CCF Transactions on High Performance Computing (2019) 1:224–239
https://doi.org/10.1007/s42514-019-00018-4

REGULAR PAPER

Performance analysis of deep learning workloads using roofline trajectories

M. Haseeb Javed[1] · **Khaled Z. Ibrahim**[2] · **Xiaoyi Lu**[1]

Received: 31 July 2019 / Accepted: 9 November 2019 / Published online: 29 November 2019

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. However, many users of such applications are not experts in Machine Learning itself. Consequently, there is limited knowledge among the community to run such applications in an optimized manner. The performance question for Deep Learning applications has typically been addressed by employing bespoke hardware (e.g., GPUs) 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 governments with resources sufficient enough to employ them at a large scale. As a result, an average user is only left with access to commodity clusters with, in many cases, only CPUs as the sole processing element. 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 CNN models and identify opportunities for black box and application-aware optimizations. Using the findings from our study, we are able to obtain up to 3.5× speedup compared to vanilla TensorFlow with default configurations.

Keywords Roofline · Deep learning · Tensorflow · MKL

1 Introduction

With the convergence of High-Performance Computing (HPC) 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 (Esteva et al. 2017), self-driving cars (Huval et al. 2015), natural language processing (Yan et al. 2016), and so on. Deep Learning frameworks and applications have been heavily leveraging HPC technologies to improve their performance and scalability.

Taking TensorFlow (Abadi et al. 2016) 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 (InfiniBand Trade Association 2017), RoCE (RDMA over Converged Ethernet 2019), High Speed Ethernet, etc. are used to improve the tensor communication performance in TensorFlow with high-performance communication libraries, such as gRPC (gRPC 2019), RDMA-gRPC (Biswas et al. 2018), MPI (Message Passing Interface Forum 2019) etc. From the computation perspective, TensorFlow-based Deep Learning workloads have been taking advantage of many advanced computing capabilities available on CPUs, GPUs, TPUs (Jouppi et al. 2017b), and so on. For CPU-based platforms, Intel TensorFlow (Intel-Tensorflow 2019) can accelerate Deep Learning workloads with the latest AVX512 technology on x86 CPUs. For GPU-based platforms, cuDNN (Chetlur et al. 2014) and NCCL (NVIDIA NCCL 2017) have become the standard building blocks

✉ M. Haseeb Javed
javed.19@osu.edu

Khaled Z. Ibrahim
kzibrahim@lbl.gov

Xiaoyi Lu
lu.932@osu.edu

[1] Department of Computer Science and Engineering, Ohio State University, Columbus, USA

[2] Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, USA

for high-performance and scalable Deep Learning training on GPU devices. DL architectures (Chen et al. 2018) specifically optimized for GPUs have also been suggested.

Even though there exists significant literature addressing performance enhancements for Deep Learning workloads, we find that there is a lack of performance models to systematically guide optimizations for Deep Learning workloads. Coming up with a useful and insightful performance model for Deep Learning workloads is not a trivial task. This is because Deep Learning workloads typically have very complex and deep software and hardware stacks (Lu et al. 2018), which can not be easily abstracted as a simple and meaningful model. Due to the lack of useful performance models for Deep Learning workloads, 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 Deep Learning researchers, we propose a simple and effective approach to systematically analyze and optimize the performance of Deep Learning workloads with the Roofline model (Williams et al. 2009). The Roofline model is a very useful and insightful visual performance model for multi-core architectures. We choose the Roofline model as our hammer to analyze Deep Learning workloads, which is because it can analyze the heavy stacked Deep Learning workloads in a black box approach.

To this end, this paper performs comprehensive profiling and analysis of Deep Learning models run on a multi-core CPU cluster using TensorFlow. In particular, we make use of the Roofline model to identify bottlenecks in a step-by-step manner and resolve them accordingly. Using this approach to optimize distributed training of DNNs, we are able to obtain up to 3.5× performance improvement over vanilla TensorFlow using the default configurations.

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. We believe that because of the simplicity of the Roofline model, many Deep Learning application researchers and developers can also use it to analyze and optimize their workloads even if they do not have a comprehensive understanding of the bulky and complex Deep Learning stacks.

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.

The paper is organized as follows. Section 2 covers background knowledge with regards to TensorFlow and Roofline model. The performance analysis methodology used in this study is detailed in Sect. 3. Section 4 discusses baseline observations while Sect. 5 presents the black box optimizations performed on the baselines established in the section prior. Application-aware optimizations are discussed in Sect. 6. Related work is summarized in Sect. 7. Finally, concluding remarks and future directions for continued work appear in Sect. 8.

2 Background

2.1 Tensorflow

Tensorflow (Abadi et al. 2016) 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 (Krizhevsky et al. 2012) 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 (Dean et al. 2012) 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.

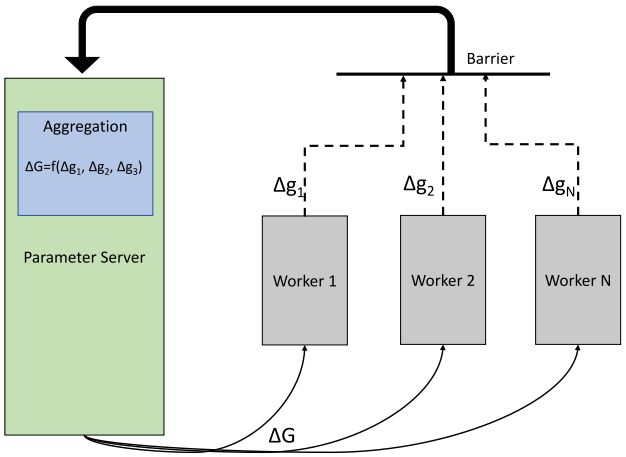


Fig. 1 Parameter server (PS) model with synchronous gradient updates: one or more PSs distribute work (through broadcast) and aggregate results (through reduction)

2.1.1 Parameter Server

The Parameter Server model (Li et al. 2014) 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 (Chen et al. 2016) 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 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 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 (Sergeev et al. 2018) 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 all-reduce (Patarasuk and Yuan 2009) aggregation. The ring-based allreduce algorithm is implemented using NVIDIA Collective Communication Library (NCCL2) (NVIDIA NCCL 2017) 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 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 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

The Roofline model (Williams et al. 2009) 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, Wang et al. (2018) introduces a data-centric variant of 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.

Figure 3 shows a typical Roofline model. 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

[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.

