Deep Learning Models on CPUs: A Methodology for Efficient Training

Quchen Fu[1,*,†], Ramesh Chukka[2], Keith Achorn[2],
Thomas Atta-fosu[2], Deepak R. Canchi[2], Zhongwei Teng[1],
Jules White[1] and Douglas C. Schmidt[1]

[1]*Dept. of Computer Science, Vanderbilt University Nashville, TN, USA*
[2]*Intel Corporation, Santa Clara, CA, USA*
E-mail: quchen.fu@vanderbilt.edu; ramesh.n.chukka@intel.com;
keith.achorn@intel.com; thomas.atta-fosu@intel.com; deepak.r.canchi@intel.com;
zhongwei.teng@vanderbilt.edu; jules.white@vanderbilt.edu;
d.schmidt@vanderbilt.edu
[†]*Corresponding Author*

Abstract

GPUs have been favored for training deep learning models due to their highly parallelized architecture. As a result, most studies on training optimization focus on GPUs. There is often a trade-off, however, between cost and efficiency when deciding how to choose the proper hardware for training. In particular, CPU servers can be beneficial if training on CPUs was more efficient, as they incur fewer hardware update costs and better utilize existing infrastructure.

This paper makes three contributions to research on training deep learning models using CPUs. First, it presents a method for optimizing the training of deep learning models on Intel CPUs and a toolkit called ProfileDNN,

---

which we developed to improve performance profiling. Second, we describe a generic training optimization method that guides our workflow and explores several case studies where we identified performance issues and then optimized the Intel® Extension for PyTorch, resulting in an overall 2x training performance increase for the RetinaNet-ResNext50 model. Third, we show how to leverage the visualization capabilities of ProfileDNN, which enabled us to pinpoint bottlenecks and create a custom focal loss kernel that was two times faster than the official reference PyTorch implementation.

Keywords: Training methodology, deep learning on CPU, performance analysis.

1 Introduction

Deep learning (DL) models have been widely used in computer vision, natural language processing, and speech-related tasks [1–4]. Popular DL frameworks include PyTorch [5], TensorFlow [6], and OpenVINO [7], etc. The hardware can range from general-purpose processors, such as CPUs and GPUs, to customizable processors, such as FPGA and ASICs, that are often called XPUs [8].

All these hardware variants make it hard to propose a universal method for efficient training of DL models. Since GPUs have dominated deep learning tasks, comparatively little attention has been paid to optimizing models running on CPUs, especially for training [9]. Previous DL model research conducted on CPUs focused largely on performance comparisons of CPUs and GPUs [10–13] or only focused on CPU inference [14].

A key question to address when optimizing training performance on CPUs is what metrics should guide the optimization process. Several metrics and benchmarks have been proposed to measure DL workload and training performance. For example, Multiply-Accumulate (MAC) has been used as a proxy for FLOPs to measure computational complexity for Convolutional Neural Network (CNN) models [15]. Time-to-Train (TTT) has been widely adopted to measure the training performance of a DL model by measuring the time models take to reach certain accuracy metrics. NetScore [16] was proposed as a universal metric for DL models that balances information density and accuracy.

Until recently, however, no widely accepted benchmark for DL models existed that incorporated a wide range of domain tasks, frameworks, and hardware. MLPerf [17] was proposed as a comprehensive DL benchmarking

suite to cover a variety of tasks and hardware. Many major tech companies have contributed to this effort by competing for better performance. Intel has been actively participating in the MLPerf challenge to improve the training performance of DL models across multiple domains.

To address portability issues related to DL models running on different hardware platforms, Intel has open-sourced the oneAPI Deep Neural Network Library (oneDNN) [18]. OneDNN is cross-platform performance library of basic deep learning primitive operations, including a benchmarking tool called *benchDNN*. Intel has also created optimized versions of popular frameworks with oneDNN, including Intel® Optimizations for TensorFlow and Intel® Extensions for PyTorch [19]. Few guidelines exist, however, for profiling and optimizing DL model training on CPUs.

Several fundamental research challenges must be addressed when training DL models on CPUs, including the following:

1. **How to locate bottlenecks**. Since frameworks with CPU-optimized kernels (such as Intel® Extention for PyTorch) are relatively new, generic model-level [20] profilers (such as the PyTorch Profiler [5]) are not oneDNN-aware. Moreover, low-level profilers like *benchDNN* can only benchmark performance at the operational level. Identifying primitive operations most critical for specific model/framework/hardware combination is thus essential so that low-level (e.g., oneDNN level) optimizations can accelerate performance significantly.

2. **How to fix bottlenecks**. While GPUs have well-established platforms (such as CUDA [21]) for kernel implementations, Deep Neural Network Libraries for CPUs are less well-known. It is therefore essential to understand how to rectify performance bottlenecks (e.g., by locating and implementing custom operation kernels for both forward and backward propagation), as well as adopting proper low-precision training so computing time can be reduced without sacrificing accuracy for CPUs.

3. **How to set achievable goals**. Projections for CPUs are often done in a crude way by dividing CPU performance in FLOPs over FLOPs required for model training. In a computation-bounded scenario, however, it is essential to create an experiment-based projection for deep learning models so that the goal is realistically achievable, i.e., not only theoretically achievable, but also considers hardware limits and kernel optimizations.

