Characterizing and Accelerating Deep Learning and Stream Processing Workloads using Roofline Trajectories

Thesis

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

By

Muhammad Haseeb Javed, B.E.

Graduate Program in Computer Science and Engineering

The Ohio State University

2019

Master's Examination Committee:

Dr. Xiaoyi Lu, Advisor

Dr. Gagan Agarwal

Abstract

Over the last decade, technologies derived from convolutional neural networks (CNNs) called Deep Learning applications, have revolutionized fields as diverse as cancer detection, self-driving cars, virtual assistants, etc. On the other hand, organizations have become heavily reliant on providing near-instantaneous insights to the end-user based on vast amounts of data collected from various sources in real-time. The most common method of increasing the processing capability of these applications is to execute them in a distributed manner over in large clusters. However, users of such applications are typically not experts in the nuances of distributed systems and find the already challenging task of scaling these applications quite difficult. Consequently, there is limited knowledge among the community to run such applications in an optimized manner. The performance question for these software stacks has typically been addressed by employing bespoke hardware better suited for such compute-intensive operations. However, such a degree of performance is only accessibly at increasingly high financial costs leaving only big corporations and government with resources sufficient enough to employ them at a large scale. For such users to make effective use of resources at their disposal, concerted efforts are necessary to figure out optimal hardware and software configurations. This study is one such step in this direction as we use the Roofline model to perform a systematic analysis of representative Deep Learning models and identify opportunities for black-box/application-aware optimizations. We also use the Roofline model to guide the architectural enhancements needed to design

accelerated Message Brokers, called Frieda, necessary for optimized stream processing pipelines. Using the findings from our study, we are able to obtain up to 3.5X speedup compared to vanilla TensorFlow with default configurations. Moreover, compared with Kafka, Frieda exhibits a reduction of up to 98% in 99.9^{th} percentile latency for micro-benchmarks and up to 31% for full-fledged stream processing pipeline constructed using Yahoo! Streaming Benchmark.

This is dedicated to the people who wait; good things are on their way.

Acknowledgments

My deepest gratitude to my advisor Dr. Xiaoyi Lu. His technical guidance and encouragement throughout my time as a graduate student has been invaluable to me.

Special thanks to my parents. Whatever I am is because of them.

My friends also deserve a mention, they mean more to me than I might lead them to believe.

Vita

2011 - 2015 ..... B.E., Software Engineering, NUST, Pakistan.

2016 - Present ..... M.S., Computer Science and Engineering, The Ohio State University, USA.

2016 - Present ..... Graduate Research/Teaching Assistant, The Ohio State University, USA.

Publications

Research Publications

M. H. Javed, K. Ibrahim, and X. Lu "Performance Analysis of Deep Learning Workloads Using Roofline Trajectories". In The Journal of CCF Transactions on High Performance Computing (THPC'19), Dec. 2019.

M. H. Javed, X. Lu, and D. K. Panda "Cutting the tail: designing high performance message brokers to reduce tail latencies in stream processing". Proceedings of the 2018 IEEE International Conference on Cluster Computing (CLUSTER'18), Sep. 2018.

X. Lu, H. Shi, R. Biswas, M. H. Javed, and D. K. Panda "DLoBD: A Comprehensive Study of Deep Learning over Big Data Stacks on HPC Clusters". In The Journal of IEEE Transactions on Multi-Scale Computing Systems (TMSCS'18), Jun. 2018.

M. H. Javed, X. Lu, and D. K. Panda "Characterization of Big Data Stream Processing Pipeline: A Case Study using Flink and Kafka". Proceedings of the Fourth IEEE/ACM International Conference on Big Data Computing, Applications and Technologies (BDCAT'17), Dec. 2017.

X. Lu, H. Shi, M. H. Javed, R. Biswas, and D. K. Panda "Characterizing Deep Learning over Big Data (DLoBD) Stacks on RDMA-Capable Networks". In 25th Annual Symposium on High-Performance Interconnects (HOTI '17), Aug. 2017.

Fields of Study

Major Field: Computer Science and Engineering

Table of Contents

Page

List of Tables

Table **Page**

List of Figures

Chapter 1: Introduction

With the convergence of High-Performance Computing (HPC), Big Data, and Artificial Intelligence (AI), researchers and developers have started paying more attention to accelerating the performance of AI models, applications, and frameworks. Under the umbrella of AI, Deep Learning (DL) has been gaining more momentum as a new promising technology to solve many challenging problems facing society, such as cancer detection [18], self-driving cars [22], natural language processing [61], and so on. Deep Learning frameworks and applications have been heavily leveraging HPC technologies to improve their performance and scalability.

Taking TensorFlow [6] as an example, a lot of optimized designs have been proposed in the community to improve its performance with different approaches. From the network perspective, InfiniBand [4], RoCE [3], High Speed Ethernet, etc. are used to improve the tensor communication performance in TensorFlow with high-performance communication libraries, such as gRPC [1], RDMA-gRPC [8], MPI [44] etc. From the computation perspective, TensorFlow-based Deep Learning workloads have been taking advantage of many advanced computing capabilities available on CPUs, GPUs, TPUs [29], and so on. For CPU-based platforms, Intel TensorFlow [2] can accelerate Deep Learning workloads with the latest AVX512 technology on x86 CPUs. For GPU-based platforms, cuDNN [10]

and NCCL [5] have become the standard building blocks for high-performance and scalable Deep Learning training on GPU devices.

In the same vein, more so than ever is there a need to process, analyze and generate insights from data in real-time i.e. before it gets obsolete. Businesses are focusing more and more to set up infrastructures that would help them gain valuable insights about their customers in real-time so that they can satisfy the changing customer needs as soon as possible and thus gain a competitive advantage over their rivals. Common examples of businesses that are heavily dependent on analyzing data in real-time include online advertisements, stock markets, etc.

To process data in real-time, a stream processing pipeline needs to be constructed. There are multiple ways to construct such a pipeline. However, in its most basic form, it is formed by stream source(s) initially writing data to an intermediate queue. A Stream Processing Engine (SPE) then consumes data from this queue to perform actual computations on it. To an uninformed observer, the purpose of the messaging queue, called the Message Broker (MB), might be unclear. The existence of a message broker in the pipeline is necessitated by the inherently ephemeral nature of the data stream. Unlike in Batch Processing, the input data in a real-time data processing system does not come from a persistent source. Instead, these streams of data come in real-time from a myriad of external sources, on which the pipeline has no control. And for this reason, in the event of a fault in the pipeline, it becomes impractical to replay the data from the stream source. Moreover, a stream source can have multiple consumers. The MB, in this case, provides an interface where multiple consumers can consume data coming from the same source without requiring different connections to the same stream source. Therefore, any production

scale streaming pipeline is bound to have some variant of message brokers facilitating the communication between stream producers and consumers.

The most practical technique to scale these applications to process large amounts of data has been to execute them in a distributed manner over a cluster of interconnected processing units. However, the task of getting the optimal performance out of such a distributed hardware and software configuration is non-trivial and is made more complicated by the unique communication and computation profile of each individual application; a set of configurations that bring the best out of one set of application may be sub-optimal for another. As many of the end-users of these applications are not distributed systems expert, the hardware resources available to them often go unutilized as the users, in many cases, do not possess the skills to dig deep and identify the performance bottlenecks their application is facing and, if possible, remove them. Therefore, to be able to get the best whatever hardware resources are available to the end-user, extensive studies need to be performed to figure out the tools, methodologies, and heuristics that allow the user to do so.

1.1 Motivation

Even though there exists significant literature addressing performance enhancements for Deep Learning and Stream Processing workloads, we find that there is a lack of performance models to systematically guide optimizations for them. Coming up with a useful and insightful performance model is not a trivial task. This is because these workloads typically have very complex and deep software and hardware stacks [42], which can not be easily abstracted as a simple and meaningful model. Due to the lack of useful performance models for these applications, researchers and developers typically use ad-hoc or experiential approaches to optimize the performance of their workloads, which may not be

efficient. These approaches also can not exactly identify where the bottlenecks lie and how much more improvement can be expected with possible further optimizations.

To shed light on how to solve these challenges facing such users, we propose a simple and effective approach to systematically analyze and optimize the performance of Deep Learning and Stream Processing workloads with the Roofline model [60]. The Roofline model is a very useful and insightful visual performance model for multi-core architectures.

To this end, this paper performs comprehensive profiling and analysis of Deep Learning and Stream Processing workloads run on various hardware configurations.

From the DL perspective, we make use of the Roofline model to identify bottlenecks in a step-by-step manner and resolve them accordingly.

The optimizations obtained in our study broadly focus on two major directions of improving the computational efficiency at minimal levels of concurrency and communicational efficiency at high levels of concurrency. We find that a visual tool that helps identify particular avenues of improvement along these directions will be very useful for the community.

Our studies demonstrate that such a performance analysis approach for Deep Learning workloads with the Roofline model is efficient for achieving near-peak performance on target platforms. In a nutshell, this study makes the following key contributions:

- Present detailed profiling and analysis of CPU-based distributed training of Deep Neural Networks (DNNs);
- Provide guidelines to show how a performance model, such as the Roofline model, may be used to optimize the execution of DNN workloads;

- Suggest optimal values for some key parameters which may be used by non-expert users to get high performance for CPU-based Deep Learning.

Using this approach to optimize distributed training of DNNs, we are able to obtain up to 3.5X performance improvement over vanilla TensorFlow using the default configurations.

On the Stream Processing end, we first used the Roofline model to identify the key areas which become a bottleneck in the stream processing pipeline and then use it to guide the architectural optimizations required to design a high-performance MB, called Frieda, which is necessary for an accelerated stream processing pipeline. Frieda provides Remote Direct Memory Address (RDMA) [55]-based communication support along with advanced designs specifically aimed towards curtailing tail latencies. Frieda not only reduces the end-to-end latency of the streaming pipeline but also increases the total throughput that it can support.

In a nutshell, from the perspective of Stream Processing, this paper makes the following key contributions:

- Analysis of design bottlenecks in Kafka which contribute to higher tail latencies.
- Mathematical model explaining the reasons for tail latency in practical systems.
- Design and implementation of Frieda, a high-performance message broker which resolves these bottlenecks.
- Extensive experimental evaluation based on message broker based micro-benchmarks as well as end-to-end streaming applications.

We run our experiments in both single broker as well as multi-broker cluster environments. We observe an improvement in latency of up to 98% for Kafka Producer micro-benchmarks and up to 32% for Yahoo! Streaming application benchmark.

1.2 Organization of this Thesis

The rest of this thesis is organized as follows.

Chapter 2 introduces the topics and concepts relevant to this thesis.

Chapter 3 contains a comprehensive performance analysis of Distributed TensorFlow and discusses the optimizations derived from it.

Chapter 4 describes the details of the implementation of Frieda, and the insights obtained from the Roofline analysis of Kafka that guided its architecture.

Chapter 5 summarizes the results of the thesis and gives pointers to future research that can be based on this work.

Chapter 2: Background

2.1 Tensorflow

Tensorflow [6] is a Machine Learning framework developed at Google which provides an implementation of various functions and modules commonly used in Machine Learning algorithms. It also provides the functionality to make use of various resources available within a system, such as multi-core processors, GPUs, etc, to accelerate the performance of the applications developed using it. Distributed TensorFlow allows users to scale applications along both inter-node and intra-node directions. Note that in this paper we adapt the data-parallelism [35] approach to partition and scale our algorithms. In this approach, multiple replicas of the same model are launched on the processing units available while the training data is partitioned equally across these replicas. Subsequently, different mechanisms, such as the ones described below, are used to aggregate the results of these replicas to obtain a global state. A contrasting approach is the model-parallelism [14] where instead of data, the layers of the machine learning model itself are partitioned across various processing units while the same data is fed to each unit. Moreover, the modular implementation of TensorFlow enables many different communication paradigms and gradient update models to be used underneath the algorithm layer. This study focuses on the two such widely used paradigms which are described in detail below.

2.1.1 Parameter Server

The Parameter Server model [36] is an approach used to perform distributed Machine Learning at scale. It includes the abstractions of the parameter server (PS) processes and worker processes. Workers execute replicas of the actual Machine Learning algorithm while the PS stores global parameters required by each replica. The process of transmission of gradients to the PS and subsequent aggregation can be performed in synchronous as well as asynchronous manner. A recent study [9] has shown that synchronous weight updates with replication for stragglers result in faster convergence and better accuracy compared to the asynchronous approach, therefore we use the synchronous approach in our experiments as well.

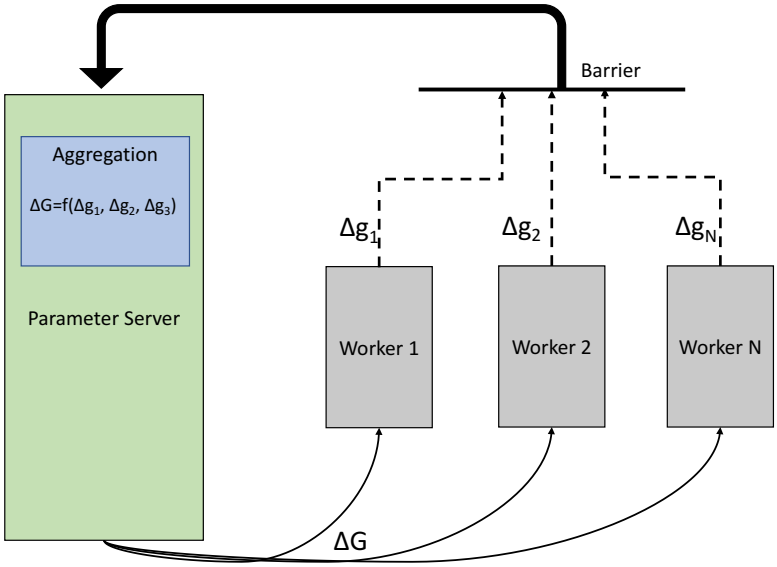


Figure 2.1: Parameter Server (PS) model with synchronous gradient updates: One or more PSs distribute work (through broadcast) and aggregate results (through reduction).

Figure 2.1 describes how the parameter server model works with synchronous updates. Each of the workers involved compute their own local gradients. After a certain number of iterations, the participating worker replicas share their local gradient vectors with the parameter server and wait at a barrier. The parameter server then aggregates all the received gradients to obtain a global view of the model, which is then broadcasted to all the workers, which can then begin the next set of iterations. For greater scalability, the ratio of parameter servers to workers can be increased. However, figuring out the optimal ratio is non-trivial and having excessive servers may saturate the network. Moreover, using the parameter server approach to scale a sequential implementation of a Deep Learning model requires significant changes in order to configure the distribution of resources and the communication pattern between them in an optimal manner.

2.1.2 Horovod

Horovod [52] is a runtime developed for decentralized distributed Machine Learning by Uber. Instead of using separate parameter servers to store the global parameters, each worker in a Horovod cluster keeps a copy of all the parameters. In the synchronization phase, each worker takes part in a bandwidth optimal ring-based allreduce [48] aggregation. The ring-based allreduce algorithm is implemented using NVIDIA Collective Communication Library (NCCL2) [5] for GPUs and MPI on CPUs. Compared to the parameter server approach, using Horovod to distribute sequential machine learning code requires minimal changes.

