Accelerated and Memory-Efficient Distributed Deep Learning: Leveraging Quantization, Parallelism Techniques, and Mix-Match Runtime Communication

Thesis

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Radha Gulhane

Computer Science and Engineering

The Ohio State University

2024

Abstract

In recent years, there has been significant research and development in Deep Learning (DL) due to its efficiency and extensive applicability across diverse domains, including Computer Vision and Large Language Models. However, the architecture of large Deep Learning models, containing dense layers, makes them compute and memory intensive. Distributed Deep Learning (Distributed DL) is the successful adaption to accelerate and enable training and inference for large-scale DL models, where it also deals with various parallel approaches, inference and training techniques, and communication optimization strategies to enhance performance.

In this thesis, we focus on accelerated and memory-efficient techniques to optimize distributed training and inference. It is broadly categorized into three different approaches: 1. Inference for scaled images using quantization, achieving a speedup of 6.5x with integer-only quantization and 1.58x with half-precision, with less than 1% accuracy degradation. 2. MPI4DL: Distributed Deep Learning Parallelism framework encompassing various parallelism techniques with integral components such as Spatial Parallelism, Bidirectional Parallelism, and Hybrid Parallelism 3. Communication optimization by leveraging MCR-DL: A distributed module for DL frameworks with support for mixed-backend communication, dynamic selection of the optimal backend, and communication optimization enhancements such as compression and tensor fusion.

Acknowledgments

My sincere gratitude goes out to everyone who contributed to the successful culmination of my thesis. I am especially thankful to Prof. D. K. Panda for granting me the invaluable opportunity to work under his supervision and alongside his research group.

I want to extend my thanks to Prof. Hari Subramoni and Dr. Aamir Shafi for their consistent guidance and assistance throughout this journey. Their mentorship has led to my growth and significant contribution.

I am extremely grateful to receive mentorship from Quentin Anthony and Nawras Alnaasan. I would especially like to thank Quentin Anthony for being an incredible mentor and person. By seeing him, you will be inspired. He not only sculpted my knowledge but also presented me with profound insights, enriching my journey.

Thanks to all NOWLAB and HiDL group members, who provided me with the opportunity to collaborate and learn about their exciting research work.

Finally, I would like to thank my lifetime supporters, family, and friends, for always being there, keeping me motivated, and making me who I am today.

Vita

2017-2021……………….….…….………………………… Bachelor of Technology
Vishwakarma Institute of Technology
Pune, India

2020-2021……………….….….……….…………………………… Software Engineer
Seagate Technology
Pune, India

2022-Present…………………..…..…………..…………………………. Master of Science
The Ohio State University
Columbus, OH, USA

2023-Present…………………..…..…………..………………………Graduate Research Associate
The Ohio State University
Columbus, OH, USA

Publications

R. Gulhane, Q. Anthony, H. Subramoni, and DK Panda, "Infer-HiRes: Accelerating Inference for High-Resolution Images with Quantization and Distributed Deep Learning," in Practice and Experience in Advanced Research Computing, Jul 2024

L. Xu, Q. Anthony, Q. Zhou, N. Alnaasan, R. Gulhane, A. Shafi, H. Subramoni, and DK Panda, "Accelerating Large Language Model Training with Hybrid GPU-based Compression," in IEEE/ACM International Symposium on Cluster, Cloud, and Internet Computing, May 2024

Fields of Study

Major Field: Computer Science and Engineering

Table of Contents

List of Tables

List of Figures

List of Equations

Chapter 1 Introduction

In the recent era of Artificial Intelligence (AI), Deep Learning (DL) has emerged as a backbone of modern research and development, exhibiting remarkable success across various domains. Its efficiency is evidenced by their application in groundbreaking technologies such as large language models like LLaMA, ChatGPT, Mistral, as well as vision models like ResNet, AmoebNet, and vision transformer utilized in medical imaging, autonomous driving, satellite imaging, and robotics.

Deep Neural Networks architecture, characterized by deep layers and many input parameters, poses significant computational and memory requirements. This becomes particularly challenging when dealing with single-GPU setups, as the models exceed the capacity of a single processing unit, necessitating out-of-core techniques for training and inference.

To address these challenges, the paradigm of Distributed Deep Learning (Distributed DL) has gained prominence. By harnessing the collective computational power of multiple GPU resources Distributed DL enables as well as accelerate training and inference on large-scale neural networks. Given the necessity for performance scalability, handling large input parameters, and accommodating deep-layered models, it becomes essential to leverage Distributed DL.

1.1 Motivation

One of the focused application areas of deep learning is high-resolution images [1] [2] [3], notably utilized in areas such as medical imaging, satellite imagery, and robotics. These images often have dimensions as large as 100,000×100,000 pixels. To meet the demands of large DL models and handle large-scale image sizes, distributed deep learning emerges as a crucial solution. While research in high-resolution image processing with DL remains essential due to its applicability, several studies [1] [4] [5] have focused on efficient training, whereas very few have explored inference [6] [7] for high-resolution images. The studies focusing on inference with high-resolution images primarily involve a single-processing unit and are limited to small-scale images. Furthermore, the exploration of inference with quantization in the context of high-resolution images in deep learning and distributed DL for scaled images has yet to be pursued. **Can we leverage quantization and distributed DL to accelerate inference and reduce memory footprints for large-scale images without compromising accuracy?**

Furthermore, there has been significant research [4] [8] dedicated to efficiently accelerating and enabling training for large DL models, leading to the evolution of various parallelism techniques. These techniques range from simpler approaches like Data parallelism (DP) and Model Parallelism (MP) to more advanced methods such as Bidirectional Parallelism (GEMS), Spatial Parallelism (SP), and Hybrid Parallelism **Are all of these easily implementable within the given design and available for researchers to perform their own experiments and select the optimal one according to their needs and available computing resources?** When discussing the benchmarks provided by Deep

Learning Parallelism Frameworks such as PyTorch and DeepSpeed, it is noted that their support is limited to Data Parallelism, Layer Parallelism, and Pipeline Parallelism, lacking support for Bidirectional Parallelism (GEMS), Spatial Parallelism, and Hybrid Parallelism (GESMS + SP).

Besides different parallelism and memory optimization techniques in Distributed DL, communication plays a pivotal role and significantly influences performance. Different parallelism uses different collective communications such as, Point-to-point, All-Reduce, All-Gather to undergo different operations such as model parameter sharing, synchronizing gradients, which can be performance bottleneck. Currently, Distributed DL supports different backends such as MPI and NCCL. Different backends provide tradeoffs for different collective and message sizes. Furthermore, rapid advancements in the MPI world, such as MVAPICH-PLUS, MVAPICH-GDR [9] [10], and other MPI libraries, along with NCCL [11], make it essential to leverage their associated features in Distributed DL. However, the distributed module provided by DL frameworks does not cover all collective operations and often requires source building to support backends like MPI. Therefore, it is also essential to provide direct support to leverage these features without depending solely on the distributed module provided by DL frameworks (e.g., PyTorch's Distributed DL). For such requirements, such as mixed-backend, direct support of any backend, optimal selection, and overall optimizing communication, **can we have implementation to support any DL framework and leverage communication optimization techniques for any DL application without depending on the DL framework's distributed Module?**

1.2 Contribution

In this thesis, we will broadly explore optimal techniques in terms of quantization, various forms of parallelism, collective communication techniques, and specifically address the questions outlined in the motivation section.

We proposed inference for scaled images leveraging quantization. We provided thorough evaluation for Single GPU and Distribute DL quantization for half-precision and Integer-Only precision quantitation. We experimented on real-world pathology datasets and achieved less than 1% accuracy degradation while accelerating and reducing memory footprint for inference. (Chapter 2)

We introduced MPI4DL, a Distributed Deep Learning Parallelism framework encompassing various parallelism techniques essential for Distributed Deep Learning. This includes Spatial Parallelism, Bidirectional Parallelism, Data Parallelism, Model Parallelism, and Hybrid Parallelism. We meticulously detailed the existing parallelism strategies, evaluating strengths and limitations. We extended support for Spatial, Bidirectional, and Hybrid Parallelism as integral components of the MPI4DL framework. Comprehensive performance evaluations are conducted for each parallelism strategy, providing insights into their effectiveness and efficiency. (Chapter 3)

We exploited the novel technique of Mix-and-Match communication (MCR-DL [12]) for communication optimization in any DL framework and application. Additionally, we conducted performance evaluations comparing different collectives in various configuration settings. Furthermore, we have provided initial implementation support for MCR-DL and its integration with Megatron-LM. (Chapter 4)

Chapter 2 Accelerating Inference for High-Resolution Images with Quantization and Distributed Deep Learning

2.1 Introduction

High-resolution images find extensive use across diverse sectors like medical imaging, satellite imagery, and surveillance. Typically, these images are in the gigapixel range, often exceeding dimensions of 100,000×100,000 pixels. For instance, the CAMELYON16 [13] digital pathology dataset features whole-slide images (WSI) with resolutions reaching approximately 100,000x200,000 pixels at a maximum 40× magnification.

With the advancement of Deep Learning (DL) and its demonstrated effectiveness across various fields, it has emerged as a popular solution for addressing challenges in High-Resolution image tasks such as classification and segmentation. Some commonly adopted DL models for these tasks include ResNet [14], U-Net [15], and AmoebaNet [16], which employ deep conventional layers. However, considering the large size of the image and several convolution layers, it provides challenges due to memory and computation limitations, as it cannot be accommodated in a single GPU memory.

Several studies [17] [6] have adopted a patch-based approach, where each whole slide image (WSI) is split into small patches with image sizes such as 256x256. This approach further requires pixel-wise annotation or classification mechanism to classify each patch to well-suited classes. But use of deep convolutions neural networks restricts the patch size due to memory limitation. For instance, image size of 8192x8192 with ResNet101 model and batch size as one becomes out-of-core model on NVIDIA-A100-40GB GPU. To facilitate scaled image sizes and improve performance, Hy-Fi [4] and GEMS [8] have made significant contributions by enabling training using Spatial Parallelism for image sizes up to 16384x16384. It further improved performance by integrating different parallelism techniques. While most studies have primarily focused on efficient deep learning training approaches for high-resolution images, optimizing inference in the context of high-resolution images remains unexplored.

We propose and evaluate quantization approach to accelerate Deep Learning inference for high-resolution images to reduce memory and computation requirements while maintaining accuracy. Quantization is a technique where model parameters are converted to low-precision such as 16-bit floating point or 8-bit integer from 32-bit floating point. This results in reducing memory utilization and latency and its proven efficiency for DL inference has been evaluated in recent surveys [18] [19] and has also been employed for Large Language Models [20] [21]. We leverage the benefits of quantization to accelerate high-resolution image inference for deep learning models. Furthermore, to enable scaled image inference, further enhance acceleration, and harness the memory and compute-efficient benefits of different parallelism, we introduce quantization support for Spatial, Layer, and Pipeline parallelism in Distributed DL.

2.2 Motivation

While research in high-resolution images with DL remains essential due to its applicability, several studies have been conducted for efficient training, whereas very few have delved into the inference for high-resolution images. The studies focusing on inference with high-resolution images primarily involve a single-processing unit and are limited to small-scale images. The exploration of inference with quantization in the context of high-resolution images in Deep Learning and Distributed DL for scaled images is yet to be pursued.