To address these challenges, we designed a structured top-down method that helped us prioritize different optimizing options for training DL models

(e.g., RetinaNet [22]) on CPUs. Incorporating this new approach, we also developed a DL performance profiling toolkit called ProfileDNN that is oneDNN-aware and supports profiling and projection at the model level, thereby bridging the gap for oneDNN-specific model-level projection.

The remainder of this paper is organized as follows: Sections 2.1 and 2.2 summarize different profile tools and their contribution to locating hot spots and discrepancies; Section 2.3 describes projection goal and procedure, as well as ProfileDNN's structure and workflow; Sections 2.4 through 2.8 discuss recommendations and approaches to enable efficient training without sacrificing accuracy; Section 3 analyzes the training efficiency and convergence under distributed situation; and Section 4 presents concluding remarks and our future work. All experiments in this paper were performed on Intel Xeon Cooper Lake processors.

2 Method Summary

This section summarizes the method component of our contribution for optimizing training on CPUs. Our goal is to provide a structured approach for users to optimize training DL models on CPUs. Our method adopts a top-down approach similar to what [23] described, which aims to *locate the critical bottlenecks in a fast and feasible manner*.

In our experience DL workflows can be categorized into three stages: profiling, projection, and optimization. Figure 1 shows how each stage can be decomposed into different tile groups. Users are advised to follow the method groups from left to right, as each group benefits from the previous group's results. Our toolkit ProfileDNN can work both as a profile tool and a projection tool. Framework-level profilers like Tensorflow (TF) profiler and Torch Profiler are popular tools partly because they are not hardware dependent (work on any hardware that runs Pytorch or Tensorflow), however, they don't have support for executing low-level primitive kernel operations, which are vital for performance projection. Low-level profiling/projection tools can
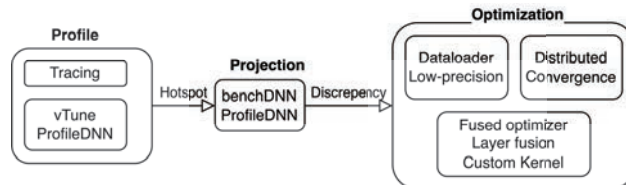

Figure 1   DL workflow method decomposition.

Table 1   Comparison of DL profiling tools

	ProfileDNN	BenchDNN	TF Profiler	DLProf	ZenDNN
Developer	Ours	Intel	Google	Nvidia	AMD
Devices Support	OneDNN hardware	CPU	CPU / GPU / TPU	GPU	CPU
Result Format	Chart/Log/Table	Log/Table	UI/Log	UI/Log	Log
Mode	Observe/Execution	Execution	Observe	Observe	Observe
Kernel Level	High/Low	Low	High	Low	Low

measure kernel execution time, therefore, are traditionally hardware-bound. Deep Learning Profiler (DLProf) was a profiling tool developed by Nvidia that map correlation between profile timing, kernel information and a Deep Learning model; ZenDNN was the equivalent product launched by AMD that supports CPU profiling; BenchDNN which runs on Intel CPU went one step further by supporting primitive operation benchmarking, thus can potentially be adapted into a projection tool. Our ProfileDNN, as shown in Table 1, has support for both high-level profiling and low-level (kernel) projection and thus can act as a bridge for framework to kernel operation translation, therefore create execution-based DL model performance projection.

2.1 Profile and Tracing

During the profile stage, users should observe the breakdown of operation kernel components of the DL model and their relative significance. Special attention should be paid to discrepancies between their model and data versus the reference implementation and original use case. For example, do all the major kernel operations of reference exist in their model? Likewise, does the kernel component percentage remain roughly the same? If the answer to either question is "no" the code may perform worse due to poor oneDNN kernel adoption.

ProfileDNN helps users better compare the distribution of the kernel components by producing intuitive visualization. This tactic was also adopted by vTune [24]. ProfileDNN supports all primitive kernels *(conv, pool, matmul, reorder, etc)* from *benchDNN*.

Convolutional Neural Networks (CNNs) [25], Recurrent Neural Networks (RNNs) [26], and Transformers [27] are some of the most popular Neural Network models today. ProfileDNN can break down the primitive operations by type and directory, as shown in Figure 2(a)–(c). We found that both CNN and RNN models spend more time doing back-propagation than forward-propagation. Transformer models consist mostly of inner product
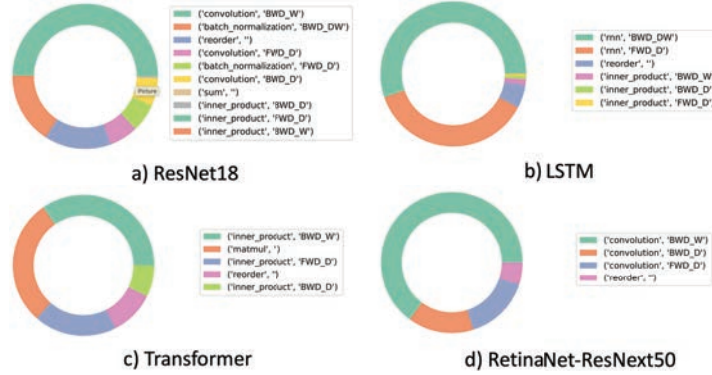
Figure 2   Comparison of primitive operations across models.

Frontend Bound		Backend Bound		Speculation &Retiring
Latency	BandWidth	Core	Memory	

Figure 3   The vTune design.

and matrix multiplication, which correspond to the `softmax` operation that is often a performance bottleneck for transformer-based models [28]. Figure 2(d) also plots the breakdown of the RetinaNet-ResNext50 model, which is a complicated object detection model whose distribution is similar to the CNN in Figure 2(a).

