ParaX: Boosting Deep Learning for Big Data Analytics on Many-Core CPUs

Lujia Yin
NUDT
Changsha, China
ylj1992nudt@gmail.com

Yiming Zhang
NiceX Lab, NUDT
Changsha, China
Contact:zym@nicexlab.com

Zhaoning Zhang
NUDT
Changsha, China
zzningxp@gmail.com

Yuxing Peng
NUDT
Changsha, China
pengyuxing@aliyun.com

Peng Zhao
Intel Research
Shanghai, China
patric.zhao@intel.com

ABSTRACT

Despite the fact that GPUs and accelerators are more efficient in deep learning (DL), commercial clouds like Facebook and Amazon now heavily use CPUs in DL computation because there are large numbers of CPUs which would otherwise sit idle during off-peak periods. Following the trend, CPU vendors have not only released high-performance many-core CPUs but also developed efficient math kernel libraries. However, current DL platforms cannot scale well to a large number of CPU cores, making many-core CPUs inefficient in DL computation. We analyze the memory access patterns of various layers and identify the root cause of the low scalability, i.e., the per-layer barriers that are implicitly imposed by current platforms which assign one single instance (i.e., one batch of input data) to a CPU. The barriers cause severe memory bandwidth contention and CPU starvation in the access-intensive layers (like activation and BN).

This paper presents a novel approach called ParaX, which boosts the performance of DL on many-core CPUs by effectively alleviating bandwidth contention and CPU starvation. Our key idea is to assign one instance to each CPU core instead of to the entire CPU, so as to remove the per-layer barriers on the executions of the many cores. ParaX designs an ultralight scheduling policy which sufficiently overlaps the access-intensive layers with the compute-intensive ones to avoid contention, and proposes a NUMA-aware gradient server mechanism for training which leverages shared memory to substantially reduce the overhead of per-iteration parameter synchronization. We have implemented ParaX on MXNet. Extensive evaluation on a two-NUMA Intel 8280 CPU shows that ParaX significantly improves the training/inference throughput for all tested models (for image recognition and natural language processing) by 1.73× ∼ 2.93×.

PVLDB Reference Format:
Lujia Yin, Yiming Zhang, Zhaoning Zhang, Yuxing Peng, and Peng Zhao. ParaX: Boosting Deep Learning for Big Data Analytics on Many-Core CPUs. PVLDB, 14(6): 864-877, 2021.
doi:10.14778/3447689.3447692

Proceedings of the VLDB Endowment, Vol. 14, No. 6 ISSN 2150-8097.

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/nicexlab/parax-source.

1 INTRODUCTION

The big data era brings severe challenge for data analytics as information becomes available in such volume, velocity and variety [7] that it cannot be processed by traditional algorithms and systems. In recent years, deep learning (DL) has been emerging as an effective approach for many fields of big data analytics [25, 55, 75] such as image recognition [39, 54, 69], natural language processing [35, 61, 62], and recommendation [27, 41, 52]. The basic processing of deep learning includes two separate phases, namely, (i) train deep neural network (DNN) models which evolve by iteratively tuning their model parameters with stochastic gradient descent (SGD) [18] on many batches of training data, and (ii) infer the results of input data by using the trained models.

DL platforms, such as TensorFlow [16], PyTorch [11], Caffe [48], and MXNet [22], support both CPUs and GPUs for training and inference of DNN models. Although CPUs are less efficient than GPUs for deep learning, they are now widely used in both training and inference in the cloud environment [12, 37, 38]. This is not only because CPUs can provide low latency (with small batch sizes) for online inference but also because there are large numbers of CPUs which would otherwise sit idle during off-peak periods. For instance, FBLearner [38] is a hybrid system of Facebook, which mainly (i) uses GPUs in training and offline inference for high throughput and (ii) adopts CPUs in online inference for low latency. Since the diurnal load cycles leave a significant number of CPUs available, FBLearner also heavily uses CPUs in its offline training and inference tasks. Further, Facebook's CPU-based inference system improves the performance of many-core CPUs by batching and co-locating inference jobs [37].

Following the trend, CPU vendors have not only released high-FLOPS (floating point operations per second) many-core CPUs but also developed high-performance math libraries (similar to cuDNN [24] for GPUs) to accelerate x86-based kernel computation on the DL platforms [16, 22, 48]. As a result, the training and inference throughput of CPUs has been effectively improved (§2),

doi:10.14778/3447689.3447692

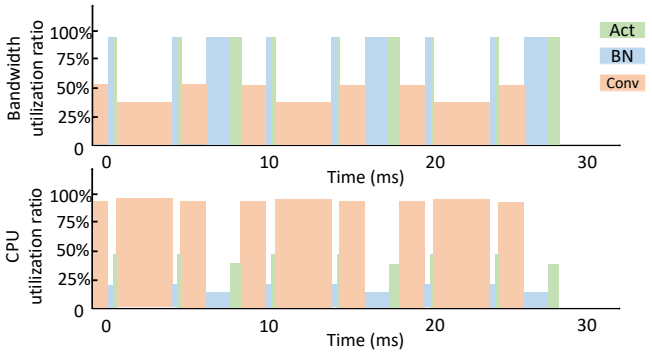

Figure 1: Utilizations of memory bandwidth and CPU cycles in the inference of ResNet50, running MXNet on Intel 8280 (one NUMA node with 28 cores).

demonstrating the potential of many-core CPUs for throughput-demanding deep learning tasks.

However, it is still challenging to fully exploit the power of many-core CPUs on the mainstream DL platforms, which are originally designed for and currently focused on GPU-based deep learning computation. This is mainly because CPUs and GPUs adopt different hardware designs especially in their memory architectures: GPUs have smaller on-chip memory with much higher bandwidth, while CPUs use main memory with lower bandwidth but much larger capacity. Current DL platforms rely on GPUs' high memory bandwidth for high throughput [33] and overlook this difference for CPU-based training and inference, making them not scale to the many cores of CPUs.

To understand this problem, we run mainstream DL platforms (MXNet [22], TensorFlow [16] and PyTorch [11]) on an Intel Xeon 8280 CPU [5] with one NUMA (non-uniform memory access) node of 28 cores to infer ResNet50 [39], and measure the utilization of memory bandwidth and CPU cycles. Figure 1 depicts the result for MXNet, where it is the memory bandwidth that bounds the executions of the activation (Act) and batch normalization (BN) layers and causes CPU starvation. About half the bandwidth is idle at the convolution (Conv) layers, which results in a low (overall) utilization of 61.3% of the precious CPU memory bandwidth. TensorFlow and PyTorch perform even worse than MXNet (not shown in Figure 1), respectively achieving only 57.0% and 54.6% memory bandwidth utilization.

The root cause of the low utilization of the precious memory bandwidth is that in current DL platforms only one single *instance* is assigned to one CPU (referred to as *one-instance-per-CPU*), where an instance is a batch of input data jointly processed in an iteration. Although assigning one instance to a GPU which has high memory bandwidth can fully exploit the parallelism inside the batch (by dividing the batch into partitions each being assigned to one core), Figure 1 shows that it is inefficient to assign only one instance to a many-core CPU which has much lower bandwidth. This is because one-instance-per-CPU implicitly imposes the *per-layer barriers* on the executions of the many cores which are jointly processing the batch.

We categorize the layers in DNNs into two classes, namely, the compute-intensive layers that execute complex arithmetic operations (like convolution and general matrix multiplication (GEMM)), and the access-intensive layers that execute much simpler element-wise operations (like activation and BN), which will be analyzed via experiments in §3. As shown in Figure 1, the per-layer barriers enforce synchronous executions of the operations, making the limited memory bandwidth of many-core CPUs become a bottleneck and causing intermittent CPU starvation at the access-intensive layers. Note that memory bandwidth would not bound the throughput when the number of cores is much smaller (in ordinary multi-core CPUs) or the bandwidth is much higher (in GPUs), in which cases the total compute capacity does not saturate the bandwidth.

Based on the observation, in this paper we present ParaX, a novel approach that boosts the performance of deep learning on many-core CPUs. Our key idea is to break the input data of an iteration (a.k.a. mini-batch [59]) into batches (i.e., instances) and assign one instance to each CPU core (referred to as *one-instance-per-core*) instead of to the entire CPU, so as to allow each core to individually process its batch and thus remove the per-layer barriers on the executions of the cores (§4.1). Note that one-instance-per-core does not increase the number of cores involved in an iteration compared to one-instance-per-CPU which can occupy all the cores for the single instance by adopting MKL-DNN (a.k.a. oneDNN) [13], but improves the memory bandwidth utilization. ParaX designs an ultralight scheduling policy which sufficiently overlaps the access–intensive layers and compute-intensive ones on different cores to avoid contention. We show this could be achieved by leveraging the randomness of the layers' execution times, complemented with delayed initiation [74] when training relatively shallow networks.

For training, ParaX follows synchronous SGD [18] to update model parameters for every iteration. However, the many instances on one CPU will potentially increase the overhead of per-iteration synchronization, which may even completely counteract the benefit of layer overlapping. To address this problem, ParaX designs a NUMA-aware gradient server mechanism (§4.2) that leverages shared memory to substantially reduce the overhead compared to the state-of-the-art parameter server (PS) [58] or ring-allreduce (RAR) [68] mechanisms, which are inefficient for synchronization of many cores on a CPU since they are designed for distributed scenarios and cannot adapt to the NUMA architecture of many-core CPUs.