Precision	**Memory Utilization (GB)**	**Memory Reduction**	**Throughput (Img/Sec)**	**Speedup**
FP32	12.48	**Baseline**	145.22	**Baseline**
FP16	7.64	**1.63×**	224.85	**1.55×**
BFLOAT16	5.28	**5.28×**	226.12	**1.56×**
INT8	2.39	**5.23×**	903.35	**6.22×**

Table 2.1 Memory and Throughput evaluation with quantization using ResNet101 for 256×256 image size and 64 Batch Size on NVIDIA A100-40GB

We evaluated the quantization effects on latency and memory footprint for an image size of 256x256 using ResNet101. Table 2.1 provides the memory and speedup evaluation by comparing quantization with baseline full-precision (FP32), half-precision (FP16, BFLOAT16), and integer-only precision (INT8). Results show a significant reduction in latency and memory utilization, with the best performance observed for INT8, reducing memory requirements by 5.23× while improving speedup by 6.22×. Further, as ResNet101 cannot scale beyond 2048×2048 or 4096×4096 image size on single GPU, to support larger images and slide level inference, we studied different parallelism implemented in Hy-fi and GEMS to enable image-sizes such as 8192×8192 and 16384×16384 and support quantization.

Consider the real-world application of digital pathology images, where inference for one whole slide image (WSI) contains an average of 500 patches, each of size 256x256. The inference time on a CPU [22] [23]can take several minutes, while on a GPU, it reduces to seconds. Utilizing GPU-enabled quantization further minimizes this time to just 1-2 seconds.

2.3 Background

2.3.1 Working Principal: Floating-Point to Integer-Only Quantization

Floating-point conversion, i.e., converting 32-bit floating point to half-precision floating point is relatively simple, as both are floating point data types and follow the same scheme representation. On contrast, converting 32-bit floating point to 8-bit integer significantly reduces value range to 256 values and requires using new scheme representation to map 32-bit float value to integer. This new representation scheme uses the range of floating-point values ([α, β] from Figure 2.1) in its representation to represent 32-bit floating point to integer values. Figure 2.1 shows mapping of floating-point range to b-bit integer values range. Equation 2.3 provides conversion calculation to represent b-bit integer value relative to floating-point, where x_q is b-bit quantized value of floating-point value x.

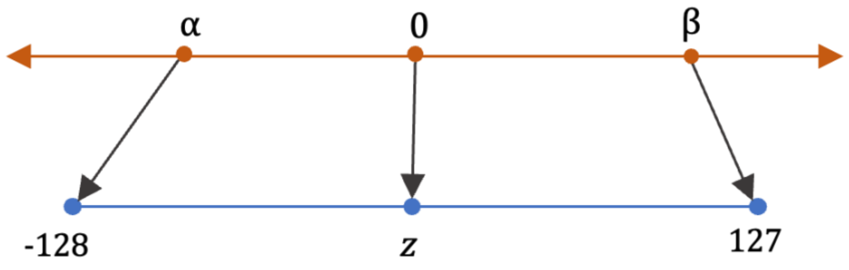


Figure 2.1 Mapping of floating-point to 8-bit values [24]

$$s = \frac{2^b - 1}{\beta - \alpha}$$ Equation 2.1

$$z = -\text{round}\,(\alpha \cdot s - 2^b - 1)$$ Equation 2.2

$$xq = \text{clip}(\text{round}\,(x \cdot s + z), -2^{b-1}, 2^{b-1} - 1)$$ Equation 2.3

Equation 2.1 and Equation 2.2 provide quantization parameters required in equation 3 scale factor (s) and zero-point value (z) respectively. The scale factor is the floating-point value and zero-point is b-bit integer value corresponding to the zero value in the floating-point representation. *clip* () maps values outside range to nearest integer representable value. To determine floating-point value range i.e. $[\alpha, \beta]$ calibration step is used which is done by performing forward pass with few given samples for model.

2.3.2 Post-Training Quantization

Post-Training Quantization (PTQ) converts the weights and activations of a pre-trained unquantized model to low-bit precision, thereby reducing memory and computation requirements for inference. PTQ is categorized into two different modes, namely dynamic quantization, and static quantization. Dynamic Quantization converts weights into low-precision values beforehand but converts activation dynamically at runtime depending observed data range. On other hand, in static quantization, the weights and activation are both quantized into low-precision values and requires calibration step to determine these values. However, for GPUs, PTQ with PyTorch [25] is limited to Static Post-Training Quantization mode via TensorRT [26], and for our work, we have used Static Post-Training Quantization. Figure 2.2 provides an overview of Post-Training Quantization Inference Pipeline.

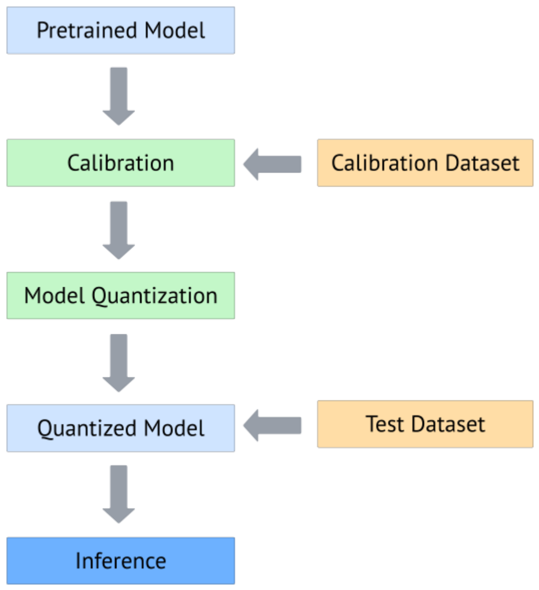


Figure 2.2 Post-Training Quantization

2.4 Proposed Solution

We propose efficient inference for high-resolution images using DL on a single GPU, as well as in Distributed DL settings, leveraging post training quantization. We exploit the quantization precision range of 16-bit floating point, with both FP16 and BFLOAT16 datatypes, and 8-bit integer. As of today, 8-bit integer is the lowest precision for GPUs supported through PyTorch.

We provide quantization support for single GPU inference, specifically to facilitate patch-based inference, a widely used approach where patch sizes are small-scale images. Furthermore, to enable scaled images, slide-level inference, and improve performance, we enable quantization for Distributed DL. We utilized Spatial, Layer, and Pipeline Parallelism for Distributed DL from the Hy-Fi implementation.

We implement our solution in PyTorch [25] and provide an inference pipeline for high-resolution images, supporting different precision quantization and Distributed DL.

2.4.1 Single-GPU Quantization

For Single-GPU quantization, we leverage the Post-Training Quantization Inference Pipeline demonstrated in Figure 2.2. We support FLOAT16, BFLOAT16, and INT8 quantization for single GPU quantization. As of today, 8-bit integer is the lowest precision for GPUs supported through PyTorch.

2.4.2 Quantization for Distributed DL

When dealing with Distributed DL models, we follow the same pipeline as shown in Figure 3.2. However, it is important to note that model quantization is performed separately on each GPU. In Distributed DL, the model is distributed across GPUs, and it can be big enough to not fit into memory. Thus, we perform model quantization for each distributed part of model. In terms of Spatial Parallelism, for each spatial part, after initialization of model with given weights, we perform model quantization independently at each GPU device. Similar is the case with Layer parallelism. Figure 2.3 shows implementation pipeline for quantization in Distributed DL.

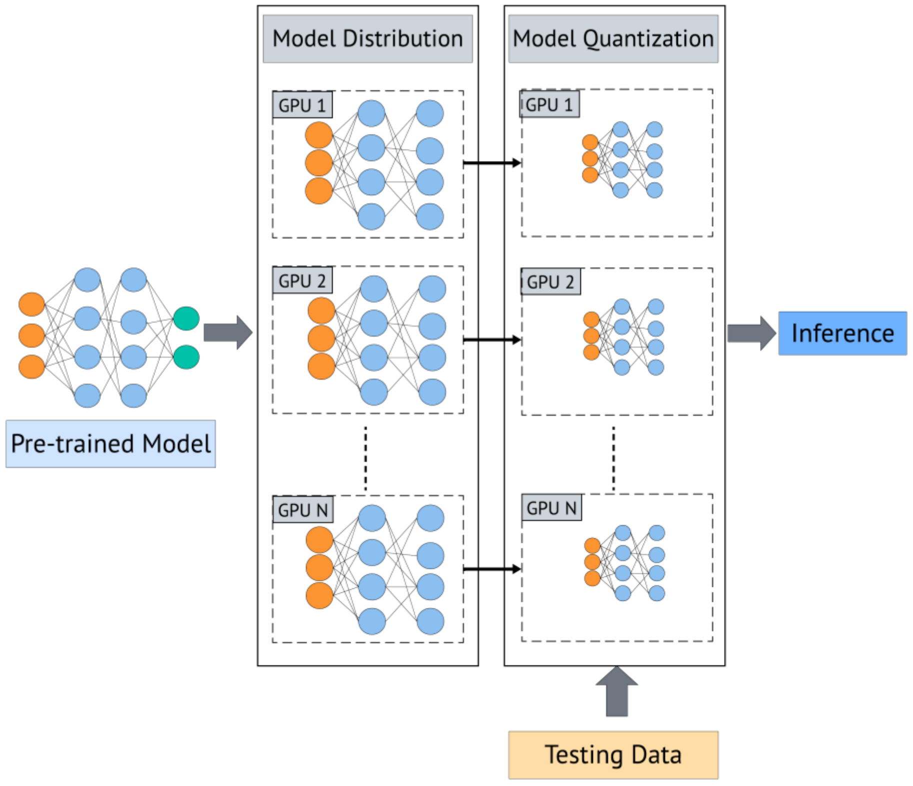


Figure 2.3 Inference using quantization in Distributed DL

Spatial parallelism requires halo-exchange to perform, as we will discuss in Chapter 3 (Figure 3.9 and Figure 3.10), where each convolutional and pooling operation, it performs point-to-point communication as part of the forward pass. In PyTorch, for GPUs, Integer-Only quantization is done via TensorRT. TensorRT takes the DL model defined in PyTorch and compiles it to support integer quantization specifically for NVIDIA GPUs. This compilation supports DL layers, such as convolutional, normalization, pooling, etc., but it does not cover collective communication function calls. Since spatial parallelism requires point-to-point communication for halo-exchange in the forward pass, TensorRT cannot resolve such communication calls during compilation, limiting Int8 quantization supported for spatial parallelism.

2.5 Performance Evaluation

Hardware Specifications

We conducted our experiments on NVIDIA A100-40 GB GPUs (2 GPUs per node) with AMD EPYC 7713 64-Core Processor.

Software Specifications

We used PyTorch v1.13.1 [25] as a Deep Learning framework and TensorRT [26] through Torch-TensorRT API for Integer-Only quantization. For collective communication in Distributed DL, we used NCCL [11] (NVIDIA Collective Communications Library) communication backend.

2.5.1 Effect of quantization on accuracy for Inference

For accuracy evaluation, we used ResNet101 and performed model quantization with different precisions. We conducted our accuracy evaluation on various datasets and compared quantization results with the baseline inference accuracy using FP32 precision. We used the following datasets: CAMELYON16 [13], ImageNet [27], CIFAR-10 [28] and Imagenette [29].

Dataset Description

CAMELYON16 is a real-world digital pathology dataset from a competition held by the International Symposium on Biomedical Imaging (ISBI) to detect metastatic breast cancer in whole slide images (WSI). It consists of 400 WSI images categorized into 2 classes: normal and tumor. ImageNet, CIFAR-10, and Imagenette are object detection datasets containing 1,431,167 images with 1000 object classes, 60,000 images with 10 classes, and 13,394 images with 10 classes, respectively.

Evaluation Methodology

For the CAMELYON16 Dataset, the total size is 300GB, and each image is around 5GB with an approximate image resolution of 100,000x200,000. Since these images cannot fit into memory, for accuracy evaluation, we used a patch-based approach. We extracted patches of size 256x256 containing the tissue region and labeled each patch based on the slide label.