Figure 2.2 shows how the ring-allreduce algorithm is used for synchronizing gradients in Horovod operates. Each node sends 2 $\times$ (N-1) messages to each of its two neighbors. First N-1 messages received are added to the receiving node's buffer whereas the second

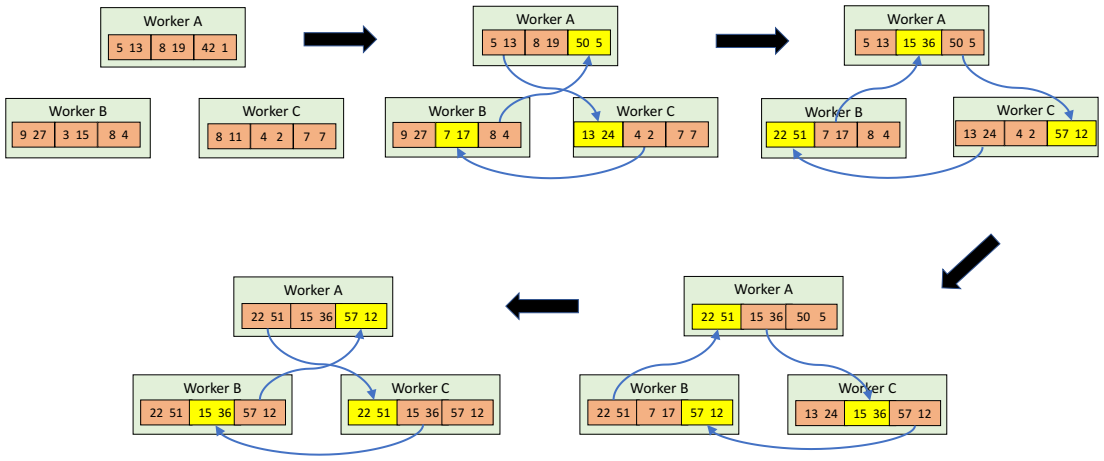


Figure 2.2: Ring-allreduce, which optimizes for bandwidth and memory usage over latency.

round of N-1 messages replaces the values held in the receiving node's buffer. After 2 $\times$ (N-1) iterations, each worker has a globally synchronized view of all the parameters.

2.2 Roofline Model and Roofline Trajectories

The Roofline model [60] is a performance analysis technique which makes use of memory access profiles and compute operations to identify if the application is memory bound or compute-bound. Traditionally, Floating Point Operations Per Second (FLOPS) have been used to quantify the compute operations performed but recently many studies have come up with bespoke, platform-specific units as well. For example, [59] introduces a data-centric variant of the Roofline model which better captures the behavior of typical applications running on commodity clusters by including not only floating point but all other integer-based operations as well. However, FLOPS is suitable for many HPC and Machine Learning applications as they are known to be fairly floating-point operations intensive.

Roofline trajectories [24] is an extension to the original Roofline model where individual points pertaining to Roofline profile at different concurrency levels are connected to generate a scaling trajectory. It is highly suited for our study considering the focus on scalability and concurrency which is why we use for our analysis.

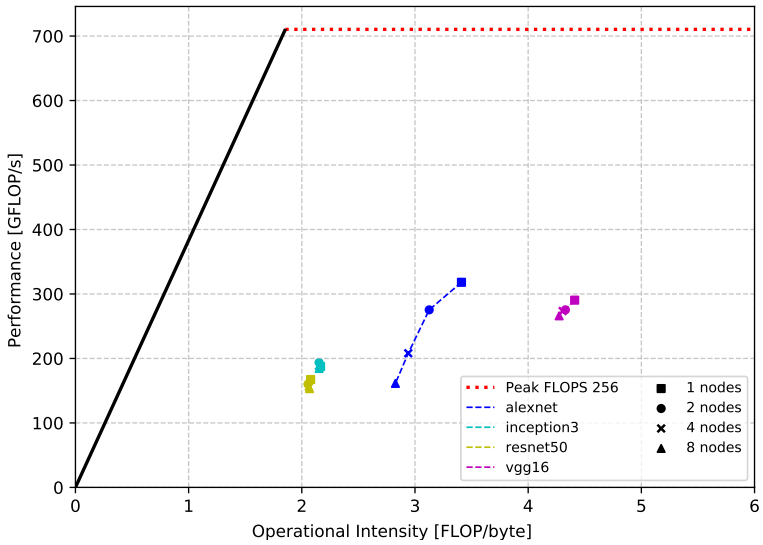


Figure 2.3: Roofline Trajectories characterize the application using operational intensity and machine computational bounds. The connected points form a line showing the general trend with which application responds with regards to varying concurrency.

Figure 2.3 shows a typical Roofline Trajectories plot. The x-axis represents Operational Intensity (OI) [1] which is a unit for measuring the floating-point operations performed per byte of memory accessed. The vertical axis represents the computational performance obtained in GFLOPS. The sloped line starting from the origin represents the range of Operational Intensity for which the performance of the application will be bound by memory bandwidth. The point at which it terminates is called the ridge point, which is the point

[1] $OI = NUM\_FLOP/MEM\_ACC$, where NUM_FLOP means the number of floating-point operations performed and MEM_ACC means the bytes of memory accessed.

beyond which all Operational Intensities represent the compute-bound region. The red horizontal line denotes the peak floating-point performance of the hardware the experiments are executed on. The individual points on the graph represent various applications and the region that they lie in based on their OIs and operations executed, which in our case are obtained by reading performance counters using tools such as perf [13]. The connected points form a line showing the general trend with which each application responds with regards to varying concurrency.

2.3 Kafka and its relation to Stream Processing

Apache Kafka is a distributed commit log designed using the publish-subscribe architecture. The bulk of the heavy lifting in a Kafka cluster is done by brokers. These are the central components of a Kafka cluster and have the responsibility of receiver messages from any number of producers, commit them to disk and replicate them in the cluster if configured to do so. They also allow consumers to consume messages from any position in the commit log. The main data abstraction in Kafka comes from the concept of a topic. A topic corresponds to an independent log corresponding to a usually logically related data. A topic in Kafka can have any number of partitions, which provide the basic level of parallelism for that topic for both writes as well as reads. Producer, at the time of topic creation, specifies the number of partitions as well as a replication factor for that topic. Each broker in the Kafka cluster will act as a leader of any number of partitions. The responsibility of leadership is equally divided among the Broker cluster. The replication factor specifies how many brokers the messages in a topic should be written to before a message is considered committed successfully. In the trivial case of the replication factor set to 1, the message is only committed to the partition leader. However, in the case of a replication factor greater

than 1, it is written to brokers who are followers of that partition, in addition to the partition leader, the total number of which aggregates to the replication factor.

Producers in a Kafka pipeline write to one particular topic, however, a consumer can consume messages from any number of topics. When a producer starts committing records to a topic, the first message that it sends is assigned the offset 0, and this offset increases atomically for each subsequent message sent. Brokers store messages for every partition in the order of increasing offsets. In the trivial case, consumers specify the offset of a particular partition in a particular topic it wants to consume messages from. It commits locally the last message offset successfully consumed so that in case of a failure, it can resume consuming messages from the position it left off. Making the Consumer stateful in this way allows the Brokers to be freed from this consumer state bookkeeping making the read and writes to the Kafka broker very cheap in terms of resource consumption.

Kafka relies on a Zookeeper [21] cluster to maintain global state about broker identities, topics, and their corresponding partition leaders, in-sync replicas, etc. This is done to allow the broker cluster to be fault-tolerant. In the case of broker failure, it can restart and fetch the global cluster state and resume its responsibility in the cluster.

Figure 2.4 shows the internal workings of Kafka. Kafka Broker consists of multiple Processor threads dedicated to handling all requests from a particular client. The Handler threads perform the actual operations of writing and reading new records, committing offsets, etc. Clients have a similar architecture to Brokers with the difference lying in Clients having a single Producer thread and no Handler threads.

Lastly, partitions of Kafka topics are guaranteed to have messages written to and read from them in an ordered fashion. This is critical for message processing semantics that are employed higher up in the stream processing pipeline where a streaming application

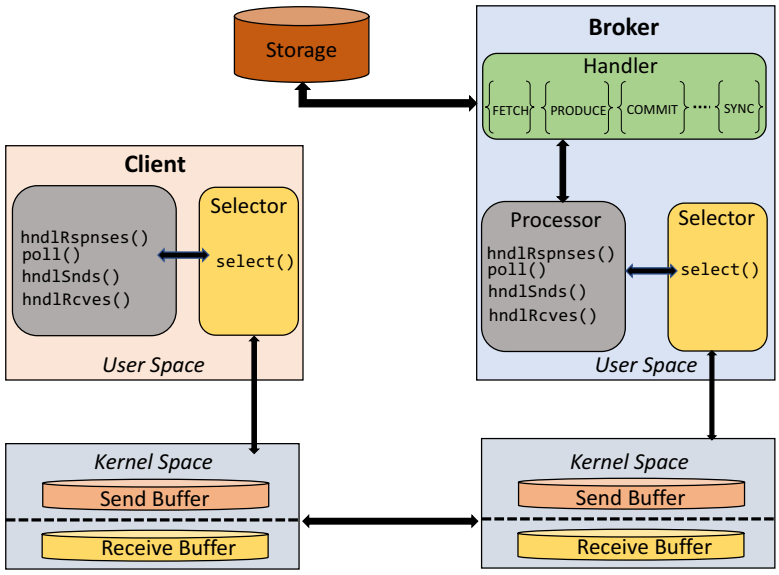


Figure 2.4: Overview of Kafka's architecture

may require at least once or even exactly-once message processing requirement. Kafka ensures this by processing requests from a channel in a strictly sequential manner, where no new requests are processed until the response for the last processed request has been sent out. This sequential processing of records, although a straightforward mechanism to ensure responses are sent out in the expected order, contributes significantly to the time taken to process requests. This is because new requests coming in from the same client are not processed until the response for the last request received is written to the respective socket.

The loose coupling of Kafka's architecture lends itself quite well to a wide variety of use cases. It can be used as a log aggregator, as a publish-subscribe system allowing a diverse set of applications to read from and write to it, etc. However, this paper focuses on Kafka's role as a Message Broker in a real-time stream processing pipeline. This role and

the motivation for choosing it have been described in detail in [26] and it also dictates the parameters we try to optimize in our designs for the enhanced Kafka library detailed in the succeeding sections.

2.4 InfiniBand & RDMA

InfiniBand is low latency, high bandwidth interconnect popular in HPC clusters. The InfiniBand Enhanced Data Rate (EDR) channel provides a bandwidth of 100Gbps with an adapter latency of 0.5ms. Besides providing fast network I/O, InfiniBand also provides advanced features including hardware offloaded communication and Remote Direct Access Memory (RDMA) based communication. As the communication in an InfiniBand channel is offloaded to the HCA, the CPU is freed to perform other operations. The semantics for performing this offloaded communication is provided by the RDMA protocol whereby remote processes can write to and write from a host's buffer without involving the host in the process. InfiniBand also provides a wrapper the Internet Protocol over InfiniBand (IPoIB) [12] wrapper protocol to allow applications using socket-based TCP/IP stack to be communicated over the InifniBand network.

Chapter 3: Performance Analysis and Optimization of Distributed TensorFlow

3.1 Performance Analysis Methodology

Determining the method to be utilized to obtain a set of performance optimizations for a particular application is a non-trivial task. Such a task is complicated even further in the case of distributed Machine Learning frameworks because of the sheer quantity of independent layers of communications and computation involved. Therefore, in this study, we adopt a step by step approach where insights from one step are used as a guide for the subsequent so that a comprehensive set of optimizations are obtained covering many different facets of the system under consideration. The first step in our methodology involves running baseline experiments to determine the performance metrics obtained using just the out-of-the-box implementation. These experiments are then analyzed to identify and isolate potential bottlenecks. In the next step, we attempt to remove these bottlenecks by performing application-agnostic, black box optimizations targeting the framework that the end-user application is running on. These optimizations do not modify the client application, which in our case is the Machine Learning algorithm, but rather optimize the operations of the framework the application executes on, which is TensorFlow for this study. Lastly, application-aware optimizations are performed to tune the application itself

to extract further performance gains based on the optimizations implemented in the preceding step. This method, when applied to distributed Machine Learning using TensorFlow, and the optimizations derived as such are explained in detail in subsequent sections.

The official TensorFlow repository provides benchmarks[2] of various Convolutional Neural Networks (CNNs) which we use in our study. Of the many networks available, we select four commonly used and extensively studied models - Alexnet [35], Inception3 [58], Resnet50 [20], and Vgg16 [54] - which are based on the ImageNet [15] image classification dataset. These represent models with varying degree of computation and communication intensities covering a broad range of implementations of the broad spectrum of Deep Learning models.

The experiments are carried on Chameleon Cloud [31], an NSF funded cloud testbed. The hardware specifications of the 'Skylake' nodes that are used to carry out all the experiments presented in this study are summarized in Table 4.1.

Resource	Specification
CPU	Intel(R) Xeon(R) Gold 6126 @ 2.60GHz
Cores $\times$ sockets	12×2
Memory	192 GB @ 119.21 GiB/s[3]
Disk	240 GB HDD
NIC	Ethernet (10 Gbps)
OS	CentOS release 7.5.1804

Table 3.1: Cluster Configuration

The names and versions of different frameworks and compilers used in this study are summarized in Table 4.2.

[2]https://github.com/tensorflow/benchmarks

[3]https://en.wikichip.org/wiki/intel/xeongold/6126

Software	Version
Tensorflow	1.13
Intel-Tensorflow	1.13
Python	2.7.1
MPICH	3.3.1
Horovod	0.16.4
gcc	4.8.5

Table 3.2: Software Configuration

Even though the focus of this study has been on homogeneous, CPU-based clusters, the approach described in this study can easily be extended to heterogeneous GPU or hybrid CPU/GPU clusters. Tools such as nvprof [4] and NVIDIA Nsight [5] kernel profile utility can be used to extract the relevant performance counters on NVIDIA GPU-based systems, as has been described in other similar works [33, 23]. As a guideline, the following steps may be performed to achieve the task for a given architecture:

1. Launch the relevant performance counter retrieval tool as a daemon. As mentioned earlier, on Intel CPU based architectures perf may be used while nvporf may be used for NVIDIA GPUs.

2. Launch the application that needs to be profiled. In our case, these were DL models executed on TensorFlow.

3. Once the application terminates, use the counters obtained to calculate the number of FLOP executed, number of bytes of memory accessed and time taken.

[4]https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview

[5]https://developer.nvidia.com/tools-overview

(a) The counters to obtain memory accesses are *CAS_COUNT.WR* and *CAS_COUNT.RD* on CPUs while *dram_read_transactions* and *dram_write_transactions* on NVIDIA GPUs.

(b) An approprite multitplier (64 for Intel Skylake, 32 for NVIDIA Volta) can be used to convert the conter values to actual memory bytes accessed.

(c)

$$OI = \frac{FLOP}{(Reads + Writes) \times Multiplier} \tag{3.1}$$

4. Use these metrics to construct a Roofline profile to guide optimizations.

3.2 Baseline Experiments

In this section, we analyze the performance characteristics of running distributed Deep Learning on vanilla Tensorflow using PS and Horovod as the variable update models. We use a variant of the Roofline model [24], in which we plot scaling trajectories, rather than points, at full concurrency; the trajectories are helpful in observing the overall trend in performance attained with respect to changes in the level of concurrency.

3.2.1 Profiling

The experiments described in this section are performed on the real ImageNet data set. However, some preliminary experiments are performed with synthetic data as well. We observe that, barring minor experimental variability, the results obtained with real data are matching the ones obtained using the synthetic data. Moreover, the actual memory bandwidth achieved on a system is often less than the theoretical peak. To measure the

maximum achievable memory bandwidth, STREAM[6] benchmark was used. All the experiments described in this study are performed with a batch size of 64, unless specified otherwise.

Parameter Server Model

Setting up a TensorFlow cluster allows for various configurations of PS and worker processes, described in Section 2.1.1, to run on the resources available. However, the number of such processes to launch and their mutual ratio is a heuristic that often needs to be optimized as it can have a significant performance impact. The launched worker processes per node (PPN) is a parameter we have tuned before we launch our detailed experiments. We run some preliminary experiments to arrive at an optimal value for PPN.