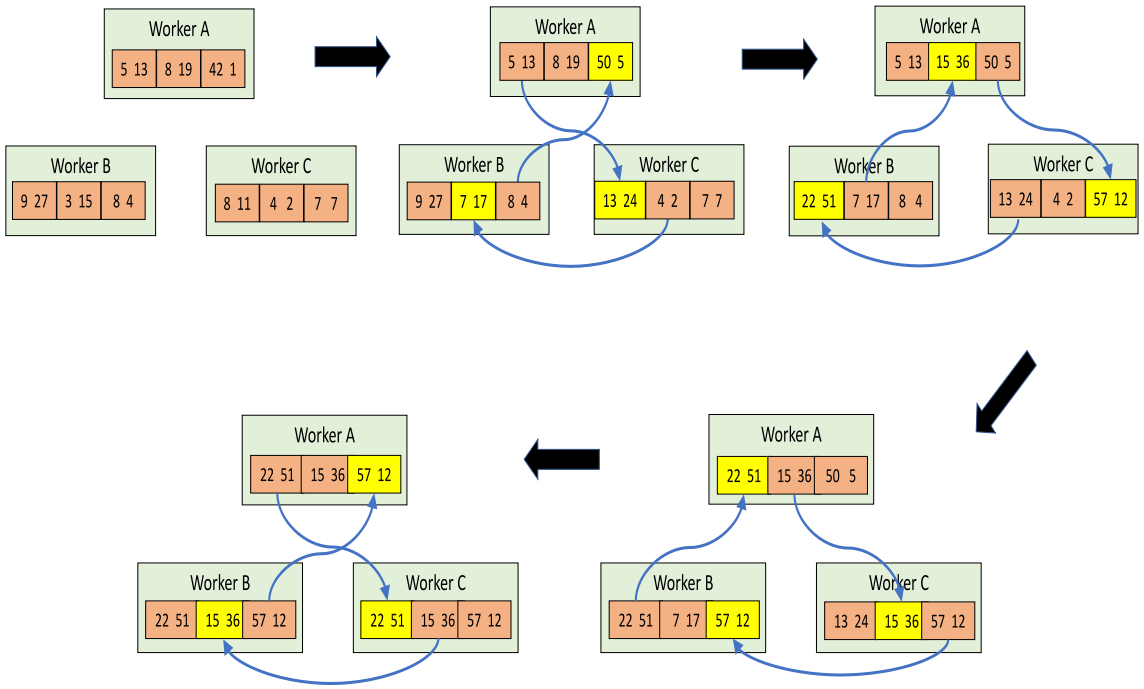


Fig. 2 Ring-allreduce, which optimizes for bandwidth and memory usage over latency

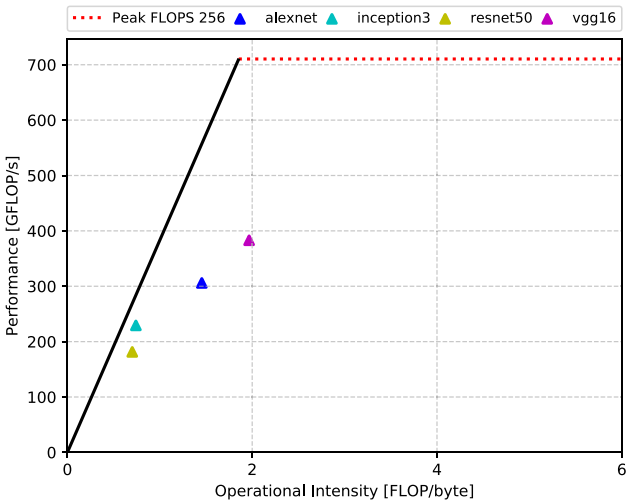


Fig. 3 The Roofline model characterizes the application using operational intensity and machine computational bounds. We could visually assess the performance optimality of a particular implementation against a empirical machine limits

will be bound by memory bandwidth. The point at which it terminates is called the ridge point, which is the point 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 (De Melo 2010).

3 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

Table 1 Cluster configuration

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

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

Table 2 Software configuration

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

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 (Krizhevsky et al. 2012), Inception3 (Szegedy et al. 2016), Resnet50 (He et al. 2016), and Vgg16 Simonyan and (Zisserman 2014)—which are based on the ImageNet (Deng et al. 2009) 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 (Keahey et al. 2019), 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 1.

The names and versions of different frameworks and compilers used in this study are summarized in Table 2.

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[3] and NVIDIA Nsight[4] 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 (Kim et al. 2011; Ibrahim et al. 2018a). 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.
   (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.

$$OI = \frac{FLOP}{(Reads + Writes) \times Multiplier}. \quad (1)$$

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

4 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 (Ibrahim et al. 2018b), 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.

4.1 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.

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

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

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

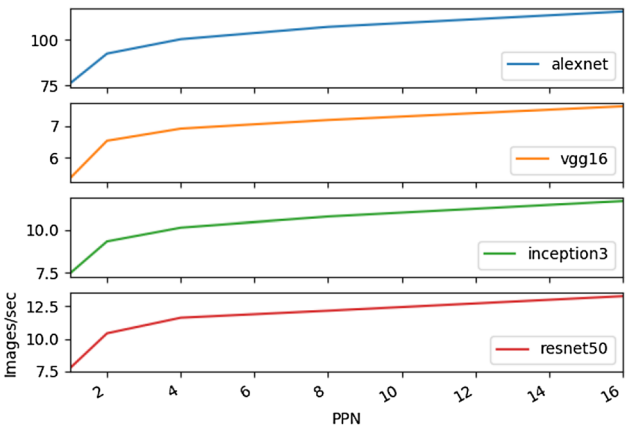


Fig. 4 Throughput variations in response to varying PPN. Increasing PPN from 1 to 2 and then 4 yields significant improvements in throughput but for PPN values beyond 4 the throughput remains fairly stable

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[5] benchmark was used. All the experiments described in this study are performed with a batch size of 64, unless specified otherwise.

4.1.1 Parameter Server Model

Setting up a TensorFlow cluster allows for various configurations of PS and worker processes, described in Sect. 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 4 shows the aggregate throughput obtained with varying number 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 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 (Lu et al. 2017).

Different levels of concurrency are tested against a varying number of PS to see how well the distributed training can scale. Figures 5 and 6 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 Sect. 4.2, 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 × 12 cores/socket × 3.7(GHz) clock rate with Turbo boost × 8 Single Precision (SP) FLOPs/cycle = 710.4 GFLOPS, which is denoted by the dotted red line "Peak FLOPS 256" in the graphs.

