Resource Sharing between Computation and Communication

Shawn Zhong
University of Wisconsin–Madison

Suyan Qu
University of Wisconsin–Madison

Abstract

Overlapping communication with computation is a commonly used strategy when training with data parallelism to reduce training time. In this project, we explore the effect of resource sharing between communication and computation on the training time. We found that the default sharing configuration used by NVIDIA NCCL is suboptimal, and better resource sharing decisions can improve the throughput by as much as 17.2% when training ResNet-101 with PyTorch DDP on a single machine with multiple GPUs. Our analysis shows that the optimal sharing decision varies based on the number of computations performed on each parameter during the backward pass. In general, more resources should be allocated to computation for compute-intensive models. In addition, the optimal sharing configuration relies heavily on the bucketing strategy used during communication.

1 Introduction

In recent years, DNN models have grown ever-larger, requiring distributed training across multiple GPUs. Such distributed training workloads require significant amount of computation as well as communication to synchronize the results of computation among worker nodes.

A lot of efforts have been devoted in reducing the communication overhead introduced in distributed training. One commonly used strategy is to hide the communication latency by overlapping communication with computation. However, evidences have shown that, when overlapped with communication, compute kernels can run for much longer than when running alone in distributed training. This is because the compute kernels typically provide enough concurrency to fully utilize the underlying accelerators. When computation is overlapped with communication, the computation workload gets a smaller share of the underlying accelerators. Consequently, running computation and communication concurrently sometimes delivers worse performance than running them sequentially.

In this project, we study the the impact of resource sharing between computation and communication on the performance of distributed training. We first modify NVIDIA NCCL's communication primitives [5] so that we can configure the amount of resources each call to the communication kernels take. We focus our study on the AllReduce kernel, which is widely used in distributed training with data parallelism to synchronize the parameters across all worker nodes for each iteration. We then vary the amount of resources consumed by the communication kernels and see how different compute kernels are affected by this scheduling decision. We also study how this scheduling decision affects the training performance of various models with different batch sizes. Finally, we attempt to summarize our experiment results in some general rules on determining how resources should be shared between computation and communication to achieve maximum training efficiency. Through our study, we found that the heuristics used by NCCL in determining the amount of resources each communication kernel takes is suboptimal and that: (1) we should allocate as much resources as possible to the last communication kernel during each training iteration since it cannot be overlapped with any computation (2) we should allocate as little resources to communication as possible while ensuring that all communication kernels (except the last one) can overlap completely with compute kernels.

2 Background

Modern deep learning workloads often involve massive computation, making it impractical to train with a single processing unit. Data parallelism is frequently used to distribute the workload among multiple processing units on single or multiple machines. In data parallelism, the training samples are distributed across the processing units on different machines, and each iteration is composed of 3 main steps: a forward pass where each node feeds its own share of training data into the model to obtain a loss, a backward pass to compute the gradient on model parameters from the loss, and a communication step (AllReduce) to sum up all gradients on each node and apply the gradient. Other strategies to distribute training across machines (model parallelism) involve more complicated communication and computation patterns and are beyond the scope of this paper.

2.1 Overlapping Computation with Communication

There has been a line of work trying to overlap communication with computation to reduce training time. Wait-Free Backpropagation (WFBP) [12] attempts to broadcast the gradient of a layer as soon as it has been computed from the backward pass. It is based on an important observation: once the gradient of a layer has been computed, the exact value of its parameters will not affect the result of the rest of the backward pass, and subsequent backpropagation will not change the gradient any further. Hence communicating gradients that have already been computed from the backward pass can be done concurrently with computing the rest of the gradients, which means the gradients of each layer can be communicated once they are ready instead of waiting for the entire backward pass to complete. WFBP thereby overlaps communication with the backward pass, but the forward pass of the next iteration has to wait for the communication to complete.

Merged-Gradient Wait-Free Backpropagation (MG-WFBP) [9] is similar to WFBP but takes a less aggressive approach. It is based on the observation that, since the number of parameters in each layer is typically small, batching the gradients of multiple layers during communication can significantly reduce overall communication time. Similar to WFBP, the forward pass of the next iteration has to wait for the backward pass to complete. This strategy, also known as bucketing, has been widely adopted in many state-of-the-art distributed learning frameworks such as PyTorch Data Distributed Parallelism (DDP) and Horovod [8].