This paper makes the following contributions. To the best of our knowledge, we are the first to (i) uncover the reason (per-layer barriers) of the inefficiency of current DL platforms on many-core CPUs, and (ii) propose one-instance-per-core for both training and inference. We have implemented ParaX on MXNet. Extensive evaluation on a two-NUMA Intel 8280 CPU (§5) shows that ParaX significantly improves the throughput of training and inference for all tested models (for image recognition and natural language processing) respectively by 1.73× ∼ 2.93× and 2.08× ∼ 2.11×.

The rest of this paper is organized as follows. §2 reviews the background. §3 analyzes the problem of deep learning on many-core CPUs. §4 introduces the design of ParaX for improving memory bandwidth utilization. §5 presents the evaluation results. §6 discusses related work. And finally §7 concludes the paper and discusses future work.

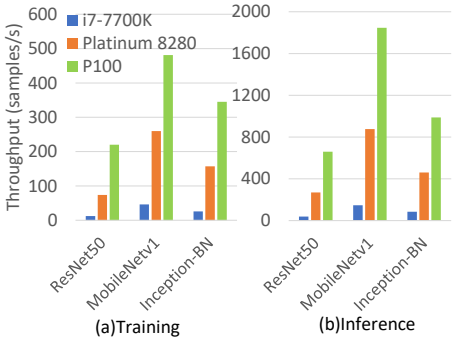

Figure 2: Throughput comparison.

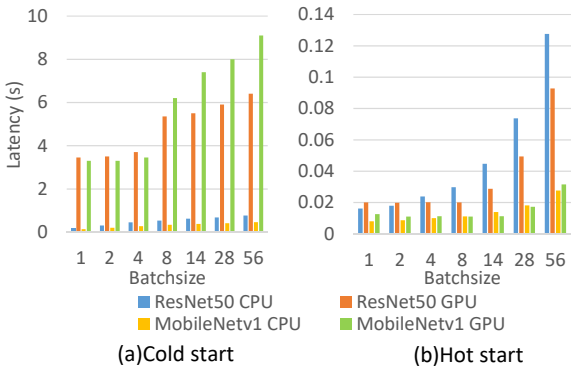

Figure 3: Inference latency comparison.

2 BACKGROUND

2.1 Recent Advances in Many-Core CPUs

The competition between CPUs and GPUs has been long lived in the market of deep learning hardware. Although in recent years it has been widely accepted that GPUs are more suitable for high-throughput tasks (model training and offline inference) and CPUs perform better in low-latency tasks (online inference), most recently CPUs are also widely used in throughput-demanding scenarios so that commercial cloud vendors could adapt to the diurnal load cycles in the cloud [12, 38]. For instance, Intel not only designs the new series of many-core CPUs [5] but also develops the math kernel DNN library [13]. This section briefly introduces the state-of-the-art many-core CPUs in hardware parameters (cores/FLOPs and memory bandwidth/capacity), libraries, and performance (latency and throughput).

Cores and FLOPs. GPUs have much more cores than CPUs. For instance, a Tesla P100 [3] GPU has 60 SM (Streaming Multiprocessor) each of which has 64 CUDA cores, so there are totally 3840 CUDA cores on P100. In contrast, Intel CPUs have at most tens of cores [5]. Counterintuitively, their difference in FLOPS is not as high as expected. For instance, the single-precision performance of NVIDIA Tesla P100 is 10.6 TFLOPS. For two-NUMA Intel Xeon Platinum 8280 CPUs (which has 56 cores with 2.70 GHz frequency with AVX512 [4] support for 64 single-precision arithmetic computation), the single-precision performance is $2.70 \times 56 \times 64 \approx 9.66$ TFLOPS.

Memory bandwidth and capacity. GPU memory has much higher bandwidth than CPU memory. For example, the peak bandwidth of P100's on-chip memory is 732 GB/sec, while the theoretical bandwidth of the state-of-the-art DDR4-SDRAM [63] is 25.6 GB/sec. Memory bandwidth is one of the main factors that limit the throughput of CPUs. On the upside, however, CPUs usually have much larger memory capacity compared to GPUs. For example, it is common for commodity servers to have 256 GB main memory, while the state-of-the-art GPUs have at most 48 GB on-chip memory [6]. Besides, the cache sizes in CPUs are also larger than that in GPUs. For example, the L3-cache size of Platinum 8280 is 39,424 KB, while P100's counterpart cache size is 4096 KB.

Libraries and applications. CPUs and GPUs adopt different mechanisms for multithreading. Intel Xeon CPUs (with MKL-DNN) relies on OpenMP [17] to awaken multiple threads from a pool, each of which executes the kernel on a data partition using one core with SIMD (Single Instruction Multiple Data) like AVX512 [4]. In contrast, GPUs (with cuDNN) relies on CUDA [14] to assign the threads to blocks which are further assigned to SMs, where threads in the same block are executed with SIMT (Single Instruction Multiple Threads) [60].

Performance. We measure the training throughput of an i7-7700k CPU (with 4 cores) and a two-NUMA Platinum 8280 CPU (with 56 cores), running ResNet50 [39], MobileNet-v1 [44], and Inception-BN [46] on MXNet (batch size = 64), respectively. For comparison, we also evaluate the same applications using a Tesla P100 GPU (with 3840 cores). The result (Figure 2) shows that (i) CPUs have already made remarkable progress in throughput, and (ii) there is still a huge performance gap between many-core CPUs and GPUs for throughput-demanding scenarios.

We also compare the inference latencies of 8280 CPU and P100 GPU, as shown in Figure 3. We run MXNet adopting one-instance-per-CPU and keep the total batch size small for emulating the online inference scenario with stringent latency requirement. Figure 3a depicts the cold start latencies (from launching programs to outputting results) where CPUs have much smaller latencies, mainly because (i) GPUs have to do extra initialization work (e.g., loading models into GPU memory) and (ii) CPUs have higher single-core frequency. Autotune [15] is turned off otherwise cuDNN would need a few more seconds to search the best kernels. Figure 3b shows the hot start latencies (from entering the networks to outputting the results), where CPUs have slightly higher latencies for inference with $B > 4$. Note that the inference latencies with cold start are important for GPUs, since it is common for context switching in the cloud environment [38] where tenants share the expensive GPUs with relatively small on-chip memory capacity.

2.2 Deep Learning on NUMA

Many-core CPUs adopt the NUMA (non-uniform memory access) architecture [51] where a CPU has m NUMA nodes ($m \geq 1$) each having its local memory shared by its cores. Although the m NUMA nodes of a CPU can access all the main memory, a NUMA can access its local memory faster than the non-local memory.

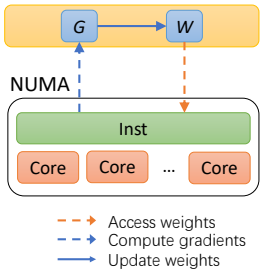

Figure 4: MXNet adopts one-instance-per-CPU for model training, where MXNet assigns one single instance to one CPU.

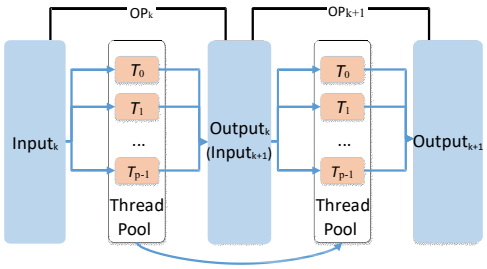

Figure 5: One-instance-per-CPU imposes per-layer barriers.

Current DL platforms (such as TensorFlow [16], PyTorch [11], Caffe [48], and MXNet [22]) all adopt the "one-instance-per-CPU" paradigm for model training and inference. A DNN model is trained by iterating a large dataset many times (i.e., *epochs*). Within each epoch, the dataset is partitioned into *mini-batches*, each being the input for a training *iteration*. A mini-batch is broken into multiple *batches*, each being a processing *instance*[1] which individually travels through the DNN model layer-by-layer. Figure 4 shows an example of the training procedure on a one-NUMA CPU. In every iteration, a single instance is assigned to the CPU and divided into partitions each being assigned to one core. The many cores jointly process the batch (exploiting the *intra-batch* parallelism) by reading the weights from memory, executing the layers one by one, computing the gradients, and updating the weights.

3 ANALYSIS

A neural network contains a sequence of successive layers where one layer's output is the next layer's input. The computation of a layer is called an operation, such as convolution, GEMM, activation, BN, etc. Current DL platforms leverage the cuDNN or MKL-DNN libraries to exploit the intra-batch parallelism, which enable all the cores of a GPU or CPU to execute the operations on the assigned batch of data.

In the state-of-the-art one-instance-per-CPU paradigm, each core runs a thread and all the cores jointly process a single instance layer-by-layer. Figure 5 illustrates the processing of the many cores on an instance (one-instance-per-CPU), where each T_i is a thread

[1]In most cases the two terms "instance" and "batch" are interchangeable.