From Roofline plots for nodes running PS shown in Fig. 5, we observe that the task is not 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 Fig. 6. Keeping the number of PS constant, increasing the number of workers leads to an upward shift for the data points in Fig. 5 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.

Figures 6 and 7 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 2× (8.2 img/s with 4 workers vs 16.3 img/s with 16 workers for Vgg16) increase in throughput for a proportional increase in workers, for all models barring Alexnet. These models

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

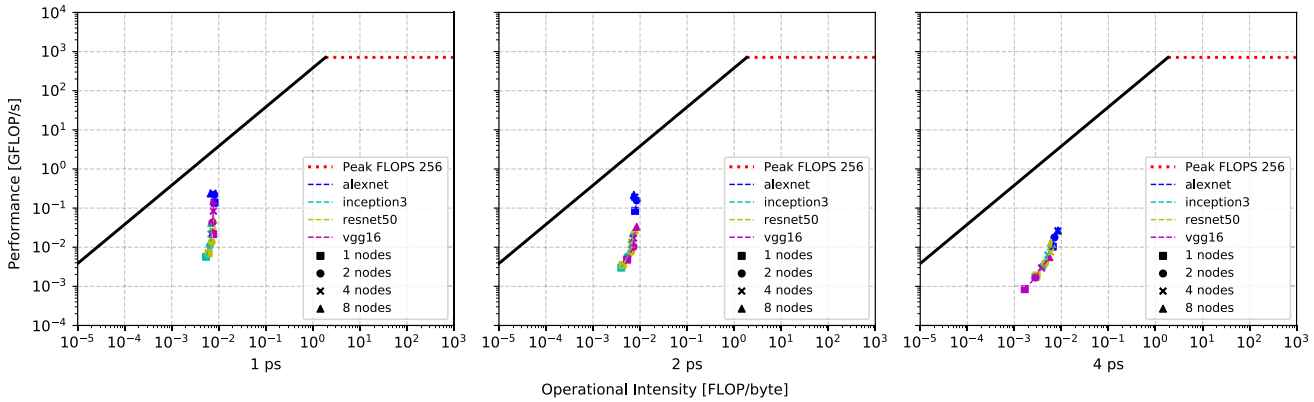


Fig. 5 Roofline plots for PS nodes. Significantly less raw operations are executed compared to the worker nodes which is expected. Increasing number of PS results in downward shift of trajectories as each PS becomes responsible for fewer network parameters to be aggregated

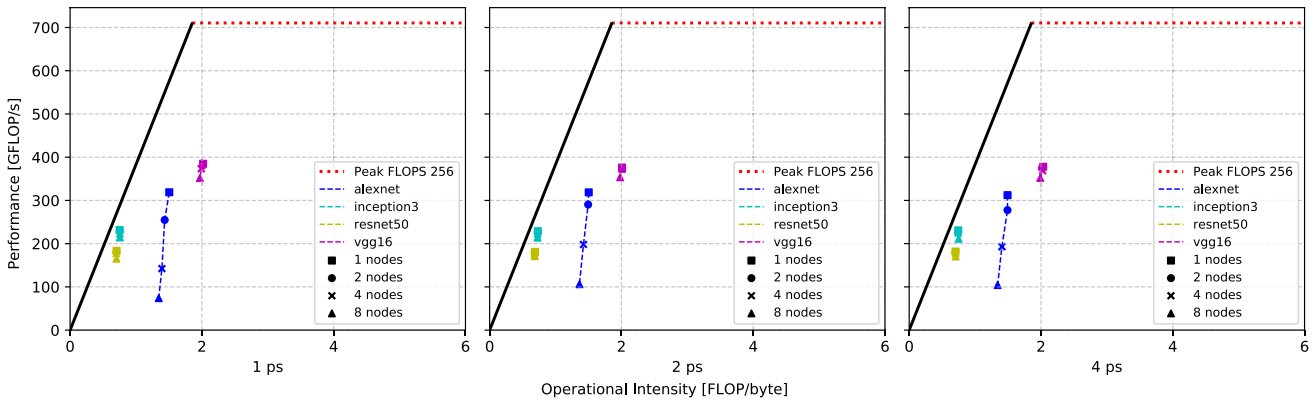


Fig. 6 Roofline plots for worker nodes. Trajectories for Alexnet show variations proportional to the level of concurrency suggesting its communication overhead is significant. A reduction in the length of trajectories for Alexnet may also be observed in response to increasing number of PS as gradient exchange becomes faster. Stable trajectories for other models indicate decent overlap between communication and computation