ByteScheduler [6] takes advantage of the fact that gradients computed later in the backward pass correspond to the shallower layers that will be used early in the forward pass of the next iteration. It partitions large tensors into smaller chunks and gives gradients corresponding to shallower layers higher priority than those corresponding to deeper layers. This approach allows it to overlap communication with both the backward pass and the forward pass of the next iteration.

2.2 NVIDIA NCCL

State-of-the-are systems rely on communication primitives provided by collective communication frameworks, most notably the NVIDIA Collective Communications Library (NCCL) [5]. NCCL implements each communication primitive with multiple algorithms and decides which algorithm to use based on hardware setup, communication kernel size, etc. NCCL internally uses two variables to determine the resources allocated to each kernel: the number of channels, and the number of threads. In NCCL, each channel maps to one CUDA block (which is mapped to GPU streaming multiprocessors), and the number of threads determines the number of threads in each CUDA block. In this paper, we follow the convention and refer to the number of channels as the number of thread blocks and number of threads as the number of threads per block.

During library initialization, NCCL finds the hardware setup and measures hardware information such as network bandwidth. It then uses heuristics to estimate the latency and bandwidth associated with different algorithms and protocols, which is used to figure out the optimal combination for each communication kernel. When each kernel is invoked, NCCL again uses heuristics to determine the number of thread blocks and the number of threads per block that should be used for the optimal algorithm and protocol and launch the kernel with the specified configuration.

2.3 PyTorch Data Distributed Parallelism

Pytorch Data Distributed Parallelism (DDP) [4] is PyTorch's native distributed training framework for data parallelism. It uses the AllReduce communication primitive to synchronize model parameters across different processing units. To overlap communication and computation, PyTorch DDP uses bucketing to group parameters and launch the communicate kernel once the gradients inside the bucket are ready (MG-WFBP). PyTorch DDP decides how tensors are grouped into buckets during graph construction, which happens dynamically during the forward pass. It adds the parameters to buckets in the reverse order of `model.parameters()`. By default, PyTorch DDP uses a maximum bucket size of 25MB.

3 Implementation

The NCCL library uses environment variables to provide coarse-grained control for the resource usage for all the collective communication functions. `NCCL_NTHREADS` is used to control the number of CUDA threads per block, and accepts values from 64, 128, 256 and 512. `NCCL_MAX_NCHANNELS` and `NCCL_MIN_NCHANNELS` are used to set the number of CUDA blocks used.

These environment variables are read by the library during `ncclCommInit*` (which is called by `torch.distributed.init_process_group()`) to set the number of blocks and the number of threads per blocks. The configuration cannot be changed after initialization and is used for all collective communication function invocations.

```
ncclResult_t ncclAllReduceCfg(
  const void *sendbuff, void *recvbuff,
  size_t count, ncclDataType_t datatype,
  ncclRedOp_t op, ncclComm_t comm,
  cudaStream_t stream,
  int nblocks, int nthread);
```

Figure 1: The modified API, `ncclAllReduceCfg`, with two additional parameters to control the number of blocks and number of threads per block.

We augment the API of NCCL collective communication functions with two additional parameters to explicitly control the number of CUDA blocks (`nblocks`) and threads per block (`nthread`) on a per invocation basis. Figure 1 shows the modified API for `ncclAllReduce`, which we call `ncclAllReduceCfg`. Newly written CUDA programs can use the API to control the resource usage for AllReduce.

In order to provide ABI compatibility of the NCCL library, we also make use of two additional environment variables, `NCCL_LOCAL_NCHANNELS` and `NCCL_LOCAL_NTHREADS`, which control the launch configuration of the NCCL AllReduce. They are read by the original `ncclAllReduce` on every invocation. We acknowledge that this is not a proper use of environment variables, but it allows users to use the modified library without source code changes to the applications that depend on it (e.g., PyTorch). Users can specify `LD_PRELOAD` to point to the shared library file to load the modified version.