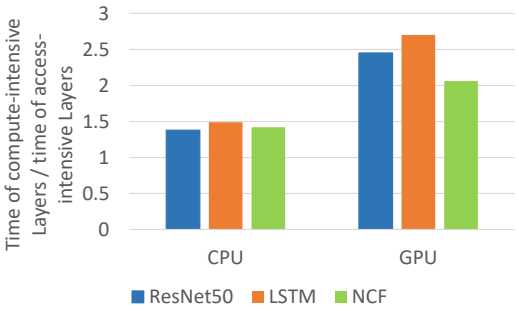

Figure 6: Inference time ratios ($B = 64$) of compute-intensive layers vs. access-intensive layers, respectively for CPU (Intel 8280) and GPU (NVIDIA P100). $B = 128$ and 256 have similar results.

running on one core. At the k^{th} layer, multiple threads are awakened from the thread pool, each executing the operation ($OP(k)$) on a partition of the instance in parallel to exploit the intra-batch parallelism. We categorize the layers into two classes, namely, the compute-intensive layers that execute compute-intensive operations (like convolution and GEMM) which conduct complex arithmetic computations, and the access-intensive layers that execute access-intensive[2] operations (like activation and BN) which are element-wise and usually have less computation complexity. When the total memory bandwidth demand at an access-intensive layer exceeds the maximum capacity, it will cause bandwidth contention which leads to starvation of the cores (Figure 1) and consequently longer execution times.

To understand the impact of bandwidth contention, we evaluate the ratios of the execution times of the compute-intensive layers to that of the access-intensive layers, running MXNet on a one-NUMA Intel 8280 CPU (with MKL-DNN), respectively for inference of ResNet50 [39], RNN (two-layer LSTM) [43], and NCF (Neural Collaborative Filtering) [40]. The result is shown in Figure 6, where the execution times of the compute-intensive layers are ($< 1.5\times$) comparable to that of the access-intensive layers. In contrast, when conducting the same experiment on a P100 GPU (with cuDNN), the execution times of the compute-intensive layers are up to 2.7× that of the access-intensive layers (also shown in Figure 6).

We further analyze the scalability of the compute-intensive operations (convolution) and the access-intensive operations (BN and activation), by executing them layer by layer using a one-NUMA 8280 CPU to infer ResNet50 on MXNet. The result (Figure 7) shows that the speedups of the three operations are similar before the numbers of threads (n) increase up to 7. Afterwards, the speedups of BN and activation increase very little, while that of convolution is still almost linear with n. This is because the bandwidth utilization of BN and activation almost reaches the maximum bandwidth capacity when $n > 7$, while that of convolution reaches only about one half the maximum capacity even when the number of cores is as high as $n = 28$.

[2]More accurately, they should be called *compute-non-intensive* since they have much less arithmetic computations (but similar amount of input/output data) compared to compute-intensive ones.

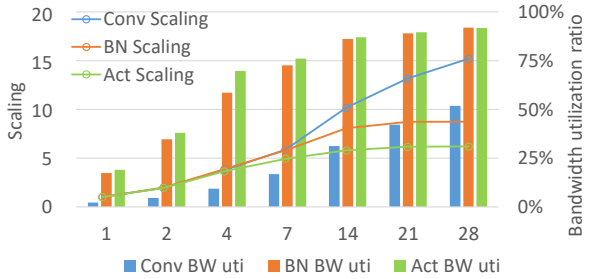

Figure 7: Speedups and utilization of different operations, as the numbers of threads increase.

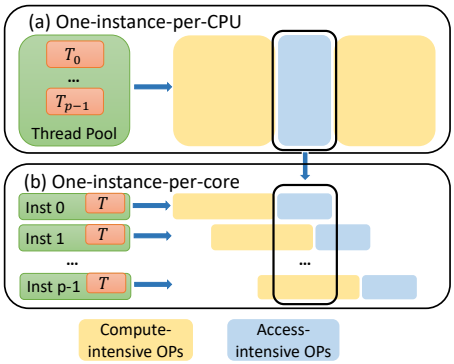

Figure 8: Layer overlapping.

Compared to the compute-intensive operations, although the access-intensive operations have less arithmetic computations in the training and inference on many-core CPUs, they have similar amount of input and output data. This makes the memory bandwidth become a potential bottleneck on the critical paths of executions of the access-intensive operations. In contrast, the compute-intensive operations perform significantly more computations per memory access, which could amortize the memory access cost. Therefore, the overall throughput on many-core CPUs is largely decided by the overall memory bandwidth utilization.

4 DESIGN

4.1 One-Instance-Per-Core

To alleviate the problem of memory bandwidth contention, our basic idea is to overlap the executions of various layers (including convolution, GEMM, activation, BN, etc.) on different cores, so that the compute-intensive layers (e.g., convolution and GEMM) could "lend" the surplus bandwidth to the access-intensive ones (e.g., activation and BN). As shown in Figure 5, however, one-instance-per-CPU implicitly imposes the per-layer barriers for the executions on all the cores that are jointly processing the instance and thus prevents "bandwidth lending" (Figure 8a).

Inspired by *data parallelism* [19, 29, 58] which is widely exploited in distributed deep learning, ParaX proposes to divide the input data of an iteration into batches (instances) and assign each CPU core with one instance, which we refer to as *one-instance-per-core*. As shown in Figure 8b, this allows each core to individually process its instance without barriers, making it possible for the access-intensive layers to make use of the surplus bandwidth of the simultaneous compute-intensive layers. Note that one-instance-per-core is different from the distributed DL scenarios [29, 58] where multiple GPU/CPU nodes process the same model and each individual node is responsible to train/infer a batch: distributed DL platforms are to integrate more resources and cannot adapt to the NUMA architecture, while ParaX is mainly focused on improving the utilization of existing resources (memory bandwidth and CPU cycles).

By removing the per-layer barriers, one-instance-per-core provides an opportunity for bandwidth lending between the many cores of a CPU, each of which runs one thread processing one instance. To improve the bandwidth utilization, intuitively ParaX need to elaborately schedule the threads on the cores so that their access-intensive layers could execute in different time slices. Since the executions of operations usually have sub-second durations, however, it is impractical to conduct fine-grained scheduling for a large number of threads.

Fortunately, we observe that there is certain randomness for the execution times of the same operations on the same-sized batches, especially when the number of threads is relatively large and the network is deep enough. Therefore, ParaX adopts an ultralight *overlap* scheduling policy, which relies on the randomness of operations' execution times to overlap the executions of access- and compute-intensive layers on different cores. Consider an i^{th} layer in a DNN model illustrated in Figure 8b. The threads randomly make different progresses during the executions of the first $(i-1)^{\text{th}}$ layers. Consequently, the executions of the access-intensive and compute-intensive layers of the threads will *statistically* overlap with each other, thus implicitly realizing memory bandwidth lending between the threads executing different instances and enabling ParaX to achieve high bandwidth utilization.

4.2 Gradient Server for Efficient Training

ParaX follows synchronous SGD [18] (for guaranteeing convergence [77]) to update the parameters by using the gradients from the instances for every training iteration. The downside of one-instance-per-core for parameter update is that it potentially increases the synchronization overhead as the numbers of instances increases, which might even completely counteract the benefit of layer overlapping (as evaluated in §5.2). Existing synchronization mechanisms of current DL platforms, such as PS (parameter server) [58] and RAR (ring-allreduce) [68], are initially targeted for distributed scenarios and thus not suitable for synchronization of the many instances on a CPU.

We briefly discuss the inefficiency of PS in synchronization. Figure 9 shows a PS-based training iteration adopting one-instance-per-core on an m-NUMA CPU ($m = 2$), where for each p-core NUMA node there is one dedicated thread occupying one core to serve as the parameter server (PS_0/PS_1), and the remaining $p-1$ (worker) cores process $p-1$ instances. A worker (i) pulls the weights ($W = \{W_i\}$ where $i = 0, 1, \cdots, m-1$) from the m servers (PS_i) by copying W_i in memory via sockets, (ii) computes its gradients (g) using its instance, and (iii) pushes g to each of the servers (g_i to PS_i) by adding g_i to the gradients (G_i). PS_i finally updates G_i to its weights (W_i). Clearly, the PS mechanism not only wastes CPU

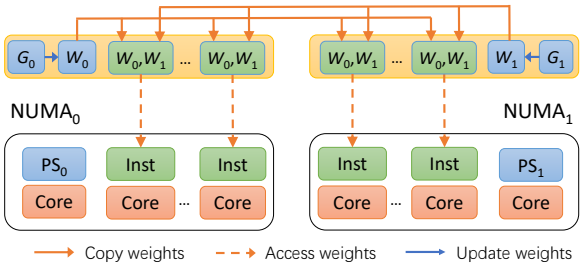

Figure 9: One-instance-per-core + PS. For brevity we omit the lines from instances to G_i for computing gradients (blue arrowed lines in Figure 4).

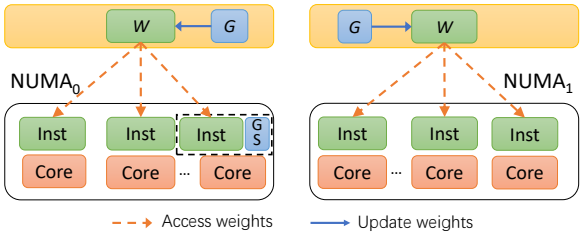

Figure 10: One-instance-per-core + gradient server (GS). All instances as well as the GS share the same weights and thus GS does not need to "copy weights" compared to PS.

cycles but also induces high synchronization delay. Different from PS, RAR reduces communication by organizing the instances into a ring which exchange gradients with the neighbors in two separate phases. However, its ring update greatly slows down the synchronization procedure (also evaluated in §5.2).