To evaluate the quantization effect on each dataset, we trained ResNet101 for a few epochs to achieve the desired training accuracy, applied PTQ to obtain a quantized model with various precision levels, and then tested the accuracy for inference on either the testing or validation dataset.

Result Evaluation

Dataset	Precision			
	FP32	FP16	BFLOAT16	INT8
CAMELYON16	70.27	70.26	70.32	70.26
ImageNet	77.62	77.57	78.41	76.85
CIFAR-10	86.02	86.05	86.04	85.99
Imagenette	75.87	75.87	75.9	75.13

Table 2.2 Inference Accuracy (%) with Different Precision Quantized ResNet Model

Table 2.2 shows accuracy evaluation with quantization on different datasets. We observed negligible variations in accuracy while using different precision and demonstrated the accuracy degradation of less than 1%.

2.5.2 Quantization with Single GPU

We evaluate quantization effects for memory utilization and throughput on different image sizes to understand the benefits of quantization on small-scale image size. Figure 2.4 and Figure 2.5 show perform evaluation on different image sizes, 256×256, 512×512, and 1024×1024 with batch size of 32 on ResNet101 model, and we compare our results with baseline FP32 precision.

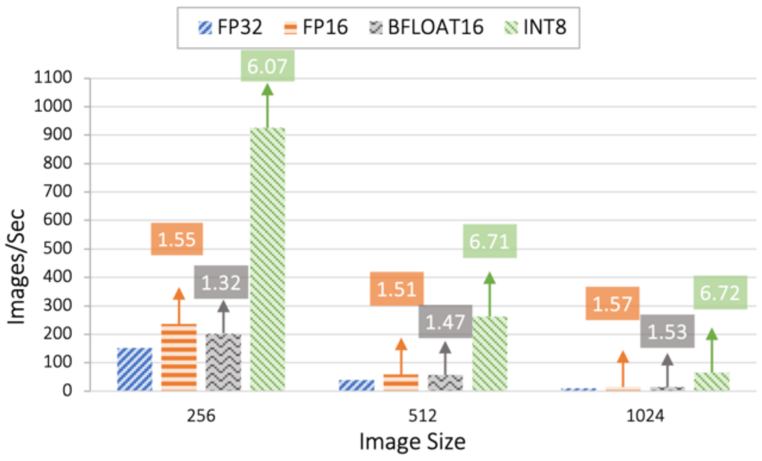


Figure 2.4 Throughput Evaluation on Single GPU

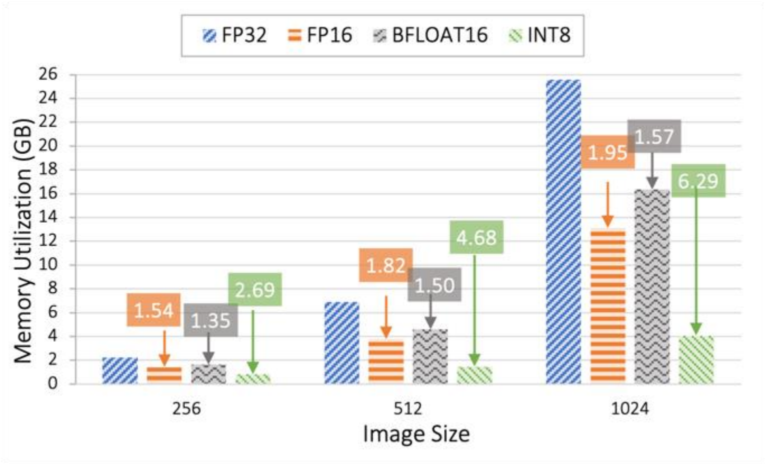


Figure 2.5 Memory Utilization Evaluation on Single GPU

Figure 2.4 illustrates the throughput evaluation. FP16 improved performance by an average of 1.54× BFLOAT16 by 1.45×, and INT8 by 6.5×. As shown in Figure 2.5, we achieved an average memory reduction of 1.77×, 1.47×, and 4.55× with FP16, BFLOAT16, and INT8 precision, respectively. Overall, INT8 precision quantization appears to be the optimal choice for small-scale images with a Single-GPU. Further, with an image size of 1024×1024, we observed a 6.29×memory reduction with a speedup improvement of 6.72×.

2.5.3 Distributed DL Quantization Performance Evaluation

In this section, first, we evaluate the performance benefits of using quantization in Distributed DL with respect to memory and throughput. Further, we evaluate benefits of spatial parallelism by enabling inference for very-high resolution images and accelerating performance. We will discuss each of these benefits in specific sections as outlined below.

Memory Evaluation

We profile memory footprints for an image size of 4096×4096 to analyze spatial parallelism with quantization. Figure 2.6 and Figure 2.7 illustrate the memory distribution on different GPUs. We perform an experiment configuration with 4 and 8 spatial parts, as shown in the figures, where one additional GPU is used for layer parallelism. Through quantization, we can halve memory requirements on each GPU compared to the memory required with a full-precision FP32 model. Overall, we achieve a memory reduction of 1.57× with FP16 and 1.40× with BFLOAT16 when compared to the baseline FP32.

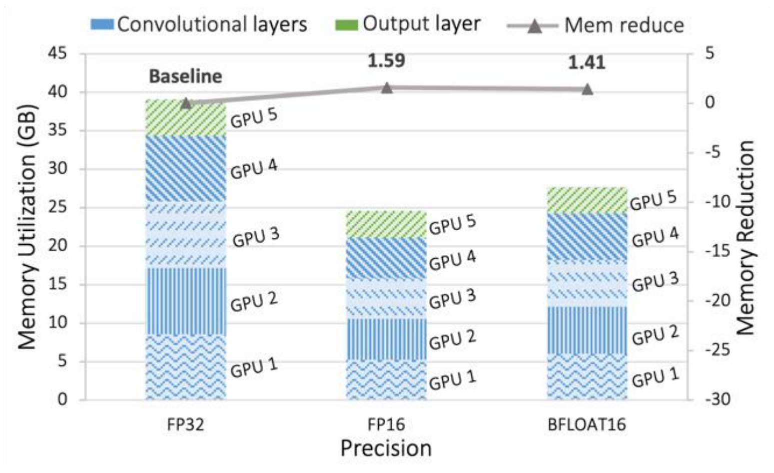


Figure 2.6 Memory footprints on 5 GPUs for SP+LP

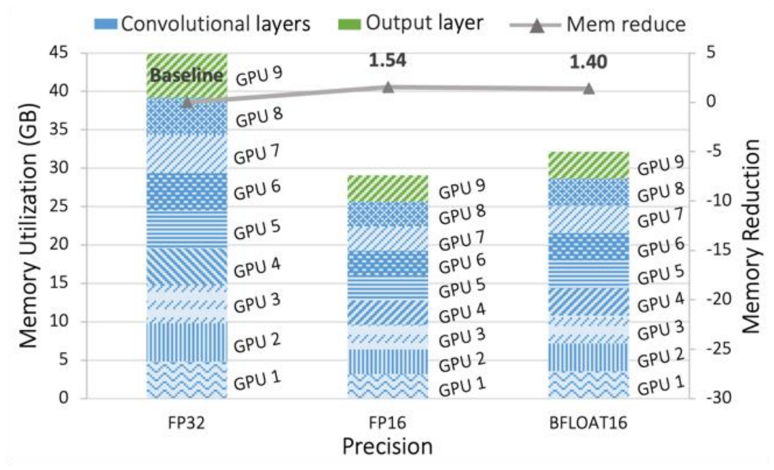


Figure 2.7 Memory footprints on 9 GPUs for SP+LP

Throughput Evaluation

We experimented with image sizes image sizes of 2048×2048 and 4096×4096, employing Spatial and layer Parallelism. We scaled the experiment with the number of spatial parts, ranging from 2, 4, to 8, where each part was distributed on a different GPU. For this experiment, we chose the maximum batch size supported for different GPU counts to utilize memory to its maximum extent. Due to partitioning into more parts across different GPUs, we enabled a higher batch size. For example, with a resolution of 2048×2048, we could not scale beyond a batch size of 16 as it would become out-of-core on a single GPU, but the batch size can be increased when we partition images using SP.

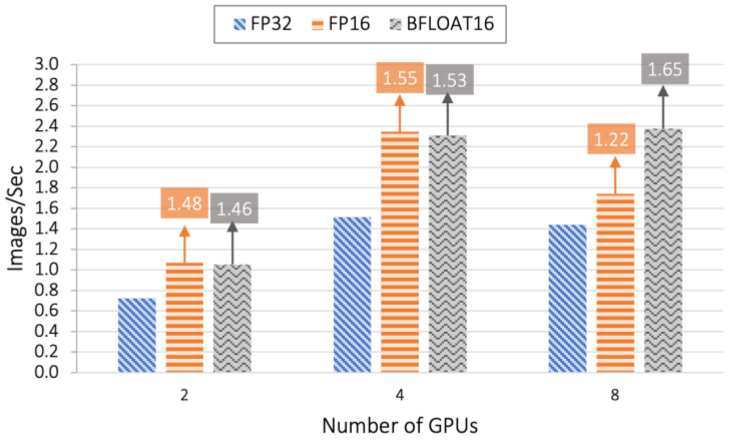


Figure 2.8 Throughput Evaluation for 2048×2048 Image Size

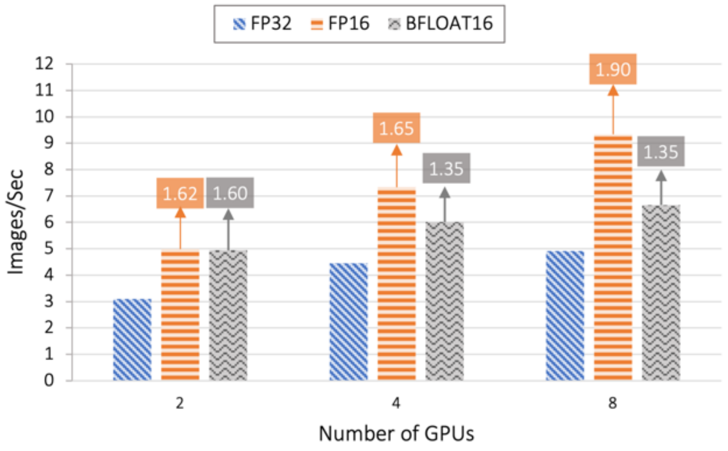


Figure 2.9 Throughput Evaluation for 4096×4096 Image Size

Figure 2.8 shows throughput for 2048×2048 with batch sizes of 16, 32, and 64, on spatial parts 2, 4, and 8, respectively. Similarly, Figure 2.9 shows throughput for 4096×4096 with batch sizes of 16, 32, and 64, on spatial parts 4, 8, and 16, respectively.

We compared the results with the baseline FP32. For 2048×2048, we achieved up to a 1.9x speedup with FP16 and 1.6× with BFLOAT16. For 4096×4096, we achieved up to a 1.55× speedup with FP16 and 1.65× with BFLOAT16.

Enabling very high-resolution images

FP32 precision ResNet101 model with image size 8192×8192, requiring total approximately 87GB memory with FP32, becomes out-of-core even with smallest batch size 1 on single GPU with 40GB memory. It requires to split image into a number of spatial parts to enable inference for 8192×8192. We enabled inference for an image size of 8192×8192 with 4 GPUs for spatial partitioning. Figure 2.10 shows the overview and performance for image size 8192×8192 with 4 and 8 GPUs.