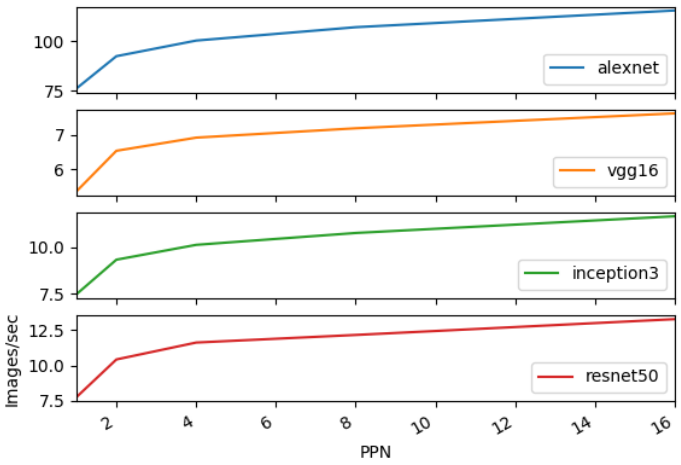


Figure 3.1: Throughput variations in response to varying PPN. Increasing PPN from 1 to 2 and then 4 yields significant improvements in throughput but for PPN values beyond 4 the throughput remains fairly stable.

[6]https://www.cs.virginia.edu/stream/

Figure 3.1 shows the aggregate throughput obtained with varying numbers of worker PPN. We observe a sharp increase in aggregate throughput when PPN is increased from 1 to 4, after which it starts to stabilize. Other factors are also needed to be considered for an optimal PPN value; launching multiple processes per node rapidly increases the job startup time and the benchmark itself would have issues outputting the correct logs for each process at higher PPN values. Taking these factors into account, we decide to set worker PPN to 4 for the experiments described in this section as it can represent a reasonable balance between desired concurrency and ease of implementation.

Binding processes to cores by setting their affinity is a frequently used optimization technique for parallel applications. We have tried various configurations of process affinities also taken into account the Non-Uniform Memory Access (NUMA) configuration of the processing element, on which the experiments are executed. Regardless, the best performance is obtained without making the use of processing affinities at all. Instead, allowing processes and the threads launched by them to freely migrate among cores can deliver the best performance for TensorFlow-based training. This phenomenon has also been observed in other Python or Java-based applications [39].

Different levels of concurrency are tested against a varying number of PS to see how well the distributed training can scale. Figure 3.2 and Figure 3.3 show Roofline plots for different DNNs executed using varying levels of concurrency, for nodes running PS and worker, respectively. As described in detail later in Section 4.1.1, unoptimized TensorFlow does not perform Advanced Vector Extensions (AVX) or Fused Multiply Add (FMA) instructions thus the peak attainable floating performance comes out to be: 2 sockets $\times$ 12

cores/socket $\times$ 3.7(GHz) clock rate with Turbo boost $\times$ 8 Single Precision (SP) FLOPs/-cycle = 710.4 GFLOPS, which is denoted by the dotted red line "Peak FLOPS 256" in the graphs.

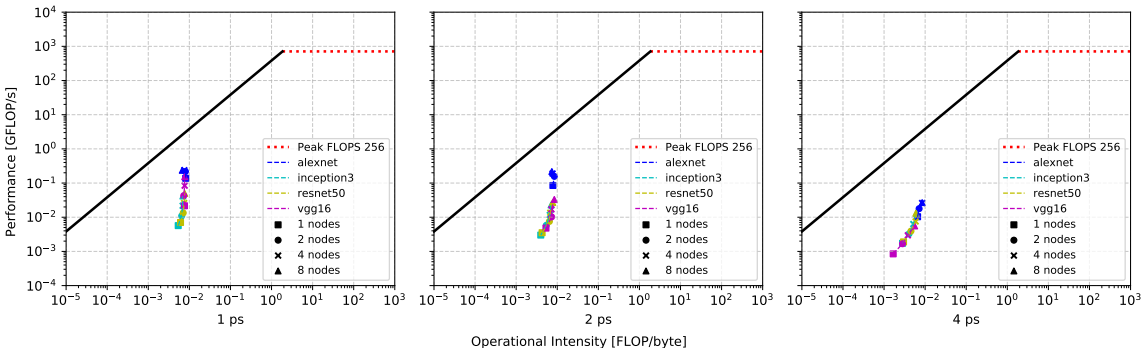


Figure 3.2: Roofline plots for PS nodes. Significantly less raw operations are executed compared to the worker nodes which is expected. Increasing number of PS results in downward shift of trajectories as each PS becomes responsible for fewer network parameters to be aggregated.

From Roofline plots for nodes running PS shown in Figure 3.2, we observe that the task is not very compute intensive as FLOPS generated are in the ballpark of 1 MFLOPS to 1 GFLOPS, which is orders of magnitude less than nodes running worker processes as shown in Figure 3.3. Keeping the number of PS constant, increasing the number of workers leads to an upward shift for the data points in Figure 3.2 as the number of workers across which the gradients need to be synchronized also increases for each PS. Similarly, for any given number of workers, adding more PS to the system leads to fewer parameters that each PS is responsible for synchronizing across the system. As a result, the corresponding Roofline trajectories also show a downward shift. Note that the results for PS nodes are plotted against a logarithmic scale.

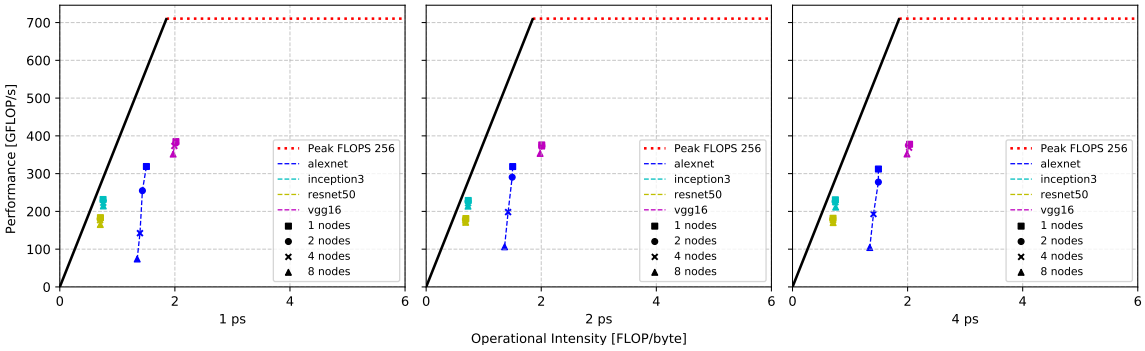


Figure 3.3: Roofline plots for worker nodes. Trajectories for Alexnet show variations proportional to the level of concurrency suggesting its communication overhead is significant. A reduction in the length of trajectories for Alexnet may also be observed in response to increasing number of PS as gradient exchange becomes faster. Stable trajectories for other models indicate decent overlap between communication and computation.

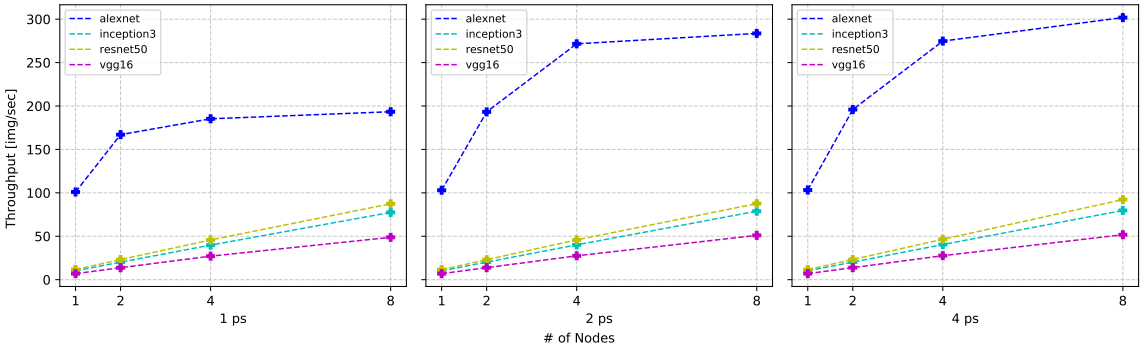


Figure 3.4: Throughput obtained by various DNNs, which may be matched to Roofline trajectories from Figure 3.3. Throughput for Alexnet almost follows a logarithmic trajectory, which shows improvement as more PS are added. All other models show linear speedup with minimal changes in response to increasing number of PS.

Figure 3.3 and Figure 3.4 show the Roofline trajectories for worker nodes and throughput obtained by the model, respectively. This helps in comparing the actual application performance with the black box approach of the Roofline model. The model which is able to generate FLOPS closest to the peak performance is Vgg16. If more workers are added to the system, while keeping the number of PS unchanged, we see an almost 2X (8.2 img/sec with 4 workers vs 16.3 img/sec with 16 workers for Vgg16) increase in throughput for a proportional increase in workers, for all models barring Alexnet. These models show a scaling efficiency upwards of 90%, with Resent50 turning out to be the most scalable model at 100% scaling efficiency. This is observable from the Roofline data points as well as the OIs for all models barring Alexnet show little change. Alexnet, however, at best shows a scaling efficiency of 37% using as many as 4 PS. The OI for Alexnet also shows a steady decline in response to an increase in concurrency in the system. This suggests that communication costs start to dominate at higher concurrency levels. When looked at in conjunction with the PS Roofline plot, we interestingly see that while the data points at the worker end decline, the ones at the PS show an upward trend indicating that PS has to perform more work to synchronize gradients across the system. Consequently, the workers have to wait longer at the barrier in-between successive steps while the synchronization takes place.

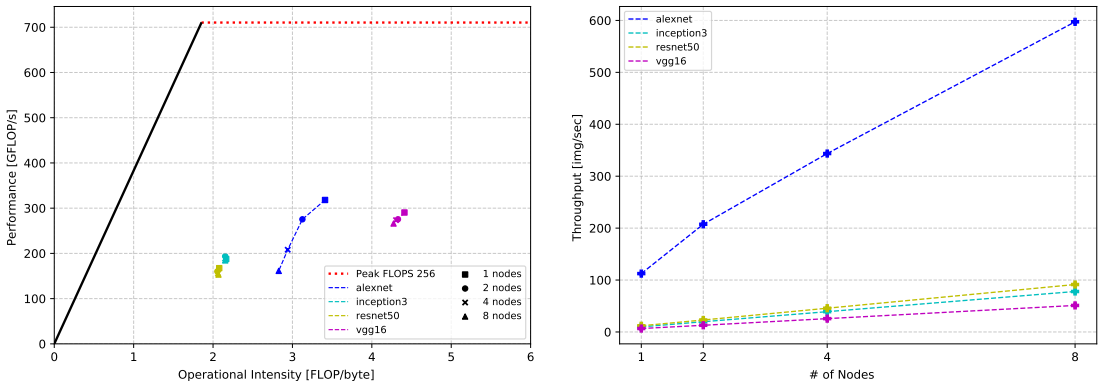
The Roofline plot for PS is not included from this section onwards for the sake of brevity as it did not show any notable change compared to the one shown in Figure 3.2.

It should be noted that all data points obtained using the PS approach are under the memory-bound region of the Roofline plot which implies that adding more computational resources will not necessarily lead to a gain in application performance. This is ever more relevant in the case of Alexnet where we observe that an increase in concurrency leads to

almost a vertical decline in raw performance without a drastic change in OI. This suggests that overheads pertinent to concurrency dominate and simple increase in processing power will not benefit the performance much. Instead, approaches that may result in an improvement in OI should be considered. The use of Horovod as a gradient update layer is a step in this direction.

Horovod

In this section, we carry out the same experiments as described in Section 3.2.1 but perform them using Horovod as the gradient update layer.



(a) Roofline plot (b) Throughput

Figure 3.5: Roofline trajectories for Horovod show increased OIs for all models compared to the PS approach (Figure 3.3). For Alexnet, there is a reduction in the length of Roofline trajectory which is also reflected in the almost linear speedup of the throughput.

From Figure 3.5, we can see a marked improvement in application throughput with Horovod as compared to the PS approach, which becomes even more pronounced at higher levels of concurrency. The OIs are also much higher for Horovod, improving as much as

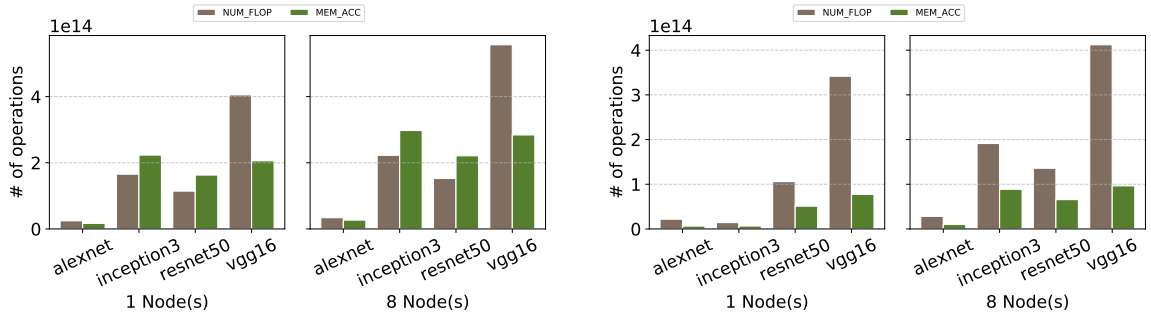
2X (2.0 FLOP/byte vs 4.2 FLOP/byte) compared to the PS approach for Vgg16. The raw floating-point performance, however, is within the same ballpark.

The scaling efficiency of experiments with Horovod is equal or better than the ones obtained with the PS approach. For Alexnet, however, the bandwidth optimal ring-allreduce algorithm shows its benefits leading to a scaling efficiency of 66%, a marked improvement from 37% obtained with as many as 4 PS in the system.

3.2.2 Analysis

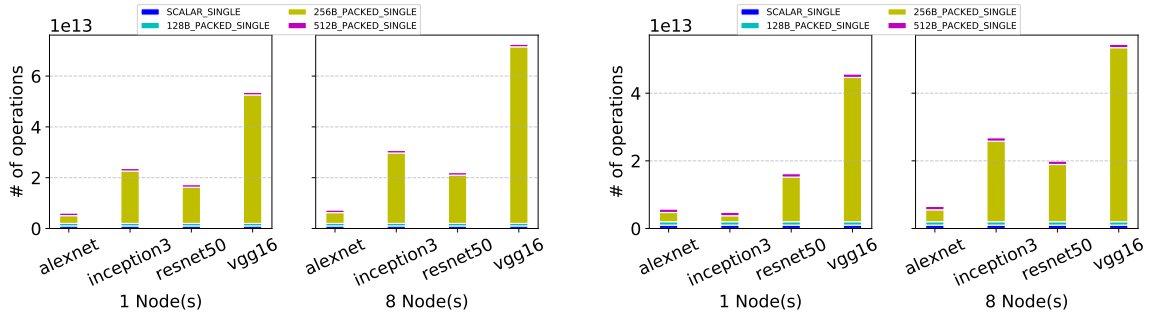
The Roofline plots discussed in Section 4.1 help understand the computational and memory/network I/O footprint of various DNNs implemented in TensorFlow. However, we need to take a deeper look at what kind of floating-point operations are performed at what degree of memory access rate to get a deeper understanding of how we can improve the performance of these applications.