4 Analysis

4.1 Experiment Setup

In this section, we study the effect of overlapping computation with communication when the GPUs reside on the same machine. We run our experiment on a CloudLab [7] Clemenson r7525 instance with 2 PCIe-attached Tesla V100S, each with 32GB memory. The GPUs are not connected through NVLink. In this setup, NCCL uses Ring Reduce implementation for AllReduce with 2 thread blocks and 256 threads per block (2×256) by default. All of our experiments, unless otherwise specified, are conducted with the PyTorch DDP framework [4].

4.2 Kernel-Level Analysis

In this section, we present our kernel-level analysis when different amounts of resources are allocated to communication.

# Blocks	Threads per Block: 64	128	256	512
1	1.23	0.93	0.82	0.88
2	0.94	0.89	1.00	1.20
3	0.88	0.99	1.05	1.42
4	0.90	0.93	1.17	1.63

Figure 2: The relative time for 25MB NCCL AllReduce with respect to the default configuration (2×256).

4.2.1 Communication Sensitivity

We start with sensitivity analysis on the NCCL AllReduce with numbers of thread blocks ranging from 1 to 4 and the number of threads per block in 64, 128, 256, and 512. We measure how long it takes to AllReduce a tensor of 25MB on each GPU. Figure 2 shows the result of this experiment. We make the following three observations.

First, the default configuration (2×256) is far from the optimal. Plenty of other configurations shown in green can indeed achieve a shorter time. Most notably 1×256 achieves 18% speedup with half of the thread blocks used and 2×128 is 11% faster using half the number of threads per block. By profiling the kernels, we found that both configurations achieve higher instruction per warp (13K and 12K) compared to the default one (7K) while the total number of instructions is about the same (≈120K).

Second, comparing 1×64 with 1×128 and 2×64, it's possible to make a tradeoff between the execution time and the resource utilization: with half the resources used, 1×64 only incurs about a 30% slowdown. During model training, if the communication time is less than the computation time, it will be helpful to use fewer resources for NCCL AllReduce and allocate them to computation.

Third, we note that it's necessary that more resources mean faster execution. The configuration 4×512 takes 63% more time to complete compared to the default one while using $2\times$ of the thread blocks and $2\times$ of the threads per block.

4.2.2 Computation Kernel Slowdown

In this experiment, we measure the slowdown experienced by different compute kernels when overlapping with a communication kernel. In a typical workload, due to bucketing, each communication kernel is typically overlapped with multiple computation kernels. We launch a NCCL AllReduce kernel that will run for long enough so that it will completely overlap with the compute kernel. We vary the number of blocks and number of threads per block for the AllReduce kernel and record the time to complete each compute kernel.

Kernel	Definition	Input Shape
Average Pooling	`nn.AdaptiveAvgPool2d((1, 1))`	(64, 64, 224, 224)
Batch Norm	`nn.BatchNorm2d(64)`	(64, 64, 224, 224)
Convolution	`nn.Conv2d(64, 64, kernel_size=7, stride=2, padding=3, bias=False)`	(64, 64, 224, 224)
Linear	`nn.Linear(224 * 224, 64)`	(64, 64, 224 × 224)
ReLU	`nn.ReLU(inplace=True)`	(64, 64, 224, 224)

Table 1: Definition of computation kernels and their input shape.

# Blocks \ Threads per Block	64	128	256	512
1	1.02	1.37	1.83	2.11
2	2.12	2.99	3.83	3.69
3	3.71	4.81	5.81	6.38
4	5.08	7.05	7.80	8.71

(a) Average Pooling

# Blocks \ Threads per Block	64	128	256	512
1	1.10	1.56	1.88	1.99
2	2.14	2.28	2.47	2.52
3	2.69	2.88	3.33	2.31
4	2.55	3.98	2.61	2.78

(b) Batch Normalization

# Blocks \ Threads per Block	64	128	256	512
1	1.08	1.37	1.82	2.22
2	1.79	1.88	2.32	2.26
3	1.79	2.13	3.21	2.40
4	2.41	2.53	2.74	2.31

(c) Convolution

# Blocks \ Threads per Block	64	128	256	512
1	1.02	1.43	1.87	2.11
2	1.45	2.44	2.67	1.65
3	1.97	2.15	3.98	2.27
4	2.50	2.82	2.86	3.00

(d) Linear

# Blocks \ Threads per Block	64	128	256	512
1	1.01	1.13	1.33	1.51
2	1.61	1.84	2.22	2.47
3	1.98	2.83	3.26	3.59
4	3.00	3.91	4.44	4.78

(e) ReLU

Figure 3: The slowdown of the computation kernel when different numbers of blocks and threads per block are allocated to NCCL AllReduce. The number shows the time relative to running the computation kernel alone.