To improve the synchronization efficiency, ParaX designs a NUMA-aware gradient server (GS) mechanism which leverages shared memory to allow worker instances to directly access the weights. As shown in Figure 10, one CPU runs one gradient server which does not possess its own weights but shares the weights with the instances, in order to eliminate the costly per-iteration memory copy. On each NUMA there is a *complete* copy of the weights in the local memory, so as to avoid frequent remote memory accesses within an iteration.

Algorithm 1 shows the pseudocode of one training iteration based on the gradient server mechanism, where m is the number of NUMA nodes, p is the number of instances (equal to the number of cores in one-instance-per-core) on a NUMA, $W_i = W$ is (the complete copy of) the shared weights (W) stored in the local memory of NUMA i ($i = 0, 1, \cdots, m-1$), G is the global gradients distributed in the local memory of the n NUMAs, and g_i^j is the gradients calculated by instance j ($j = 0, 1, \cdots, p-1$) on NUMA i. Each instance individually performs `Forward()` and `Backward()` to calculate the local gradients (g_i^j), and aggregates g_i^j to the global gradients (G). Note that the aggregate step (G += g_i^j) may access the remote memory of another NUMA, but which is infrequent and will not affect the overall performance. After all local gradients have been aggregated to G, the gradient server will add G to the m copies of the weights W_i ($i = 0, 1, \cdots, m-1$) in the m NUMA's local memory. Also note that the m copies of W_i are identical because they are updated in exactly the same way.

Algorithm 1 A Training Iteration Based on GS

```
for all i = 0, 1, ··· , m-1 do in parallel
  for all j = 0, 1, ··· , p-1 do in parallel
    /**** Initialization ****/
    instance_i^j.bind(core_i^j)
    if Network is shallow then // Discussed in §4.4
      Calculate delay t according to its launching group
      instance_i^j.sleep(t)
    end if
    /**** Forward/Backward ****/
    Forward(d_i^j, W_i)
    g_i^j=Backward(d_i^j, W_i)
    G += g_i^j // Mutual exclusion
  end for
end for
/**** Synchronization ****/
for all i = 0, 1, ··· , m-1 do in parallel
  W_i += G · lr // Executed by gradient server
end for
```

4.3 One-Instance-Per-x-Core

Compared to the existing one-instance-per-CPU paradigm, one-instance-per-core increases the number of instances and effectively improves memory bandwidth utilization for many-core CPUs. For distributed training, however, too many instances in an iteration may affect the training accuracy [34, 50]. Consider an n-machine cluster where each machine has m p-core NUMAs, one-instance-per-core will lead to $n \times m \times p$ instances each processing one batch of input data in an iteration. Too many instances decreases the training accuracy, because either the total batch size of all instances is too large or the per-instance sample number is too small [65, 73].

To avoid the inaccuracy problem, we design the more general one-instance-per-x-core paradigm by extending one-instance-per-core: a CPU is assigned with multiple instances each of which is jointly processed by x cores. Existing studies [53] have shown that for distributed training in moderate-sized clusters the accuracy will almost keep unchanged if issuing at most $k = 4$ instances per NUMA node with appropriate batch sizes (e.g., 64) and adopting a linear scaling learning rate. For example, for distributed training with multiple machines each having a one-NUMA CPU ($p = 28$), assigning $k = 4$ instances to each machine (i.e., each instance is jointly processed by $x = p/k = 7$ cores) can guarantee the accuracy.

We have partially implemented one-instance-per-x-core for ParaX on a single machine, which will be evaluated together with one-instance-per-core in §5.2 and §5.4. For distributed training with multiple machines, however, one-instance-per-x-core requires tight cooperation between the NUMA-aware synchronization mechanism (i.e., gradient server discussed in §4.2) and the distributed communication framework (PS or RAR) for both intra- and inter-machine parameter updates, which has not yet been implemented in ParaX and will be studied in our future work.

4.4 Delayed Initiation

The randomness-based overlap scheduling policy (§4.1) effectively overlaps the executions of access- and compute-intensive operations for DNN models. However, a potential problem of overlap scheduling is that at the first few layers of a *training* iteration, the randomness might not be enough for sufficient operation overlapping. Although not a problem for deep networks with a large number of layers, it may affect the performance of relatively *shallow* networks of which the total number of layers is small. Therefore, ParaX complements overlap scheduling with *delayed initiation* [74] to address this issue.

Consider a CPU with m NUMA nodes each having p cores participate in the training. For one-instance-per-core, ParaX has totally $m \times p$ worker threads organized into m thread pools, each processing one instance on one core. We divide the p threads of each NUMA into n subsets each having p/n threads, and select one subset from each of the m pools to form a *launching group* (consisting of totally mp/n threads). At the beginning of an iteration, ParaX in turn initiates the n launching groups with an interval of t between two successive initiations, where t is a configurable parameter (usually at the scale of tens of milliseconds) to explicitly overlap all the launching groups within the duration of the first group's first few layers.

4.5 Discussion

To satisfy the performance requirement of common desktop, server, and cloud applications, currently CPU memory has much lower latency and higher capacity than GPU memory. However, CPUs view memory bandwidth as a secondary performance metric, as most applications running on CPUs do not have such high bandwidth requirement.

As demonstrated in Figure 1, the low CPU memory bandwidth causes severe bandwidth contention and affects the execution of access-intensive operations in DL training and inference, consequently lowering the performance of CPU-based DL. Although ParaX effectively alleviates the bandwidth contention problem by overlapping the access-intensive operations with the compute-intensive ones, the software-based solution will be possibly not able to fully exploit the computing capacity of modern many-core CPUs, as they have increasingly higher FLOPS which will be even comparable with GPUs in the near future. Our analysis on bandwidth requirement of memory accesses (§3) implies that the bandwidth contention problem could be addressed by designing new DL-friendly CPU memory architecture, where we could add an additional level of on-chip high-bandwidth cache with less stringent latency requirement that can be below or in parallel with the L3-cache, possibly integrated with different cache scheduling policies.

5 EXPERIMENT

We have implemented ParaX on MXNet (v1.5 with default configuration). We choose MXNet as the basis for implementing ParaX, mainly because (i) MXNet performs better than other mainstream platforms in many-core CPU based DL (as evaluated in §1), and (ii) the modular design of MXNet facilitates the integration of ParaX. We have realized one-instance-per-core (as well as one-instance-per-x-core) with overlap scheduling and gradient servers. Delayed initiation is disabled in our evaluation. The machine installs one two-NUMA Intel Platinum 8280 CPU with 28 2.70 GHz cores per NUMA (totally 56 cores), 39424KB L3-Cache, 192GB six-channel DDR4-2933 memory, and two Intel 750 PCIe 400GB NVM-Express SSDs.

We evaluate ParaX in training and inferring various DNN models including image recognition (ResNet [39], MobileNet [44], and Inception-BN [46]) and natural language processing (LSTM [43] and GNMT [71]),

Next we first briefly introduce these DNN models.

First, ResNet adds shortcut connections between layers in the DNN networks, so as to avoid the problems of gradient vanishing and exploding when the number of layers is high.

Second, MobileNet replaces common convolutions with depth-wise convolutions to reduce parameters while keeping accuracy for mobile image processing.

Third, Inception-BN adds BN layers after the convolution layers in the Inception network, which contains various convolution kernels in each sub-module to adapt to features at different scales.

Fourth, LSTM (long short-term memory) uses feedback connections to selectively remember useful patterns in long sequences. An LSTM has four GEMM layers, four `sigmoid` activation layers, and two `tanh` activation layers.

If not specified, the data type used in our evaluation is FP32. The setting of per batch size in ParaX is relatively straightforward: too large (> 64) batch sizes will tend to exceed the memory capacity (§5.3) since ParaX adopts one-instance-per-core for many-core CPUs, and too small batch sizes (< 32) will lead to inaccuracy for training and low throughput for inference [37, 38]. Therefore, if not specified, for training the per-instance batch size is $B = 64$ (which ensures convergence) and thus the *total* batch size is B times the number of instances, and for inference the *total* batch size is a fixed value ($64 \times 56 = 3,584$). An exception is that for ResNet50/ResNet101 on ImageNet, when training with 56 instances ParaX uses $B = 32$ (instead of 64) and thus the total batch size is 1792, and when inferring ParaX uses a fixed total batch size of 1792 (instead of 3584), to avoid exceeding the memory capacity. This is because higher batch sizes for them will exceed the maximum memory capacity of the machine (as evaluated in §5.3).

In the rest of this section, our experiments seek to answer the following questions:

- How does ParaX perform when training and inferring various DNN models, compared to the original MXNet that assigns one single instance to the many-core CPU? And what is the resource utilization and scalability (§5.1)?
- How (and why) does the gradient server mechanism outperform the state-of-the-art PS and RAR communication mechanisms in per-iteration parameter update for model training (§5.2)?
- What is the impact of the total batch sizes on the training and inference performance of ParaX as well as the memory footprint (§5.3)?
- And how does ParaX perform in various applications including image recognition and NLP (§5.4)?

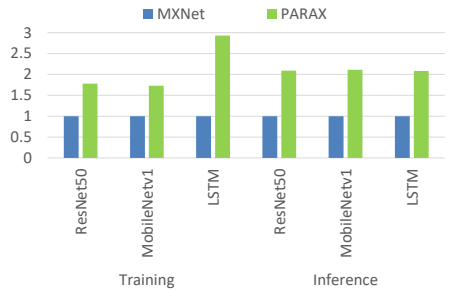

Figure 11: ParaX vs. MXNet.