Figure 3.6 is helpful in understanding why there is a difference in OI for the same experiments run using PS and Horovod approach. Figure 3.6(a) shows the OI breakdown at worker nodes of experiments run with varying numbers of PS while Figure 3.6(b) is for Horovod. As expected, NUM_FLOP and MEM_ACC do not vary much if we add more PS to the system while increasing the number of workers leads to an increase in both NUM_FLOP and MEM_ACC. That is because adding more PS reduces the workload for each PS but the worker task remains unchanged. As far as Horovod is concerned, the bars of NUM_FLOP in Figure 3.6(b) are generally not as high as for similar PS based experiments. However, MEM_ACCs are, on average 2X lower for all models leading to a much higher OI.



(a) Parameter Server (b) Horovod

Figure 3.6: OI breakdown of TensorFlow worker with PS and Horovod. Memory accesses are higher than FLOP executed for Inception and Resnet50 resulting in an OI of less than 1 for PS. For Horovod, however, the ratio between NUM_FLOP and MEM_ACC is much higher than the ones observed for PS.



(a) Parameter Server (b) Horovod

Figure 3.7: FP breakdown of TensorFlow worker with PS and Horovod. 256-bit SP instructions constitute the majority of all FP instructions executed despite Horovod using a different aggregation mechanism.

Based on the above profiling results, we can conclude that Horovod is much more efficient with the data that it processes as for each byte of memory accessed, it performs more number of FLOP than using the PS approach. Next, we take a look at the distribution of different kinds of floating-point (FP) instructions performed by TensorFlow using different methods of gradient update.

From Figure 3.7, we can see that vanilla TensorFlow mostly makes use 256-bit SP FP instructions. As a result, the maximum FLOPS that can be performed has an upper bound denoted by Peak FLOPS 256 line on the Roofline graph. For the DNNs to perform closer to the theoretical peak, the multiple FMA units available have to be used in conjunction with the 512-bit AVX registers.

3.2.3 Insights

The Roofline plots coupled with the throughput gradients are quite helpful in understanding the general behavior of DNN models. For instance, the Roofline trajectories are shown in Figure 3.3 of all models barring Alexnet suggest that these are computationally dense models with communicational requirements which do not become a bottleneck even at higher levels of concurrency. The corresponding throughput numbers, as described in Figure 3.4, verify this claim as we see an almost linear speedup for all models barring Alexnet.

Alexnet seems to be an anomaly in this case requiring deeper investigation. There is a drop in performance per node of almost 70% (311 GLOPS vs 98 GFLOPS) for Alexnet even with 4 PS in the system. This corresponds with the throughput numbers for Alexnet which show sub-par speedup with an almost logarithmic trajectory. However, even with poor speedup, the absolute speedup for Alexnet is orders of magnitude higher than that

of the next best performing model (i.e., Resent50). This indicates that, as opposed to other models, Alexnet is quite intensive in terms of communication but is not too dense computationally. The observation can be verified by analyzing the number of operations that need to be executed by the processing element and network parameters that need to be exchanged. Alexnet has 60 million network parameters which have to be synchronized relatively frequently as it contains only 25 layers. Resent50, on the other hand, has only 25 million network parameters spread out over 50 layers thus not requiring synchronization as often.

3.3 Black Box Optimizations

From the graphs discussed in the previous section, we find that in order to improve the application performance and take the Roofline data points closer to the theoretical peak, some optimizations need to be performed. The data discussed so far indicates that the choice of underlying gradient update model significantly influences whether the computation is bandwidth bound, i.e. PS-based approach, or compute-bound i.e. Horovod.

3.3.1 Profiling

To start with, we decide to use Intel MKL enabled TensorFlow. Intel-TensorFlow makes use of AVX, AVX2 and AVX512 registers to perform Fused Multiply Addition (FMA) instructions enabling applications to perform FP operations much closer to the theoretical peak provided by the hardware. Note that the peak attainable performance with AVX512 FMA instructions may not be calculated using the formula described in Section 3.2.1, as having 512-bit vector instructions reduces the maximum clock rate[7]. Therefore we use the

[7]https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-scalable-spec-update.pdf

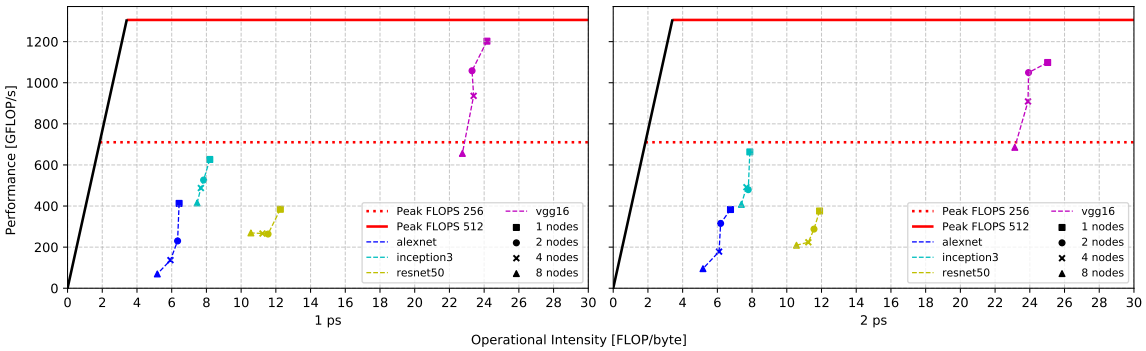


Figure 3.8: MKL enabled TensorFlow worker with PS. The use of vectorized instructions pushes the maximum attainable performance to **Peak FLOPS 512**. As a result, the raw application performance also shows an upward movement compared to the ones shown without MKL in Figure 3.3.

value of 652.8 $\times$ 2 sockets = 1305.6 GFLOPS provided by Intel in their official documentation [8] as the maximum attainable FLOPS with AVX512 instructions, denoted by the solid red "Peak FLOPS 512" line.

We are able to figure out the appropriate configuration to get the most out of MKL enhanced Intel-TensorFlow. We have tried a series of configurations and at two MPI processes/node and 12 OMP threads/MPI process, we are able to obtain the best scalability. Note that for vanilla TensorFlow, the best performance is achieved by using four MPI processes/node.

Parameter Server Model

Figure 3.8 shows the Roofline plot of MKL enabled TensorFlow with PS for gradient update while Figure 3.9 summarizes the throughput achieved. The use of AVX-512 enabled FMA instructions by MKL not only leads to a significant increase in OI for all models but

[8] https://www.intel.com/content/dam/support/us/en/documents/processors/APP-for-Intel-Xeon-Processors.pdf

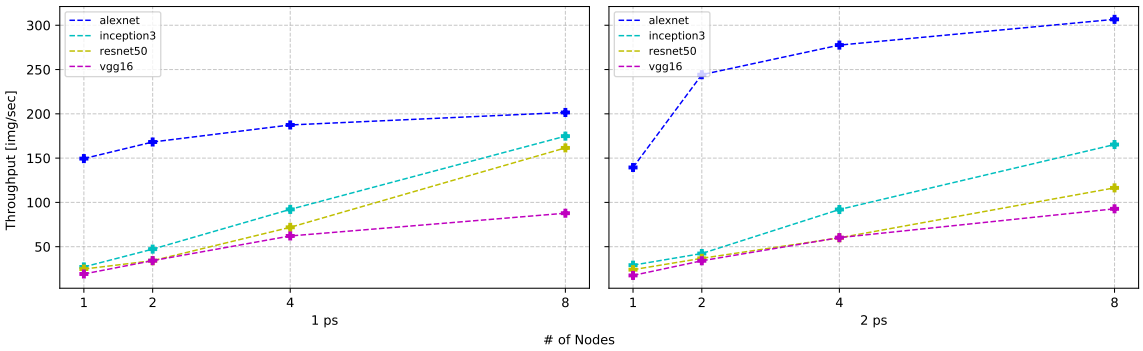
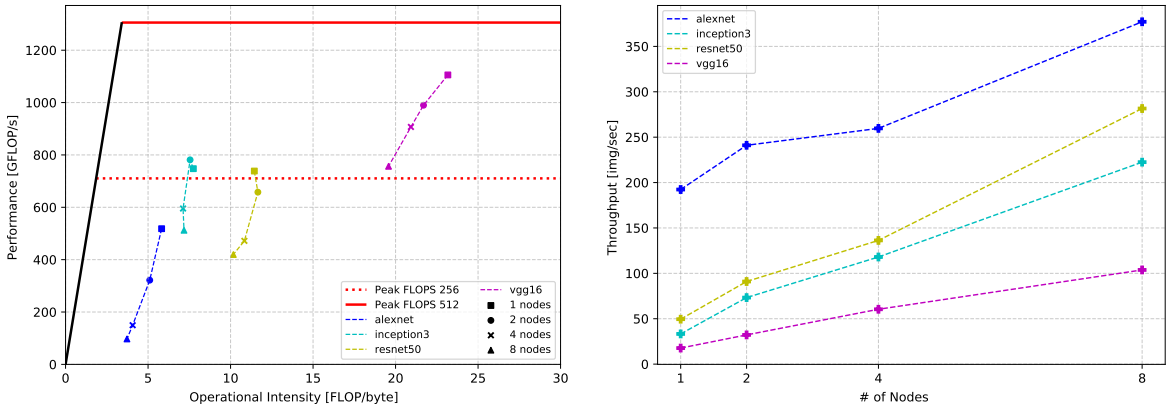


Figure 3.9: Throughput of MKL enabled TensorFlow worker with PS. Significant improvement in speedup is observed for all models except Alexnet as gains from faster computation are compensated with losses from more frequent gradient exchanges resulting in minimal net improvement.

also results in the Roofline trajectories shifting upwards. The result is pronounced with Vgg16 with four workers performing at 1200 GFLOPS, which is merely 8% less than the peak performance. The improvement in raw performance is supported by throughput numbers as well with almost 2X improvement in performance for Inception (72 img/sec vs 170 img/sec), Resnet50 (74 img/sec vs 130 img/sec), and Vgg16 (50 img/sec vs 100 img/sec with 32 workers) using only 2 PS. Alexnet, however, does not show marked improvement in throughput, which can be seen from the Roofline plot as well as it shows the least improvement in raw performance compared to other DNNs tested. We can also observe an increase in length and variability in Roofline trajectories compared to vanilla TensorFlow (Figure 3.3) indicating that although the use of vectorized instructions speed up the pass through the layers of a DNN, the network parameters have to be synchronized more often which leads to a decline in raw performance per node proportional to the level of concurrency in the system.

Horovod



(a) Roofline plot

(b) Throughput

Figure 3.10: MKL enabled TensorFlow worker with Horovod. Similar trends in Roofline trajectories and throughput are observed as in Figure 3.8 and Figure 3.9 for PS, respectively. However, the bandwith optimal ringa-allreduce algorithm seems to overlap computation and communication more effectively resulting noticeable improvements over the PS counterpart.

From Figure 3.10, we can see that using Horovod for gradient update seems to bring the most out of MKL enabled TensorFlow as a quicker pass through DNN layers (because of vectorized instructions) is complemented by a bandwidth optimal gradient update algorithm (i.e., ring-based allreduce). MKL-enabled TensorFlow with Horovod leads to less time spent in both computation and communication with neither becoming a bottleneck for the other. This is not observed to be the case in any of the prior experiments.

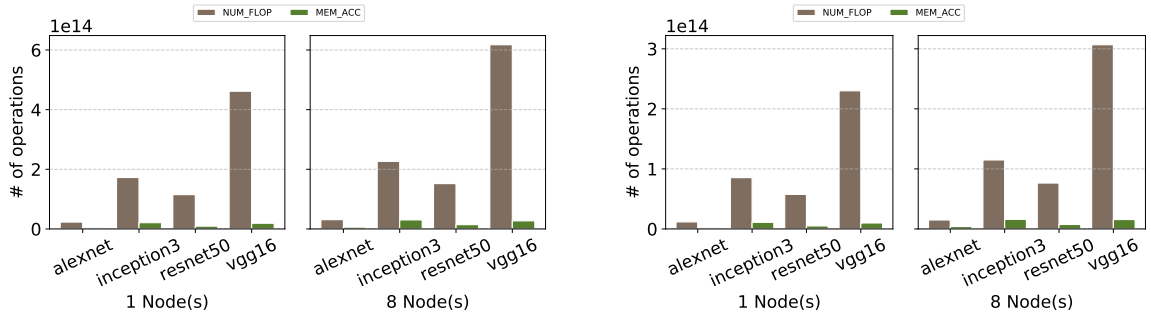
We see greater than 2X improvement for Resnet50 (98 img/sec vs 273 img/sec with 16 workers) and Vgg16 (49 img/sec vs 101 img/sec with 16 workers) while Inception3 shows a speedup of 3X (73 img/sec vs 215 img/sec with 16 workers) compared to vanilla TensorFlow. Alexnet, however, shows severely poor scalability as at eight nodes it shows a

decline of 25% in throughput compared to vanilla TensorFlow even though at one node the observed speedup is 2X (101 img/sec vs 198 img/sec). The Roofline trajectory for Alexnet provides cues for this behavior. Comparing Figure 3.5 and Figure 3.10 at eight nodes, the raw performance also shows a decline of 41% however the performance at minimal concurrency (i.e., two workers on one node) improves by about 70%. This indicates that although vectorized instructions improve pass through the layers significantly, the overhead incurred from synchronizing 60 million network parameters frequently at higher levels of concurrency actually leads to performance degradation.

As shown in Figure 3.10, at minimal concurrency (i.e., one node) we do get very close to the theoretical peak performance for Vgg16. However, Intel-TensorFlow does not scale as well as vanilla. Doubling the number of resources for vanilla TensorFlow leads to a proportional increase in throughput as well. However, for Intel-TensorFlow it is less than proportional. This trend is evident in the Roofline plot as well where data points for Intel-TensorFlow show a much greater decline in response to an increase in resources (higher communication costs) than vanilla TensorFlow.

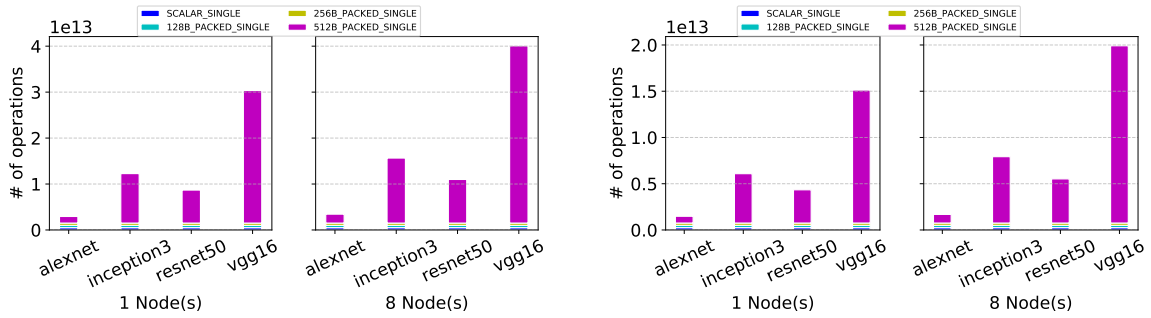
3.3.2 Analysis

The graphs discussed in Section 3.3 indicate that using optimizations provided by MKL, the throughput of DNNs run on TensorFlow is improved by at least an order of magnitude, which is also indicated by the upward movement Roofline trajectories crossing the Peak 256 line and getting ever so closer to the theoretical peak of the Peak 512 line. Further analysis of the data is performed to understand the exact cause of this improvement and if further insights can be obtained leading to even greater benefits.



(a) Parameter Server (b) Horovod

Figure 3.11: OI breakdown of MKL enabled TensorFlow worker with PS and Horovod. There is a significant decline in memory accesses compared to vanilla TensorFlow (Figure 3.6(a)). As a result the OIs are orders of magnitude higher.



