIA System Management Interface for monitor computation usage on each GPU, it only gives very coarse utilization percentage of compute cores to be either 0 or 100%. Recent literature also reports similar findings (Yu & Chowdhury, 2020; Jeon et al., 2019). Indeed, even *Volatile-GPU-Util* indicates 100% computation usage, when we dive into each computational kernel statistics from NVIDIA Visual Profiler (NVIDIA, 2020b), almost all the kernels' active core usage (i.e. occupancy rate) is much lower than 100%. Thus, we report occupancy rate, instead of *Volatile-GPU-Util*, as the real active computation usage in the paper.

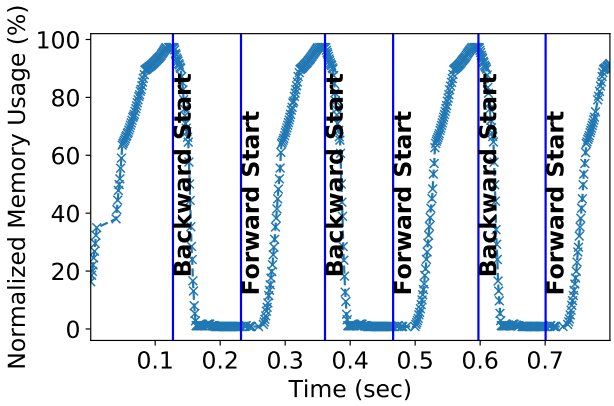
Even with coarse-grained measurements from *Volatile-GPU-Util*, the GPU utility patterns we found still exist. For example, Figure 16 shows GPU computation and memory utility of training DenseNet-160 (Huang et al., 2017) on 2 V100 GPUs via data parallelism. As shown in Figure 16(a), the GPU memory peak and valley periods are well-defined and consistent. Regarding the computation usage illustrated in Fig. 16(b), only ∼40% cores are used with little variance across different training iterations.

Figure 17 shows the corresponding on-device memory and computation utilization with `wavelet`'s Tick-Tock scheduling in the same data parallel training setting as mentioned above. Similar as Figure 2, `wavelet` achieves near-optimal memory usage (Figure 17(a)) and almost double the compute resource utility (∼80% core utilization as shown in Figure 17(b)) when compared with gang-scheduled baseline.

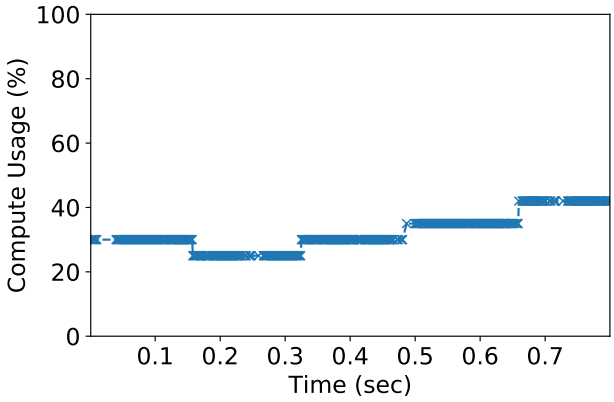
B IMPLEMENTATION

Realizing tick-tock scheduling is more than pure engineering efforts. Here we discuss some implementation details to optimize `wavelet` performance in both data parallel and model parallel training cases.

[1]Similar as query on *nvmlDeviceGetUtilizationRates* from NVIDIA Management Library (NVML) API



(a) Memory Usage



(b) Computation Usage

Figure 16. GPU memory and computation usage of data-parallel training job using 2 V100 GPUs with gang-scheduling (DenseNet-160).

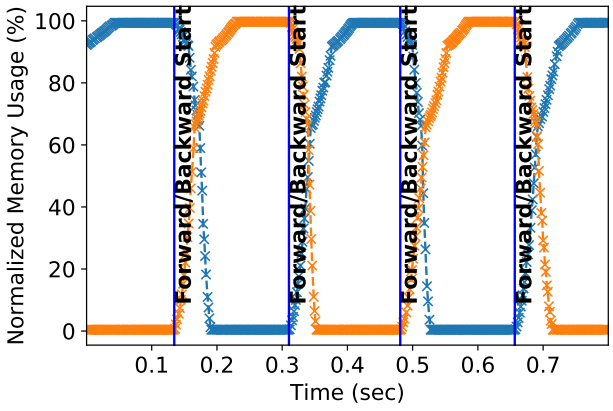
B.1 Data parallel Wavelet

In data parallel `wavelet` training, two main challenges need to be solved. First, how to align the launching time of tasks between tick and tock waves. Second, how to mitigate the extra overheads of cross-wave model synchronization.

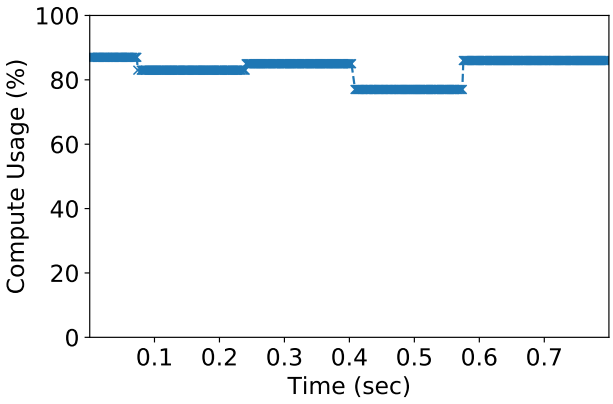
B.1.1 Task alignment between tick and tock waves

To reduce GPU accessing interference caused by multiple processes (Yu & Chowdhury, 2020), we create two threads and two model replicas inside a job's training process and attach each model replica to one thread. Then each thread is responsible for the task executions of 1 wave (either tick or tock). We launch all the tasks in 1 wave over 1 cudaStream to ensure its sequential execution order.

We injects signal events at the end of both forward and backward propagation in each training iterations, so that tick thread and tock thread can align forward/backward



(a) Memory Usage



(b) Computation Usage

Figure 17. GPU memory and computation usage of data-parallel training job using 2 V100 GPUs with Tick-Tock scheduling (DenseNet-160).

executions properly. For example, in Figure 9, once tick thread finishes the first forward propagation task F_0, tock thread can be synchronized on F_0*_end()* event signal and start its forward computation F_1. Similarly, after tock thread emits signal F_1*_end()*, F_2 on tick thread can be launched, so on and so forth. End signals on backward propagation tasks (B_i) follow the same rule.

B.1.2 Reducing cross-wave model synchronization overhead

One extra type of overheads we add into the training procedure is cross-wave model synchronization. Cross-wave model synchronization consists of two part: data transfer overheads (e.g. $g_0^i\prime, g_1^i\prime$ in Figure 9) and computation overheads for averaging gradients across two waves defined as Equation 1.

As shown in Figure 9, we hide the communication of $g_m^i\prime$ in the background of sender's local in-wave model synchronization period and receiver's local backward propagation period. And we impose cross-wave gradient averaging after the receiver generates its local gradient.

B.2 Model parallel wavelet

Different from data parallel training which only inserts 1 tock wave training procedure over the baseline tick wave tasks, we concurrently launch multiple tock waves in model parallel training with `wavelet`. The problem of how to insert multiple tock waves training without interfering the original tick wave tasks remains to be addressed.

We calculate the maximum number (M) of tock waves to insert based on the memory and compute usage profiling procedure (the middle stage of Figure 7) we conduct before training. Then we run a second round profiling on the target GPU to adjust (i.e. addictive decrease from M) to proper number of tock waves training without significantly introducing latency on the original tick wave tasks.