A primitives-level breakdown is often sufficient to locate model bottlenecks since many DL training tasks are computation-bound. In the case of a memory/cache bounded scenario, however, trace analysis is needed to inspect the orders in which each operation runs. A trace is an ordered set of span sequences, where each span has an operation name, a start and end timestamp, as well as relations to other spans (child process, etc). If a trace is highly fragmented there is significant context switching, so a custom merged operator may help improve performance.

VTune is another powerful tool for profiling CPU performance based on a top-down method [23]. vTune divides the CPU workflow pipeline into frontend and backend, with the former bounded by latency and bandwidth, and the latter bounded by core (computation) and memory (cache), as shown in Figure 3. The first round of profiling should be a generic hot spot analysis on training the model to determine costly operations.

The profiling round can be followed by micro-architecture exploration that measures CPU utilization rate (spinning time), memory bandwidth and
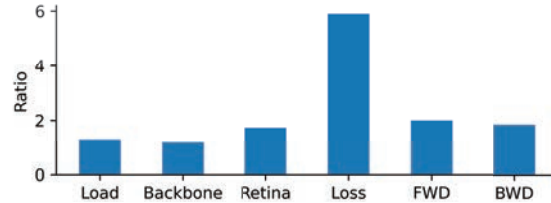
Figure 4 Open image vs COCO training time ratio breakdown.

cache (L1, L2, or L3) miss rate. After pinpointing the primitive operation with the most computation-heavy footprint, algorithm- or implementation-level optimizations can be applied. If memory is the bottleneck, memory access and IO analysis can also be performed on individual operations.

2.2 Data Discrepancy

A commonly overlooked discrepancy is the difference between the reference dataset and the custom dataset. The data distribution can not only affect the performance of the same model, it can sometimes change the model itself. For example, RetinaNet-ResNext50 is a classification model that changes structure based on the number of classes from the dataset.

After we switched the dataset from COCO [29] to OpenImage [30], the training time increased dramatically. We found that the dataset size increased 10 times, but the training time per epoch increased 20 times, which is not proportional. Part of this increase can be attributed to a bigger fully-connected (FC) layer in the backbone. In particular, we found that the major increase in time is within the focal loss calculation caused by three times more classes, as shown in the detailed breakdown in Figure 4.

Trace analysis also supported this conclusion by showing that one-third of the backward calculation time was spent on focal loss. We addressed this issue by implementing our custom focal loss kernel, as discussed in Section 7.

2.3 Projection and Toolkit Structure

Projection of DL models aims to determine the theoretical performance ceiling of a specific model/framework/hardware combination. Intel has an internal tool that can perform projection for DL models, though this tool currently requires significant manual setting and tuning. *BenchDNN* can be used to predict performance on specific hardware automatically, but only on one operation at a time. We therefore designed ProfileDNN to combine
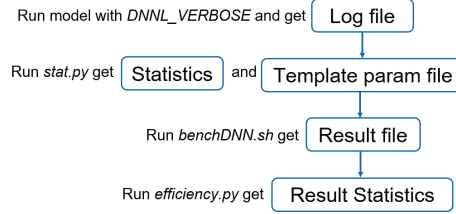
Figure 5   Toolkit structure and flow pipeline.

Table 2   Summary of benchDNN parameters

Name	Example
Driver	conv, relu, matmul, rnn, bnorm
Configuration	u8s8u8, s8f32
Directory	FWD_I, BWD_D, BWD_W
Post_Ops	sum+eltwise_relu
Algorithm	DIRECT
Problem_batchsize	mb1, mb32
Problem_input	id4ih32iw32
Problem_output	id16ih16iw16
Problem_stride	sd2sh4sw4
Problem_kernel	kd2kh3kw3
Problem_padding	pd1ph1pw1
Problem_channel	ic16oc32

the advantages of both existing tools since it can perform predictions for the whole DL model with little manual effort.

As is shown in Figure 5, ProfileDNN takes in an arbitrary log file produced by running deep learning models on a platform that supports oneDNN with DNNL_VERBOSE set to 1. The *stats.py* file then collects and cleans the raw log file into CSV format, produces a template parameter file, calculates and plots the component distribution of primitive operations. The *benchDNN.sh* file runs each primitive operation multiple times and takes the average. The *efficiency.py* then takes a weighted sum of all operations' time by the number of calls and produces an efficiency ratio number.

To ensure our toolkit can accurately reproduce the running behavior of the kernels from the original model, we ensure both the computation resources and the problem descriptions are the same. We use *numactl* to control the number of CPU cores and memory binding and the *mode* is set to *p (performance)* in *benchDNN* to optimize performance. These parameters are carefully controlled and summarized in Table 2.

2.4 Dataloader and Memory Layout

By examining the DL training process from the same vTune top-down perspective shown in Figure 3, the dataloader can be seen as a frontend bounded by bandwidth and latency. There are three sources of bottlenecks for the dataloader: I/O, decoding, and preprocessing. We found similar performance for data in NVMe or loaded to RAM and the I/O overhead is negligible. We observed better decoding performance by adopting Pillow-SIMD and accimage as the backend in torchvision.

A PyTorch dataloader parameter controls the number of worker processes, which are usually set to prevent blocking the main process when training on GPUs. For training on CPUs, however, this number should not be set to minimize memory overhead. Since CPU RAM memory is usually larger than GPU memory (but has a smaller bandwidth) training on CPUs has the advantage of allowing larger batch size and training larger models [10].