(a) Parameter Server (b) Horovod

Figure 3.12: Breakdown of FP operations performed by MKL-enabled TensorFlow worker with PS and Horovod. MKL-enabled TensorFlow almost exclusively makes use of 512-bit SP vectorized instructions resulting in a much higher FP operations count than that observed with vanilla TensorFlow (Figure 3.12(a)).

Comparing Figure 3.3 and Figure 3.8, we can see that the OIs obtained with the same experiments run with MKL-enabled TensorFlow are higher than those obtained with vanilla TensorFlow. That is because, as depicted in Figure 3.11(a) and Figure 3.11(b), the number of raw FP operations executed by vanilla TensorFlow are much higher than those of Intel-TensorFlow, no matter which gradient update layer is used. However, Intel-TensorFlow performs fewer memory accesses to execute the same number of floating-point operations which results in it having much higher OI, with both PS and Horovod approaches.

It can be observed from Figure 3.12(a) and Figure 3.12(b) that MKL enabled TensorFlow almost exclusively performs 512-bit SP floating-point instructions, which helps it to achieve FLOPS much closer to the theoretical peak. We also see that vanilla TensorFlow mostly uses 256-bit SP instructions where one such instruction performs 8 actual FLOPs. Intel-TensorFlow, on the other hand, almost exclusively uses 512-bit SP instructions which are equal to 16 FLOPs. Even though the magnitude of 256-bit SP instructions executed by vanilla TensorFlow is much higher than 512-bit SP instructions executed by Intel-TensorFlow, the actual number of FP instructions executed does not vary much, as is evidenced by Figure 3.11(b).

3.3.3 Insights

Initially, the experiments with MKL were executed with the worker PPN set to 4. However, even though it yields better performance than other values of PPN, the gains in speedup were not nearly as much as expected from using vectorized instructions. We suspected that thread management might be an issue. A cursory analysis indicated that, with default configurations, as many as 200 threads were launched by TensorFlow per worker. As MKL enabled TensorFlow makes use of OpenMP threads to distribute workload among

all the cores available, launching extraneous threads has detrimental effects on the performance. We did some further experimentation and concluded that with MKL enabled, the case of worker PPN setting to 2 can give the best performance which is what we have used for all experiments described in this section. This has been verified by other studies as well[9].

Guidelines provided by Intel to get the best performance also suggest using channel first NCHW *(Batch Size, Channel, Height, Width)* data encoding format instead of the channel last NHWC *(Batch Size, Height, Width, Channel)* data encoding format, which is the default data format used for all experiments described in this study. We have performed a number of experiments with the recommended NCHW format as well however we could not observe any observable benefits. It should be noted that this does not necessarily indicate that there are no performance gains to be had from using the NCHW format for training on CPUs. Instead, it merely indicates that for the experiments performed in this study, the choice of data format could not have significant effects.

3.4 Application-aware Optimizations

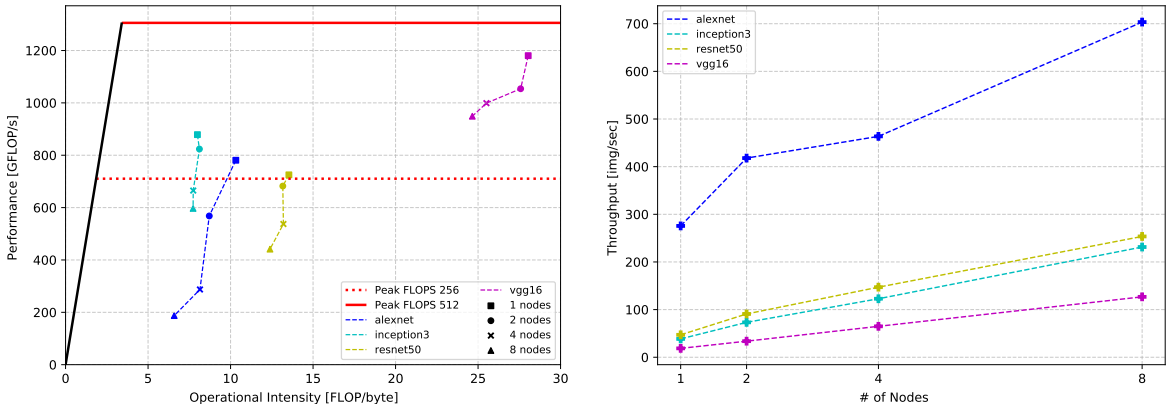
This section describes our attempts to tweak the performance derived from using vectorized instructions in TensorFlow even further.

3.4.1 Profiling

The black box optimizations performed in the previous section yield performance enhancements for all DNNs tested. However, the gains are less pronounced for Alexnet than for other models. To address this discrepancy, characteristics particular to Alexnet have

[9]https://software.intel.com/en-us/articles/maximize-tensorflow-performance-on-cpu-considerations-and-recommendations-for-inference

to be considered. Knowing that network communication significantly impacts the performance of Alexnet, we decide to increase the batch size of the input data to 128. This is done particularly to improve the performance of Alexnet. From the perspective of the DNN model itself, increasing the batch size results in more data being processed before gradients have to be synchronized. This should be helpful for models such as Alexnet with a large number of network parameters spread out over not as many layers.



(a) Roofline plot

(b) Throughput

Figure 3.13: Roofline plots and throughputs using Horovod and batch size=128. Increased batch size benefits Alexnet the most, leading to an upward shift by more than 200 GFLOPS for each Roofline data point compared to similar experiments with batch size=64 (Figure 3.10(a)). This translates to improvement in throughput as well with 2X improvement at 8 nodes compared to Figure 3.10(b).

Figure 3.13 shows the Roofline plots using Horovod for gradient update with batch size increased to 128. As can be seen from the graph, increasing the batch size significantly improves the throughput obtained by Alexnet which is 3.5X improvement compared to vanilla Tensorflow with 1 PS and 8 Nodes for workers (700 img/sec vs 200 img/sec). This indicates that because Alexnet is more communication-bound than other models, when we

increase the batch size, the gradients have to be updated after relatively larger amounts of data is processed thus amortizing the communication costs over a longer stretch of computation. The increase in throughput is not as pronounced for other models indicating their implementations overlap computation and communication to a reasonable degree.

3.4.2 Analysis

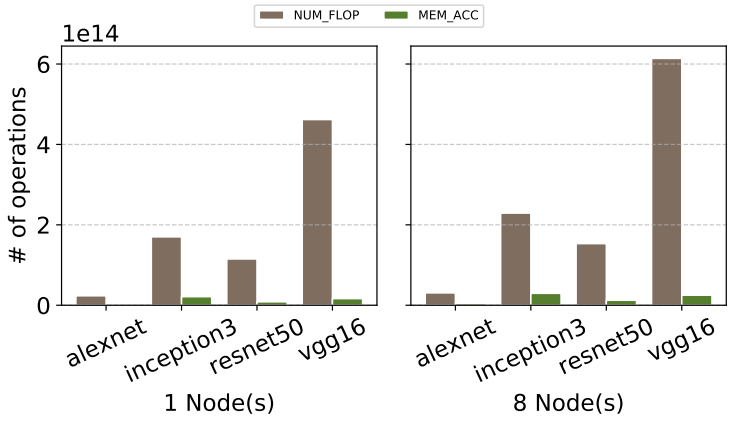


Figure 3.14: OI breakdown of MKL enabled TensorFlow worker with Horovod and batch size=128. The reason for improvement in the performance of Alexnet can be seen here as the FP operations performed increase by a factor of two compared to experiments with batch size=64 (Figure 3.11(b)).

Figure 3.14 shows the breakdown of OIs for experiments with batch size set to 128. Focusing on Alexnet and comparing its OI breakdown results with batch size set to 64 in Figure 3.11(b), we observe that the number of floating-point operations performed gets increased by a factor of two when the batch size is set to 128. However, memory accesses remain constant. As a result, there is an upward shift in not only the Roofline trajectories of all models but also an increase in the application throughput obtained, most pronounced with Alexnet.

3.5 Related Work

The original Roofline [60] paper suggests various optimizations that could be performed on workload bound by memory bandwidth and/or computational power and applies them to traditional scientific workloads. Since then it has been used to profile and optimize various architectures such as Intel KNL [16], NVIDIA GPUs [38], Google TPUs [28] and applications, including but not limited to, disaster detection [46], large scale simulations [33], wireless network detection [51] and even matrix multiplication [34].

There have not been many studies analyzing and profiling the performance of distributed Deep Learning. The few that exist [7, 53, 42, 32] do not focus on Roofline model based performance analysis and optimizations on CPUs. [7] and [53] analyze the overall execution time of different frameworks but do not discuss techniques for performance improvement. In [32], the authors use Alexnet alone as a representative model to analyze the performance of different distributed Machine Learning frameworks using GPUs. They can speed up training by 2X using only framework-specific options. However, their study does not include the effects of gradient exchange between nodes over a network. Recently, the Roofline model has also been applied to analyze the performance of Deep Learning models, mostly focusing on bespoke FPGA accelerator implementations, such as in [63] and [43].

Chapter 4: Characterization of Kafka and Architectural Details of Frieda

4.1 Analysis and Modeling

In this section, we first use the Roofline model to analyze the raw compute throughput generated by Kafka running at different levels of concurrency and then use that to identify potential bottlenecks. From there on, we dive deeper into the causes of the bottlenecks and the ways they could be mitigated.

4.1.1 Analysis

To start our analysis, we conduct a basic experiment where we run a typical Kafka workload with varying levels of concurrency for the number of producers participating in the experiments and generate a Roofline plot to see how the performance at the broker end varies as a result. The results of this experiment are shown in Figure 4.1.

We observe a steady increase in raw performance at the broker end as the number of concurrent producers are increased from 2 to 8, which is because there is a proportional increase in the work that the broker has to do log and persist the messages coming from each client. However, at 16 producers, we observe a sharp decline. At this level of concurrency, Kafka becomes overwhelmed by the sheer amount of messages coming it's way

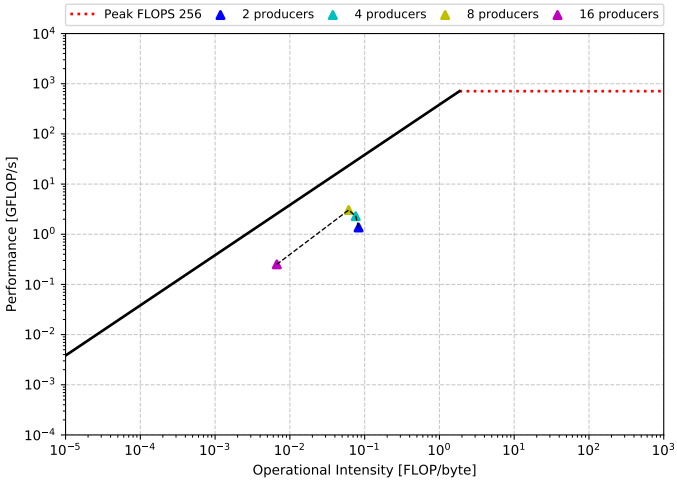


Figure 4.1: Roofline plot of Kafka producer benchmark with varrying levels of concurrent producers

and is unable to process them within reasonable latency bounds causing the entire system to just crash. This tells indicates that there is a bottleneck in Kafka's design which gets magnified at higher levels of concurrency, However, we need to dig deeper to first identify the bottleneck and discuss its possible solutions.

There are a variety of reasons why observed latencies of big data pipelines, and stream processing pipelines in general, may vary on a spectrum. However, it is possible to design a pipeline where the latency remains constant throughout the application's life cycle. Figure 4.2(a) describes such a hypothetical system with a sender sending messages to a receiver which does some processing and sends out a corresponding response. Both the sender and receiver allocate send and receive buffers to store the messages before they are flushed. However, in this case, requests are sent by the sender in a completely sequential manner i.e. a new request is only sent after the response for the previous request has been

received. This allows both the sender and receiver to reuse the send and receive buffers, as a result, there is no extra cost of transferring messages as once the pre-allocation and preparation of buffers are done for the first message, there is no need to do it again. Consequently, the latency of this pipeline remains constant as $T_1 = T_2 = T_3$.

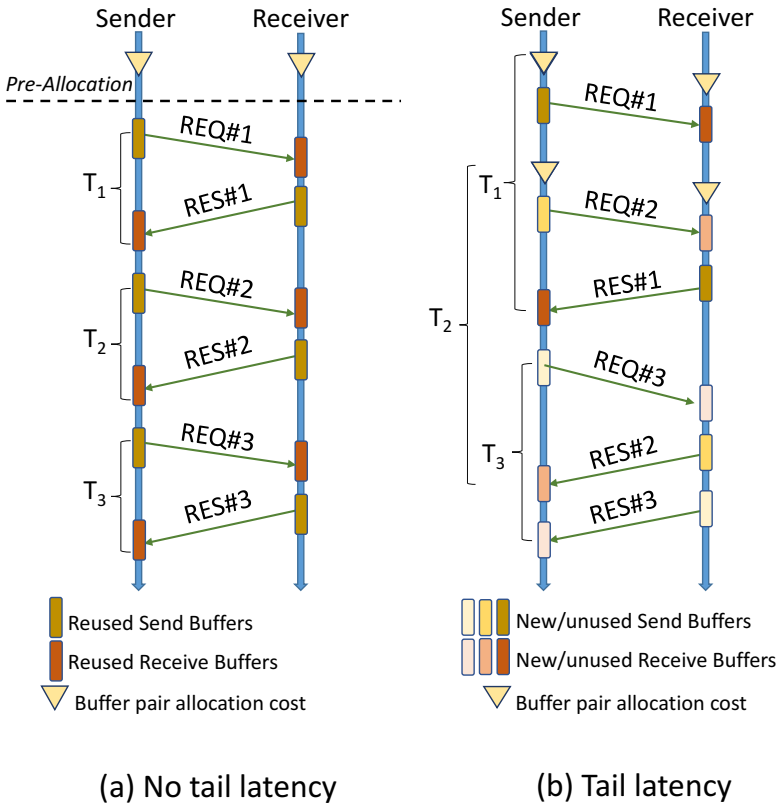


Figure 4.2: A scenario which results in tail latency

Even though latency guarantees provided by the system mentioned above are highly desirable, the throughput provided by such a system is untenable as such a system has no message transfer overlap. However, for practical systems, the transfer of messages has to be overlapped to ensure the system has acceptable throughput guarantees. In order to support

overlap of message transfer, resource contention is an unavoidable side effect, as is shown in Figure 4.2(b). Messages are sent one after the other, without having to wait for their responses. However, new send/receive buffers have to be allocated at both the sender and the receiver each time a message is transferred. This system has a higher throughput than the one in Figure 4.2(a), however, the overhead incurred compounds as more messages are sent. Consequently, we observe that the message latencies in such vary on a range, as is shown in the figure where $T_1 \neq T_2 \neq T_3$ i.e. when the size of the messages is similar. The latency for REQ#2 i.e. T_2, can be considered to fall on the tail-end of the latency spectrum. One important thing to note here is that the reuse of buffers is one of the many optimizations that can be done in a system where there is no contention for resources (Figure 4.2(a)). This includes buffer management, message reordering overheads, I/O scheduling, etc.