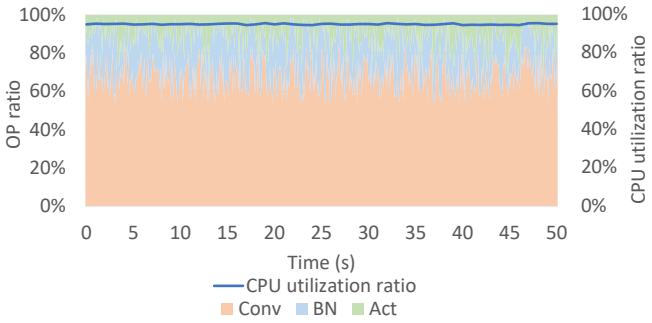

Figure 12: OP ratios & CPU utilization.

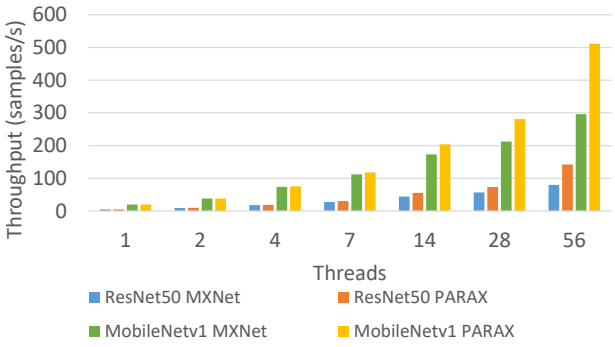

Figure 13: Training scalability.

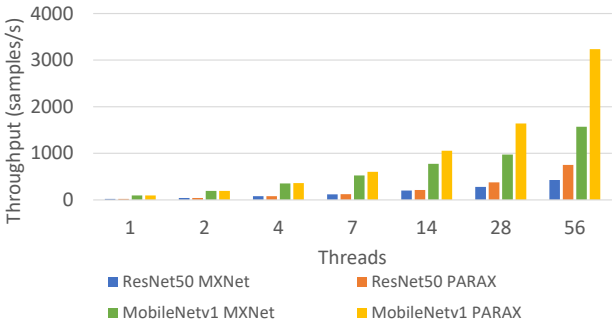

Figure 14: Inference scalability.

5.1 Micro Benchmarks

We evaluate the throughput of ParaX in training and inferring ResNet (with 50 layers and using ImageNet [30]), MobileNet-v1 (with 28 layers and using ImageNet), and LSTM (with 2 layers 650 hidden neurons per layer and using the Sherlock Holmes dataset [8]). ParaX adopts one-instance-per-core and gradient servers, and relies on randomness for layer overlapping. ParaX does not adopt delayed initiation since the networks are deep enough. For comparison we also evaluate MXNet, which adopts one-instance-per-CPU and thus has no synchronization overhead for training.

Figure 11 shows the speedups of ParaX to MXNet, where for training and inference the speedups are respectively 1.73× ∼ 2.93× and 2.08× ∼ 2.11×. Unlike ResNet and MobileNet-v1, LSTM training is sensitive to the per-instance batch size (B), and thus we also evaluate the throughput of MXNet for LSTM with $B = 128$ and 256. The result (not shown in Figure 11 for brevity) shows that ParaX has the speedups of 2.76× and 2.49×, respectively.

The advantage of ParaX mainly comes from two factors, namely, (i) the gradient server update mechanism (which we will evaluate in §5.2), and (ii) the overlapping of the compute- and access-intensive operations. Therefore, we also measure the degree of overlapping, i.e., the ratios of the convolution/BN/Activation operations during the inference of ResNet50 on ParaX. The result is shown in Figure 12, which demonstrates that the executions of the different kinds of operations are effectively overlapped.

We further measure the effectiveness of overlapping scheduling, i.e., the CPU utilization during the inference of ResNet50 on ParaX, with a total batch size of 1792. Figure 12 shows the CPU utilization for a period of 50 seconds (measured with vtune [9]), where the utilization keeps always high. The mean utilization of the entire training procedure achieves as high as 94.8%. We also measure the mean memory bandwidth utilization during the inference, which (not shown in this figure) is about 88.6%, much higher than that of the original MXNet (Figure 1). Compared to the original one-instance-per-CPU paradigm adopted by current DL platforms, ParaX significantly improves the CPU and memory bandwidth utilizations.

We also evaluate the scalability of ParaX and compare it with that of MXNet, respectively in training and inferring ResNet50 and MobileNet-v1. We increase the number of involved cores from 1 to 56, and measure the corresponding throughput of ParaX and MXNet. The results are depicted in Figures 13 and 14, where ParaX has much better scalability than MXNet in both training and inference. This is because ParaX not only maximizes the utilizations of memory bandwidth and CPU cycles (in both training and inference) but also minimizes the synchronization overhead of multi-instance parameter updates (in training).

5.2 Gradient Servers

This subsection compares GS (gradient server) of ParaX with the state-of-the-art PS (parameter server) and RAR (ring-allreduce) mechanisms for per-training-iteration parameter update.

We first evaluate the throughput when training ResNet50 on ParaX with different numbers of instances, adopting gradient server (GS), parameter server (PS), and ring-allreduce (RAR), respectively. The PS mechanism has already been integrated in the original MXNet, and RAR is ported from Horovod [68]. Since PS and RAR

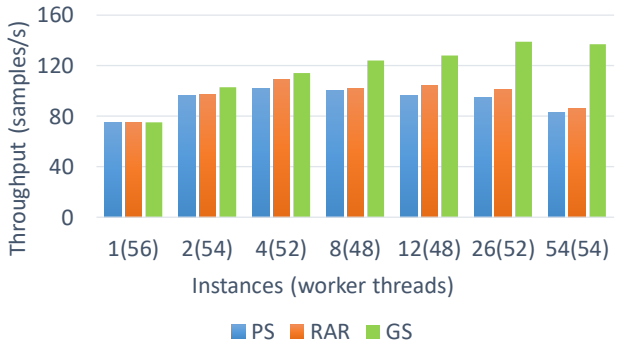

Figure 15: Gradient server vs. PS/RAR (adopting one-instance-per-x-core). For example, 4(52) means 4 instances on 52 cores, i.e., one-instance-per-13-core.

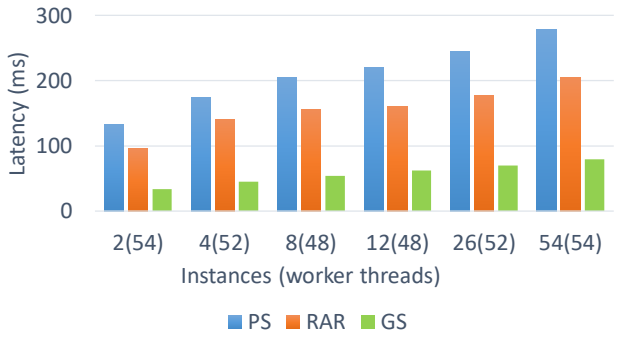

Figure 16: Update latency comparison (adopting one-instance-per-x-core).

are originally targeted for distributed scenarios, they both need a dedicated server thread occupying one core on each NUMA node. Consequently, there only remain at most $56 - 2 = 54$ cores on the 2-NUMA CPU available for training instances. Since all the instances must occupy the same numbers of cores, several cores have to be wasted in some of the tests. For example, if we issue four instances then each instance has at most 13 cores, and thus the total number of cores available for training is 52. Consequently, $54 - 52 = 2$ cores are wasted.

For fairness we use the same numbers of cores when comparing different mechanisms (even though all the 56 cores can be used for training in GS). The result is shown in Figure 15, where the numbers in the parenthesis represent the actual numbers of cores (*worker threads*) used for specific numbers of *instances*. For example, the result of 4(52) represents the training throughput when using 4 instances and 52 cores.

The throughput almost always keeps growing for the GS mechanism as the number of instances increases, except for 54 instances when ParaX has to use batch size $B = 32$ (instead of 64) due to the memory capacity limitation (evaluated in a later experiment in §5.3). In contrast, both the PS and RAR mechanisms suffer from lower throughput when using more than four instances. When using 54 instances with PS and RAR, the per-iteration synchronization overhead almost counteracts *all* benefit of layer overlapping,

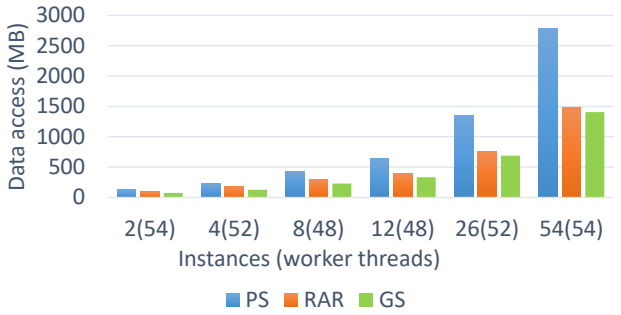

Figure 17: Data accessed for parameter synchronization in an iteration (adopting one-instance-per-x-core).

if we compare the two results of 1(56) and 54(54). This is because (as the numbers of instances increase) the cost of per-iteration parameter update greatly counteracts the benefit of layer overlapping. Note that when using only one instance for all the 56 cores all the three mechanisms have the same throughput, because no synchronization occurs between the cores.