Here we define n as batchsize, c as channel, h as height, and w as width. The recommended memory layout in Intel® Extension for PyTorch is *nhwc* (channel last) for more efficient training, though the default layout in *benchDNN* is *nchw*. We set the default behavior of ProfileDNN to adopt *nchw* based on known best-practices. If the log input specifies the memory layout, ProfileDNN automatically overrides the default.

2.5 Library Optimization

Substituting slow operation implementations with a more efficient library can improve performance significantly, as we discovered by replacing the official PyTorch implementation with the Intel® Extension for PyTorch counterpart. ProfileDNN helped identify a discrepancy between the number of backward convolution calls between the official PyTorch vs. the Intel® Extension for PyTorch library. Using a detailed analysis of the computation graph and our ProfileDNN-based visualization, we found calls emanated from the frozen layers in the pre-trained model (ResNext backbone).

Our analysis helped increase the performance of RetinaNet-ResNext50 model training with 2 fixed layers by 16%. We also found that the primitive operation `frozenbachnorm2d` was missing in Figure 2(d) and `torchvision.ops.misc.FrozenBatchNorm2d` was interpreted as `mul` and `add` ops, which meant it was not a single oneDNN kernel operation.

Our analysis indicated that bandwidth-limited operations made the `torchvision.ops.misc.FrozenBatchNorm2d` operation inefficient. It therefore cannot be fused with other operations to reduce memory

accesses. Training performance increased by 29.8% after we replaced the `torchvision.ops.misc.FrozenBatchNorm2d` operation with `IPEX.nn.FrozenBatchNorm2d`.

2.6 Low-precision Training

Low-precision training has proven an efficient way for high-performance computing and BF16 (Brain Floating Point) is widely supported by various DL hardware. BF16 is unique since it has the same range as float32 but uses fewer bits to represent the fraction (7 bits). This BF16 datatype characteristic can be beneficial when computing speed is important, but can also lead to accuracy loss when compared with float32 in calculating the loss. As shown in Figure 6, computation time is almost half when done in BF16 compared to float32 (Improvements are intentionally plotted relative so as not to release the actual data for compliance with Intel data policy).

There is a significant discrepancy between the forward/backward training time ratio compared with that of bare-bone kernel time. This discrepancy indicates highly inefficient non-kernel code in the forward pass. We found that the loss function does not scale well and comprises a significant portion of computation time.

After locating the focal loss as having significant overhead, we implemented our version of the focal loss kernel, further discussed in Section 2.8. However, the loss result is different from the original implementation. We pinpointed the accuracy loss as happening during low-precision casting to BF16 by `torch.cpu.amp.autocast`. Unless convergence can be guaranteed, therefore, casting data into BF16 should be avoided for loss calculation, especially when reduction operations are involved.

2.7 Layer Fusion and Optimizer Fusion

In inference mode, certain layers can be fused for a forward pass to save cache copying operation since an intermediate is not needed. In training mode, however, the layers containing trainable weights need to save the intermediate for backpropagation. When oneDNN runs in inference mode, it enables `batchnorm+relu` and `conv+relu` respectively, but not `frozenbatchnorm (FBN)+relu`. OneDNN already supports `eltwise (linear, relu)` post-ops for `conv` and chaining of post-ops. We therefore treat FBN as a per-channel linear operation to enable `conv+FBN+relu`. This fusion potentially increases performance 30% and is work-in-progress (WIP).

Modern-day deep learning frameworks invariably support automatic differentiation and modularity of deep learning building blocks, which facilitate the creation of deep learning models by lowering the entry barrier. However, it is common knowledge in software development that there exists a trade-off between modularity and performance. As pointed out by Jiang et al. 2021 [31], eager execution which executes forward propagation, gradient calculation, and parameter updating in serialized stages may harm the model performance, while optimizer fusion aims to improve locality and parallelism by reordering the three procedures. Intel® Extension for PyTorch currently supports fusion of SGD [32] and the Lamb [33] optimizer, partly by fusing operations, thus separate the *grad*, *parameters* and *intermediate* into small groups for better caching mechanism. We tested a fused/unfused Lamb optimizer with RetinaNet and found a 5.5X reduction in parameter updating time when the optimizer is fused.

2.8 Custom Operation Kernel

Custom operation kernels are essential to optimize performance by eliminating computation overhead, e.g., unnecessary copying and intermediates. These kernel implementations must be mathematically equivalent to the reference code. They can also show significant performance gains under all or most circumstances, as discussed below.

2.8.1 Theoretical deduction

Instead of relying on the PyTorch implementation (Appendix 5) of forward pass for focal loss and adopting the default generated backward pass, we implemented a custom focal loss kernel for both forward and backward pass (backward kernel implementation is optional, as implicit autograd can be generated). Focal loss can be represent as in Equation (1) and we adopt $\gamma = 2$ and $\alpha = 0.25$.