4.1.2 Modeling

As is described in the above Section 4.1.1, it is possible to design a distributed system with little or no variation throughout its latency spectrum i.e Figure 4.2(a). The average (T_{avg}) and tail latency (T_{tail}) for such a system, as represented by the 99th percentile, can be expressed formally using Equations 1 and 2 respectively.

$$(1) \quad T_{avg} = (\sum_{i=1}^{N}(L + \frac{MsgSz_i}{B}))/N$$

$$(2) \quad T_{tail} = Max_{i=1..N}^{99th} \left\{L + \frac{MsgSz_i}{B}\right\}$$

L and *B* denote the latency and the bandwidth of the underlying network whereas the size of each message is represented by *MsgSz* and *N* represents the number of messages

sent. As messages in such a system are in a strictly sequential manner, the T_{avg} is essentially just the time taken to transfer each message, averaged over all the messages sent (N). T_{tail}, on the other hand, is represented by the latency of the 99th percentile message processed ranked in order of increasing latency.

However, such a system is impractical for real-life applications due to the poor throughput it offers. As a consequence, practical systems have a degree of overlap in the manner in which messages are sent and received, whereby messages of certain window size (W) will be sent in an asynchronous manner. The $T^{'}_{avg}$ and $T^{'}_{tail}$ of such a system, as shown in Figure 4.2(b), can be elaborated using Equations 3 and 4.

$$(3) \quad T^{'}_{avg} = \left(\sum_{i=1}^{N/W} \left(L + \frac{\sum_{j=1}^{W} MsgSz_j}{B} \right) / W \right) \; / \frac{N}{W}$$

$$(4) \quad T^{'}_{tail} = Max^{99th}_{i=1..N/W} \left\{ Max^{99th}_{j=1..W} \left\{ L_j + \frac{MsgSz_j}{B} \right\} \right\}$$

For such a system, $T^{'}_{avg}$ is the latency of sending all the messages in any one window, averaged over all windows, the number of which is denoted by (N/W). Similarly, $T^{'}_{tail}$, can be calculated by first obtaining the 99th percentile latency of messages in each window, and then determining the 99th percentile of this set of latencies.

The reason why messages within a window might have varying latencies and thus contributing to a more pronounced tail of the latency distribution, is explained in Equation 5.

$$(5) \quad Max^{99th}_{j=1..W} \left\{ L_j + \frac{MsgSz_j}{B} \right\} = L + \frac{MsgSz}{B} + O(W, R)$$

The latency of a message *j* in a window is given by $L + \frac{MsgSz}{B} + O(W, R)$. The $L + \frac{MsgSz}{B}$ is common from Equation 2. However, there is a *contention overhead*, denoted by $O(W, R)$, to ensure the overlapped transfer of messages. This overhead is a function of both the number of messages transferred in that window (*W*), as well as the contention in the resources (*R*) which the sending of these messages in an asynchronous fashion incurred. Buffer management, as described in earlier sections, is just one of such overheads that could be incurred and are going to exist in every streaming pipelines, regardless of the frameworks utilized to implement the pipeline. In subsequent sections, we examine what these overheads are in a Kafka based streaming pipeline.

4.2 Design

This section first identifies the bottlenecks in vanilla Kafka implementation and then describes the design of Frieda which attempts to mitigate the effect of these bottlenecks.

4.2.1 Sources of Tail-Latency

In Section 4.1.1, we described how certain design choices in data processing frameworks could contribute to varying message processing latencies. In this section, we will discuss, from the perspective of Kafka as Message Broker in a stream processing pipeline, what are the design choices that end up being a source of uneven latencies. Figure 4.3 gives an overview of these sources.

① **TCP/IP:** Kafka uses TCP/IP stack for communication between its various components. As is shown in [19], network I/O over TCP/IP is slow as it involves context switches from user space to kernel space which is detrimental for low-latency applications. Moreover, as the network-related bookkeeping tasks such as flow/congestion control and

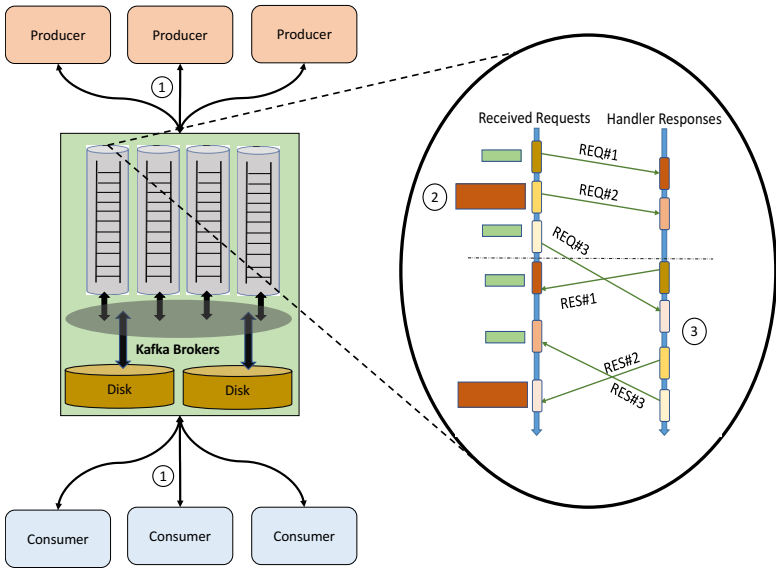


Figure 4.3: Sources of Tail Latency in Kafka

handling of unordered messages has to be handled by the CPU, certain messages end up waiting for more than others for them to be processed by CPU.

② **Handling Ordered Responses:** Kafka provides guarantees to its clients that all the requests will be responded to in-order. To ensure that, Kafka processes each request from a client in sequential order i.e. a new request will not be offloaded to handler threads to be processed until the response for the previous request has been sent. As a result, some messages end up waiting for more than the others before they are being processed and handled by Kafka.

③ **Buffer Management:** For every new message that is sent or received by Kafka, a buffer is created on the fly. Operations required for preparing memory buffers, such as

allocation and regsitration [57][45], are known to be costly operations affecting the performance of applications. This issue has already been explained in detail in Section 4.1 using Figure 4.2.

4.2.2 High-Performance design of Frieda

We design Frieda from the ground up to use the high-performance, RDMA-based communication engine. We also implement some key designs ensuring the bottlenecks contributing to Kafkas's subpar performance are resolved in our framework. The use of RDMA as Frieda's communication paradigm implies that we offload all the network I/O related tasks to the Host Channel Adapter (HCA). This not only allows the CPU to focus more on Kafka's framework-specific tasks but also paves the way for a cleaner implementation of sophisticated designs to restrict tail latencies. A high-level view of Frieda's architecture is shown in Figure 4.4.

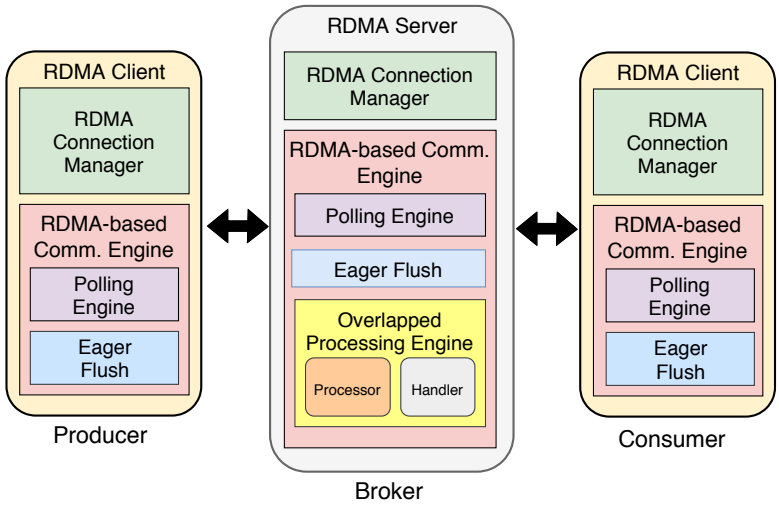


Figure 4.4: High Performance Design of Frieda

In the following subsections, we will describe advanced architectural designs incorporated in Frieda to drastically reduce tail latencies in the pipeline.

Dynamic Eager Flush

Frieda's communication runtime is based on a Dynamic Eager Flush policy. We implement an abstraction of RdmChannel, which performs the same role as SocketChannel in TCP based Kafka. However, it gives us fine-grained control of the exact semantics according to which messages can be written to the wire. The Kafka protocol ensures that each message in Kafka is prepended by 4 bytes representing the size of the overall message. Every time a new message's header is written to Kafka, we use this information to keep track of the size of the entire message. Kafka's higher-level writes to channel in distinct invocations of RdmaChannel's multiple write methods. The tracker notifies the RdmaChannel as soon as the full message has been collected. When this event occurs, we flush the message to the HCA which in turn immediately writes it to the wire. Unlike for Kafka, we do not have to send or receive buffers of a predefined size which have to be filled before the messages are eventually flushed. This Dynamic Eager Flush design helps remove the bottleneck of unnecessary waiting of messaging in socket queues which we observed in Kafka's design.

Continuous Polling

Another design feature in Frieda to ensure its sensitivity to tail latency is what we call the Continuous Polling design. We implement a dedicated Polling Engine whose sole responsibility is to poll the HCA for new messages coming in from any of the connected RdmaChannels. This ensures that the Processors need not poll for new messages at each iteration of their thread loop. Instead, the Polling Engine distributed the messages in queues

unique to each processor thread thus overlapping the process of polling for new records and performing other operations.

Adaptive Buffer Management

In order to ensure that buffer management does not become a performance bottleneck for Frieda, we implement an approach of pre-registering a pool of buffers of different sizes. Whenever a new buffer is required at the application layer, we intercept the request and return an idle buffer from our pool which closely matches the required size. When a buffer completes its lifecycle through the application logic, we put it back to the pool. In this, we ensure that once buffers are pre-registered at the start of the application, there is no further cost of issuing buffers for new requests.

Hardware-offloaded Message Ordering

Frieda is implemented using the RDMA RC protocol which ensures the ordering of messages, performed by the HCA. As a result, Frieda ensures that the CPU is free from performing the expensive task of message re-ordering. However, there is a catch to using this approach. As shown in this paper [41], in order to achieve the optimum performance from RDMA, an approach for selecting between Eager and Rendevous protocol based on message size has to be adopted. As shown in Figure 4.5, this approach leads to cases where even though a Request to Send (RTS) for a large message is received before subsequent small messages, the actual transfer of these small messages finished before the prior large message. That is because the small messages are sent using the Eager protocol where the message itself is piggybacked with its header and delivered in just a single message. The large message, on the other hand, first has its RTS delivered to the remote side which then does on RDMA READ from the sender processes memory. When this is done, the remote

process sends the sender the FIN message indicating the message transfer has finished. However, during the time this Rendezvous operation takes place, the messages delivered using the Eager protocol are already received at the remote side and as a result, the remote side receives messages in a different order than how they were sent by the sender. Similar out-of-order scenarios may happen for RDMA Write based rendezvous protocols as well.

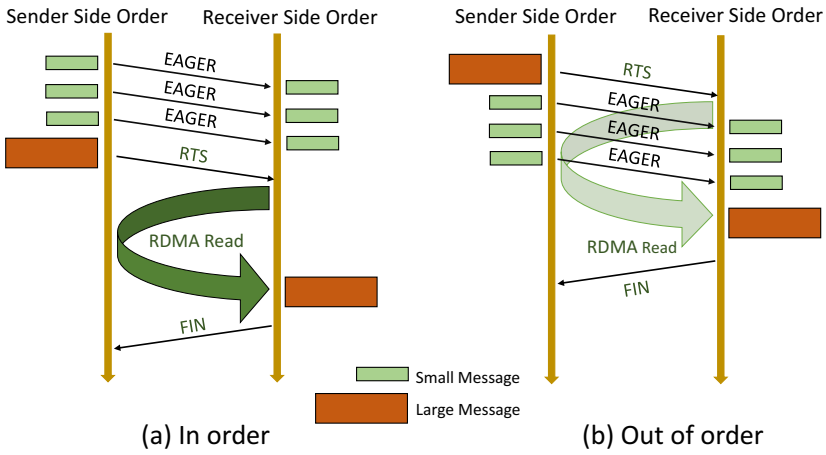


Figure 4.5: Un-ordered Messages with Eager and Rendezvous protocols

In order to resolve this issue and also process the incoming messages in a parallel manner, we introduce an Asynchronous Message Processing Engine (AMPE). Using AMPE, we remove the bottleneck of having to process messages from each client in a sequential manner to ensure ordered delivery of responses. In order to achieve this, each new request is offloaded to a handler thread as soon as it is received, but we maintain a data structure that keeps track of the order in which new requests are received by putting them in appropriate slots dictated by their sequence IDs. When the handler threads return a response for a particular message, it is put in its corresponding request's slot. Consequently, when the

processing thread comes around to sending the responses, it polls the data structures for the first contiguous set of requests for which responses have been put in the slots. This not only ensures that the processing of messages is parallelized but guarantees that the responses are delivered in order.

Overlapped Processing

Unlike Kafka, we design Frieda in a way so as to allow it to process multiple requests in parallel. Using our Overlapped Processing design, we avoid processing requests from clients in a sequential manner to maintain message order. Instead, we process multiple requests from a client in an overlapped fashion. To ensure the ordered delivery of responses, we keep a track of the sequence in which messages from a particular client were received while associating an empty response bucket with each of them. These requests are then dispatched to the handler threads without requiring them to wait. The fully formed responses are then put in the buckets of appropriate requests maintaining their order. When the processor threads in their event loop come around to actually writing completed responses, we write the first set of contiguous responses that have been completed. In this way, we not only maintain the order in which the requests are responded to but also enable the overlapped processing of messages by the broker.

As shown by our evaluations in the subsequent section, these designs help in drastically reducing the message processing time, not only in terms of tail latencies but across the spectrum.

Design Choices: While designing a distributed system, the decision of whether to implement certain functionalities on the client or the server is a crucial one. In our case, most of the optimizations mentioned above could be implemented on both the broker and the client end. However, such a discussion is more nuanced when it comes to ensuring Overlapped

Processing in the pipeline. As a consequence of Overlapped Processing and the fact that Kafka clients expect ordered responses to their requests, there is a need to reorder the responses produced by handler threads at the server. We first implemented the bucket based message reordering logic mentioned above at the broker end. However, this did not yield good performance as the broker becomes an overburdened entity in the pipeline ensuring message ordering for all the participating clients. We decided to stick with Kafka's design philosophy of having a stateless server and pushing all such extraneous logic to the clients. This not only guarantees the correctness of the system but also ensures high performance in the entire pipeline.

4.3 Evaluation and Results

We use a two-pronged strategy to evaluate Frieda. First, we perform some microbenchmarks to evaluate the performance of basic message broker operations. Then, we plug in Frieda in a full-fledged streaming pipeline using HiBench to evaluate the benefit of having a high-performance Message Broker on the entire pipeline. The experiments were carried on two clusters, an in-house OSU-RI2 (Cluster A) and SDSC Comet (Cluster B), the specifications of which are summarised in Table 4.1. Software and their corresponding versions used for the experiments performed are described in Table 4.2. Smaller-scale experiments were run on Cluster A whereas Cluster B was used for larger-scale runs. In all the experiments mentioned below, the broker, as well as each of the producers, was launched on separate nodes. A colocated Zookeeper was launched for each of the brokers used in the experiment.