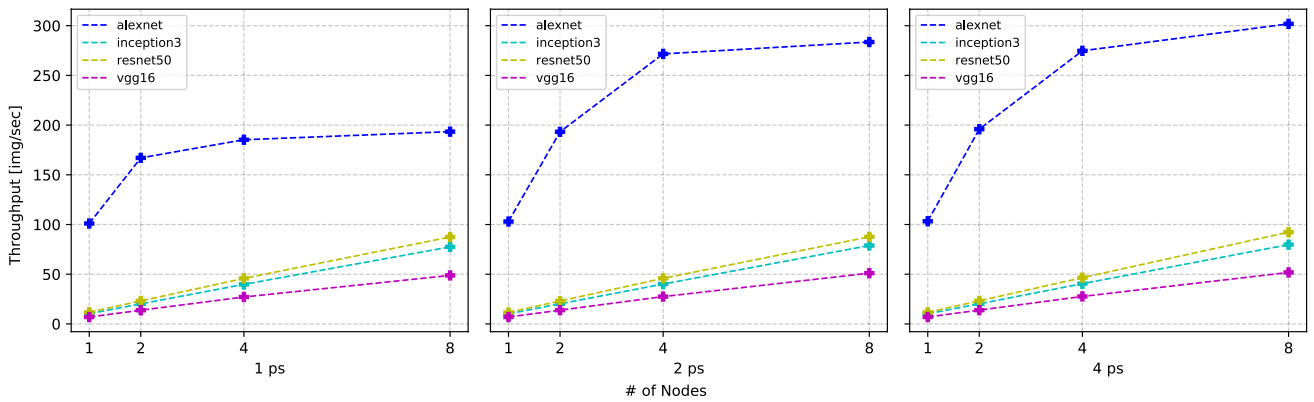
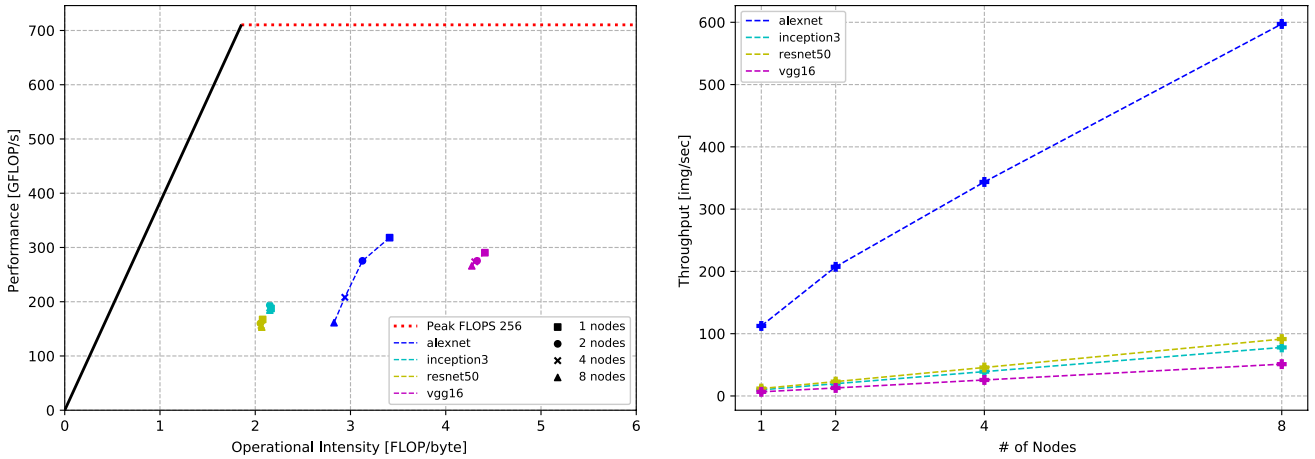


Fig. 7 Throughput obtained by various DNNs, which may be matched to Roofline trajectories from Fig. 6. Throughput for Alexnet almost follows a logarithmic trajectory, which shows improvement as more PS are added. All other models show linear speedup with minimal changes in response to increasing number of PS



(a) Roofline plot **(b)** Throughput

Fig. 8 Roofline trajectories for Horovod show increased OIs for all models compared to the PS approach (Fig. 6). For Alexnet, there is a reduction in the length of Roofline trajectory which is also reflected in the almost linear speedup of the throughput

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 Fig. 5.

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.

4.1.2 Horovod

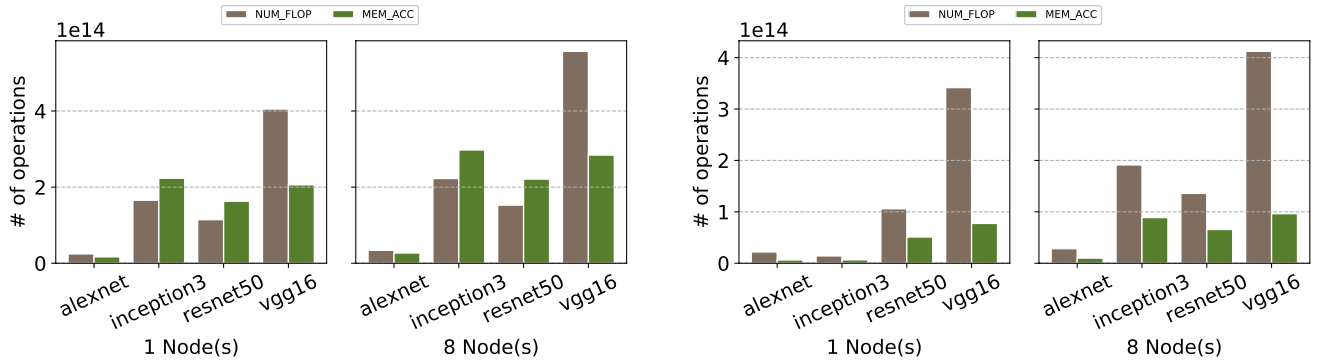
In this section, we carry out the same experiments as described in Sect. 4.1.1 but perform them using Horovod as the gradient update layer.

From Fig. 8, 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 2× (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.

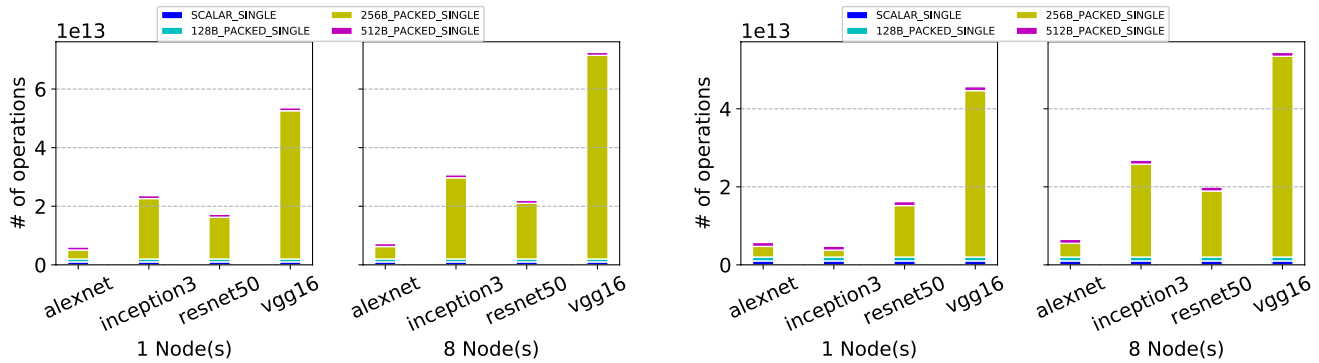
4.2 Analysis

The Roofline plots discussed in Sect. 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 into how we can improve the performance of these applications.



(a) Parameter Server **(b)** Horovod

Fig. 9 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. Increase in concurrency leads to a proportional increase in both NUM_FLOP and MEM_ACC for both programming models



(a) Parameter Server **(b)** Horovod

Fig. 10 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

Figure 9 is helpful in understanding why there is a difference in OI for the same experiments run using PS and Horovod approach. Figure 9a shows the OI breakdown at worker nodes of experiments run with varying number of PS while Fig. 9b is for Horovod. As expected, NUM_FLOP and MEM_ACC do not vary much if we add more PS to the system while increasing 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 Fig. 9b are generally not as high as for similar PS based experiments. However, MEM_ACCs are, on average 2× lower for all models leading to a much higher OI.