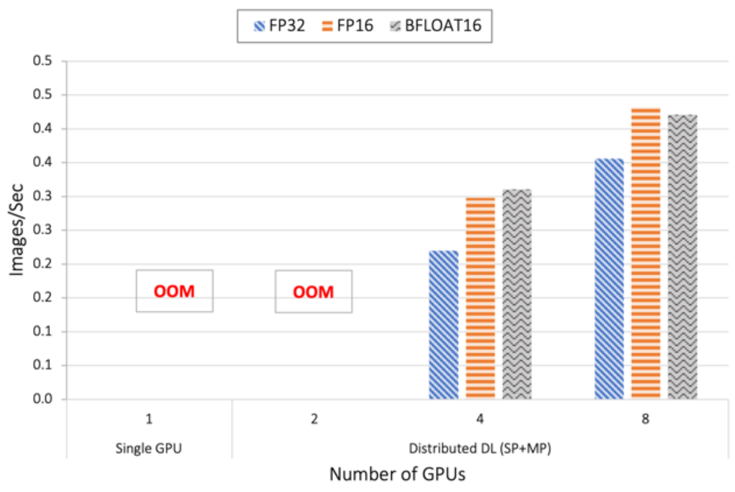


Figure 2.10 Enabling inference for 8192×8192 with FP16

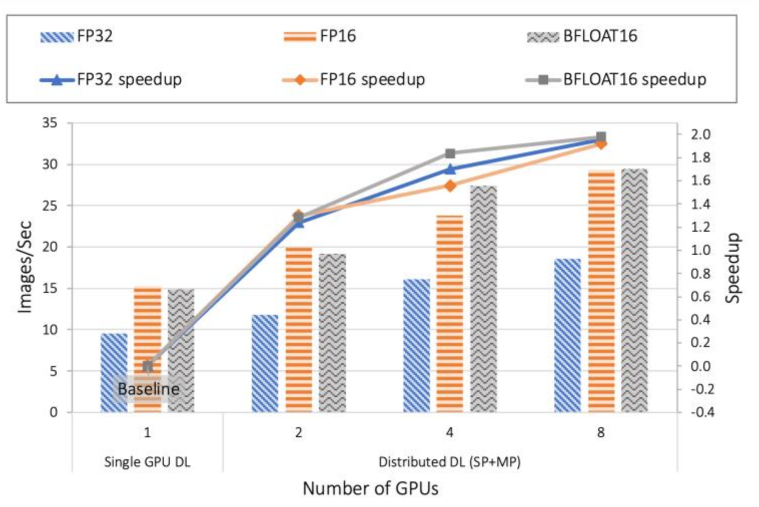


Figure 2.11 Accelerating performance with SP

We further evaluate Spatial Parallelism to accelerate performance while scaling with respect to the number of GPUs. Figure 2.11 shows the performance comparison of SP on different GPUs (2, 4, and 8) with a baseline of a Single GPU. We achieve linear scaling, attaining up to a 1.8× and 2× speedup on 4 and 8 GPUs with BFLOAT16.

2.6 Summary

High-resolution images with Deep Learning come with their own set of challenges due to the large size of the image and deep networked DL models, making it computationally and memory intensive. However, research in high resolution in DL is crucial due to its applicability and efficiency, for example in digital pathology. Our efforts are focused on making trained DL models accessible for high-resolution image inference by reducing computation time and resource requirements.

We proposed accelerated inference for high-resolution images utilizing quantization technique while reducing memory and computation and without accuracy degradation. We provided support for single GPU as well as multi-GPU Distributed DL inference. We achieved overall 6.5x speedup and 4.55x memory reduction with single GPU with INT8 quantization. With Distributed DL, we enabled inference for scaled images. We achieved 1.58x speedup and 1.57x memory reduction using half-precision Distributed DL. We further accelerate performance by 2x using SP compared to single GPU.

We hope that our work will facilitate researchers in achieving accessibility and efficiency in Deep Learning inference while reducing computational costs for their innovative research in the field of high-resolution images.

Chapter 3 MPI4DL: Distributed Deep Learning Parallelism Framework

3.1 Introduction

Deep learning (DL) has emerged as a pivotal tool in advancing vision tasks, offering a multitude of advantages that elevate its importance in this field. Its capability is well proven by its effectiveness in vision tasks such as image classification, object detection, and semantic segmentation, utilized in a wide range of real-world applications such as autonomous vehicles, healthcare imaging, surveillance, and augmented reality. We closely looked at one of such applications in digital pathology in previous Chapter 2.

The architecture of DL models, characterized by deep convolutional layers, makes them memory, and compute intensive. Moreover, considering the large input and batch sizes, processing becomes time-consuming and may even not be able to accommodate in the available GPU memory. To enable training and inference for these models, as well as scale performance, we utilize Distributed Deep Learning with parallelism techniques such as Data Parallelism and Model Parallelism. Furthermore, advanced model parallelism techniques such as GEMS [8] and PipeDream-Flush [30] have been proposed to reduce GPU underutilization. These techniques employ memory-aware designs, which we will discuss in detail in Section 3.4.1. To overcome limitations in model parallelism and enable the training of large input images or larger batch sizes, Spatial Parallelism [4] [5] has been proposed. Furthermore, Hybrid Parallelism leverages the benefits of different parallelism paradigms by integrating them.

However, the aforementioned advanced Model Parallelism, Spatial parallelism, and Hybrid parallelism still remain unavailable, restricting researchers from utilizing this work. Furthermore, it hinders researchers from conducting their own experiments to explore various parallelism techniques and select the parallelism method best suited to their specific requirements and available computing resources. When discussing the benchmarks provided by Deep Learning Parallelism Frameworks such as PyTorch [25] and DeepSpeed [31], it is noted that their support is limited to Data Parallelism, Layer Parallelism, and Pipeline Parallelism, lacking support for Bidirectional Parallelism, Spatial Parallelism, and Hybrid Parallelism.

We understood the requirement of enable and accelerate large DL models and inaccessibility to recently proposed parallelism techniques for enabling and accelerating training. This hinders researchers to utilize proposed parallelism techniques for vision tasks and leverage its benefits. Thus, to overcome this challenge we proposed MPI4DL, Distributed Deep Learning Parallelism Framework. The effectiveness of this work is also evident, with over 755 clones to date. The implementation is available at https://github.com/OSU-Nowlab/MPI4DL.

3.2 Proposed Solution

MPI4DL is a Distributed Deep Learning Parallelism Framework implemented in PyTorch. It consists of GPU-enabled, accelerated, memory-aware parallelism techniques for ResNet and AmoebaNet vision models. The parallelism techniques include Data Parallelism, Layer Parallelism, Pipeline Parallelism, Bidirectional Parallelism (a memory-aware parallelism to reduce underutilization of GPUs), Spatial Parallelism (parallelism technique to enable very large image sizes), and Hybrid Parallelism (integrates different parallelism techniques and leverage their benefits).

3.3 Challenges and Solutions

When it comes to different parallelism techniques, they involve various point-to-point and collective operations, each with its unique communication pattern and workflow. These patterns become even more complex with Bidirectional Parallelism and Spatial Parallelism. For example, consider Figure 3.8 illustrating Bidirectional Parallelism. Models 1 and 2 are distributed across four different GPUs. Model 1 performs pipeline parallelism and uses point-to-point communication consecutively from GPU0 to GPU3, while the point-to-point communication for Model 2 occurs in reverse order, from GPU3 to GPU0. Looking at another example, in Spatial Parallelism, we need to understand the halo exchange performed for convolution operations to obtain adjacent pixels. As shown in Figure 3.1, when we split image into 4 GPUs, GPU 1 will need to perform halo exchange with GPU 2, 3, and 4. When we split image onto 8 GPUs as shown Figure 3.2, in GPU5 needs to communicate with 8 other GPUs (GPU1, 2, 3, 4, 6 ,7, 8,9). Thus, understanding the patterns of these communications is significant for efforts in implementing support.

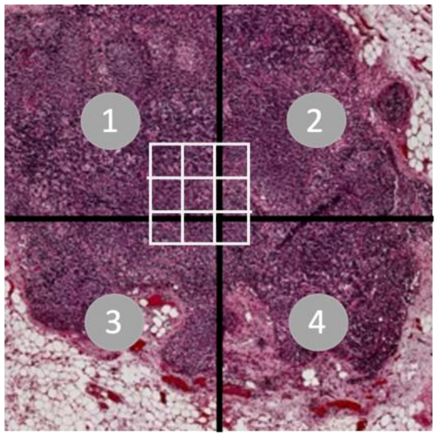


Figure 3.1 Distribution of spatial part on 4 GPUs in SP

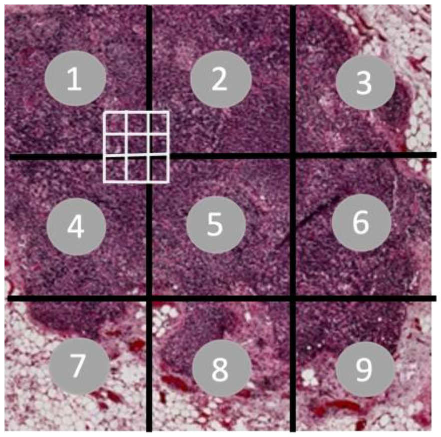
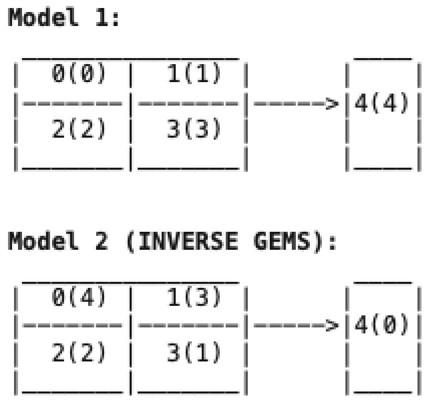


Figure 3.2 Distribution of spatial part on 8 GPUs in SP

In the case of Spatial Parallelism, it was previously implemented for either 4 or 8 spatial parts, with each spatial part assigned to a separate GPU. However, this approach limited scalability. To address this limitation, we provided scalable implementation that supports configurations for any number of spatial parts. Our solution has been evaluated for scalability across a range of 2 to 128 GPUs.