The forward pass can be simplified further by assuming x and y are real in Equation (2). Lastly, since y is a binary matrix, all the terms that contains y(y-1) equals to 0 and can be removed as shown in Equation (3). The backward equation is shown in Appendix 6.

$$FL(p) = \begin{cases} -\alpha(1-p)^\gamma \log(p), & y = 1 \\ -(1-\alpha)p^\gamma \log(1-p), & \text{otherwise} \end{cases} \tag{1}$$

$$FL = (a(2y - 1) - y + 1) \left(\frac{-e^x y + e^x + y}{e^x + 1} \right)^\gamma (\log(e^x + 1) - xy) \quad (2)$$

$$FL_{sp} = \left(\frac{-e^x y + e^x + y}{e^x + 1} \right)^\gamma ((\alpha(2y - 1) - y + 1) \log(e^x + 1) - \alpha xy)$$

$$(3)$$

2.8.2 Implementation and assessment

The operators in ATEN of PyTorch can be roughly categorized into two types: in-place operation and standard operation, with the former suffixed by _ (as in *add_*. Since in-place operations modify the Tensor directly, the overhead of copying or creating new spaces in the cache is avoided. The implementation shown in Appendix 7 heavily adopts in-place operation as much as possible, which enhances efficiency.

After confirming that our kernel implementation is mathematical equivalent to the reference implementation, we tested our kernel against the reference code under both float32 and BF16 settings. As shown in Figure 6, the custom forward kernel is 2.6 times faster than the default implementation under the BF16 setting.

Although the PyTorch framework can generate implicit autograd for our custom kernel, its performance is not ideal. The custom backward kernel is 1.3 times faster than the reference implementation and 1.45 times faster than the generated implicit autograd kernel. We also discovered that the custom backward kernel can boost forward kernel performance and we suspect that the explicit backward kernel can prevent the forward kernel from saving unnecessary intermediates. The combined improvement from custom focal loss kernel is two times faster. Our code has been integrated into Intel® Extension for PyTorch and will be available in that library shortly.
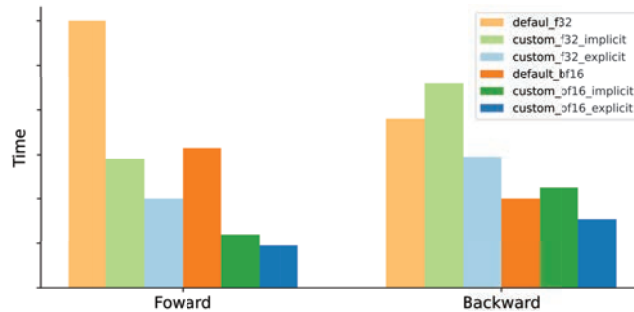

Figure 6   Comparison of custom focal loss time vs default.

3 Distributed Training

Compared to inference (which can be scaled-out amongst independent nodes), training DL models often require much greater computing power working synchronously. Meeting this need can be accomplished by scaling-up nodes with additional CPU resources or by scaling-out amongst multiple nodes. When training a system at scale – whether multiple nodes, multiple sockets, or even a single socket – it is necessary to distribute the workload across multiple workers.

Coordination among distributed workers requires communication between them. Distributing workloads on CPUs can be performed via multiple protocols and middleware, such as MPI (Message Passing Interface) [34] and Gloo [35]. We use MPI terminology in subsequent sections.

3.1 Distributed Training Performance

To maximize training performance, a training workload should target one thread per CPU core of each system node. For example, an 8-socket system with 28 cores per socket should target 224 total threads. The total threads may be apportioned across several workers identified by their *rank*, e.g., 8 ranks of 28 threads, 16 ranks of 14 threads, 32 ranks of 8 threads, etc. The selection of ranks and threads should not cause any rank to span multiple sockets.

In practice, better performance may be achieved by utilizing more ranks with fewer threads each, rather than fewer ranks with more threads each at the same global batch size. Table 3 shows how the throughput goes up diagonally from bottom-left to top-right. However, the number of available ranks is limited by the available system memory, model size, and batch size. The system memory is divided amongst the ranks, so each rank must have sufficient memory to support the model and host functions to avoid workload failure.

While the top-end CPUs can already perform on par with their GPU counterpart (Intel 4th Gen Xeon processors could train ResNet-50 model in less than 90 minutes [36]), the real benefit lies in that CPU training

Table 3   Scalability (normalized throughput)

Threads/Worker	Number of Workers				
	1	2	4	8	16
7	1.00	2.00	3.82	7.04	11.87
14	1.86	3.7	6.8	11.59	–
28	3.27	6.51	11.20	–	–
56	5.11	10.18	–	–	–

democratize the availability of training DL models to people who don't have access to GPUs, or companies with existing CPU clusters and on a tight budget. Since the inefficiency of CPU training is largely due to limited bandwidth, better software optimization (Intel® Extension for PyTorch, etc) and low-level kernel support (oneDNN, etc) can alleviate the issue by breaking and group operations into proper chunks for better caching. The improvement can be quite significant, as we found a 2X performance increase with Intel® Extension for PyTorch compared to the default PyTorch. AI accelerator is another potential platform for training, according to the latest MLPerf benchmark, the Gaudi2 processor has 2X the throughput of the A100 on ResNet-50 and BERT [37].

3.2 Training Convergence

As a training system is scaled-out to more nodes, sockets, or ranks, two factors are known to degrade the model's convergence time: weak scaling efficiency and convergence point. Weak scaling efficiency is a ratio of *the performance of a system to N systems doing N times as much work* and tends to lag behind the linear rate at which resources are added. This phenomenon and its causes are well-documented [38] across hardware types and is not explored further in this paper.

A model's convergence point is the second factor that impacts convergence time as a training system scales. In particular, the global batch size increases as a distributed system scales out, even though the local batch size per worker remains constant. For instance, if a 2-socket system launches a combined 8 ranks with a global batch size of 64 (BS = 8 per rank), when scaled out to 8-sockets, the global batch size becomes 256 even though each rank has the same local batch size.