Our study involves five different compute kernels that are commonly used in deep neural networks. The definition of the kernels and the input shape are shown in Table 1. Note that the computation kernel and the AllReduce kernel use different input tensors.

Figure 3 shows the slowdown of the computation kernels when overlapped with the NCCL AllReduce. With the least amount of resource allocated to AllReduce (1×64), the computation kernels only show less than 10% slow down. Notably, the slowdown is less than 3% for Average Pooling, Linear, and ReLU. The default configuration of AllReduce (2×256) slows down the computation kernel by $2.22\times$ to $3.83\times$. The result of this experiment shows that with fewer resources allocated to communication, it's possible to achieve great speedup for computation.

4.3 Case Study: ResNet-18

We then proceed to analyze how different amounts of resources allocated to communication affect the training time of the entire model. We first train ResNet-18 [2] with a batch size of 32 and vary the number of blocks and number of threads per block. We record the throughput achieved in this setup and compute its performance under various configurations relative to the default configuration.

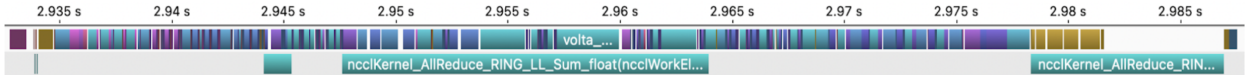
Figure 4 shows our experiment results. Our first observation is that the heatmap we obtained in this experiment is similar to the heatmaps presented in Figure 3. Compared with the default configuration, we can obtain a throughput as much as 7.1% higher by reducing the amount of resource allocated to communication. And with more resources allocated to communication, the performance becomes worse. A noticeable exception is when we reduce the number of blocks to 1 and the number of threads per block to 64. To understand this, we present the timeline when training with different configurations in Figure 5.

# Blocks \ Threads per Block	64	128	256	512
1	0.2%	6.6%	7.1%	3.2%
2	4.2%	3.2%	0.0%	-11.3%
3	-5.2%	0.0%	-4.6%	-17.3%
4	3.4%	1.2%	-6.2%	-21.2%

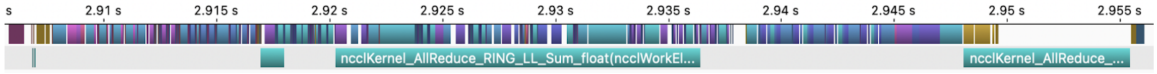
Figure 4: Throughput for training ResNet-18 with batch size 32 per GPU. The heat map shows the improvement relative to the default configuration (2×256).

4.3.1 Timeline

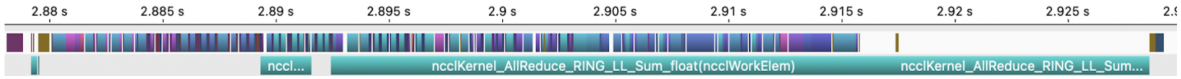
Comparing Figure 5a to Figure 5b, surprisingly, we notice that, when overlapped with computation, the time taken to complete each AllReduce remains largely unaffected when we reduce the number of blocks allocated to AllReduce, which confirms the finding in Section 4.2.1. More importantly, in both cases, all communication kernels, except the last one, are completely overlapped with the compute kernels. As a result, since computation becomes faster when less resources are allocated to the communication kernels, the computation



(a) Default configuration (2×256): 42 ms computation and 26 ms communication with 18 ms overlap.



(b) Optimal configuration (1×256): 38 ms computation and 26 ms communication with 17 ms overlap.



(c) Suboptimal configuration (1×64): 35 ms computation and 39 ms communication with 25 ms overlap.

Figure 5: The timeline for training ResNet-18 for one iteration with a batch size of 32

time when AllReduce takes only 1 thread block becomes shorter, reducing the total training time.