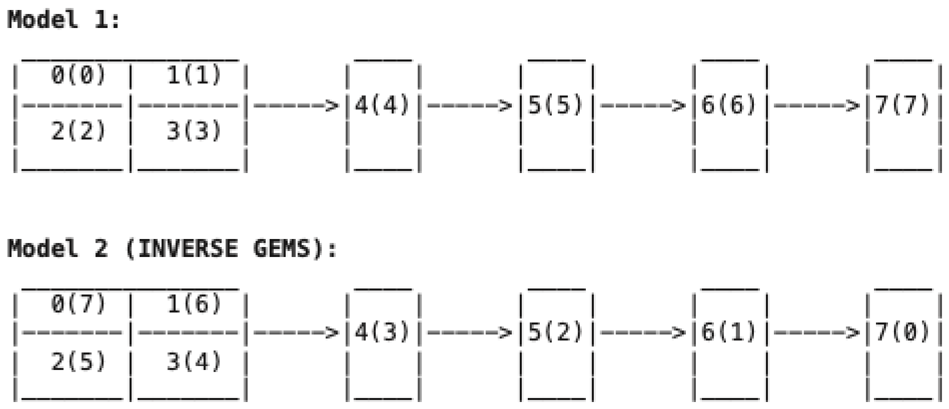
It is also required to meticulously understand different configurations for parallelism degree associated with different parallelism techniques to avoid deadlocks, and runtime errors. For example, for Hybrid Parallelism (Integrating Spatial, Model, Bidirectional Parallelisms), we need to make sure model replicas in Bidirectional Parallelism will not overlap with GPUs associated with Spatial Parallelism. Figure 3.3 and Figure 3.4 shows Hybrid Parallelism with different degrees of parallelism used for Spatial and Model Parallelism. Numbers inside the brackets '()' refer to world rank of GPUs, whereas outside numbers refer to ranks used by model replica. In Figure 3.3, GPUs (1), (2), and (3) are involved in both model replicas of Bidirectional Parallelism, with each model replica also undergoing Spatial Parallelism. This will lead to a deadlock since these GPUs are required to perform halo-exchange as part of spatial parallelism and point-to-point communication as part of bidirectional parallelism. Thus, to avoid a deadlock condition, we ensure that there is overlap between the Spatial Parallelism (SP) of the first model replica and the SP of the second model replica, as shown in Figure 3.4. We recognized these challenges and addressed them by implementing validity checks before initiating parallelism. Furthermore, we provided benchmarks associated with all different parallelism techniques, accompanied by examples demonstrating configurations of varying degrees of parallelism.



```
Model 1:
 _______________          ____
|  0(0) |  1(1) |        |    |
|-------|-------|----->|4(4)|
|  2(2) |  3(3) |        |    |
|_______|_______|        |____|

Model 2 (INVERSE GEMS):
 _______________          ____
|  0(4) |  1(3) |        |    |
|-------|-------|----->|4(0)|
|  2(2) |  3(1) |        |    |
|_______|_______|        |____|
```

Figure 3.3 Hybrid Parallelism with 4 spatial parts and 2 model parts (INVALID configurations for parallelism degree)



```
Model 1:
 _______________        ____        ____        ____        ____
|  0(0)  |  1(1) |      |    |      |    |      |    |      |    |
|--------|-------|----->|4(4)|----->|5(5)|----->|6(6)|----->|7(7)|
|  2(2)  |  3(3) |      |    |      |    |      |    |      |    |
|________|_______|      |____|      |____|      |____|      |____|

Model 2 (INVERSE GEMS):
 _______________        ____        ____        ____        ____
|  0(7)  |  1(6) |      |    |      |    |      |    |      |    |
|--------|-------|----->|4(3)|----->|5(2)|----->|6(1)|----->|7(0)|
|  2(5)  |  3(4) |      |    |      |    |      |    |      |    |
|________|_______|      |____|      |____|      |____|      |____|
```

Figure 3.4 Hybrid Parallelism with 4 spatial parts and 5 model parts
(VALID configurations for parallelism degree)

3.4 Design and Background

In this section we will provide an overview of parallelism techniques and DL models utilized by MPI4DL.

3.4.1 Distributed Parallelism Techniques

Data Parallelism (DP)

Data Parallelism is one of the simplest yet most efficient scalable approaches, where each model is replicated across GPU resources, and batches of data are distributed to each GPU for parallel processing. The gradients are then synchronized using the all-reduce collective operation. However, for large models or inputs of considerable scale, where the model cannot fit into a single GPU memory, Data Parallelism is not applicable.

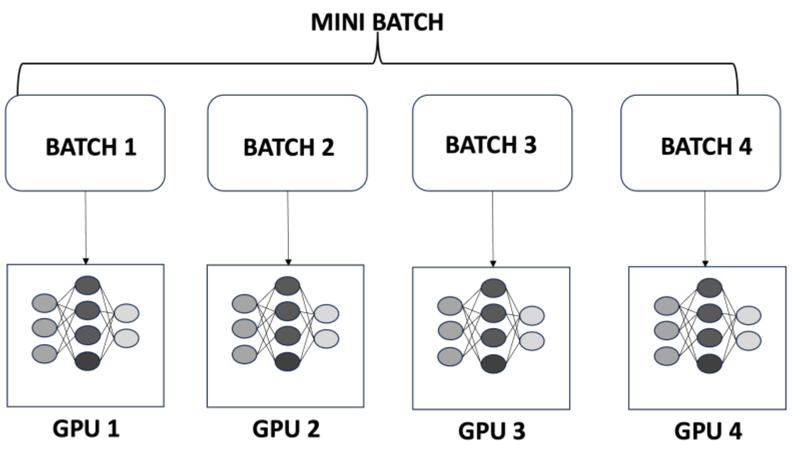


Figure 3.5 Data Parallelism

Model Parallelism (MP)

Model Parallelism overcomes the limitation of Data Parallelism by partitioning the entire model across different GPU resources. Figure 3.6 illustrates Naïve Model Parallelism (also referred to as Layer Parallelism (LP)), where the data dependency between layers results in sequential processing. Consequently, only one GPU operates at any given instance, rendering the remaining GPUs idle. This inefficiency leads to the underutilization of GPU compute and memory resources.

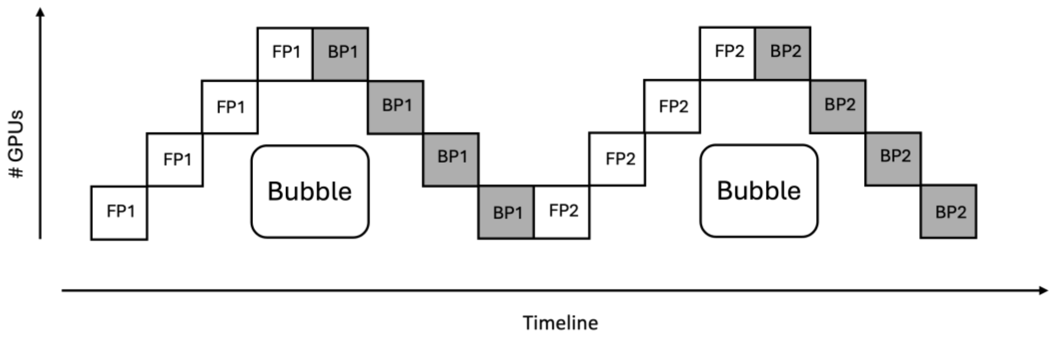


Figure 3.6 Layer Parallelism

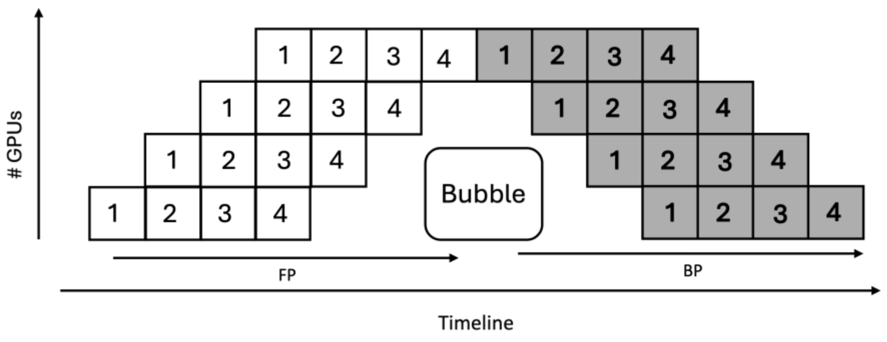
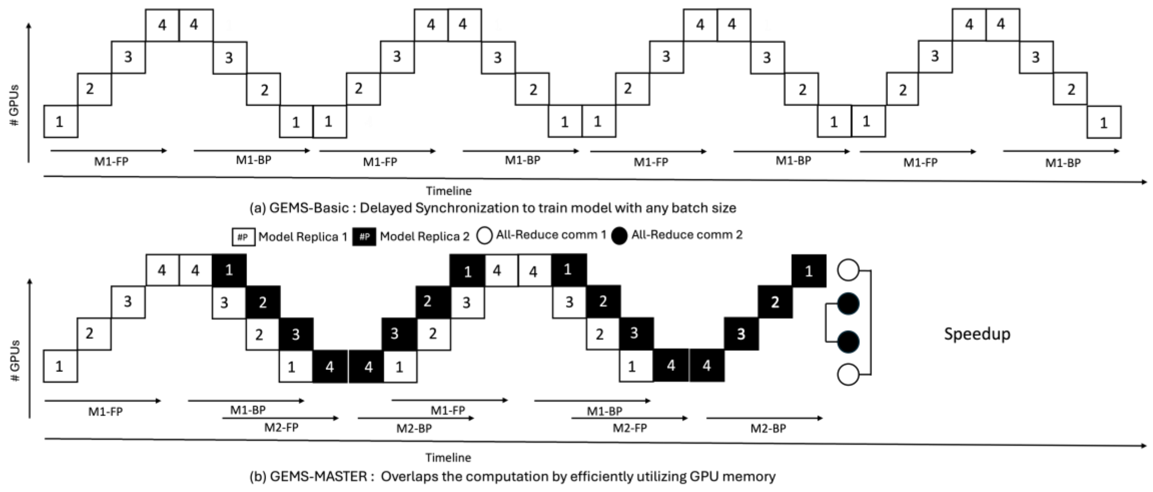


Figure 3.7 Pipeline Parallelism

Pipeline Parallelism (PP) overcomes the sequential execution inherent in Layer Parallelism by dividing the batch size into micro-batch sizes and executing each micro-batch in a pipeline manner. Figure 3.7 illustrates the execution of Pipeline Parallelism across 4 GPU nodes. However, as shown in Figure 3.7, Pipeline Parallelism still encounters bubbles, where GPUs are not utilized.

Bidirectional Parallelism

Bidirectional Parallelism aims to decrease the bubble encountered in Model Parallelism by enabling model replicas to run in parallel. As shown in Figure 3.8, model replica 2 begins the forward pass once the last model part of model replica 1 completes the backward pass. This utilization of GPU resources by model replicas was ideal during the bubble region in the previously observed model parallelism. The gradients will be synchronized using point-to-point operations once both model replicas complete the execution of the batch size. The effective batch size equals the number of model replicas batch size equals the number of model replicas multiplied by the batch size for each model replica.



# GPUs
M1-FP
M1-BP
Timeline
(a) GEMS-Basic : Delayed Synchronization to train model with any batch size
Model Replica 1
Model Replica 2
All-Reduce comm 1
All-Reduce comm 2
M2-FP
M2-BP
Speedup
(b) GEMS-MASTER : Overlaps the computation by efficiently utilizing GPU memory

Figure 3.8 Bidirectional Parallelism (GEMS) [8]

Spatial Parallelism (SP)

Spatial Parallelism solves the problem where the large input image size and given deep convolutional layer makes even single model layer not fit into GPU memory. For example, image size of 8192x8192 with ResNet101 model and batch size as one becomes out-of-core model on NVIDIA-A100-40GB GPU. In Spatial Parallelism, the whole image is partitioned into smaller non-overlapping spatial parts and distributed across different GPUs. Further, convolution and pooling layers of the DL model are replicated on GPUs containing spatial parts and lastly, output layer will be replicated on single GPU undergoing LP.