As the number of epochs required to converge at a model's target accuracy increases the global batch size of a training workload also increases, as shown in Figure 7. This increase in the epochs to reach a convergence point can detract substantially from the increased resources. When planning a system scale-out, it is therefore critical to account for the resulting convergence point and mitigate it by reducing the local batch size if possible [39].

4 Concluding Remarks

This paper explores various aspects of optimizing the training of deep learning (DL) models on CPUs, in addition to a method guide. We present a
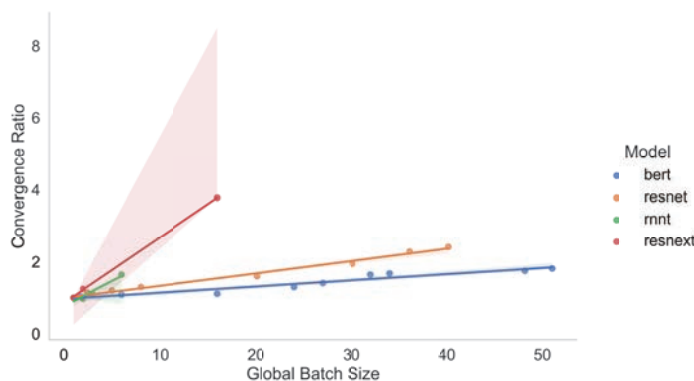
Figure 7   Convergence ratio vs global batch size (normalized).

DL profile/projection toolkit called ProfileDNN that helped us locate several issues for training RetinaNet-ResNext50, which when fixed increased performance by a factor of two. We also created a custom Focal Loss kernel that is 1.5 times faster than the PyTorch reference implementation when running on CPUs.

The following is a summary of the lessons learned from our study of training deep learning models using CPUs:

- Efficient DL frameworks that are optimised for CPUs (such as the Intel® extension for PyTorch) can reduce training time dramatically with little cost.
- Model profiling should be done both on the reference code and custom implementations, especially when the data set is changed. Discrepancies between different implementations and corresponding low-level op distributions can help pinpoint the bottlenecks.
- Implementing both forward pass and backward pass explicitly for custom kernels yields the best training performance.
- Local batch size is highly correlated with convergence point and should be reduced properly when planning a system scale out.

Ammar et al. [40] found that compared to hardware architecture, software libraries have a more significant impact on DL model training performance, therefore more research should focus on hardware-software co-design. Deep-Mind [41] recently discovered a novel algorithm with reinforcement learning for doing matrix multiplication by jointly profiling and finetuning hardware and software together. The algorithm, albeit hardware-dependent, increased performance dramatically and could not have been discovered by traditional

algorithm analysis. The same idea can be applied to discover more hardware-specific efficient kernel implementation. It can be misleading to compare hardware on DL model training solely using Time-to-Train (ToT), as pointed out by Sparsh, et al. [42], and choosing which system for DL training depends on other factors like energy efficiency, throughput, and latency. Our future work will focus on testing our method and ProfileDNN toolkit on other popular models and conducting a more in-depth study on optimizing training DL models with distributed CPU clusters. We will also work on improving MLperf to include more comprehensive metrics for DL model training.

Appendix

5 Reference Focal Loss Code [43]

```python
import torch
import torch.nn.functional as F
import time
def sigmoid_focal_loss(
    inputs: torch.Tensor,
    targets: torch.Tensor,
    alpha: float = 0.25,
    gamma: float = 2,
    reduction: str = "none",
):
    inputs = inputs.to(dtype=torch.float32)
    targets = targets.to(dtype=torch.float32)
    p = torch.sigmoid(inputs)
    ce_loss = F.binary_cross_entropy_with_logits(
        inputs, targets, reduction="none"
    )
    p_t = p * targets + (1 - p) * (1 - targets)
    loss = ce_loss * ((1 - p_t) ** gamma)
    if alpha >= 0:
        alpha_t = alpha * targets + (1 - alpha) * (1 -
    targets)
        loss = alpha_t * loss
    if reduction == "mean":
        loss = loss.mean()
    elif reduction == "sum":
        loss = loss.sum()
    return loss
```

6 Focal Loss Derivative

$$\frac{\partial}{\partial x}\left(\left(1-\left(y\times\frac{1}{1+e^{-x}}+\left(1-\frac{1}{1+e^{-x}}\right)(1-y)\right)\right)^{\gamma}\right.$$
$$\left(-y\log\left(\frac{1}{1+e^{-x}}\right)-(1-y)\log\left(1-\frac{1}{1+e^{-x}}\right)\right)(ay+(1-a)(1-y))\right)=$$
$$-\left(\left((a(2y-1)-y+1)\left(\frac{-e^x y+e^x+y}{e^x+1}\right)^{\gamma}\right.\right.$$
$$\left(e^{2x}y^2-\gamma e^x(2y^2-3y+1)\log\left(\frac{1}{e^x+1}\right)-2e^{2x}y+\gamma e^x(2y-1)\right.$$
$$\left.\left.\left.y\log\left(\frac{1}{e^{-x}+1}\right)+e^{2x}-y^2\right)\right)/((e^x+1)(e^x(y-1)-y))\right)$$

Figure 8   Backward kernel equation.

$$-(e^x+1)^{-\gamma-1}(y-e^x(y-1))^{\gamma-1}$$
$$\left(-\alpha\gamma e^x xy+\gamma e^x\log(e^x+1)(\alpha+y-1)+(\alpha-1)e^{2x}(1-y)+\alpha y\right)$$

Figure 9   Simplified backward kernel.