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 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 Fig. 10, 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.

4.3 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 shown

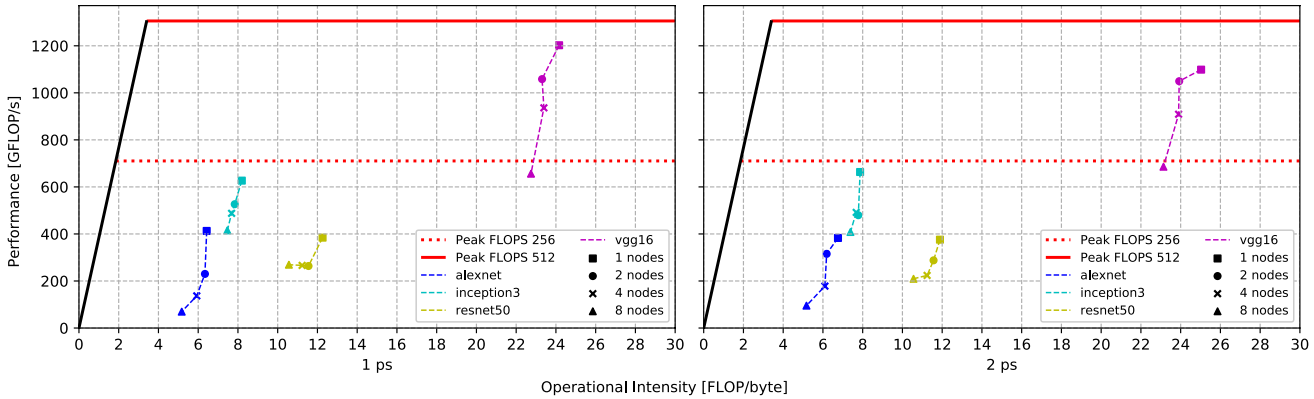


Fig. 11 MKL enabled TensorFlow worker with PS. The use of vectorized instructions pushes the maximum attainable performance to **Peak FLOPS 512**. As a result, the raw application performance also shows an upward movement compared to the ones shown without MKL in Fig. 6

in Fig. 6 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 Fig. 7, 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.

5 Black box optimizations

From the graphs discussed in the previous section, we find that 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.

5.1 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 Sect. 4.1.1, as having 512 byte vector instructions reduces the maximum clock rate[6]. Therefore we use the value of 652.8×2 sockets $= 1305.6$ GFLOPS provided by Intel in their official documentation[7] 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 can obtain the best scalability. Note that for vanilla TensorFlow, the best performance is achieved by using four MPI processes/node.

5.1.1 Parameter server model

Figure 11 shows the Roofline plot of MKL enabled TensorFlow with PS for gradient update while Fig. 12

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

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

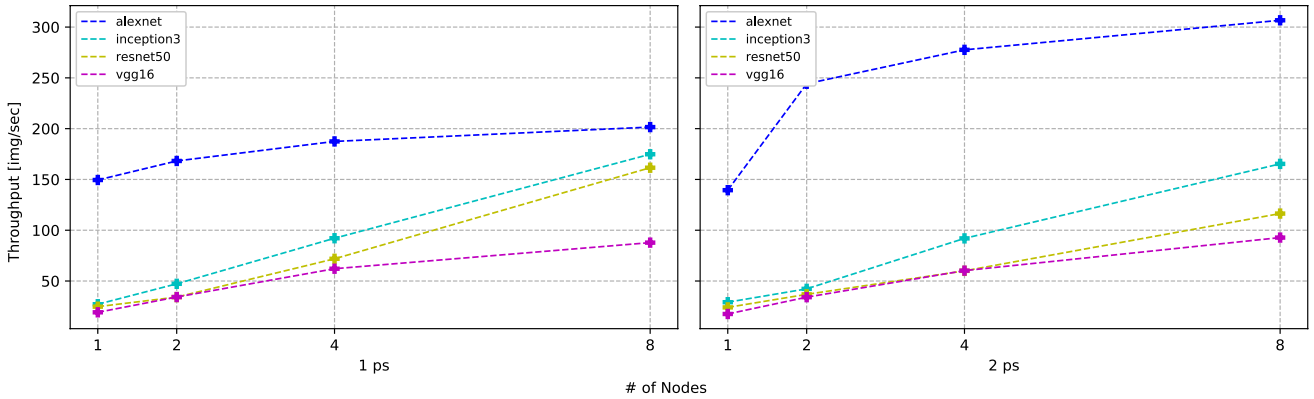
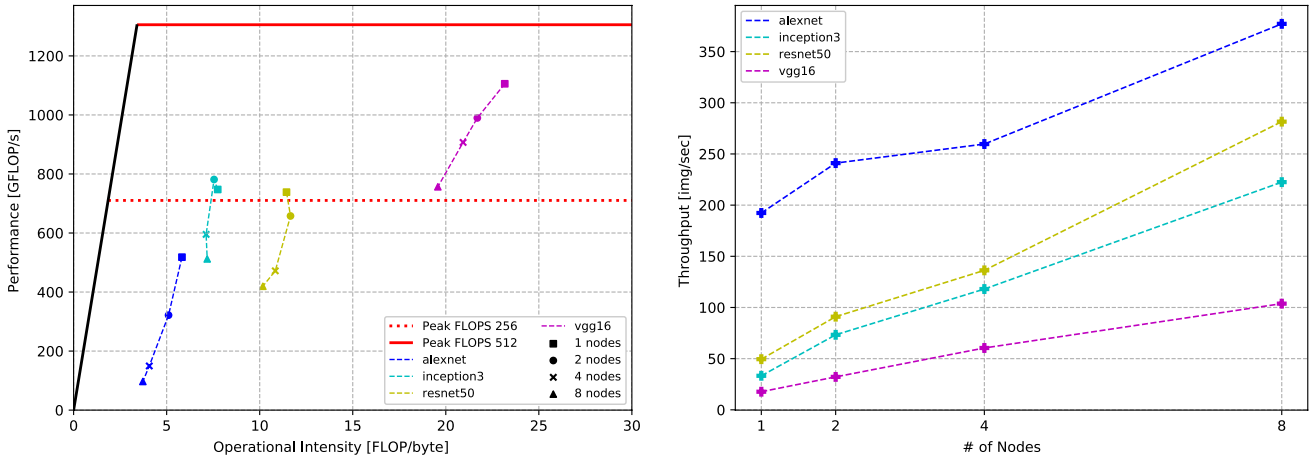