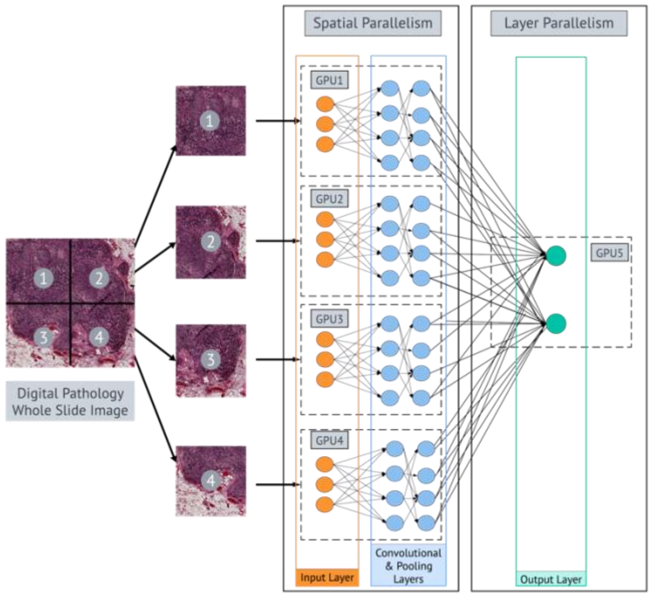


Figure 3.9 Spatial Parallelism

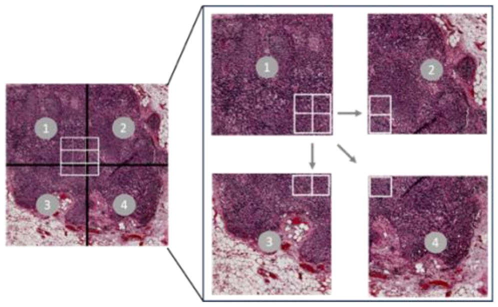


Figure 3.10 Halo Exchange

Figure 3.9 shows overview of Spatial and Layer Parallelism. The digital pathology image is partitioned into 4 spatial parts, and the model is split into 2 parts. The first model split consists of compute and memory-intensive convolution and pooling layers, while the second and final model partitions contain the output layer. Each spatial part performs convolution and pooling operations given by first model split, and finally, the outputs are aggregated by second model split.

In SP, Halo-Exchange Communication is required for convolution and pooling operations to receive adjacent pixels. For pixels located on the boundaries of the spatial segment, their adjacent pixels will be on different GPUs. Figure 3.10 illustrates the halo-exchange required by the first spatial part with different GPUs to obtain adjacent pixels.

Hybrid Parallelism

Hybrid Parallelism leverages the efficiency of each of the Parallelisms. For example, feature of enabling training for scaled images using spatial parallelism, will be integrated with bidirectional parallelism, which provides accelerated and memory efficient parallelism, Further, Data parallelism can also be integrated to scale and accelerate training. Figure 3.11 shows Hybrid Parallelism integrating different parallelisms techniques, namely Data Parallelism, Layer Parallelism, Pipeline Parallelism, Bidirectional Parallelism and Spatial Parallelism.

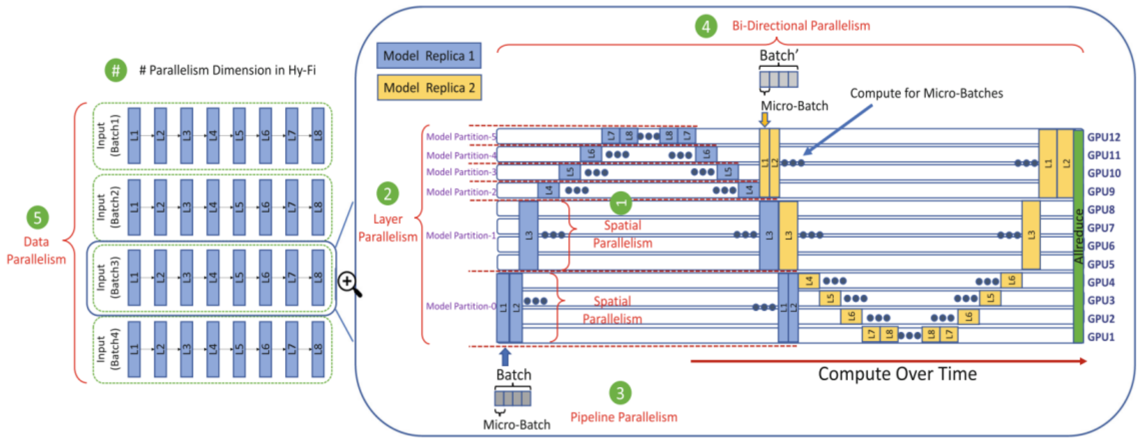


Figure 3.11 Hybrid Parallelism [4]

3.4.2 Support for Deep Learning Models

MPI4DL utilizes ResNet [14] and AmoebaNet-D [16]; Both ResNet and AmoebaNet have excelled in various computer vision tasks, particularly image classification, where they have achieved state-of-the-art performance. ResNet revolutionized deep learning architecture with its pioneering use of residual blocks, featuring shortcut connections to alleviate the vanishing gradient problem in deep networks. By facilitating direct gradient flow, ResNet achieves state-of-the-art performance in tasks like image classification. AmoebaNet-D is the evolution of the original AmoebaNet, developed by researchers at Google Brain. AmoebaNet incorporates a technique of automatically discovering optimal network architectures for a given task. It involves searching through a large space of possible network architectures to find the most effective ones. AmoebaNet-D was obtained by manually extrapolating the evolutionary process and optimizing the resulting architecture for training speed.

3.5 Performance Evaluation

For all experiments, the following hardware and software setups were utilized.

Hardware Specifications

NVIDIA A100-40 GB GPUs (2 GPUs per node) with AMD EPYC 7713 64-Core Processor. GPU nodes are connected via Mellanox HDR (200 Gbps) InfiniBand.

Software Specifications

PyTorch v1.13.1 [25] is used as DL framework. For the distributed communication backend, we used MVAPICH-GDR 2.3.7 [10].

3.5.1 Enabling training for scaled Images and accelerating training with SP

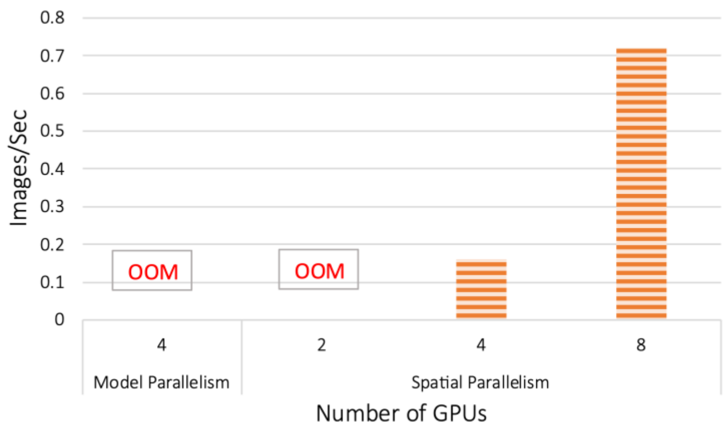


Figure 3.12 Enabling and Accelerating DL training using Spatial Parallelism for 2048 × 2048 Image Size with ResNet101 Model

In this experiment, we evaluated ResNet101 with an image size of 2048 × 2048 using Model Parallelism and Spatial Parallelism. In the case of Model Parallelism, ResNet101 was partitioned across 4 GPUs, resulting in a very large batch size to accommodate GPU memory. When Spatial Parallelism is used, we enable training by splitting spatial parts across 4 GPUs. Figure 3.12 also demonstrates the performance scalability with Spatial Parallelism as it scales with the number of GPUs.

3.5.2 Model Parallelism and Spatial Parallelism

Figure 3.13 and Figure 3.14 show the performance comparison of Model and Spatial Parallelism using the same computer resources of 5 GPUs. In the case of Model Parallelism, we partition the model into 5 parts across 5 GPUs. For a batch size of 2, we utilize Pipeline Parallelism. For Spatial Parallelism, we partition spatial parts on 4 GPUs, and one GPU will be utilized for the last output layer (fully connected and SoftMax layer as shown in Figure 3.9).

For both experimented image sizes, Spatial Parallelism outperforms Model Parallelism. Specifically, it provides 1.5× performance improvement with 1024 × 1024 (Batch Size = 2) and 2.4× performance improvement with 2048×2048 (Batch Size = 2).

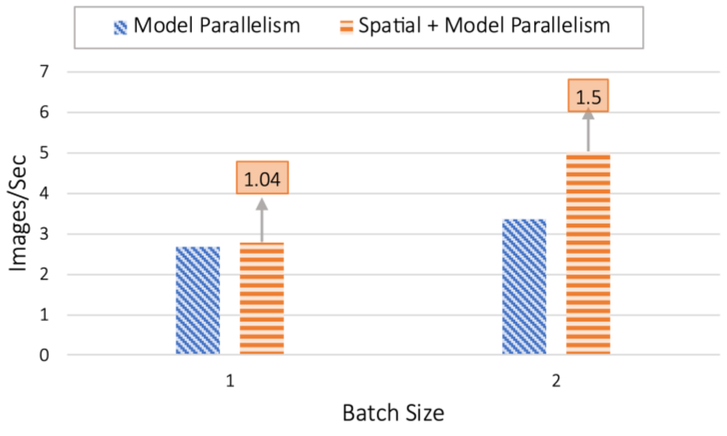


Figure 3.13 Performance Evaluation of Pipeline Parallelism and Spatial Parallelism for AmoebaNet Model with 1024 × 1024 image size

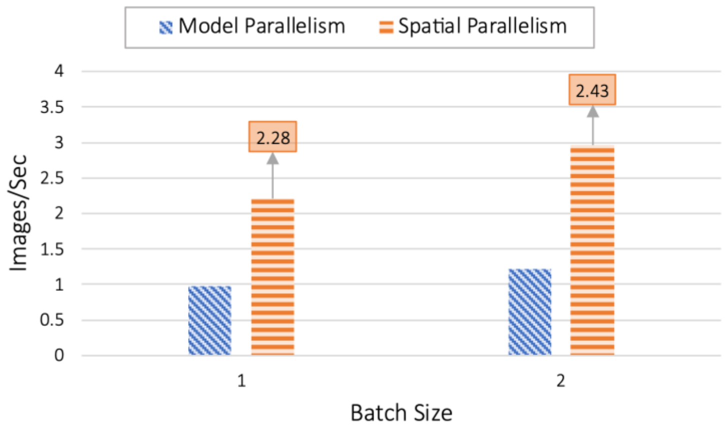


Figure 3.14 Performance Evaluation of Pipeline Parallelism and Spatial Parallelism for AmoebaNet Model with 2048×2048 Image Size

3.5.3 Hybrid Parallelism: Spatial + Bidirectional Parallelism (GEMS)

Next, we evaluate Hybrid Parallelism which integrates Spatial, Bidirectional, and Model Parallelism are used with two model replicas. We conducted experiments for AmoebaNet and ResNet with a 1024 × 1024 image size. When spatial parallelism is integrated with Bidirectional Parallelism, it provides up to a 2.5× improvement for AmoebaNet with a batch size of 4 and performs 2× better for ResNet with a Batch Size of 4. Figure 3.15 and Figure 3.16 show the performance evaluation for Spatial Parallelism and Hybrid Parallelism.