7 Custom Focal Loss Kernel Code

```cpp
at::Tensor _focal_loss_forward(const at::Tensor& input,
   const at::Tensor& target, const float alpha, const
   float gamma, const int64_t reduction) {
 at::Tensor loss;
 loss=(((alpha*(-input).mul_(target)).add_(((2*alpha-1)*
target+(1-alpha)).mul_(((input.exp_() + 1).log_())))).
mul_(((target -1).mul_(input).add_(-target)).pow_(gamma
))).div_((input + 1).pow_(gamma));
 return apply_loss_reduction(loss, reduction);
}

at::Tensor _focal_loss_backward(const at::Tensor& grad,
   const at::Tensor& input, const at::Tensor& target,
   const float alpha, const float gamma, const int64_t
   reduction) {
 at::Tensor grad_input;
 grad_input=-((input.exp() + 1).pow(-gamma-1)).mul((
target.add((1-target).mul(input.exp()))).pow(gamma - 1)
).mul(((-alpha*gamma*input).mul(target).mul(input.exp()
)).add(gamma*(target+alpha-1).mul(input.exp()).mul(((
input.exp()+1).log())))).add(alpha*target).add((alpha-1)
*(1-target).mul((input.exp()).pow(2)))).mul(grad);
```

```
    if (reduction == at::Reduction::Mean) {
        return grad_input / input.numel();
    }
    return grad_input;
}
```

References

[1] Peter Mattson, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, David Patterson, Guenther Schmuelling, Hanlin Tang, et al. Mlperf: An industry standard benchmark suite for machine learning performance. *IEEE Micro*, 40(2):8–16, 2020.

[2] Quchen Fu, Zhongwei Teng, Jules White, and Douglas C. Schmidt. A transformer-based approach for translating natural language to bash commands. In *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1245–1248, 2021.

[3] Quchen Fu, Zhongwei Teng, Jules White, Maria E. Powell, and Douglas C. Schmidt. Fastaudio: A learnable audio front-end for spoof speech detection. In *ICASSP 2022 – 2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3693–3697, 2022.

[4] Jee weon Jung, Hemlata Tak, Hye jin Shim, Hee-Soo Heo, Bong-Jin Lee, Soo-Whan Chung, Ha jin Yu, Nicholas W. D. Evans, and Tomi H. Kinnunen. Sasv 2022: The first spoofing-aware speaker verification challenge. *Interspeech 2022*, 2022.

[5] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: A system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[7] Yury Gorbachev, Mikhail Fedorov, Iliya Slavutin, Artyom Tugarev, Marat Fatekhov, and Yaroslav Tarkan. Openvino deep learning workbench: Comprehensive analysis and tuning of neural networks
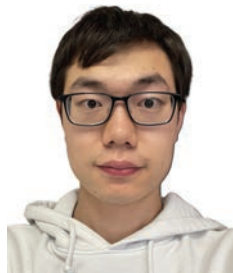
inference. In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*, pages 0–0, 2019.

[8] James R Reinders. Sycl, dpc++, xpus, oneapi. In *International Workshop on OpenCL*, pages 1–1, 2021.

[9] Dhiraj Kalamkar, Evangelos Georganas, Sudarshan Srinivasan, Jianping Chen, Mikhail Shiryaev, and Alexander Heinecke. Optimizing deep learning recommender systems training on cpu cluster architectures. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[10] Yu Emma Wang, Gu-Yeon Wei, and David Brooks. Benchmarking tpu, gpu, and cpu platforms for deep learning. *arXiv preprint arXiv:1907.10701*, 2019.

[11] Ebubekir Buber and DIRI Banu. Performance analysis and cpu vs gpu comparison for deep learning. In *2018 6th International Conference on Control Engineering & Information Technology (CEIT)*, pages 1–6. IEEE, 2018.

[12] Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. Benchmarking state-of-the-art deep learning software tools. In *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, pages 99–104. IEEE, 2016.

[13] Wei Dai and Daniel Berleant. Benchmarking contemporary deep learning hardware and frameworks: A survey of qualitative metrics. In *2019 IEEE First International Conference on Cognitive Machine Intelligence (CogMI)*, pages 148–155. IEEE, 2019.

[14] Yanli Qian. *Profiling and characterization of deep learning model inference on CPU*. PhD thesis, 2020.

[15] Jiho Chang, Yoonsung Choi, Taegyoung Lee, and Junhee Cho. Reducing mac operation in convolutional neural network with sign prediction. In *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 177–182. IEEE, 2018.

[16] Alexander Wong. Netscore: towards universal metrics for large-scale performance analysis of deep neural networks for practical on-device edge usage. In *International Conference on Image Analysis and Recognition*, pages 15–26. Springer, 2019.

[17] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim Hazelwood, Andrew Hock, Xinyuan Huang, Atsushi Ike, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao,

Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Tsuguchika Tabaru, Carole-Jean Wu, Lingjie Xu, Masafumi Yamazaki, Cliff Young, and Matei Zaharia. Mlperf training benchmark, 2019.

[18] Oneapi deep neural network library (onednn). https://github.com/oneap i-src/oneDNN.

[19] Intel extension for pytorch. https://github.com/intel/intel-extension-for -pytorch.