Fig. 12 Throughput of MKL enabled TensorFlow worker with PS. Significant improvement in speedup is observed for all models except Alexnet as gains from faster computation are compensated with losses from more frequent gradient exchanges resulting in minimal net improvement



(a) Roofline plot

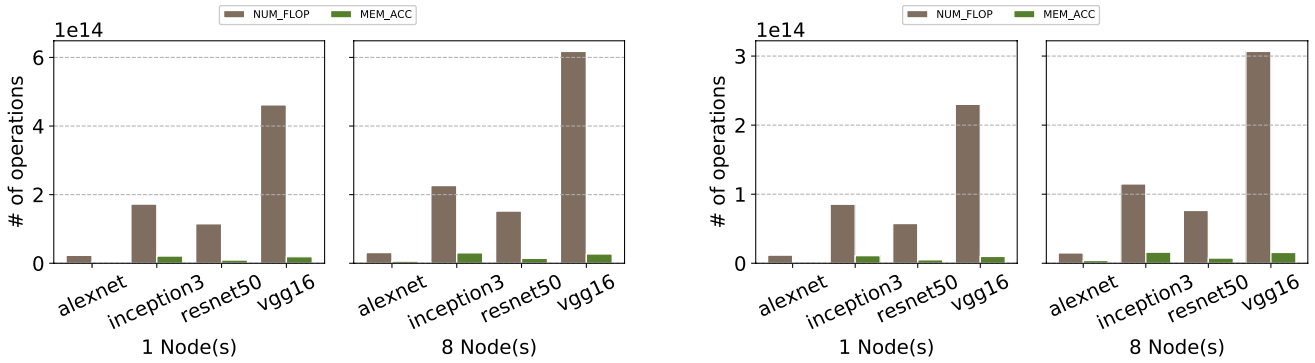
(b) Throughput

Fig. 13 MKL enabled TensorFlow worker with Horovod. Similar trends in Roofline trajectories and throughput are observed as in Figs. 11 and 12 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

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 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 2× improvement in performance for Inception (72 img/s vs 170 img/sec), Resnet50 (74 img/s vs 130 img/s), and Vgg16 (50 img/s vs 100 img/s 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 (Fig. 6) 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.

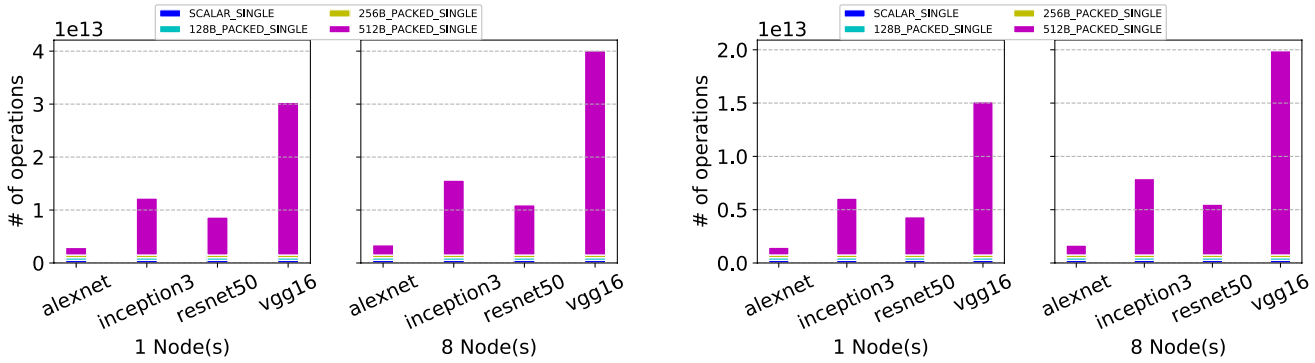
5.1.2 Horovod

From Fig. 13, 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.



(a) Parameter Server **(b)** Horovod

Fig. 14 OI breakdown of MKL enabled TensorFlow worker with PS and Horovod. There is a significant decline in memory accesses compared to vanilla TensorFlow (Fig. 9a). As a result the OIs are orders of magnitude higher



(a) Parameter Server **(b)** Horovod

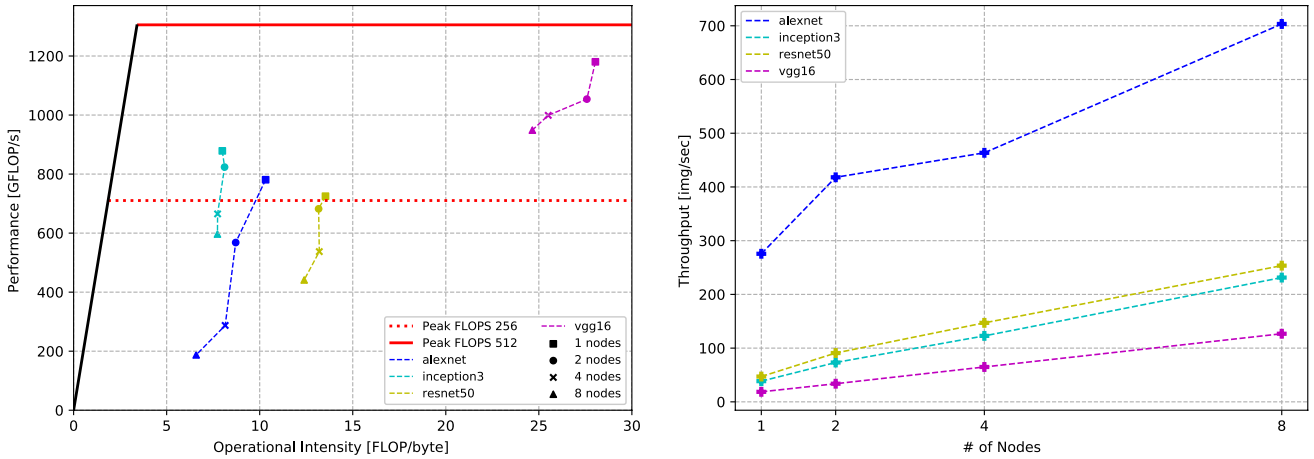
Fig. 15 Breakdown of FP operations performed by MKL-enabled TensorFlow worker with PS and Horovod. MKL-enabled TensorFlow almost exclusively makes use of 512-bit SP vectorized instructions resulting in a much higher FP operations count than that observed with vanilla TensorFlow (Fig. 15a)

We see greater than 2× improvement for Resnet50 (98 img/s vs 273 img/s with 16 workers) and Vgg16 (49 img/s vs 101 img/s with 16 workers) while Inception3 shows a speedup of 3× (73 img/s vs 215 img/s 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 2× (101 img/s vs 198 img/sec). The Roofline trajectory for Alexnet provides cues for this behavior. Comparing Figs. 8 and 13 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 a performance degradation.