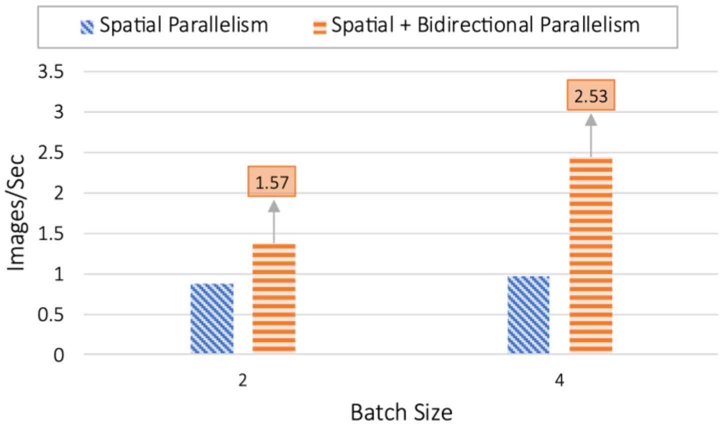


Figure 3.15 Performance Evaluation of Spatial Parallelism and Bidirectional Parallelism for AmoebaNet Model with 1024 ×1024 Image Size

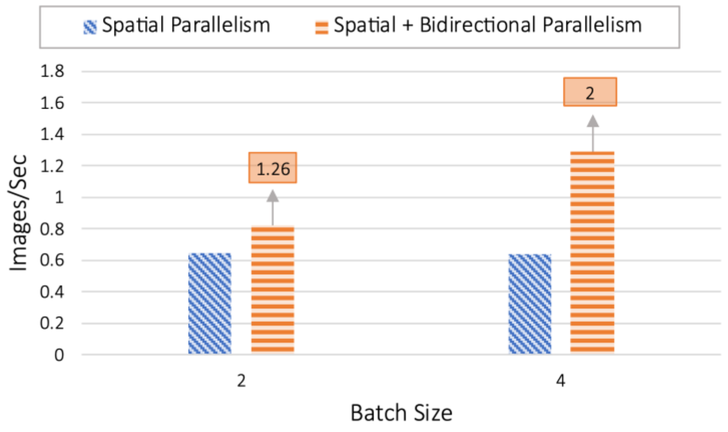


Figure 3.16 Performance Evaluation of Spatial Parallelism and Bidirectional for ResNet101 Model with 1024 × 1024 Image Size

3.6 Summary

Deep Learning has been widely used in vision tasks due to its remarkable capabilities. To accommodate large models, input size, and enhance performance, various parallelism techniques are utilized. Prominent research has been conducted in this area, leading to the existence of different parallelism techniques. However, not all of these parallelism techniques are yet available or supported by any Deep Learning Parallelism framework. Speaking about the benchmarks provided by PyTorch and DeepSpeed, their support is limited to Data Parallelism, Layer Parallelism, and Pipeline Parallelism, but does not include support for Bidirectional Parallelism, Spatial Parallelism, and Hybrid Parallelism.

Thus, we proposed MPI4DL, Distributed Deep Learning Parallelism Framework. As of today, MPI4DL, is the first open-source work which provides all the state-of-the-art GPU-Enabled, memory-efficient, accelerated parallelisms techniques at single place. It includes ResNet and AmoebaNet model support with parallelism, namely Data Parallelism, Layer Parallelism, Pipeline Parallelism, Bidirectional Parallelism, Hybrid Parallelism, Spatial Parallelism. The success of this work is also evident today, it has 755 clones. The implementation is made available at https://github.com/OSU-Nowlab/MPI4DL. We aim for our work to assist researchers in utilizing cutting-edge parallelism techniques in Deep Learning vision tasks, speeding up training and lowering computational expenses for their research.

Chapter 4 Leveraging MCR-DL: Mix-and-Match Communication Runtime for Deep Learning

4.1 Introduction

In the last two chapters, 0 and Chapter 3, we focused on accelerating training and inference by quantization and exploiting various parallelism techniques. However, it is essential to note that communication in Distributed DL significantly influences performance. Distributed DL uses Point-to-Point communication and various collective communications depending on the specific parallelism approach used. For instance, Tensor Parallelism utilizes All-Reduce, Sequence Parallelism employs Reduce-Scatter and All-Gather, Pipeline Parallelism relies on Point-to-Point communication, and Data Parallelism utilizes All-Reduce. Furthermore, these communications also vary in terms of message sizes. Existing Distributed DL frameworks lack support for all communication operations and backends, and they do not support mixed-backend communication. MCR-DL [12] addresses this limitation by providing support for all communication operations. Additionally, for performance optimization, it facilitates Mixed-Backend communication and functionality for dynamically selecting the optimal backend based on the message size and collective operation requirements.

Furthermore, rapid advancements in the MPI world, such as MVAPICH-PLUS, MVAPICH-GDR [9] [10], and other MPI libraries, along with NCCL [11], make it essential to leverage their associated features in Distributed DL. However, the distributed module provided by DL frameworks does not cover all collective operations and often requires source building to support backends like MPI. MCR-DL simplifies this requirement by decoupling communication backends from PyTorch's distributed module.

Understanding the significance of communication optimization in Distributed DL, we recognize the benefits of MCR-DL, such as providing mixed-backend support, dynamically selecting the optimal backend, and enabling easy integration of MPI backend for Distributed DL frameworks. Therefore, we are initiating efforts to make it available. The efforts to provide the open-source version of MCR-DL are in progress. Currently, version 0.1 of MCR-DL has been open-sourced and is accessible at https://github.com/OSU-Nowlab/MCR-DL.

4.2 Design

MCR-DL is a distributed module for DL frameworks with its mixed-backend communication support, dynamic selection of the optimal backend for a given collective based on the message size at runtime, and communication optimization enhancements such as compression and tensor fusion.

MCR-DL is implemented as a C++ backbone beneath a thin Python interface, where importantly all the collectives are written in C++. The Python wrapper makes it accessible to all existing DL Distributed Frameworks, while also adding support for mixed communication backends and dynamically selecting the optimal backend for communication operations.

4.2.1 API Usage:

MCR-DL offers easy access and seamless integration with DL frameworks, requiring minimal code changes and programming efforts. Below is an example comparing *torch.distributed* and *mcr_dl* as Distributed Modules, highlighting the minimal effort required to integrate MCR-DL.

In Code 1, *torch.distributed* is used as a distributed module with 'nccl' as the communication backend, while in Example 2, *mcr_dl* is used as a distributed module with 'nccl' as the communication backend. The code changes are highlighted in both examples. Additionally, mcr_dl can be further configured with different backends by passing a list of user-available multiple backends, such as ['nccl', 'mpi']. These backends will later be used by mcr_dl to dynamically select the optimal backend for a given collective and message size, thereby improving performance.

Example 1: Converting PyTorch Distributed Framework code to use MCR-DL

Code 1: PyTorch with *torch.distributed* as a Distributed Module:

```
import torch
import torch.distributed as dist


def all_reduce_bench():
    x = torch.ones(1, 3).to("cuda") * (dist.get_rank() + 1)
    sum_of_ranks = (dist.get_world_size() * (dist.get_world_size() + 1)) // 2
    result = torch.ones(1, 3).to("cuda") * sum_of_ranks
    dist.all_reduce(x)
    assert torch.all(x == result)

def init_process(backend="nccl"):
    # initialization necessary environment variables
    ....
    dist.init_process_group(backend)

if __name__ == "__main__":
    init_process(backend="nccl")
    all_reduce_bench()

```

Code 2: PyTorch with *mcr_dl* as a Distributed Module:

```
import torch
import mcr_dl as dist


def all_reduce_bench():
    x = torch.ones(1, 3).to("cuda") * (dist.get_rank() + 1)
    sum_of_ranks = (dist.get_world_size() * (dist.get_world_size() + 1)) // 2
    result = torch.ones(1, 3).to("cuda") * sum_of_ranks
    dist.all_reduce(x)
    assert torch.all(x == result)

def init_process(backend="nccl"):
    # initialization necessary environment variables
    ....
    dist.init_distributed(backend)

if __name__ == "__main__":
    init_process(backend="nccl")
```

4.2.2 Mixed-Backend Communication

We can initialize multiple communication backends through mcr_dl and later leverage these communication backends for mixing communication, as demonstrated in Example 2. To prevent deadlocks and ensure synchronization, each operation by independent communication backends needs to be synchronized, as shown in lines 15 and 16.

Example 2: Mixed-Backend Communication with *mcr_dl*

```
1.  import torch
2.  import mcr_dl as dist
3.
4.  def tensor():
5.      return torch.rand(1,1)
6.
7.  x = tensor().cuda()
8.  y = tensor().cuda()
9.  z = tensor().cuda()
10. dist.init_distributed(['nccl', 'mpi'])
11.
12. h1 = dist.all_reduce('nccl', x, async_op=True)
13. h2 = dist.all_reduce('mpi', y, async_op=True)
14. z = z + z
15. h1.wait()
16. h2.wait()
17. result = x + y + z
```

4.3 Performance Evaluation

For all experiments, the following hardware and software setups were utilized.

Hardware Specifications:

NVIDIA A100-40 GB GPUs (2 GPUs per node) with AMD EPYC 7713 64-Core Processor. GPU nodes are connected via Mellanox HDR (200 Gbps) InfiniBand.

Software Specifications

PyTorch v2.0.1 [25] is used as DL framework. For the distributed communication backend, we used NCCL [11] and MVAPICH-GDR 2.3.7 [10].

The following performance evaluations demonstrate the need for MCR-DL, specifically in terms of dynamically selecting the backend for different communication operations and message sizes. However, as the support of MCR-DL in Megatron-LM is experimental and currently in progress, application-level results are yet to be collected.

Experiment Configuration Description

The following experiments use the configurations 'torch nccl', 'torch mpi', 'pure nccl', and 'pure mpi'. In this section, we provide explanations for each term to better understand the performance results.

‘torch nccl’ and ‘torch mpi’:

PyTorch comes with its own distributed module, which can be initialized with a given communication backend, such as 'nccl' and 'mpi'. When we use PyTorch's distributed module and initialize it with the 'nccl' and 'mpi' backends, we refer to them as 'torch nccl' and 'torch mpi', respectively.

‘pure nccl’ and ‘pure mpi’:

MCR-DL distributed module comes with its own implementation for using 'nccl' and 'mpi' as backend communication, instead of relying on PyTorch's distributed module for collective operations. When we utilize the MCR-DL distributed module with the 'nccl' and 'mpi' backend implementations provided by MCR-DL, we refer to them as 'pure nccl' and 'pure mpi', respectively

4.3.1 Performance Analysis with Different Configurations:

Figure 4.1 and Figure 4.2 show the experiments using different configuration setups for All-Reduce and All-to-all collective operations. As can be observed, 'pure nccl' performs best for smaller message sizes (from 16B to 8MB), whereas 'torch nccl' provides better performance for message sizes above 8MB. In terms of the All-to-all benchmark, 'pure nccl' performs consistently across all message ranges, whereas 'torch nccl' performs significantly better for large message sizes (from 1MB to 2GB).

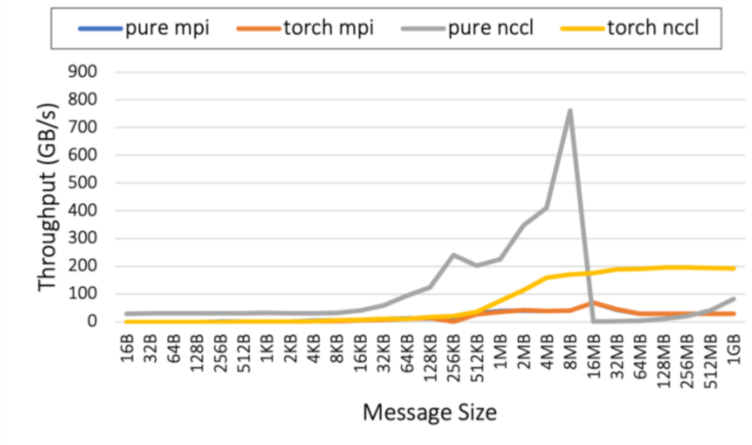


Figure 4.1 All-Reduce Performance with 2 GPUs using different configurations

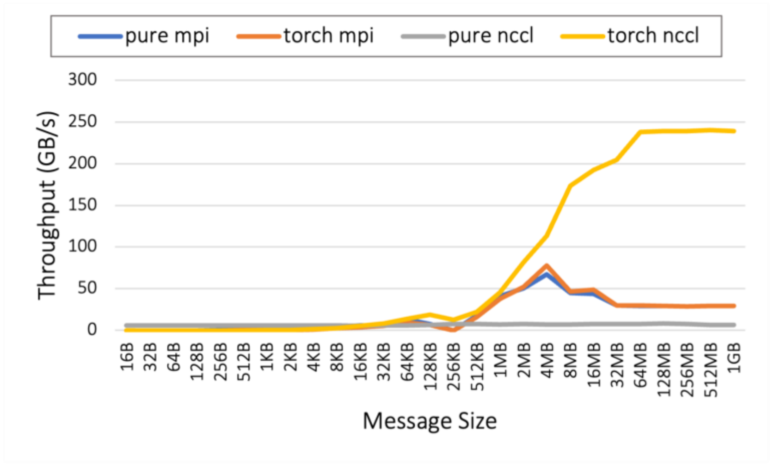


Figure 4.2 All-to-all Performance with 2 GPUs using different configurations

4.3.2 PyTorch's Distributed Vs 'pure nccl'/'pure mpi' via MCR-DL:

Figure 4.3 shows the performance results for the All-Gather collective operation with 'pure nccl' and 'torch nccl' on a 2-GPU setup. For large message sizes, 'pure nccl' performs significantly better than 'torch nccl'. However, when compared to the All-to-all benchmark as shown in Figure 4.4, 'torch nccl' performs better than 'pure nccl'.