[20] Luis A Torres, Carlos J Barrios, and Yves Denneulin. Computational resource consumption in convolutional neural network training–a focus on memory. *Supercomputing Frontiers and Innovations*, 8(1):45–61, 2021.

[21] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020.

[22] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988, 2017.

[23] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44. IEEE, 2014.

[24] James Reinders. *VTune performance analyzer essentials*, volume 9. Intel Press Santa Clara, 2005.

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

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

[27] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[28] Jiachen Lu, Jinghan Yao, Junge Zhang, Xiatian Zhu, Hang Xu, Weiguo Gao, Chunjing Xu, Tao Xiang, and Li Zhang. Soft: Softmax-free transformer with linear complexity. *Advances in Neural Information Processing Systems*, 34, 2021.

[29] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[30] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Alexander Kolesnikov, et al. The open images dataset v4. *International Journal of Computer Vision*, 128(7):1956–1981, 2020.

[31] Zixuan Jiang, Jiaqi Gu, Mingjie Liu, Keren Zhu, and David Z Pan. Optimizer fusion: Efficient training with better locality and parallelism. *arXiv preprint arXiv:2104.00237*, 2021.

[32] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[33] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes. *arXiv preprint arXiv:1904.00962*, 2019.

[34] William Gropp, William D Gropp, Ewing Lusk, Anthony Skjellum, and Argonne Distinguished Fellow Emeritus Ewing Lusk. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[35] Gloo. https://github.com/facebookincubator/gloo.

[36] Intel delivers leading ai performance results on mlperf v2.1 industry benchmark for dl training, 2022.

[37] Ml commons v2.1 result, 2022.

[38] Srinivas Sridharan, Karthikeyan Vaidyanathan, Dhiraj Kalamkar, Dipankar Das, Mikhail E Smorkalov, Mikhail Shiryaev, Dheevatsa Mudigere, Naveen Mellempudi, Sasikanth Avancha, Bharat Kaul, et al. On scale-out deep learning training for cloud and hpc. *arXiv preprint arXiv:1801.08030*, 2018.

[39] Mlcommons rcp. https://github.com/mlcommons/logging/tree/master /mlperf_logging/rcp_checker/training_2.0.0.

[40] Ammar Ahmad Awan, Hari Subramoni, and Dhabaleswar K Panda. An in-depth performance characterization of cpu-and gpu-based dnn training on modern architectures. In *Proceedings of the Machine Learning on HPC Environments*, pages 1–8. 2017.

[41] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J R Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, et al.

Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.

[42] Sparsh Mittal, Poonam Rajput, and Sreenivas Subramoney. A survey of deep learning on cpus: opportunities and co-optimizations. *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[43] Sébastien Marcel and Yann Rodriguez. Torchvision the machine-vision package of torch. In *Proceedings of the 18th ACM international conference on Multimedia*, pages 1485–1488, 2010.

[44] https://github.com/intel/intel-extension-for-pytorch/commit/d09f3409 65bbd2421a00317b466bbad1bf3fcad0.

[45] https://github.com/intel/intel-extension-for-pytorch/commit/5f1f32ed 2754e5df767ff21e1894ea49f189c030.

Biographies

Quchen Fu Fu is a Ph.D. student at Vanderbilt University major in Computer Science, his research interest is NLP and Deep Learning. He got his Master's degree in CMU and he was TA for multiple courses including Cloud Computing and Cybersecurity. He interned at multiple companies including Tencent, Intel, and Microsoft. He is now a research assistant in Magnum research group under Dr. Jules White.
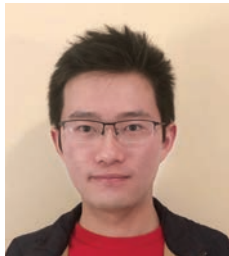
Zhongwei Teng is pursuing a Ph.D. in Computer Science in Vanderbilt University. His research interests include speech verification, NLP and machine learning.

Jules White is Associate Dean of Strategic Learning Programs in the School of Engineering and Associate Professor of Computer Science in the Dept. of Computer Science at Vanderbilt University. He is a National Science Foundation CAREER Award recipient. His research has won multiple Best Paper Awards. He has also published over 150 papers. Dr. White's research focuses on cyber-security and mobile/cloud computing in domains ranging from healthcare to manufacturing. His research has been licensed and transitioned to industry, where it won an Innovation Award at CES 2013, attended by over 150,000 people, was a finalist for the Technical Achievement at Award at SXSW Interactive, and was a top 3 for mobile in the Accelerator Awards at SXSW 2013. He has raised over $12 million in venture backing for his startup companies. His research is conducted through the Mobile Application computinG, optimizatoN, and secUrity Methods (MAGNUM) Group at Vanderbilt University, which he directs.

Douglas C. Schmidt is the Cornelius Vanderbilt Professor of Computer Science, Associate Chair of Computer Science, and a Senior Researcher at the Institute for Software Integrated Systems, all at Vanderbilt University. His research covers a range of software-related topics, including patterns, optimization techniques, and empirical analyses of frameworks and model-driven engineering tools that facilitate the development of mission-critical middleware for distributed real-time embedded (DRE) systems and intelligent mobile cloud computing applications. Dr. Schmidt received B.A. and M.A. degrees in Sociology from the College of William and Mary in Williamsburg, Virginia, and an M.S. and a Ph.D. in Computer Science from the University of California, Irvine (UCI) in 1984, 1986, 1990, and 1994, respectively.