To understand the throughput advantage of GS (gradient server) over PS (parameter server) and RAR (ring-allreduce), we measure the latency of per-iteration parameter update in training the ResNet50 model, adopting the same configurations as the above experiment. The result is shown in Figure 16, where the latency increases as the numbers of instances increase for all the three update mechanisms. The GS mechanism always has much lower latency compared to PS and RAR, because the NUMA-aware GS mechanism uses shared memory to keep one copy of shared weights for each NUMA, so as to eliminate the costly memory copy overhead. Besides lower update overhead, another important advantage not shown in this experiment is that GS can use all the cores for training while PS and RAR have to waste at least two cores dedicated for updating parameters.

We also measure the volumes of data access (including both the volume of data read and the volume of data write) for parameter synchronization in a training iteration, respectively for the GS, PS, and RAR mechanisms. The result is shown in Figure 17, where the volume of data access of the PS mechanism is about twice that of GS. This is because in PS both the `pull` and `push` operations perform memory copies while in GS only `push` needs to do so owing to the shared weights mechanism. Note that although RAR has only slightly higher data accesses than GS, it has much higher synchronization overhead as the instances are organized into a ring and exchange gradients with the clockwise neighbors in two separate phases.

5.3 Impact of Batch Sizes

To understand the impact of batch sizes on the performance, in this subsection we evaluate the training and inference throughput of ResNet50 and MobileNet-v1 on ParaX, as a function of the total batch sizes. For training, we use a fixed per-instance batch size $B = 64$ and vary the numbers of instances (n) from 1 to 56. For inference, we first use 1, 4, 14, and 56 instances each with $B = 1$, and afterwards fix the number of instances (= 56) and increase B up to 64. The results are shown in Figures 18 and 19, where the

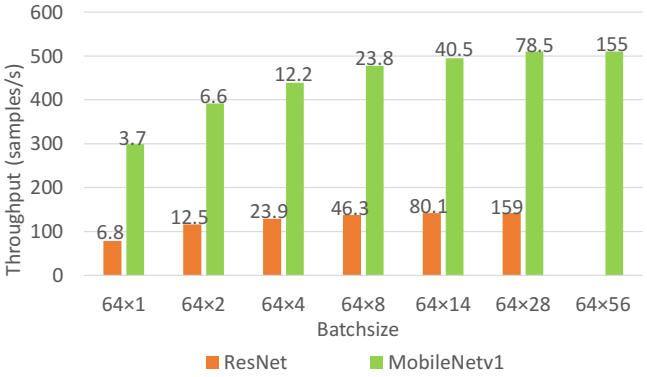

Figure 18: Training throughput vs. total batch size = $B \times n$ (n is instance number) on ParaX. Memory footprint (in GB) is labeled on the bars.

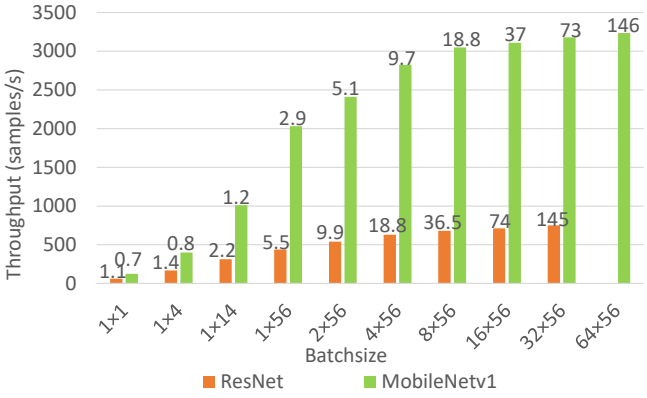

Figure 19: Inference throughput vs. total batch size = $B \times n$ (n is instance number) on ParaX. Memory footprint (in GB) is labeled on the bars.

throughput increases fast at first and slowly after the total batch sizes are relatively high. For training, this is because after 64×14 the numbers of instances have the marginal effect on memory bandwidth utilization; while for inference, this is because after 16×56 the batch sizes have only marginal effect on computation efficiency.

Figures 18 and 19 also label the memory footprint (in GB) on the bars for different total batch sizes, respectively when training and inferring ResNet50 and MobileNet-v1 on ParaX. Training has slightly more memory footprint than inference, because it experiences more complex processing (e.g., back propagation). As shown in the two figures, ResNet cannot use the total batch size 3584 ($= 64 \times 56$) because it will exceed the maximum memory capacity (192 GB). Since the operations have already been sufficiently overlapped, the batch size and memory capacity are not a bottleneck for the performance of ParaX. Almost all the models (except RestNet50 and ResNet101) evaluated in this section can have a total batch size of 3584 when we have 192 GB memory. We also test various total batch sizes on P100, and the result (not depicted due to lack of space) shows that its maximum total batch size is 448 for both ResNet50

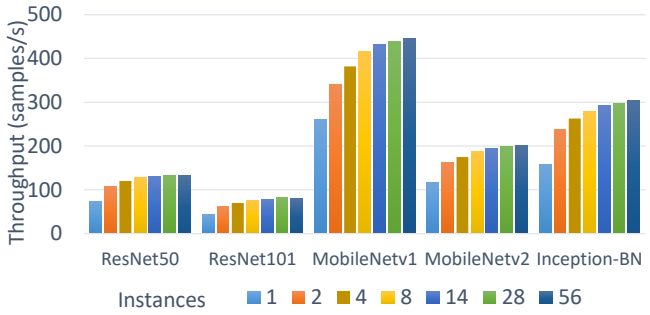

Figure 20: Training with ImageNet.

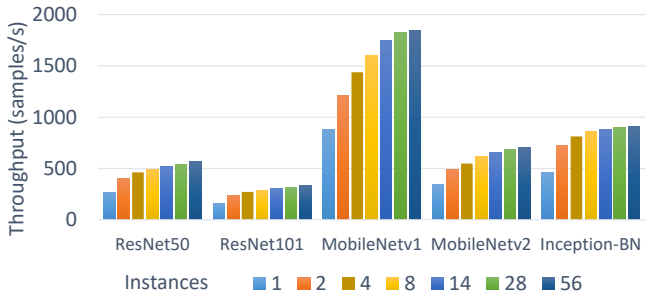

Figure 21: Inference with ImageNet.

and MobileNet-v1 due to the relatively small GPU memory capacity (16 GB for P100).

5.4 Applications

In this section we in turn evaluate ParaX with more applications (for image recognition and natural language processing).

5.4.1 Image Recognition. We evaluate ParaX in training and inferring various models (ResNet, MobileNet, and Inception-BN) for image recognition. On the Intel 8280 CPU with two NUMA nodes each having 28 cores, we vary the numbers of instances (from 1 to 56) to compare the throughput of various ParaX configurations of one-instance-per-CPU, one-instance-per-x-core, and one-instance-per-core. The datasets include ImageNet (ILSVRC2012) [30] and CIFAR10 [1]. ImageNet has 1,280,000 training images from 1000 categories. The images in ImageNet are all natural images with high resolutions (224×224). In contrast, the CIFAR10 dataset contains 60,000 relatively small (32×32) images.

Figures 20 and 21 show the throughput of five CNN models (ResNet50, ResNet101, MobileNet-v1, MobileNet-v2, and Inception-BN) on ImageNet, respectively for training and inference. As introduced at the beginning of §5, in training the per-instance batch size is $B = 64$ (except for 56-instance ResNet $B = 32$), and in inference the total batch size is a fixed value of 3584 (except for ResNet it is 1792). Note that all the cores are used in these experiments no matter the numbers of instances. ParaX achieves the best performance when issuing 56 instances for all the five models, demonstrating that the one-instance-per-core paradigm effectively improves the utilizations of memory bandwidth and CPU cycles.

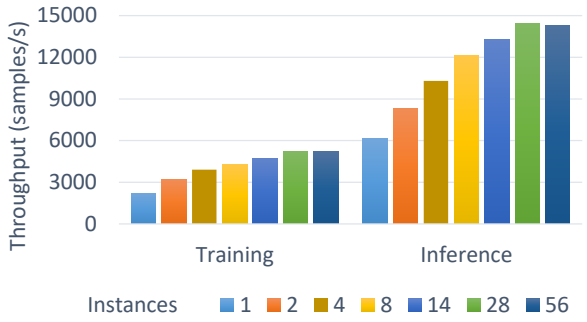

Figure 22: Cifar10-ResNet50.

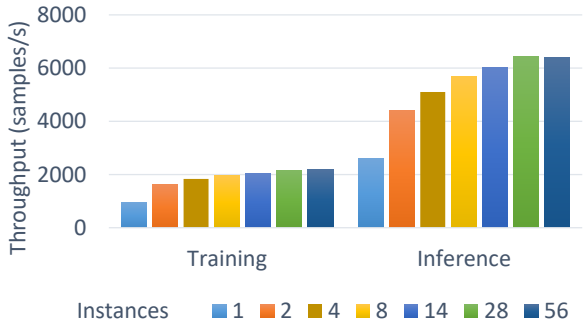

Figure 23: Cifar10-ResNet101.