	Cluster A	Cluster B
Nodes	20	1,984
CPU	2.4GHz 14 cores x2	2.5GHz 12 cores x2
Memory	512 GB	128 GB
Disk	400 GB SSD & 4 x 2 TB HDD	320 GB SSD & 80GB HDD
HCA	IB EDR (100Gbps)	IB FDR (54Gbps)
OS	CentOS release 7.2	CentOS release 6.7

Table 4.1: Cluster Configuration

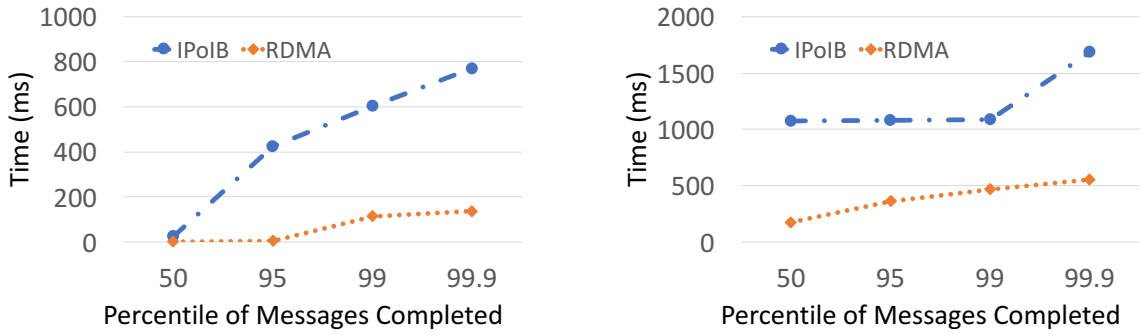
Software	Version
Apache Kafka	0.11.1.0
Apache Zookeeper	3.4.8

Table 4.2: Software Configuration

4.3.1 Microbenchmarks

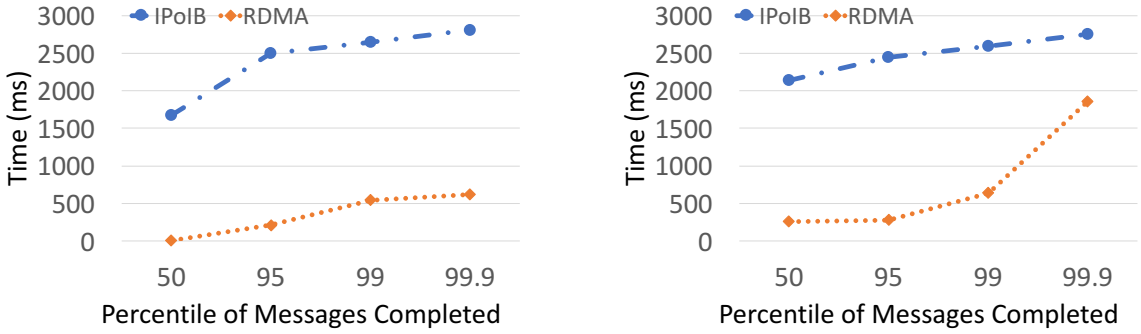
Kafka comes bundled with a utility to evaluate producer performance and we use the same utility to compare the performance of Kafka and Frieda. The utility operates by writing messages of configurable message size and replication factor. For every record that is sent by the producer, the broker sends out an acknowledgment, and the time it takes for a record to be sent to the broker and its acknowledgment to arrive back at the producer considered to be the latency of the record. As an RDMA implementation for Kafka is not available, we run Kafka using the Internet Protocol over InfiniBand (IPoIB) [12] over the InfiniBand network. Note that in all the graphs presented, the legends are marked based on the network protocol that is used by each framework i.e. Kafka utilizes IPoIB whereas Frieda makes use of RDMA.

1 Broker



(a) Small Message (100B) (b) Large Message (10000B)

Figure 4.6: Percentile Latencies for 2 Producers

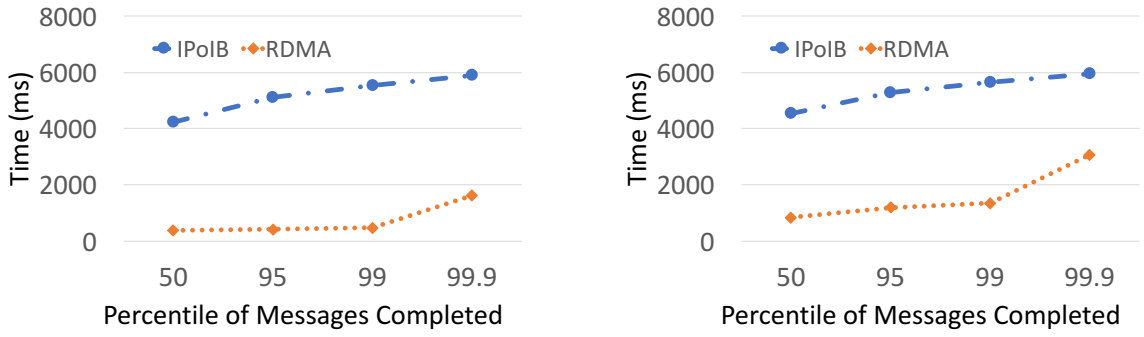


(a) Small Message (100B) (b) Large Message (10000B)

Figure 4.7: Percentile Latencies for 4 Producers

In the first step, we have a single broker - multiple producer setup wherein for different cluster sizes we vary the size of messages emitted to evaluate how latency varies in response. These experiments were carried on Cluster A. Figure 4.6, 4.7, 4.8 and 4.9 show the results of these experiments in terms of percentile latencies. As expected, increasing the number of Producers significantly increases the latency of the pipeline. However, the latency in Frieda degrades gracefully as opposed to vanilla Kafka. We observe significantly reduced tail latencies for Frieda throughout these experiments. For 2 concurrent clients, we

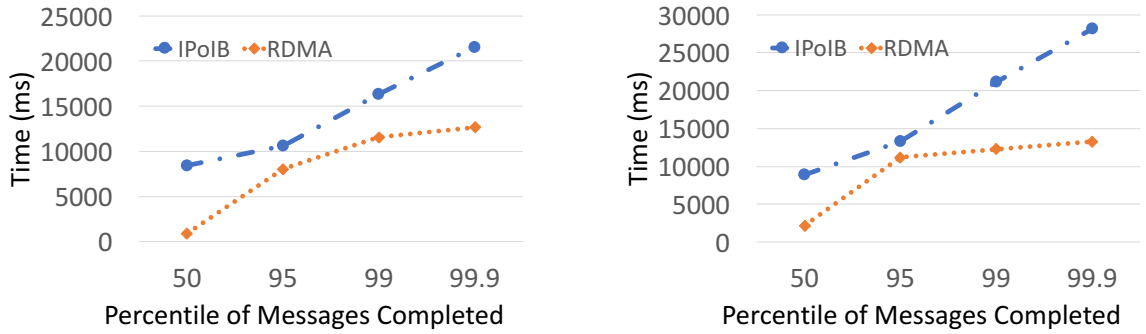
see a decrease of 82% and 67% in 99.9th percentile latency of Frieda compared with Kafka for small and large messages respectively. This trend continues as we increase the number of concurrent producers, getting improvements of 78% and 33%, 72% to 48% and 41% to 53% in 99.9th percentile latency of small and large messages for 4, 8 and 16 Producers respectively.



(a) Small Message (100B)

(b) Large Message (10000B)

Figure 4.8: Percentile Latencies for 8 Producers



(a) Small Message (100B)

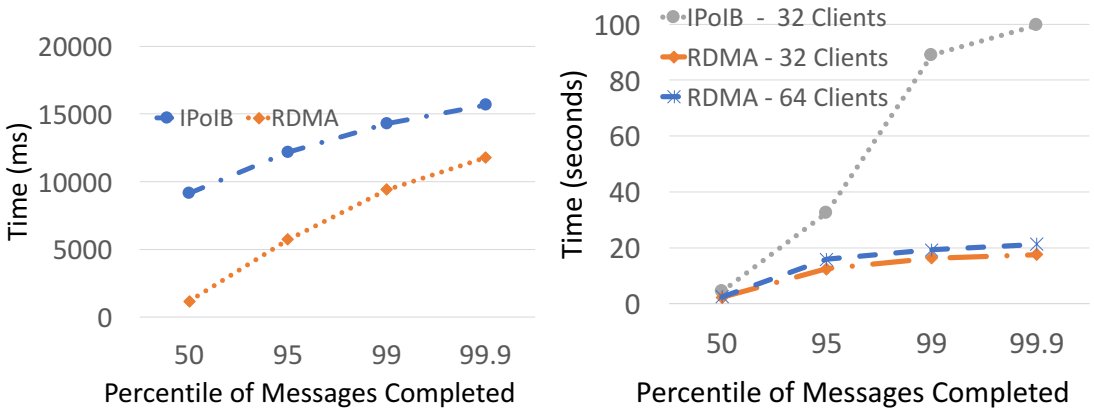
(b) Large Message (10000B)

Figure 4.9: Percentile Latencies for 16 Producers

4 Brokers with Replication

In most cases, production environments set up a replicated cluster of Kafka brokers whereby each record written to anyone broker is replicated to K number of other brokers.

This value of K is configurable using the *replication-factor* property set during topic creation. This is done to ensure the durability of the committed records, such that even if only one of the replicated brokers is alive, the cluster would still be able to function though with compromised performance.



(a) 16 Producers (b) 32 and 64 Producers

Figure 4.10: Latencies with 4x Replication

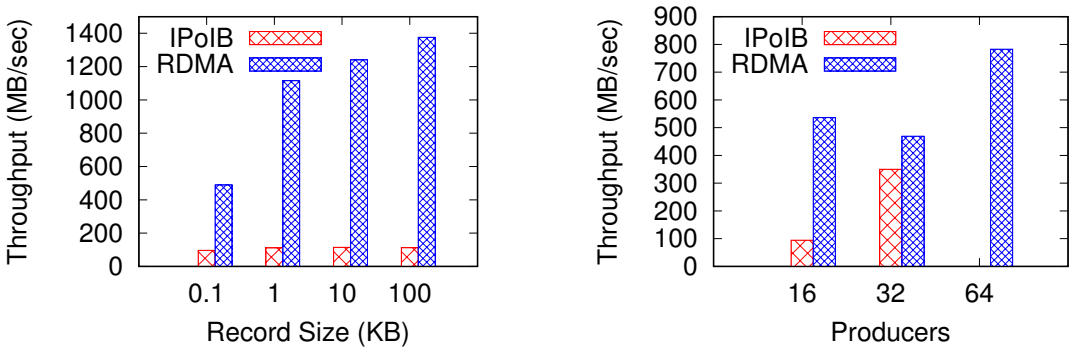
In order to test our Frieda in this environment, we set up a cluster of 4 Kafka brokers while also setting the *replication-factor* = *4*, implying that every message has to be replicated to all 4 Kafka brokers for it to be considered fully committed. In this experiment, we also try to evaluate how well does Frieda scales compared to vanilla Kafka and in order to do so, we take large-scale numbers using 16, 32 and 64 producers, respectively. As the effect of message size on the performance has been thoroughly evaluated in the first set of experiments, we restrict this experiment to a payload of 100 bytes, which can be considered a representative message size for a typical streaming processing environment. Cluster B was used to perform these experiments. The effects on percentile latency of these experiments are shown Figure 4.10(a) and 4.10(b) respectively. For 16 concurrent clients in a

replicated setting, we see a reduction of about 25% in 99.9th percentile latency for Frieda. However, as we increase the number of producers to 32, we observe that the percentile latencies for Kafka explode while Frieda's performance is still commendable, showing a decrease of about 98% in 99.9th percentile latency compared to Kafka. The degradation in performance for Kafka is so severe that with 64 concurrent producers, Kafka Brokers quickly become overwhelmed by the huge amount records needing to be processed causing producers to throw timeout exceed warnings as multiple records fail to be acknowledged by brokers within a specified duration. Even after multiple runs, we were unable to finish the experiment for Kafka for this configuration so we show the results for Frieda only for which the performance for this configuration is comparable to 32 concurrent producers. This behavior of Kafka can be attributed to the fact that TCP based communication, which Kafka is based on, takes away precious CPU cycles leaving not nearly enough resources for the application to operate at the desired performance level. To see if the default Kafka implementation would fare any better in the case of 64 clients but with replication disabled, we restart the experiment with *replication-factor = 1*. However, even in these somewhat relaxed circumstances, the default Kafka cluster was unable to cope with such a high rate of incoming messages.

Throughput Analysis

Although minimizing latency is the main goal of our work, our designs do not compromise on throughput for improved latency. We present the results of producer benchmark in terms of aggregated throughput for both single and multi-broker cluster in Figure 4.11(a) and 4.11(b). Increasing the number of producers increases the total throughput of the Kafka broker, up to the point where the broker is saturated to its processing limits. We observe, from Figure 4.11(a), an increase in throughput of 4x to 10x for different message sizes for

8 Producers respectively. Similar improvements are observed for the multi-broker environment as shown in Figure 4.11(b). Similar to the earlier latency based results, we were unable to obtain throughput numbers for 64 producers for Kafka as it became unresponsive under the extensive workload.



(a) 1 Broker with 8 Producers

(b) 4 Brokers with 32/64 Producers

Figure 4.11: Throughput with different cluster configurations

This improvement primarily originates from the NIC offloaded nature of RDMA communication, where host CPUs are not involved in the actual process of data communication from the network, thus allowing them to spend more clock cycles on non-network tasks.

4.3.2 Yahoo! Streaming Benchmark

After ensuring enhanced performance for producer based operations across various cluster configurations and message sizes, we test Frieda with a real-life streaming application. For this purpose, we use the Yahoo! Streaming Benchmark (YSB) [11]. The benchmark uses a streaming pipeline based on an advertisement application. Kafka producer(s) emits messages containing data pertaining to different advertisement campaigns to a Kafka cluster. An SPE than fetches those messages, deserializes them and filters out a set

of events based on a predefined predicate. It then takes a projection of the relevant field and performs a join on them with data contained in an in-memory Redis instance. This database is used to maintain a windowed count of relevant events processed per campaign. Finally, the latency is calculated as the time from the start of a window until the acknowledgment for the last event in the window is received. This benchmark was originally designed to test the performance of various SPEs in a typical streaming pipeline. The SPEs that can be benchmarked with this application include Storm, Spark Streaming, Flink as well as Kafka Streams. Kafka Streams[10] is a streaming framework built on top of Kafka. For our experiments, we were able to seamlessly integrate Frieda in the streaming pipeline by providing required client libraries used to read and write data for the consumers and producers in the pipeline, respectively.

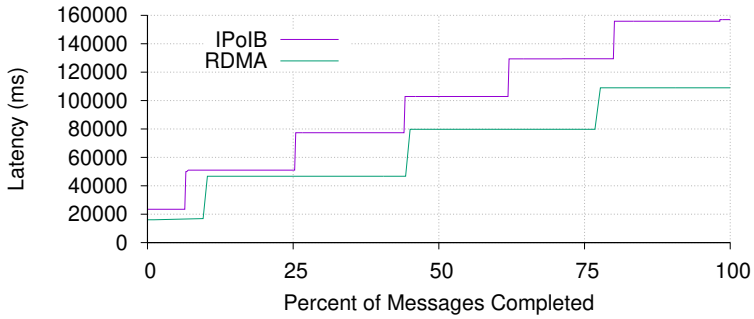


Figure 4.12: Latency distribution of Kafka and Frieda with YSB

For this experiment, we used a single broker with 8 concurrent producers emitting data at an aggregated rate of 120,000 records/sec. The results of this experiment are shown in Figure 4.12. The jumps in latency for Kafka based IPoIB and Frieda based RDMA

[10]https://kafka.apache.org/documentation/streams

experiments are due to the *committime=1000ms* configured for both runs. Kafka Streams uses *committime* parameter to specify after how much time would the results of a streaming pipeline will be committed to a persistent store. This is similar to the mini-batch adopted in Spark Streaming [62]. The lower end of the latency spectrum is similar for both Kafka as well as Frieda with the latter having slightly lower latencies. However, from the 50th percentile onwards, Frieda significantly outperforms Kafka as shown by a reduction of about 31% in 99 percentile latency.

