Performance Study on CPU-based Machine Learning with PyTorch

Smeet Chheda
schheda@cs.stonybrook.edu
Institute for Advanced Computational Science, Stony Brook University
Stony Brook, New York, USA

Anthony Curtis
Institute for Advanced Computational Science, Stony Brook University
Stony Brook, USA
anthony.curtis@stonybrook.edu

Eva Siegmann
Institute for Advanced Computational Science, Stony Brook University
Stony Brook, USA
eva.siegmann@stonybrook.edu

Barbara Chapman
Institute for Advanced Computational Science, Stony Brook University
Stony Brook, New York, USA
barbara.chapman@stonybrook.edu

ABSTRACT

Over the past decade we have seen a surge in research in Machine Learning. Deep neural networks represent a subclass of machine learning and are computationally intensive. Traditionally, GPUs have been leveraged to accelerate the training of such deep networks by taking advantage of parallelization and the many core architecture. As the datasets and models grow larger, scaling the training or inference task can help reduce the time to solution for research or production purposes. The Supercomputer Fugaku established state of the art results in multiple benchmarks in machine learning by scaling ARM based CPU technology. To that end, we study and present the performance of machine learning training and inference tasks on 64-bit ARM CPU architecture by exploiting its features namely the Scalable Vector Extensions (SVE) in the ARMv8-A.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; *Distributed computing methodologies*; *Concurrent computing methodologies*.

KEYWORDS

Machine Learning, Scalability, Distributed Learning, High Performance Computing

ACM Reference Format:
Smeet Chheda, Anthony Curtis, Eva Siegmann, and Barbara Chapman. 2023. Performance Study on CPU-based Machine Learning with PyTorch. In *International Conference on High Performance Computing in Asia-Pacific Region Workshops (HPCASIAWORKSHOP 2023), February 27-March 2, 2023, Raffles Blvd, Singapore*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3581576.3581615

HPCASIAWORKSHOP 2023, February 27-March 2, 2023, Raffles Blvd, Singapore

ACM ISBN 978-1-4503-9989-0/23/02…$15.00
https://doi.org/10.1145/3581576.3581615

1 INTRODUCTION

Deep learning has revolutionized the fields of natural language processing, computer vision, pattern recognition, recommendation systems, etc. Massive efforts from industry and academia have been poured into research and development in the last couple of years. Deep learning represents a subset of machine learning and is one of the most popular topics these days. Complexity in the operations, layers, connections, gradient optimization algorithms make deep neural networks computationally expensive to train. To address the growing size of datasets and large/deep models, it is beneficial to adopt techniques from HPC to accelerate training and inference while taking maximum advantage of the underlying hardware. Scaling deep learning workloads is a viable option to reduce time to solution.

The concept of a neural network is not recent. A deep neural network named LeNet [25] was a deep convolutional neural network introduced in 1998. However, it was computationally expensive to train with the available resources at that time. A decade ago Alexnet [23] was featured, which presented significant prediction improvements in the field of object recognition compared to previous studies, and demonstrated the use of GPUs to accelerate training. This reinvigorated research in artificial neural networks and we can see multiple neural architectures, optimization algorithms, frameworks, etc. pop up.

ARM architecture is widely used in the world, from mobile devices to HPC. Until recently (June 2022), The Fugaku Supercomputer [33] (hereafter just "Fugaku") was ranked highest in the Top500 [38], HPCG [37] and HPL-AI [17] benchmarks. At its heart lies the A64FX processor developed by Fujitsu Limited based on the ARM v8.2 instruction set architecture [12]. The key feature of this chip includes the Scalable Vector Extension (SVE), which provides SIMD (Single Instruction Multiple Data) instructions of size 512 bits, and high bandwidth memory. Traditionally, GPUs have been used to accelerate deep learning training and inference. Our interest in ARM technology is motivated by the fact that scaling CPU based technology efficiently has achieved commendable performance in major benchmarks [10].

In this work, we evaluate the performance of PyTorch [32], a popular machine learning framework, on the A64FX processor.

We assess state of the art models in computer vision tasks over image datasets in both intra- and inter-node fashion, and utilize benchdnn for performance benchmarking of primitives provided by oneDNN [19]. The primary goal of the paper is to record the maximum throughput that can be achieved, in images per second. Therefore, we perform weak scaling for inter-node tasks, and model convergence is not considered.

This paper is laid out in the following manner. Section 2 provides information on related works and 3 describes the ML libraries used along with their versions and modifications. Section 4 describes the programming environment of the underlying systems, including the software stack, hardware and interconnect used. Section 5 we describe the benchmark datasets and models used for experimentation in section 6. We make note of certain observations in section 7 and conclude in section 8.