Figures 22 and 23 respectively show the throughput of ResNet50 and ResNet101 with CIFAR10, for both training (with per-instance batch size $B = 64$) and inference. (with the total batch size = 64×56 = 3584). The result is similar to that with ImageNet, except that for inference the best throughput is achieved with 28 instances in both ResNet50 and ResNet101. This is mainly because the number of channels of ResNet is compressed when training with Cifar10, which increases the dependence on the memory bandwidth and thus lowers the effectiveness of the overlapping scheduling.

5.4.2 Natural Language Processing. We evaluate ParaX in training and inferring the Word-LM (word language model) and GNMT (Google neural machine translation) models for natural language processing, as the numbers of instances increases from 1 to 56. We use the Sherlock Holmes [8] dataset for Word-LM and IWSLT2015 [31] for GNMT.

For Word-LM, we train and infer a two-layer LSTM, which has 650 hidden neurons on each layer. Figure 24 shows the throughput of Word-LM for both training (with per-instance batch size $B = 64$) and inference (with the total batch size = 64×56 = 3584). The result is similar to that for CNN models (§5.4.1), where the throughput increases as the number of instances increases, but Word-LM scales better than CNN models because it mainly contains GEMM (rather than convolution) operations.

We evaluate the GNMT model which has 128 hidden neurons on each layer with the same configurations as the above experiment. GNMT consists of three separate networks (encoder, decoder, and attention), where the encoder and decode contain an eight-layer LSTM each so as to translate entire sentences with high accuracy.

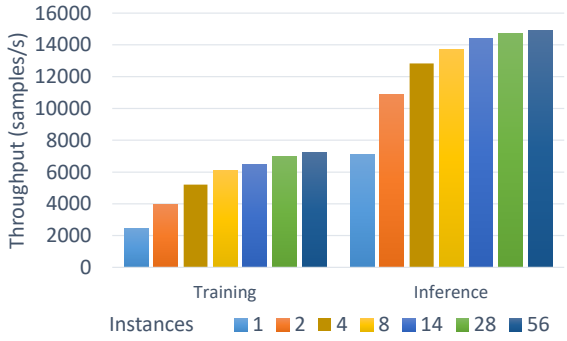

Figure 24: Word-LM.

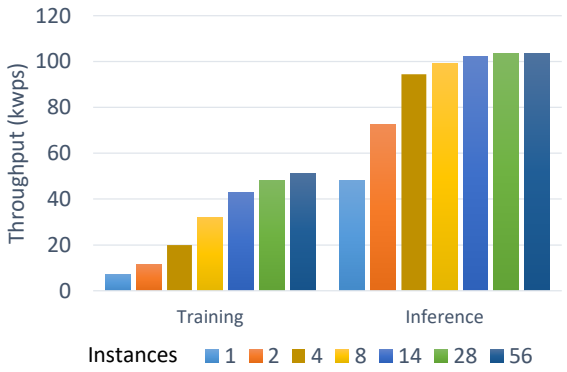

Figure 25: GNMT.

The result is shown in Figure 25, where the throughput is measured as the number of (1000) words translated per second. From the result we get similar conclusion as in the Word-LM experiment.

6 RELATED WORK

6.1 Memory Wall for DL Computation

With the development of semiconductor technology, many-core CPUs have broken through the power limitation and greatly improved the performance via core parallelism. However, due to the mismatch between the total compute capacity and the memory bandwidth, memory wall [42, 72] lowers the DL performance of CPUs compared to GPUs and accelerators (such as Brainwave [32], Centaur [45], and TPU [64]). The memory bandwidth has become a major performance bottleneck as the number of CPU cores greatly increases.

Memory bandwidth contention has been widely studied for deep learning on GPUs and accelerators. For example, Prophet [20] and Baymax [21] realize precise QoS prediction on non-preemptive accelerators to improve bandwidth utilization. Compared to these studies, ParaX removes the per-layer barriers to mix different operations and improves utilization based on operation execution randomness without precise prediction.

Deep learning on CPUs has drawn less attention than on GPUs and accelerators. For example, TensorDIMM [56], DeepRecSys [36],

and RecNMP [49] improve CPU-based recommendation performance with tensor operations and near-memory processing; DeepCPU [76] improves CPU-based RNNs by sharing weights in L3 cache and leveraging fusion. Compared to these studies that focus on specific deep learning applications and models, ParaX is applicable to all applications and accelerates all models based on one-instance-per-core, where shared weights in L3-cache is just a byproduct of gradient servers and fusion is orthogonal to ParaX.

For x86-based CPU architectures, the math kernel library for Deep Neural Networks (MKL-DNN a.k.a. oneDNN [28]) has developed a series of optimizations for specific operations (like convolution). MKL-DNN alleviates the mismatch between compute capacity and memory bandwidth via architecture-specific techniques such as optimal threading, cache-blocking, vectorization, and register-blocking. For example, Intel has changed the memory layout for many-core CPUs from "NCHW" to "nChw8c" [28], which makes memory accesses in the innermost loops as contiguous as possible to increase the cache hit ratio. These optimizations demonstrate the problem of memory bandwidth contention and inspire our design.

6.2 DL on Multi-Core CPUs

DimmWitted [75] was an early work which studied three approaches for data analytics based on multi-core (but not many-core) CPUs with last-level cache (LLC) optimization, namely, PerCore, PerNode, and PerMachine, respectively corresponding to one-instance-per-core/-node/-CPU in this paper.

DimmWitted mainly targeted linear models like support vector machine, logistic regression, least squares regression, and linear programming. For neural networks, DimmWitted updated model parameters for each *epoch* instead of each *iteration*, which is different from all state-of-the-art methods including ParaX. With the per-epoch update paradigm, the authors found that the PerNode approach with communication through the last-level cache performed the best for a shallow (seven-layer) neural network on a small dataset (MNIST [2]).

DimmWitted achieved different conclusion from ParaX's result that one-instance-per-core is optimal for most cases (in, e.g., Figures 20 $\sim$ 25). This is not only because DimmWitted was applied to a shallow network on multi-core CPUs (with 6 $\sim$ 10 cores per NUMA) using a non-mainstream (per-epoch) parameter update paradigm, but also because its PerCore approach did not adopt the NUMA-aware GS mechanism (§4.2).

6.3 Optimizations for DL

Recent studies propose to limit the model sizes to satisfy the low latency requirement of emerging applications. For example, Quantization [47] converts floating-point arithmetic in DNNs into fixed-point (e.g., from `fp32` to `int8`) which could theoretically achieve four times speedup for inference making real-time inference practical on mobile devices.

Further, researchers propose to use fewer bits for quantification with higher speedups [26, 57, 66]. For example, BWN (Binary Neural Network) [26] uses binary weights, TWN (Ternary Weight Network) [57] uses weights of +1, 0 and −1, and XNOR-Net [66] adopts binary convolutional neural networks. However, quantization inevitably reduces the accuracy of the models due to compression and approximation.

Tensor compilers (such as TVM [23], GLOW [67], and Tensor-Comprehensions [70]) provide end-to-end optimizations under different architectures, so as to ease model deployment for developers by summarizing the optimization experiences of various DL operations from a high level of abstraction and allowing users to explore efficient implementation space in an automated or semi-automated way. These designs are orthogonal to the one-instance-per-core paradigm and we will integrate ParaX with these compilers in our future work.

7 CONCLUSION

Large numbers of CPUs are now heavily used for DL in the cloud, routinely running training and inference tasks since they would otherwise sit idle during off-peak periods. Unfortunately, the state-of-the-art platforms cannot support efficient DL on many-core CPUs, because they overlook the low memory bandwidth property of many-core CPUs and assign a single instance to one CPU (in the same way as in GPU-based DL). This causes synchronous executions of operations (layers) and consequently results in intermittent memory bandwidth contention and CPU starvation.

This paper proposes ParaX, an effective method that improves memory bandwidth utilization for scaling DL to many CPU cores. The key idea behind ParaX is to assign one instance to each core (instead of to each CPU), so as to overlap the cores' executions of different operations. ParaX designs the ultralight scheduling policy and gradient server mechanism. We have implemented ParaX and evaluated it with various models. The results show that ParaX achieves much higher DL performance compared to existing methods on many-core CPUs.

In the future, we plan to enhance ParaX to fully support distributed deep learning on multiple machines installing many-core CPUs. Besides, ParaX follows synchronous SGD to ensure convergence, and we will study ParaX with asynchronous SGD. We expect ParaX to achieve similar accelerations in DL on many-core CPUs for other MKL-DNN-based platforms like TensorFlow and PyTorch, because they all suffer from the low memory bandwidth utilization problem caused by the per-layer execution barriers. Currently ParaX is implemented on MXNet, and we will implement ParaX on other popular deep learning platforms. We will also integrate ParaX with tensor compilers. The bandwidth contention problem could be addressed by adding an additional level of on-chip high-bandwidth cache, which will also be studied in the future. The source code of ParaX is available at [10].

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments. We thank Zheng Qin and Huiba Li for the discussion at PDL. Lujia Yin and Yiming Zhang are co-primary authors. This work was supported by the National Key Research and Development Program of China (2016YFB1000101) and the National Natural Science Foundation of China (NSFC) under Grant Numbers 61772541, 61872376, and 61932001.

REFERENCES

[1] 2009. https://www.cs.toronto.edu/~kriz/cifar.html. Online; accessed Jan 31, 2021.

[2] 2012. http://yann.lecun.com/exdb/mnist/. Online; accessed Jan 31, 2021.

[3] 2016. https://www.nvidia.com/en-us/data-center/tesla-p100/. Online; accessed Jan 31, 2021.