4.4 Related Work

Researches on streaming Big Data processing and the associated processing engines have been performed in the literature. Most of these studies typically focus on investigating the performance impact on end-to-end average latency of the streaming pipeline in response to varying input message rate. Qian et al. [49] compare the performance of Spark Streaming, Storm, and Samza on standardized micro-benchmarks such as grep, projection, etc. Chintapalli et al. [11] evaluate the performance of Storm, Spark Streaming, and Flink using an application designed to mimic a real-life stream processing scenario. Both of these studies base their results on end-to-end average latency of the streaming pipeline. Moreover, these studies are limited to low-speed networks, such as 1GigE. Inoubli et al. [25] perform an exhaustive performance analysis of streaming as well as batch frameworks. They evaluate Spark, Storm, Flink, and Hadoop on the metrics of performance, scalability and resource utilization. The performance comparison focuses mainly on the framework's internal processing rather than the overall pipeline while also excluding the messaging middleware from the picture. In our previous work [26], we evaluated the performance of Flink

and Kafka on cutting-edge networking technologies such as 40GigE and InfiniBand EDR, which show the potential of using high-speed interconnects for streaming data processing.

Du and Gupta [17] implement adaptive timeout and load balancing techniques to curtail tail latency in Apache Storm. Suresh et al. [56] make use of a replica based scheme to minimize tail latency in their proposed system C3. Kamburugamuve et al. [30] use RDMA to improve the performance of Heron streaming framework and show considerable benefits. However, their implementation can not fit in an arbitrary stream processing pipeline as shown by the need to modify the Yahoo! Streaming Benchmarking (YSB). YSB, mimicking a generic stream processing pipeline, uses Kafka producers to generate data into the streaming pipeline. However, for this work, the authors had to implement custom Heron spouts to perform this task.

Recently, a lot of research works have been proposed in the field for accelerating Big Data stacks to fully take advantage of modern high-performance interconnects and protocols. Several research works have focused on exploiting advanced features like RDMA in key-value stores including RDMA-Memcached [27], RAMCloud [47], MICA [37], to improve the response time and throughput of the application servers deployed on HPC clusters. For offline data-intensive workloads such as Hadoop and Spark, high-performance solutions such as RDMA-Hadoop [50] and RDMA-Spark [40] have been designed to deliver much higher performance compared to just using the traditional TCP/IP or IP-over-IB based protocols. However, these studies are not focusing on streaming data processing and they do not pay attention to the problem that how to significantly reduce tail latency with RDMA technology.

Compared to the related work described above, this paper mainly focuses on how to significantly reduce both average latency and tail latency for the modern streaming data

processing pipeline by proposing several tools and technologies, including but not limited to RDMA. The motivation, design goals, and achievements are quite different than the related studies as listed above. To the best of our knowledge, this paper is the first work to propose native RDMA-based, latency-centric designs for message brokers from the perspective of a stream processing pipeline.

Chapter 5: Contributions and Future Work

In this study, we propose the use of the Roofline model to analyze various CNN models implemented in TensorFlow for CPU. We are able to identify various bottlenecks that allow us to significantly improve the performance of these CNN models. We hope that our study would be helpful for scientists, especially those who may not have enough knowledge of low-level systems, to optimize their Deep Learning model training processes and maximize the performance. Using various optimizations described in this study, we are able to achieve a maximum speedup in throughput of 3.5X for Alexnet at a concurrency level of 8 nodes.

Moreover, we proposed using high-performance message brokers to accelerate an arbitrary stream processing pipeline. To this effect, we designed and implemented Frieda, an RDMA-enhanced high-performance message broker with a plug and play interface for any arbitrary stream processing pipelines. Through our evaluations, we were able to show that tail latencies in a stream processing pipeline could be significantly reduced by the use of a message broker like Frieda specialized to deal with this issue. As more and more HPC resources get utilized by Big-Data applications, Frieda poses as an ideal Message Broker enabling streaming pipelines to best utilize the resources provided.

For future work, we would like to understand how the choice of network interconnect - InfiniBand, RoCE, High-Speed Ethernet, etc. - and the network channels used to communicate over them - gRPC, MPI, etc. - influence the performance of Deep Learning models.

Furthermore, we would also like to extend our study to incorporate the ever-expanding landscape of processing elements suitable for Deep Learning, such as GPUs, TPUs, and FPGAs. Moreover, since DL applications are notorious for their soaring power requirements, we would also like to explore if our approach can be used to generate optimizations that can reduce the energy consumption of the applications without incurring a significant performance penalty.

In addition to that, we would like to optimize our designs for large messages as these do not show as much benefit for small messages which are typical of stream processing. This would render Frieda usable for other use-cases where large messages are common such as log aggregation. Moreover, we would like to evaluate Frieda with SPEs other than Kafka Streams to determine if our designs our generic enough to provide benefits regardless of the SPE used.

Bibliography

[1] gRPC. http://grpc.io/.

[2] Intel-Tensorflow. https://github.com/Intel-tensorflow/tensorflow.

[3] RDMA over Converged Ethernet. http://www.roceinitiative.org/.

[4] InfiniBand Trade Association. http://www.infinibandta.org, Feb. 2017.

[5] NVIDIA NCCL. https://github.com/NVIDIA/nccl, Feb. 2017.

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A System For Large-Scale Machine Learning. In 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), pages 265–283, 2016.

[7] Soheil Bahrampour, Naveen Ramakrishnan, Lukas Schott, and Mohak Shah. Comparative study of caffe, neon, theano, and torch for deep learning. 2016.

[8] Rajarshi Biswas, Xiaoyi Lu, and Dhabaleswar K Panda. Accelerating TensorFlow with Adaptive RDMA-based gRPC. In 2018 IEEE 25th International Conference on High Performance Computing (HiPC), pages 2–11. IEEE, 2018.

[9] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting Distributed Synchronous SGD. arXiv preprint arXiv:1604.00981, 2016.

[10] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. CoRR, abs/1410.0759, 2014.

[11] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. Benchmarking streaming computation engines: Storm, Flink and Spark streaming. In Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International, pages 1789–1792. IEEE, 2016.

[12] Jerry Chu and Vivek Kashyap. Transmission of IP over Infiniband (IPoIB). Technical report, 2006.

[13] Arnaldo Carvalho De Melo. The New Linux perf Tool. In Slides from Linux Kongress, volume 18, 2010.

[14] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In Advances in neural information processing systems, pages 1223–1231, 2012.

[15] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In CVPR09, 2009.

[16] Douglas Doerfler, Jack Deslippe, Samuel Williams, Leonid Oliker, Brandon Cook, Thorsten Kurth, Mathieu Lobet, Tareq Malas, Jean-Luc Vay, and Henri Vincenti. Applying the Roofline Performance Model to the Intel Xeon Phi Knights Landing Processor. In International Conference on High Performance Computing, pages 339–353. Springer, 2016.

[17] Guangxiang Du and Indranil Gupta. New Techniques to Curtail the Tail Latency in Stream Processing Systems. In Proceedings of the 4th Workshop on Distributed Cloud Computing, page 7. ACM, 2016.

[18] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. Dermatologist-level Classification of Skin Cancer with Deep Neural Networks. Nature, 542(7639):115, 2017.

[19] Annie P Foong, Thomas R Huff, Herbert H Hum, Jaidev R Patwardhan, and Greg J Regnier. TCP Performance Re-visited. In Performance Analysis of Systems and Software, 2003. ISPASS. 2003 IEEE International Symposium on, pages 70–79. IEEE, 2003.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 770–778, 2016.

[21] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In USENIX annual technical conference, volume 8. Boston, MA, USA, 2010.

[22] Brody Huval, Tao Wang, Sameep Tandon, Jeff Kiske, Will Song, Joel Pazhayampallil, Mykhaylo Andriluka, Pranav Rajpurkar, Toki Migimatsu, Royce Cheng-Yue, et al. An Empirical Evaluation of Deep Learning on Highway Driving. arXiv preprint arXiv:1504.01716, 2015.

[23] Khaled Ibrahim, Samuel Williams, and Leonid Oliker. Performance Analysis of GPU Programming Models using the Roofline Scaling Trajectories. In 2018 International Symposium on Benchmarking, Measuring and Optimizing (Bench'19).

[24] Khaled Ibrahim, Samuel Williams, and Leonid Oliker. Roofline Scaling Trajectories: A Method for Parallel Application and Architectural Performance Analysis. In 2018 International Conference on High Performance Computing & Simulation (HPCS), pages 350–358. IEEE, 2018.

[25] Wissem Inoubli, Sabeur Aridhi, Haithem Mezni, and Alexander Jung. Big Data Frameworks: A Comparative Study. CoRR, abs/1610.09962, 2016.

[26] M. Haseeb Javed, Xiaoyi Lu, and Dhabaleswar K. Panda. Characterization of Big Data Stream Processing Pipeline: A Case Study using Flink and Kafka. In Big Data Computing, Applications and Technologies (BDCAT), 2017 IEEE/ACM International Conference on. IEEE, 2017.

[27] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md. Wasi-ur Rahman, Nusrat S. Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, and Dhabaleswar K. Panda. Memcached Design on High Performance RDMA Capable Interconnects. In Proceedings of the 2011 International Conference on Parallel Processing, ICPP '11, Washington, DC, USA, 2011.

[28] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Agrawal, et al. In-datacenter performance analysis of a tensor processing unit. In 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2017.

[29] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter Performance Analysis of a Tensor Processing Unit. In 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), pages 1–12. IEEE, 2017.

[30] Supun Kamburugamuve, Karthik Ramasamy, Martin Swany, and Geoffrey Fox. Low Latency Stream Processing: Twitter Heron with Infiniband and Omni-Path.

[31] Kate Keahey, Pierre Riteau, Dan Stanzione, Tim Cockerill, Joe Mambretti, Paul Rad, and Paul Ruth. Chameleon: a Scalable Production Testbed for Computer Science Research. Contemporary High Performance Computing: From Petascale toward Exascale, 2019.

[32] Heehoon Kim, Hyoungwook Nam, Wookeun Jung, and Jaejin Lee. Performance Analysis of CNN Frameworks for GPUs. In 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 55–64. IEEE, 2017.

[33] Ki-Hwan Kim, KyoungHo Kim, and Q-Han Park. Performance Analysis and Optimization of Three-Dimensional FDTD on GPU using Roofline Model. Computer Physics Communications, 182(6):1201–1207, 2011.

[34] Martin Kong, Louis-Noël Pouchet, and P. Sadayappan. A Roofline-Based Performance Estimator for Distributed Matrix-Multiply on Intel CnC. 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, pages 1241–1250, 2015.

[35] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In Advances in neural information processing systems, pages 1097–1105, 2012.

[36] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling Distributed Machine Learning with the Parameter Server. In 11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14), pages 583–598, 2014.

[37] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI '14), April 2014.

[38] Andre Lopes, Frederico Pratas, Leonel Sousa, and Aleksandar Ilic. Exploring GPU Performance, Power and Energy-Efficiency Bounds with Cache-aware Roofline Modeling. In 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 1–12. IEEE, 2017.

[39] X. Lu, H. Shi, D. Shankar, and D. K. Panda. Performance Characterization and Acceleration of Big Data Workloads on OpenPOWER System. In 2017 IEEE International Conference on Big Data (Big Data), pages 213–222, Dec 2017.

[40] Xiaoyi Lu, Dipti Shankar, Shashank Gugnani, and Dhabaleswar K DK Panda. High-Performance Design of Apache Spark with RDMA and Its Benefits on Various Workloads. In 2016 IEEE International Conference on Big Data (BigData '16), pages 253–262. IEEE, 2016.

[41] Xiaoyi Lu, Dipti Shankar, Shashank Gugnani, Hari Subramoni, and Dhabaleswar K Panda. Impact of HPC Cloud Networking Technologies on Accelerating Hadoop RPC and HBase. In Cloud Computing Technology and Science (CloudCom), 2016 IEEE International Conference on, pages 310–317. IEEE, 2016.

[42] Xiaoyi Lu, Haiyang Shi, Rajarshi Biswas, M Haseeb Javed, and Dhabaleswar K Panda. DLoBD: A Comprehensive Study of Deep Learning over Big Data Stacks on HPC Clusters. IEEE Transactions on Multi-Scale Computing Systems, 4(4):635–648, 2018.

[43] Paolo Meloni, Gianfranco Deriu, Francesco Conti, Igor Loi, Luigi Raffo, and Luca Benini. Curbing the Roofline: A Scalable and Flexible Architecture for CNNs on FPGA. In Proceedings of the ACM International Conference on Computing Frontiers, pages 376–383. ACM, 2016.

[44] Message Passing Interface Forum. `http://www.mpi-forum.org/`.

[45] Jeffrey C Mogul. TCP Offload Is a Dumb Idea Whose Time Has Come. In HotOS, pages 25–30, 2003.

[46] Kohei Nagasu, Kentaro Sano, Fumiya Kono, and Naohito Nakasato. FPGA-based Tsunami Simulation: Performance Comparison with GPUs, and Roofline Model for Scalability Analysis. Journal of Parallel and Distributed Computing, 106:153–169, 2017.

[47] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. ACM SIGOPS Operating Systems Review, 43(4):92–105, 2010.

[48] Pitch Patarasuk and Xin Yuan. Bandwidth Optimal all-reduce Algorithms for Clusters of Workstations. Journal of Parallel and Distributed Computing, 69(2):117–124, 2009.

[49] Shilei Qian, Gang Wu, Jie Huang, and Tathagata Das. Benchmarking Modern Distributed Streaming Platforms. In Industrial Technology (ICIT), 2016 IEEE International Conference on, pages 592–598. IEEE, 2016.

[50] M. W. Rahman, Xiaoyi Lu, Nusrat S. Islam, and D. K. Panda. HOMR: A Hybrid Approach to Exploit Maximum Overlapping in MapReduce over High Performance Interconnects. In International Conference on Supercomputing (ICS), Munich, Germany, 2014.

[51] Jahangir H Sarker, Mahbub Hassan, and Seppo J Halme. Power Level Selection Schemes to Improve Throughput and Stability of Slotted ALOHA under Heavy Load. Computer Communications, 25(18):1719–1726, 2002.

[52] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. arXiv preprint arXiv:1802.05799, 2018.

[53] Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. Benchmarking state-of-the-art Deep Learning Software Tools. In 2016 7th International Conference on Cloud Computing and Big Data (CCBD), pages 99–104. IEEE, 2016.

[54] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv preprint arXiv:1409.1556, 2014.

[55] Viswanath Subramanian, Michael R Krause, and Ramesh VelurEunni. Remote Direct Memory Access (RDMA) Completion, August 14 2012. US Patent 8,244,825.

[56] P Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In NSDI, pages 513–527, 2015.

[57] Bernhard Suter, TV Lakshman, Dimitrios Stiliadis, and Abhijit K Choudhury. Buffer Management Schemes for Supporting TCP in Gigabit Routers with per-flow Queueing. IEEE Journal on Selected Areas in Communications, 17(6):1159–1169, 1999.

[58] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 2818–2826, 2016.

[59] Lei Wang, Jianfeng Zhan, Wanling Gao, Rui Ren, Xiwen He, Chunjie Luo, Gang Lu, and Jingwei Li. BOPS, Not FLOPS! A New Metric, Measuring Tool, and Roofline Performance Model For Datacenter Computing. CoRR, abs/1801.09212, 2018.

[60] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Floating-point Programs and Multicore Architectures. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2009.

[61] Rui Yan, Yiping Song, and Hua Wu. Learning to Respond with Deep Neural Networks for Retrieval-based Human-Computer Conversation System. In Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval, pages 55–64. ACM, 2016.

[62] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pages 423–438. ACM, 2013.

[63] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pages 161–170. ACM, 2015.