As shown in Fig. 13, 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.

5.2 Analysis

The graphs discussed in Sect. 5 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



(a) Roofline plot **(b)** Throughput

Fig. 16 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 (Fig. 13a). This translates to improvement in throughput as well with 2× improvement at 8 nodes compared to Fig. 13b

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.

Comparing Figs. 6 and 11, 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 Fig. 14a, 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 less 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 Fig. 15a, 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 Fig. 14b.

5.3 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[8].

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 several 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.

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

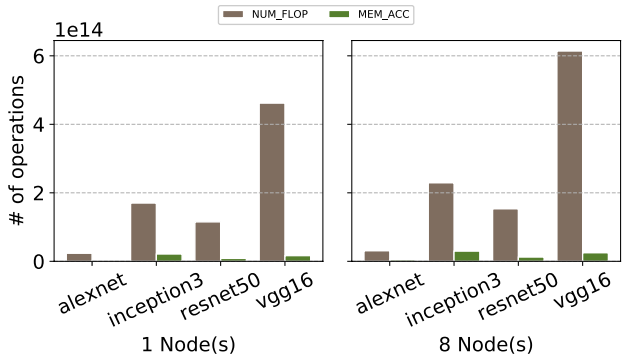


Fig. 17 OI breakdown of MKL enabled TensorFlow worker with Horovod and batch size = 128. The reason for improvement in the performance of Alexnet can be seen here as the FP operations performed increase by a factor of two compared to experiments with batch size=64 (Fig. 14b)

6 Application-aware optimizations

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

6.1 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 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.

Figure 16 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 3.5× improvement compared to vanilla Tensorflow with 1 PS and 8 Nodes for workers (700 img/s vs 200 img/s). 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.

6.2 Analysis

Figure 17 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 Fig. 14b, 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.

7 Related work

The original Roofline (Williams et al. 2009) 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 (Doerfler et al. 2016), NVIDIA GPUs (Lopes et al. 2017), Google TPUs (Jouppi et al. 2017a) and applications, including but not limited to, disaster detection Nagasu et al. (2017), large scale simulations (Kim et al. 2011), wireless network detection (Sarker et al. 2002) and even matrix multiplication (Kong et al. 2015).

There have not been many studies analyzing and profiling the performance of distributed Deep Learning. The few that exist (Bahrampour et al. 2016; Shi et al. 2016; Lu et al. 2018; Kim et al. (2017) do not focus on Roofline model based performance analysis and optimizations on CPUs. Bahrampour et al. (2016) and Shi et al. (2016) analyze the overall execution time of different frameworks but do not discuss techniques for performance improvement. In Kim et al. (2017), 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 2× 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 Zhang et al. (2015) and Meloni et al. (2016).

8 Conclusion

In this paper, 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 3.5× for Alexnet at a concurrency level of 8 nodes.

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.

Acknowledgements Results presented in this paper are obtained on the Chameleon Cloud testbed supported by the National Science Foundation. This work has been supported by Lawrence Berkeley National Laboratory under Contract no. DE-AC02-05CH11231 with the U.S. Department of Energy. Moreover, this research is supported in part by National Science Foundation Grant CCF#1822987.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M. et al.: Tensorflow: a system for large-scale machine learning. In: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), pp. 265–283 (2016)

Bahrampour, S., Ramakrishnan, N., Schott, L., Shah, M.: Comparative study of caffe, neon, theano, and torch for deep learning (2016)

Biswas, R., Lu, X., Panda, D.K.: Accelerating TensorFlow with adaptive RDMA-based gRPC. In: 2018 IEEE 25th International Conference on High Performance Computing (HiPC). IEEE, pp. 2–11 (2018)

Chen, J., Pan, X., Monga, R., Bengio, S., Jozefowicz, R.: Revisiting distributed synchronous SGD (2016). arXiv preprint arXiv:1604.00981

Chen, J., Li, K., Bilal, K., Li, K., Philip, S.Y., et al.: A bi-layered parallel training architecture for large-scale convolutional neural networks. IEEE Trans. Parallel Distrib. Syst. **30**(5), 965–976 (2018)

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., Shelhamer, E.: cuDNN: Efficient primitives for deep learning, CoRR, vol. abs/1410.0759 (2014). [Online]. http://arxiv.org/abs/1410.0759

De Melo, A.C.: The new Linux perf tool. In: Slides from Linux Kongress, vol. 18 (2010)

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Senior, A., Tucker, P., Yang, K., Le, Q.V. et al.: Large scale distributed deep networks. In: Advances in Neural Information Processing Systems, pp. 1223–1231 (2012)

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., Fei-Fei, L.: ImageNet: a large-scale hierarchical image database. In: CVPR09 (2009)

Doerfler, D., Deslippe, J., Williams, S., Oliker, L., Cook, B., Kurth, T., Lobet, M., Malas, T., Vay, J.-L., Vincenti, H.: Applying the roofline performance model to the intel xeon phi knights landing processor. In: International Conference on High Performance Computing. Springer, pp. 339–353 (2016)

Esteva, A., Kuprel, B., Novoa, R.A., Ko, J., Swetter, S.M., Blau, H.M., Thrun, S.: Dermatologist-level Classification of skin cancer with deep neural networks. Nature **542**(7639), 115 (2017)

gRPC (2019). http://grpc.io/

He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)

Huval, B., Wang, T., Tandon, S., Kiske, J., Song, W., Pazhayampallil, J., Andriluka, M., Rajpurkar, P., Migimatsu, T., Cheng-Yue, R. et al.: An empirical evaluation of deep learning on highway driving. arXiv preprint arXiv:1504.01716, (2015)

Ibrahim, K., Williams, S., Oliker, L.: Performance analysis of GPU programming models using the roofline scaling trajectories. In: 2018 International Symposium on Benchmarking, Measuring and Optimizing (Bench'19) (2018a)