[4] 2017. https://software.intel.com/en-us/articles/intel-avx-512-instructions. Online; accessed Jan 31, 2021.

[5] 2019. https://software.intel.com/en-us/articles/performance-boosting-in-seldon. Online; accessed Jan 31, 2021.

[6] 2019. https://www.nvidia.com/en-us/design-visualization/quadro/rtx-8000/. Online; accessed Jan 31, 2021.

[7] 2020. https://blog.udacity.com/2020/08/machine-learning-for-big-data.html. Online; accessed Jan 31, 2021.

[8] 2020. https://www.kaggle.com/idevji1/sherlock-holmes-stories. Online; accessed Jan 31, 2021.

[9] 2020. https://software.intel.com/en-us/vtune. Online; accessed Jan 31, 2021.

[10] 2020. https://github.com/nicexlab/parax-source. Online; accessed Jan 31, 2021.

[11] 2021. https://pytorch.org/. Online; accessed Jan 31, 2021.

[12] 2021. docs.aws.amazon.com/dlami/latest/devguide/deep-learning-containers-eks-tutorials-cpu-training.html. Online; accessed Jan 31, 2021.

[13] 2021. https://github.com/oneapi-src/oneDNN. Online; accessed Jan 31, 2021.

[14] 2021. https://developer.nvidia.com/about-cuda. Online; accessed Jan 31, 2021.

[15] 2021. https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html. Online; accessed Jan 31, 2021.

[16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 265–283.

[17] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. 2001. *Parallel programming in OpenMP*. Morgan kaufmann.

[18] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Józefowicz. 2016. Revisiting Distributed Synchronous SGD. *CoRR* abs/1604.00981 (2016). arXiv:1604.00981 http://arxiv.org/abs/1604.00981

[19] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. 2016. Revisiting distributed synchronous SGD. *arXiv preprint arXiv:1604.00981* (2016).

[20] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. 2017. Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*. ACM, 17–32.

[21] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. 2016. Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, Tom Conte and Yuanyuan Zhou (Eds.). ACM, 681–696. https://doi.org/10.1145/2872362.2872368

[22] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[23] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, and Luis Ceze. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[24] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[25] Tyson Condie, Paul Mineiro, Neoklis Polyzotis, and Markus Weimer. 2013. Machine learning for big data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 939–942. https://doi.org/10.1145/2463676.2465338

[26] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*. 3123–3131.

[27] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*. ACM, 191–198.

[28] Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidynathan, Srinivas Sridharan, Dhiraj Kalamkar, Bharat Kaul, and Pradeep Dubey. 2016. Distributed deep learning using synchronous stochastic gradient descent. *arXiv preprint arXiv:1602.06709* (2016).

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

[30] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[31] Nadir Durrani, Barry Haddow, Philipp Koehn, and Kenneth Heafield. 2014. Edinburgh's phrase-based machine translation systems for WMT-14. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*. 97–104.

[32] Jeremy Fowers, Kalin Ovtcharov, Michael K. Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2019. Inside Project Brainwave's Cloud-Scale, Real-Time AI Processor. *IEEE Micro* 39, 3 (2019), 20–28. https://doi.org/10.1109/MM.2019.2910506

[33] Evangelos Georganas, Kunal Banerjee, Dhiraj Kalamkar, Sasikanth Avancha, Anand Venkat, Michael Anderson, Greg Henry, Hans Pabst, and Alexander Heinecke. 2019. High-Performance Deep Learning via a Single Building Block. *arXiv preprint arXiv:1906.06440* (2019).

[34] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[35] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. 2013. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, 6645–6649.

[36] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S. Lee, David Brooks, and Carole-Jean Wu. 2020. DeepRecSys: A System for Optimizing End-To-End At-scale Neural Recommendation Inference. In *ISCA 2020*.

[37] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim M. Hazelwood, Mark Hempstead, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. 2020. The Architectural Implications of Facebook's DNN-Based Personalized Recommendation. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22-26, 2020*. IEEE, 488–501. https://doi.org/10.1109/HPCA47549.2020.00047

[38] Kim M. Hazelwood, Sarah Bird, David M. Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*. 620–629. https://doi.org/10.1109/HPCA.2018.00059

[39] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[40] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*. International World Wide Web Conferences Steering Committee, 173–182.

[41] Jonathan L Herlocker, Joseph A Konstan, Loren G Terveen, and John T Riedl. 2004. Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems (TOIS)* 22, 1 (2004), 5–53.

[42] Mark D Hill and Michael R Marty. 2008. Amdahl's law in the multicore era. *Computer* 41, 7 (2008), 33–38.

[43] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[44] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[45] Ranggi Hwang, Taehun Kim, Youngeun Kwon, and Minsoo Rhu. 2020. Centaur: A Chiplet-based, Hybrid Sparse-Dense Accelerator for Personalized Recommendations. (2020).

[46] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).

[47] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2704–2713.

[48] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 675–678.

[49] Liu Ke, Udit Gupta, Carole-Jean Wu, Benjamin Youngjae Cho, Mark Hempstead, Brandon Reagen, Xuan Zhang, David M. Brooks, Vikas Chandra, Utku

Diril, Amin Firoozshahian, Kim M. Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, and Xiaodong Wang. 2020. RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing. In *ISCA 2020*.

[50] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2016. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836* (2016).

[51] Jeffrey S Kimmel, Robert A Alfieri, A Miles, William K McGrath, Michael J McLeod, Mark A O'connell, and Guy A Simpson. 2000. Operating system for a non-uniform memory access multiprocessor system. US Patent 6,105,053.

[52] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix factorization techniques for recommender systems. *Computer* 8 (2009), 30–37.

[53] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997* (2014).

[54] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[55] Arun Kumar, Matthias Boehm, and Jun Yang. 2017. Data Management in Machine Learning: Challenges, Techniques, and Systems. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 1717–1722. https://doi.org/10.1145/3035918.3054775

[56] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. 2019. TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 740–753. https://doi.org/10.1145/3352460.3358284

[57] Fengfu Li, Bo Zhang, and Bin Liu. 2016. Ternary weight networks. *arXiv preprint arXiv:1605.04711* (2016).

[58] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 583–598.

[59] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J. Smola. 2014. Efficient mini-batch training for stochastic optimization. In *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*, Sofus A. Macskassy, Claudia Perlich, Jure Leskovec, Wei Wang, and Rayid Ghani (Eds.). ACM, 661–670. https://doi.org/10.1145/2623330.2623612

[60] Yongchao Liu, Bertil Schmidt, and Douglas L Maskell. 2010. CUDASW++ 2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. *BMC research notes* 3, 1 (2010), 93.

[61] Christopher D Manning, Christopher D Manning, and Hinrich Schütze. 1999. *Foundations of statistical natural language processing*. MIT press.

[62] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*.

[63] Janani Mukundan, Hillery Hunter, Kyu-hyoun Kim, Jeffrey Stuecheli, and José F Martínez. 2013. Understanding and mitigating refresh overheads in high-density DDR4 DRAM systems. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 48–59.

[64] Eric B. Olsen. 2018. RNS Hardware Matrix Multiplier for High Precision Neural Network Acceleration: "RNS TPU". In *IEEE International Symposium on Circuits and Systems, ISCAS 2018, 27-30 May 2018, Florence, Italy*. IEEE, 1–5. https://doi.org/10.1109/ISCAS.2018.8351352

[65] Chao Peng, Tete Xiao, Zeming Li, Yuning Jiang, Xiangyu Zhang, Kai Jia, Gang Yu, and Jian Sun. 2018. Megdet: A large mini-batch object detector. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 6181–6189.

[66] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*. Springer, 525–542.

[67] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. 2018. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907* (2018).

[68] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *CoRR* abs/1802.05799 (2018). arXiv:1802.05799 http://arxiv.org/abs/1802.05799

[69] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.

[70] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[71] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *CoRR* abs/1609.08144 (2016). arXiv:1609.08144 http://arxiv.org/abs/1609.08144

[72] Wm A Wulf and Sally A McKee. 1995. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news* 23, 1 (1995), 20–24.

[73] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. 2018. Imagenet training in minutes. In *Proceedings of the 47th International Conference on Parallel Processing*. ACM, 1.

[74] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*. 265–278.

[75] Ce Zhang and Christopher Ré. 2014. DimmWitted: A Study of Main-Memory Statistical Analytics. *Proc. VLDB Endow.* 7, 12 (2014), 1283–1294. https://doi.org/10.14778/2732977.2733001

[76] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, Elton Zheng, Olatunji Ruwase, Jeff Rasley, Jason Li, Junhua Wang, and Yuxiong He. 2019. Accelerating Large Scale Deep Learning Inference through DeepCPU at Microsoft. In *2019 USENIX Conference on Operational Machine Learning, OpML 2019, Santa Clara, CA, USA, May 20, 2019*, Bharath Ramsundar and Nisha Talagala (Eds.). USENIX Association, 5–7. https://www.usenix.org/conference/opml19/presentation/zhang-minjia

[77] Shuxin Zheng, Qi Meng, Taifeng Wang, Wei Chen, Nenghai Yu, Zhiming Ma, and Tie-Yan Liu. 2016. Asynchronous Stochastic Gradient Descent with Delay Compensation for Distributed Deep Learning. *CoRR* abs/1609.08326 (2016). arXiv:1609.08326 http://arxiv.org/abs/1609.08326