Figure 5c, on the other hand, shows what happens when too few resources are allocated to communication. When we only allocate one thread block and 64 threads per block to AllReduce, the communication time becomes significantly longer than before. As a result, even though the compute time becomes much shorter than before, communication starts to dominate the total training time, leading to suboptimal performance. This observation indicates that a potentially good scheduling strategy when training with data parallelism on a single multi-GPU machine is to allocate as few resources to communication as possible while ensuring that all communication kernels (except the last one) are completely overlapped with computation. As for the last communication kernel, since there is no way we can overlap it with any compute kernels, we choose the configuration that makes it finishes fastest.

4.3.2 Effect of Batch Size

In this experiment, we study how batch size affects the optimal resource allocation between computation and communication. We train ResNet-18 with various batch sizes and record the configuration under which it achieves the optimal configuration. The training time breakdown of the default and optimal configurations are shown in Figure 6. The breakdown is gathered by PyTorch profiler, which records CUDA events with starting and ending timestamps. We categorize the events as communication if it starts with `nccl` and then post-process the trace to compute the overlap.

By increasing the batch size, we effectively increase the amount of computation for each iteration while keeping the amount of data communicated constantly. This agrees with Figure 6: as batch size increases, computation takes significantly longer time, while the communication time remains roughly the same. Consequently,

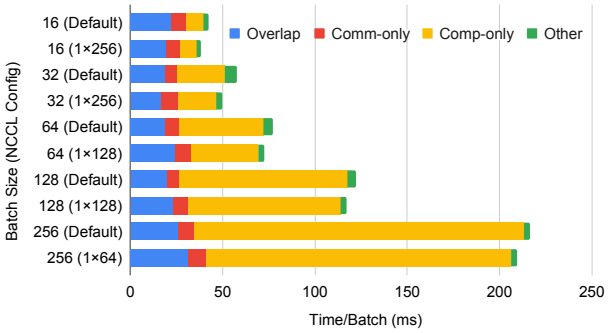


Figure 6: The time breakdown of one iteration with varying batch sizes. For each batch size, we show the training time for default and optimal configuration.

with larger batch size, the optimal configuration would allocate fewer resources to communication, since the computation is intense enough to fully overshadow the communication kernels (except the last one).

The second observation is that, with a larger batch size, the improvement of optimal configuration relative to default tends to become smaller. This is because the benefit obtained through better resource allocation decisions is limited by the amount of communication. In fact, the difference in training time between default and optimal configuration remains relatively constant, with the 3.97 ms improvement when batch size is 16, and 6.92 ms improvement when batch size is 256.

4.4 Model-Level Analysis

4.4.1 Model Intensity

To quantify the amount of computation relative to communication when training a particular model, we borrow the model intensity metric introduced in [10]. Let C be the number of floating-point operations involved in the forward pass with a batch size of 1 and D be the model

	#Param D	#FLOP C	Intensity C/D	Batch Size 16	32	64	128
AlexNet	61.1 M	1.4 G	23	1×256 14.7%	1×256 13.5%	1×256 15.3%	1×256 9.8%
VGG-16	138.4 M	31 G	224	1×256 6.2%	1×256 2.3%	3×64 10%	2×64 9.8%
VGG-19	143.7 M	39.3 G	274	1×256 8.7%	4×64 0.5%	4×64 8.5%	2×64 10.7%
ResNet-18	11.7 M	3.6 G	311	1×256 10.5%	4×64 12.2%	1×64 9.5%	1×64 8.2%
ResNet-50	25.6 M	8.2 G	321	1×256 10.8%	1×64 9.3%	1×64 8.9%	1×64 3.2%
ResNet-101	44.6 M	15.7 G	351	1×128 15.3%	1×64 7.2%	1×64 17.2%	1×64 10.6%
ViT-Base/16	86.6 M	58.5 G	675	1×256 6.5%	1×128 7.7%	1×64 9.4%	1×64 6.0%

Table 2: The best configuration and the relative improvement compared to the default configuration for each model and batch size.

size, the model intensity I is defined as:

$$I = \frac{C}{D}$$

This metric captures the specific model computation and communication pattern. Intuitively, models with larger I are more compute-intensive and models with smaller I are more communicate-intensive.