2 RELATED WORK

Scaling is of crucial importance. Computation and communication should be optimized and overlapped to deliver the best performance. Popular machine learning libraries have had support for distributing workloads in a data parallel or model parallel manner. A notable piece of research to accelerate training was done by a team at Facebook that accurately trained ImageNet [8] within an hour by utilizing GPUs across multiple servers [14]. This partly inspired other research groups to develop distributed frameworks to efficiently use computation resources. Uber being one among them, released their framework Horovod [34] to the community in the following year, and now it has become a popular API for multi-node deep learning.

Many works have been done on performance evaluation of Deep Learning libraries on accelerators and CPUs [4], [30], [39]. Primarily these have focused on CPU vs GPU comparisons, or framework comparisons over GPUs and the CPUs have always been x86 CPUs, to the best of our knowledge. [27] used the Cori system, an XC40 Cray machine, at NERSC to train their 3D CNN at scale on Intel's Xeon Phi (KNL) with the Cray ML Plugin to achieve high scaling up to 8192 nodes. A team at HPE created a framework to model the scalability of Distributed Machine Learning [39]. [24] introduced novel communication strategies in synchronous distributed learning with the goal to overlap computation and communication and hide communication latency. They claim to achieve near linear scaling with 27,600 NVIDIA V100 GPUs on the Summit Supercomputer. These papers show that there has been commendable work on accelerating Deep Learning at scale on CPUs and GPUs.

There has been some study on Deep Learning and DL at scale on A64FX micro-architecture. [10] introduces a standard for benchmarking large-scale scientific machine learning workloads, and Fugaku achieves a high ranking in both the benchmark applications - CosmoFlow and DeepCAM. [35] have shown some important work in exploring a hybrid data and model parallelism approach to scaling the CosmoFlow problem from the MLPerf HPC benchmark training suite. Their study involves performance analysis, and more importantly performance tuning of TensorFlow, I/O performance of the filesystem, Tofu network topology, tuned MPI collective operations and generating tuned code for aarch64 for A64FX processors on Fugaku. [9] analyse the performance of convolution operators on A64FX. Their interest lies in exploring the performance benefits for latency constrained deep learning workloads by in integrating long SIMD units in multicore processors and evaluate 3 convolution implementations.

Fugaku employs the A64FX processor in the FX1000 node system developed by Fujitsu and based on the ARMv8-A ISA. This chip is the first to implement the Scalable Vector Extensions. The MLPerf HPC training benchmark results include results from Fugaku with their TensorFlow and PyTorch extensions [31] over multiple benchmark applications. They show comparable runtime performance to GPU centric results. Therefore, we dive a little deeper in the PyTorch framework on the Ookami cluster [20] to see its performance with other benchmark applications and models.

There have been studies on the A64FX processor, including but not limited to, OpenMP benchmarking [29], [28], parallel benchmarks [5], domain science applications [11], [6], [15]. PyTorch, a hybrid MPI+X framework, acts as a good test of the systems computation and communication capabilities.

3 ML LIBRARIES

Here we describe the libraries used for experimentation and any changes that were made to them for the same. Table 1 briefly mentions the libraries we used for experimentation. All were compiled from source and we used venv to create a virtual environment for execution.

Table 1: Libraries

Library	Version
Python	3.8.2
Numpy	1.22.4
PyTorch	1.10.0
Torchvision	0.11.0
oneDNN	2.4.3
Horovod	0.24.3

3.1 PyTorch

PyTorch is a high performance machine learning library which supports tensor computations, strong GPU acceleration and reverse mode automatic differentiation. It provides a front-end Python API, but is written in C++ to achieve high performance. Multi-threading is implemented in C++, which bypasses Python's global interpreter lock. The adoption rate of PyTorch has increased over the years [32]. Due to its popularity and ease of use, we based our experiments on this library. Other popular machine libraries include TensorFlow [3], Chainer [36], and MxNet [7].

We follow Fujitsu's setup instructions, as published on their GitHub repository for `PyTorch v1.7` [13]. We do not use this branch for our evaluations because it is specific for the A64FX processors. Instead, we use `PyTorch v1.10.0` cloned directly from PyTorch's GitHub page to establish consistency across all architectures. Our experiments are based on the image classification problem and for that, we make use of `torchvision v0.11.0` to provide optimized models. BLAS, LAPACK, OpenMP and MPI CMake files are modified to accommodate Fujitsu's BLAS library, OpenMP flags

and MPI wrapper to compile PyTorch. For the ARM compiler, only the BLAS and LAPACK libraries are modified to accept the ARM Performance Libraries, others can be used as is. With the GNU compiler, no changes are necessary and PyTorch can be compiled out of the box. With the recent GNU compiler (v11.x), we remove the `-Werror=format` option in the CMakelists.txt file to avoid compilation errors.