Ibrahim, K., Williams, S., Oliker, L.: Roofline scaling trajectories: a method for parallel application and architectural performance analysis. In: 2018 International Conference on High Performance Computing and Simulation (HPCS). IEEE, pp. 350–358 (2018b)

InfiniBand Trade Association (2017). [Online]. http://www.infinibandta.org

Intel-Tensorflow (2019). [Online]. https://github.com/Intel-tensorflow/tensorflow

Jouppi, N.P., Young, C., Patil, N., Patterson, D., 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 (2017a)

Jouppi, N.P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al.: In-datacenter performance analysis of a tensor processing unit. In: 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA). IEEE, pp. 1–12 (2017b)

Keahey, K., Riteau, P., Stanzione, D., Cockerill, T., Mambretti, J., Rad, P., Ruth, P.: Chameleon: a scalable production testbed for computer science research. From Petascale toward Exascale, Contemporary High Performance Computing (2019)

Kim, K.-H., Kim, K., Park, Q.-H.: Performance analysis and optimization of three-dimensional FDTD on GPU using roofline model. Comput. Phys. Commun. **182**(6), 1201–1207 (2011)

Kim, H., Nam, H., Jung, W., Lee, J.: Performance analysis of CNN frameworks for GPUs. In: 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, pp. 55–64 (2017)

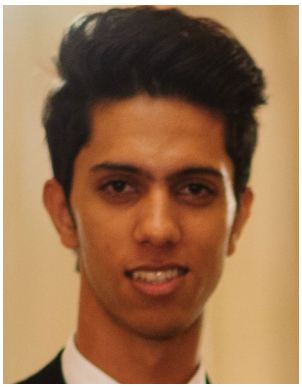
Kong, M., Pouchet, L.-N., Sadayappan, P.: A roofline-based performance estimator for distributed matrix-multiply on Intel CnC. 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, pp. 1241–1250 (2015)

Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1097–1105 (2012)

Li, M., Andersen, D.G., Park, J.W., Smola, A.J., Ahmed, A., Josifovski, V., Long, J., Shekita, E.J., Su, B.-Y.: Scaling distributed machine learning with the parameter server. In: 11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14), (2014), pp. 583–598

Lopes, A., Pratas, F., Sousa, L., Ilic, A.: 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). IEEE, pp. 1–12 (2017)
Lu, X., Shi, H., Shankar, D., Panda, D.K.: Performance characterization and acceleration of big data workloads on OpenPOWER system. In: 2017 IEEE International Conference on Big Data (Big Data), pp. 213–222 (2017)
Lu, X., Shi, H., Biswas, R., Javed, M.H., Panda, D.K.: DLoBD: a comprehensive study of deep learning over big data stacks on HPC clusters. IEEE Trans. MultiScale Comput. Syst. **4**(4), 635–648 (2018)
Meloni, P., Deriu, G., Conti, F., Loi, I., Raffo, L., Benini, L.: Curbing the roofline: a scalable and flexible architecture for CNNs on FPGA. In: Proceedings of the ACM International Conference on Computing Frontiers. ACM, pp. 376–383 (2016)
Message Passing Interface Forum (2019). [Online]. http://www.mpi-forum.org/
Nagasu, K., Sano, K., Kono, F., Nakasato, N.: FPGA-based tsunami simulation: performance comparison with GPUs, and roofline model for scalability analysis. J. Parallel Distrib. Comput. **106**, 153–169 (2017)
NVIDIA NCCL (2017). [Online]. https://github.com/NVIDIA/nccl
Patarasuk, P., Yuan, X.: Bandwidth optimal all-reduce algorithms for clusters of workstations. J. Parallel Distrib. Comput. **69**(2), 117–124 (2009)
RDMA over Converged Ethernet (2019). http://www.roceinitiative.org/
Sarker, J.H., Hassan, M., Halme, S.J.: Power level selection schemes to improve throughput and stability of slotted ALOHA under heavy load. Comput. Commun. **25**(18), 1719–1726 (2002)
Sergeev, A., Del Balso, M.: Horovod: fast and easy distributed deep learning in tensorflow (2018). arXiv preprint arXiv:1802.05799
Shi, S., Wang, Q., Xu, P., Chu, X.: Benchmarking state-of-the-art deep learning software tools. In: 2016 7th International Conference on Cloud Computing and Big Data (CCBD). IEEE, pp. 99–104 (2016)
Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. (2014). arXiv preprint arXiv:1409.1556
Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., Wojna, Z.: Rethinking the inception architecture for computer vision. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 2818–2826 (2016)
Wang, L., Zhan, J., Gao, W., Ren, R., He, X., Luo, C., Lu, G., Li, J.: BOPS, not FLOPS! A new metric, measuring tool, and roofline performance model for datacenter computing, CoRR, vol. abs/1801.09212. [Online]. (2018). http://arxiv.org/abs/1801.09212
Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for floating-point programs and multicore architectures, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), Tech. Rep. (2009)
Yan, R., Song, Y., Wu, H.: 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. ACM, pp. 55–64 (2016)
Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., Cong, J.: Optimizing FPGA-based accelerator design for deep convolutional neural networks. In: Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM, pp. 161–170 (2015)



M. Haseeb Javed received his undergraduate degree in software engineering from the National University of Science and Technology (NUST), Pakistan. He is working towards a graduate degree at The Ohio State University and is a research assistant in Dr. Xiaoyi Lu's lab. His current focus is on systems research cross-cutting the domain of big data and high-performance computing.

Khaled Z. Ibrahim is a computer scientist in the Computational Research Division at Lawrence Berkeley National Laboratory (LBNL). He obtained his PhD in computer engineering from North Carolina State University in 2003. His research interests include high-performance computing, virtualization and cloud computing environments, high-performance runtime systems, and code optimization.

Xiaoyi Lu is a Research Assistant Professor in the Department of Computer Science and Engineering, The Ohio State University, Ohio. He received his Ph.D. degree in Computer Science and Technology from Institute of Computing Technology, Chinese Academy of Sciences, Beijing, in 2012. His current research interests include high performance interconnects and protocols, Big Data Analytics, Parallel Computing Models, Virtualization, Cloud Computing, and Deep Learning systems. He has published more than 100 papers in major International conferences, workshops, and journals with multiple Best (Student) Paper Awards or Nominations. He has been actively involved in various professional activities in academic journals and conferences.

He is a member of the IEEE and ACM. More details about Dr. Lu are available at http://web.cse.ohio-state.edu/~luxi.