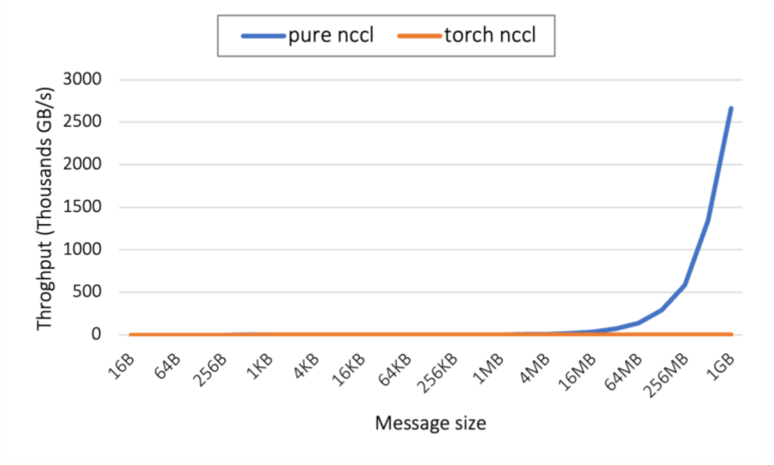


Figure 4.3 All-Gather Performance with 2 GPUs using different distributed modules

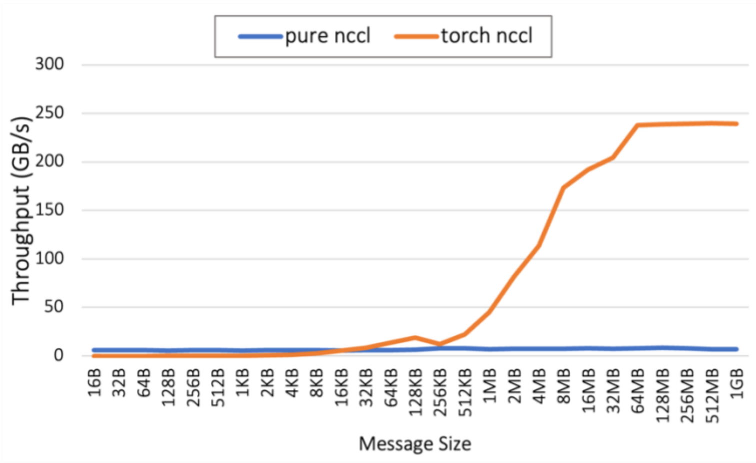


Figure 4.4 All-to-all Performance with 2 GPUs using different distributed modules

Conclusively, as demonstrated in section 4.3.1 and 4.3.2, different collectives with different message sizes exhibit different performance and MCR-DL overcomes this limitation by dynamically choosing the best configuration for optimal performance.

4.4 Summary

In Distributed DL, communication plays a vital role in influencing performance, with various parallelisms employing different communication methods. Depending on the type of communication and message sizes, different communication backends offer varying performance advantages and disadvantages. Furthermore, existing DL frameworks often lack comprehensive support for communication operations and require intricate source building for backend support like MPI. To address these deficiencies, MCR-DL emerges as a solution, providing support for all communication operations, facilitating mixed-backend communication, and dynamically selecting the most optimal backend based on specific needs. This further simplifies the utilization of any backend without dependence on the distributed module of a particular DL framework. Recognizing the benefits of MCR-

DL we initiated efforts to make it available, and its success is evident by having more than 250 downloads. These initiatives aim to help researchers by providing communication optimization for their DL applications across different Distributed DL frameworks.

Chapter 5 Conclusion and Future Work

In this thesis, we present quantization techniques, parallelism strategies, and communication optimization techniques aimed at improving the performance of Distributed Deep Learning applications. Firstly, we focused on inference leveraging quantization to reduce memory footprint and accelerate inference without accuracy degradation. We introduced support for up to INT8 quantization for single GPU setups and half-precision quantization in Distributed DL settings for scaled image inference. Secondly, we explored various parallelisms, their benefits, and drawbacks. We incorporated these designs as part of MPI4DL to leverage accelerated and memory-efficient training, further enabling scaled image training. Lastly, we looked into a novel approach of mix-and-match communication to improve the performance by mixing communication and dynamically selecting communication backend to achieve optimal performance and overall optimize Distributed DL communication.

In Chapter 2, we examined quantization, achieving significant performance improvements with int8 quantization while experiencing less than 1% accuracy degradation for inference. Recent research [32] [33] has also explored INT4 quantization on GPUs for transformer models, which led to a curiosity about potential of INT4 quantization for vision models. Furthermore, extending INT4 and INT8 for Distributed DL could have a substantial impact. At present, we use TensorRT for INT8 quantization, which also restricts support for INT8 quantization in Distributed DL. Additionally, supporting mixed precision training in MPI4DL could accelerate performance. Given that MCR-DL can be integrated with any Distributed DL application, MPI4DL stands as an example of an application that could further benefit from MCR-DL for communication optimization.

Bibliography

[1] R. Feng, X. Liu, J. Chen, D. Z. Chen, H. Gao and J. Wu, "A deep learning approach for colonoscopy pathology WSI analysis: accurate segmentation and classification," in *IEEE Journal of Biomedical and Health Informatics*, 2020.

[2] M. Soori, B. Arezoo and R. Dastres., "Artificial intelligence, machine learning and deep learning in advanced robotics, a review," *Cognitive Robotics,* 2023.

[3] B. Neupane, T. Horanont and J. Aryal, "Deep learning-based semantic segmentation of urban features in satellite images: A review and meta-analysis," *Remote Sensing,* vol. 13, 2021.

[4] A. Jain, A. Shafi, Q. Anthony, P. Kousha, H. Subramoni and a. D. K. Panda, "Hy-Fi: Hy brid Fi ve-Dimensional Parallel DNN Training on High-Performance GPU Clusters," in *International Conference on High Performance Computing*, 2022.

[5] X. Sheng, L. Li, D. Liu and H. Li, "Spatial Decomposition and Temporal Fusion based Inter Prediction for Learned Video Compression," in *EEE Transactions on Circuits and Systems for Video Technology*, 2024.

[6] W. Li, M. Mikailov and W. Chen, "Scaling the inference of digital pathology deep learning models using cpu-based high-performance computing," in *IEEE Transactions on Artificial Intelligence*, 2023.

[7] A. Pedersen, M. Valla, A. M. Bofin, J. P. D. Frutos, I. Reinertsen and E. Smistad, "FastPathology: An open-source platform for deep learning-based research and decision support in digital pathology," *IEEE Access,* vol. 9, 2021.

[8] A. Jain, "Gems: Gpu-enabled memory-aware model-parallelism system for distributed dnn training," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis,* 2020.

[9] D. K. Panda, H. Subramoni, C.-H. Chu and M. Bayatpour, "The MVAPICH project: Transforming research into high-performance MPI library for HPC community," *Journal of Computational Science,* vol. 52, 2021.

[10] "MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, RoCE, and Slingshot," [Online]. Available: https://mvapich.cse.ohio-state.edu/userguide/gdr/. [Accessed 15 April 2024].

[11] "NVIDIA Collective Communications Library (NCCL)," [Online]. Available: https://developer.nvidia.com/nccl. [Accessed 15 April 2024].

[12] Q. Anthony, A. A. Awan, J. Rasley, Y. He, A. Shafi, M. Abduljabbar, H. Subramoni and D. Panda, "Mcr-dl: Mix-and-match communication runtime for deep learning," in *EEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023.

[13] "Camelyon 2016," 2016. [Online]. Available: https://camelyon16.grand-challenge.org/. [Accessed 15 April 2024].

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

[15] O. Ronneberger, P. Fischer and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference*, Munich, Germany, 2015.

[16] E. Real, A. Aggarwal, Y. Huang and Q. V. Le, "Regularized evolution for image classifier architecture search," in *Proceedings of the aaai conference on artificial intelligence*, 2019.

[17] O. F. K. K. K. M. R. Iizuka, K. Arihiro and M. Tsuneki, "Deep learning models for histopathological classification of gastric and colonic epithelial tumours," *Scientific reports*, vol. 10, 2020.

[18] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney and K. Keutzer, "A survey of quantization methods for efficient neural network inference," in *Low-Power Computer Vision*, 2022.

[19] H. Wu, P. Judd, X. Zhang, M. Isaev and P. Micikevicius, "Integer quantization for deep learning inference: Principles and empirical evaluation," 2020.

[20] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal and a. J. Dean, "Efficiently scaling transformer inference," in *Machine Learning and Systems*, 2023.

[21] G. J. L. M. S. Xiao, H. Wu, J. Demouth and S. Han, "Smoothquant: Accurate and efficient post-training quantization for large language models," in *International Conference on Machine Learning*, 2023.

[22] J. Braatz, P. Rajpurkar, S. Zhang, A. Y. Ng and J. Shen, "Deep learning-based sparse whole-slide image analysis for the diagnosis of gastric intestinal metaplasia," 2022.

[23] J. R. Kaczmarzyk, A. O'Callaghan, F. Inglis, S. Gat, T. Kurc, R. Gupta, E. Bremer, P. Bankhead and J. H. Saltz, "Open and reusable deep learning for pathology with WSInfer and QuPath," *NPJ Precision Oncology*, vol. 8, 2024.

[24] H. P. J. Wu, X. Zhang, M. Isaev and P. Micikevicius, "Integer quantization for deep learning inference: Principles and empirical evaluation," 2020.

[25] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen and e. al, "Pytorch: An imperative style, high-performance deep learning library.," in *Advances in neural information processing systems 32*, 2019.

[26] "NVIDIA TensorRT," 2019. [Online]. Available: https://developer.nvidia.com/tensorrt/. [Accessed 15 April 2024].

[27] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *IEEE conference on computer vision and pattern recognition*, 2009.

[28] "The CIFAR-10 Dataset," 2014. [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html. [Accessed 15 April 2024].

[29] "GitHub - fastai/imagenette: A smaller subset of 10 easily classified classes from Imagenet, and a little more French," [Online]. Available: https://github.com/fastai/imagenette. [Accessed 15 April 2024].

[30] D. Narayanan, A. Phanishayee, K. Shi, X. Chen and M. Zaharia, "Memory-efficient pipeline-parallel dnn training," in *International Conference on Machine Learning*, 2021.

[31] J. Rasley, S. Rajbhandari, O. Ruwase and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020.

[32] X. Wu, C. Li, R. Y. Aminabadi, Z. Yao and Y. He, "Understanding int4 quantization for language models: latency speedup, composability, and failure cases," in *International Conference on Machine Learning*, 2023.

[33] H. Xi, C. Li, J. Chen and J. Zhu, "Training transformers with 4-bit integers," in *Advances in Neural Information Processing Systems*, 2023.