With sequential models (all parameters are involved during training), each node sends and receives $2 \cdot D$ parameters during the communication step. Since the number of flops involved in a backward pass is roughly twice as much as the number of flops involved in a forward pass, with a batch size B, the total number of computations involved in one iteration is approximate $3 \cdot B \cdot C$, and the compute-to-communicate ratio involved in the iteration is roughly

$$\frac{3 \cdot B \cdot C}{2 \cdot D} \propto B \cdot I$$

4.4.2 Grid Search

For model-level analysis, we choose 7 popular vision models for comparison: AlexNet [3], VGG-16 [11], VGG-19 [11], ResNet-18 [2], ResNet-50 [2], ResNet-101 [2], and ViT-Base/16 [1]. For each model, we use the batch size from 16 to 128 and performed a grid search for the NCCL configuration that achieves the largest throughput in term of images per second. Specifically, we train the model for 5 batches (per GPU), compute the throughout with batch size divided by CPU time, and use the median value over 10 data points (across 2 GPUs) as the final throughout.

Table 2 lists the best configuration and the relative improvement compared to the default configuration (2×256). The full results are shown in Appendix A.

We make the following observations from the results. (1) We observe up to 17.2% speedup for ResNet-101 with batch size 64 using configuration 1×64. (2) For each model, a general trend is that as the batch size increases, the optimal configuration tends to shift towards allocating less total number of threads for AllReduce. Since the amount of communication is the same when the batch size increases, more computation needs to be done, and thus it's preferable to allocate more resource to computation. (3) In terms of the model intensity, the models with larger intensity is more "computationally hungry", so they reach the minimal configuration for AllReduce (1×64) faster with increasing batch size.

4.4.3 Time Breakdown

Figure 7 shows the time breakdown between the default configuration and the optimal configuration for models with batch size 64. The breakdown is gathered by PyTorch profiler and post-processed using the same method as introduced in Section 4.3.2.

AlexNet with a low model intensity spent most of the time on communication, which almost entirely overlaps with computation. The 6ms computation only may come from the forward pass and thus does not change when the resource allocated to NCCL AllReduce changes. The optimal configuration saves time from both communication (135 ms to 116 ms) and computation (41 ms to 28 ms). The save in communication time with lower resource usage agrees with the findings in Section 4.2.1.

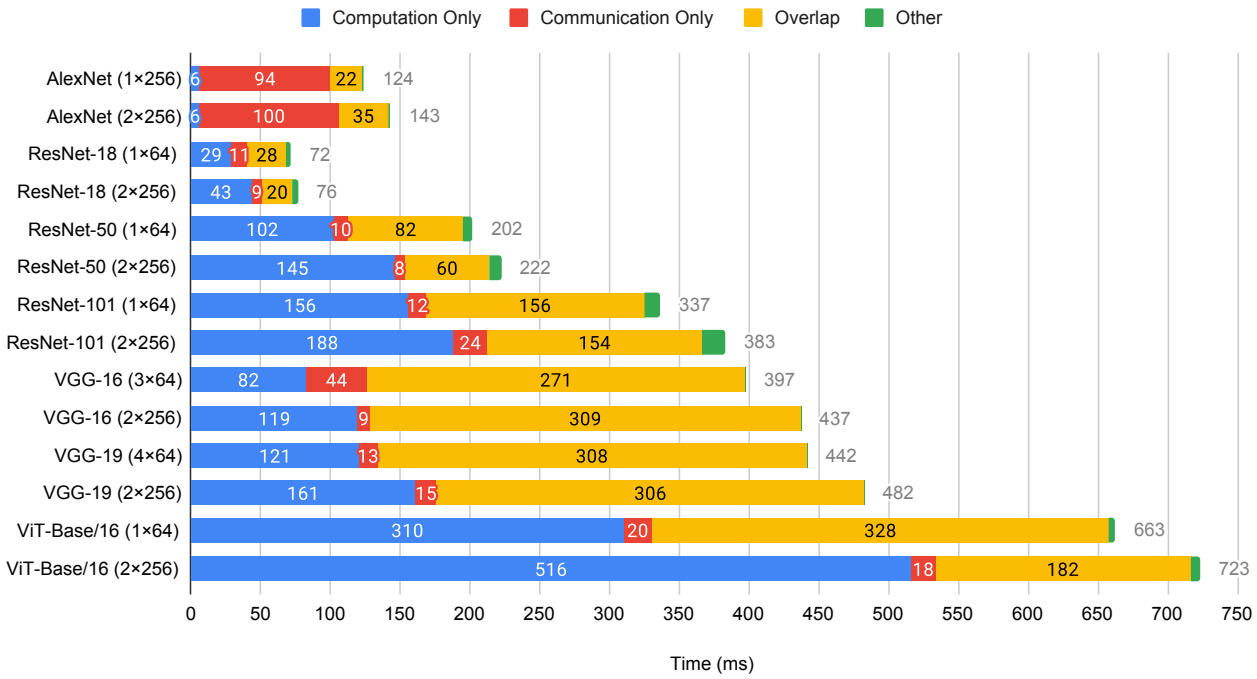


Figure 7: Time breakdown for different models. We compare the optimal configuration and the default configuration (2×256) for each model.

The ResNet models all achieve the optimal configuration at 1×64. For all the configurations, the communication can overlap well computation as indicated by low numbers for the communication-only bar. The save in time for ResNet models mostly comes from a reduction of the computation time as more resources are allocated for computation. The cost is that the communication spent more times (29 ms to 39 ms for ResNet-18, 68 ms to 92 ms for ResNet-50, and 168 ms to 178 ms for ResNet-101), but that's fine since communication can still be overlapped as indicated by a longer bar for the overlap time.

The optimal configuration for VGG-16 indeed achieves worse overlap (309 ms to 271 ms) and longer communication-only time (9 ms to 44 ms), possibly due to longer time spent on the last AllReduce, which cannot be overlapped with any communication. Nevertheless, the reduction in the computation time is significant and thus results in an 10% improvement in the total time. The time spent on the communication for VGG-19 is about the same, and the improvement comes form faster computation as computation-only reduces from 161 ms to 121 ms.

For ViT-Base/16, the default configuration achieves a better balance between computation and communication as the overlap time increases significantly. As more resources are allocated to computation, the computation time reduces from 698 ms to 638 ms.

5 Future work

Fine-grained scheduling. For all the experiments, we set a fixed number of thread blocks and the number of threads per block across the entire training process. From Section 4.3.1, we found that a better strategy would be to allocate less resource to communication as possible while ensuring that the AllReduce kernels (except the last one) can be overlapped with computation kernels. As for the last AllReduce kernel invocation, since there is no computation kernel, we want to finish it as fast as possible by allocating the appropiate amount of resources. With the interface described in Section 3, we'd like to integrate it with PyTorch to set different resource usages to different invocations to AllReduces.

Cross-node communication. In this project, we only analyzed the single-node multi-GPU case. For cross-node communication, the network delay may dominate the AllReduce execution time. We want to investigate whether tuning the resource allocated to AllReduce still has a major impact on the communication time.

Horovod. So far, all the measurements are conducted with the PyTorch DDP framework. Horovod uses a different bucketing strategy and allocates smaller buckets to the later ones. We would like to analyze how different bucketing strategies affect the optimal configuration.

Bucketing size. Bucketing size is yet another tunable

configuration for distributed training. We currently use the default configuration of 25MB max. It would be interesting to see how the bucketing size affects the optimal configuration.

6 Conclusion

In this project, we explore the effect of resource sharing between communication and computation on the training time. We found that the default sharing configuration used by NVIDIA NCCL is suboptimal, and better resource sharing decisions can improve the throughput by as much as 17.2% when training ResNet-101 with PyTorch DDP on a single machine with multiple GPUs.

Through our study, we found that the heuristics used by NCCL in determining the amount of resources each communication kernel takes is suboptimal and that: (1) we should allocate as much resources as possible to the last communication kernel during each training iteration since it cannot be overlapped with any computation (2) we should allocate as little resources to communication as possible while ensuring that all communication kernels (except the last one) can overlap completely with compute kernels.

References

[1] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[2] 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 (CVPR)*, June 2016.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

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

[5] NVIDIA. Nvidia collective communication library (nccl).

[6] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 16–29, 2019.

[7] R. Ricci, E. Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), Dec. 2014.

[8] A. Sergeev and M. Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[9] S. Shi, X. Chu, and B. Li. Mg-wfbp: Efficient data communication for distributed synchronous sgd algorithms. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 172–180. IEEE, 2019.

[10] S. Shi, Z. Tang, X. Chu, C. Liu, W. Wang, and B. Li. A quantitative survey of communication optimizations in distributed deep learning. *IEEE Network*, 35(3):230–237, 2020.

[11] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[12] H. Zhang, Z. Hu, J. Wei, P. Xie, G. Kim, Q. Ho, and E. Xing. Poseidon: A system architecture for efficient gpu-based deep learning on multiple machines. *arXiv preprint arXiv:1512.06216*, 2015.

A Appendix

We present the full results of Section 4.4.2 in the appendix.

Name	BS	AllReduce Configurations (Num Thread Block × Num Thread per Block)																Speedup w.r.t. 2 × 256
		1				2				3				4				
		64	128	256	512	64	128	256	512	64	128	256	512	64	128	256	512	
AlexNet	16	95	127	**134**	132	119	122	117	96	126	119	121	86	126	127	99	76	14.7%
	32	190	243	**276**	260	234	250	243	192	250	240	238	170	246	243	203	151	13.5%
	64	381	467	**517**	496	461	467	448	382	489	463	461	335	478	478	394	301	15.3%
	128	714	906	**980**	956	881	935	893	741	925	901	893	648	904	912	778	570	9.8%
ResNet-18	16	357	380	**399**	328	392	383	361	327	385	349	353	293	384	352	331	262	10.5%
	32	595	603	602	606	595	594	543	509	538	562	564	462	**610**	579	556	455	12.2%
	64	888	**885**	877	868	842	859	811	775	864	810	793	744	841	847	801	702	9.5%
	128	1078	995	**1058**	1046	1037	1046	997	984	1028	1037	1027	967	1036	1042	997	910	8.2%
ResNet-50	16	147	162	**169**	159	167	146	152	140	164	156	147	114	160	153	135	114	10.8%
	32	249	**241**	241	233	235	224	228	212	227	226	222	195	225	220	216	175	9.3%
	64	313	**304**	297	289	302	298	288	276	298	287	286	265	291	291	283	254	8.9%
	128	345	**341**	340	334	340	333	334	325	337	335	333	321	335	335	329	304	3.2%
ResNet-101	16	85	**98**	91	89	94	92	85	74	94	89	83	67	94	90	83	67	15.3%
	32	144	**143**	136	135	140	128	134	116	129	137	123	112	129	131	121	106	7.2%
	64	191	**181**	180	176	181	163	163	163	174	174	172	158	174	161	166	150	17.2%
	128	214	196	**205**	204	204	202	193	190	203	199	196	192	201	199	196	186	10.6%
VGG-16	16	40	51	**55**	53	48	52	52	43	52	53	49	37	51	52	48	34	6.2%
	32	74	89	**97**	95	88	96	95	80	95	93	92	69	90	95	85	65	2.3%
	64	128	142	155	150	146	153	146	134	**161**	152	142	126	156	150	131	108	10.0%
	128	188	204	197	193	**209**	198	190	179	207	193	184	165	200	189	178	159	9.8%
VGG-19	16	38	47	**53**	51	46	49	49	40	49	49	47	36	49	49	42	32	8.7%
	32	69	84	90	90	78	88	89	76	89	90	85	67	**90**	88	75	61	0.5%
	64	113	134	139	132	137	139	132	119	142	136	126	105	**143**	132	122	96	8.5%
	128	172	177	171	164	**182**	171	165	160	181	166	161	147	174	164	156	138	10.7%
ViT-Base/16	16	49	56	**57**	55	53	56	54	47	55	55	52	42	55	52	50	40	6.5%
	32	76	**77**	75	74	76	74	72	67	74	72	70	63	73	72	68	58	7.7%
	64	94	89	88	88	**90**	87	86	84	87	86	85	81	87	86	83	78	9.4%
	128	99	**96**	95	93	95	94	93	91	94	93	92	89	93	93	91	88	6.0%

Figure 8: Median training throughput across 10 data points (5 batches × 2 GPUs). The unit is in images per second. The rows show models with different batch sizes. The best result for each row is highlighted in green with bold font and the second best one is highlighted in yellow. The column varies the amount of resources allocated for AllReduce.