3.1.1 oneDNN. oneDNN, a part of oneAPI, is an open source library providing highly optimized hardware-aware primitives for building deep learning applications. This has been adopted in PyTorch for inference for some time now. However, training support was introduced in PyTorch v1.9.

oneDNN was originally optimized for Intel CPUs by taking advantage of vectorization (SSE, AVX instruction sets on Intel CPUs) and better cache reuse. To port this library to Aarch64, Fujitsu developed an AArch64 version of `xbyak` which is the just-in-time assembler for x86 [21]. Along with that, they also built a binary translator `xbyak_translator_aarch64` to convert runtime generated x86 code to Aarch64 (ARMv8-A ISA specifically). This work has been upstreamed into oneDNN.

Earlier, one would have to build this translator on the A64FX chip before building oneDNN on A64FX. Since the necessary headers have been upstreamed, one can easily build oneDNN without having to build the aforementioned translator. We refer to MKL-DNN as "onDNN" hereafter, and the blocking style provided by oneDNN as "onDNN block format". To use the efficient primitives in PyTorch during training, one can follow these steps:

```
input = torch.randn((10,10)).to_mkldnn()
output = model(input)
```

And during inference:

```
from torch.utils import mkldnn
input = torch.randn((10,10)).to_mkldnn()
model = mkldnn.to_mkldnn(model)
output = model(input)
```

For other compilers, we can build oneDNN out of the box with respective BLAS libraries by setting `DNNL_BLAS_VENDOR` environment variable during configuration.

3.1.2 PyTorch JIT. PyTorch provides just-in-time compilation via `torch.jit`. Default training in PyTorch is done in "Eager mode" execution where the computational graph is built at runtime and managed by the Python process. The forward pass is supposed to dynamically create this graph and the backward pass is supposed to apply losses and then destroy it. This is not desirable for performance and deployment, in which case PyTorch also supports graph execution. In graph execution the computation graph is built once, and the underlying process (which can be a C++ process) manages this state. Graph execution is difficult to debug, but is expected to be faster than eager execution and hence, once can take advantage of this mode if their models are not supported by oneDNN. One can use the `torch.jit.script` or `torch.jit.trace` functionality to use just-in-time compilation for their custom model.

3.2 Horovod

Horovod is a popular, easy-to-use distributed deep learning framework introduced by Uber and currently hosted by LF AI & Data Foundation [34]. Currently, it supports TensorFlow, Keras, PyTorch and MXNet Machine Learning frameworks and can be built with MPI, NCCL, Gloo and oneCCL backends for Tensor operations. In our experiments we use Horovod for CPUs built with Open MPI. To use horovod with PyTorch, one can initialize the library `init()`, apply the distributed sampler to distribute the dataset into equal shards, wrap the optimizer object with the Distributed Optimizer `DistributedOptimizer()`, broadcast model parameters from rank 0 `broadcast_parameters()` and train the model as one would. It is important to note that the user should modify the learning rate based upon the optimization algorithm and gradient averaging technique used by horovod. A general rule of thumb is to scale the learning rate proportional to the number of workers or ranks, unless one uses AdaSum to perform reduction, then the learning rate should be scaled by an empirical constant factor of 2-2.5 [2]. The batch size is decided based upon the scaling style (*weak vs strong*). Setting the batch size in the DataLoader object sets the rank local batch size. To build horovod, we add a CMake file to find Fujitsu's MPI wrapper. With the other libraries, no changes are required. Horovod is built with PyTorch and MPI support by setting `HOROVOD_WITH_MPI=1` and `HOROVOD_WITH_PYTORCH=1`

With horovod, we evaluate data-parallel training in a weak scaling setting. Here, each worker has a copy of the model and all workers work on exclusive shards of the dataset. Model parallel training is also possible but out of scope of this work. If the model is too large and does not fit on a single node or single workers resources, one may have to implement it.

4 COMPILERS AND HARDWARE

Our primary focus is to evaluate ARM architectures for Machine Learning applications. Fugaku has been deemed to become the core infrastructure providing a high performance AI platform for processing and training over large amounts of data efficiently at scale. So, we evaluate the intra- and inter-node performance of PyTorch on the A64FX FX700 system with different compiler toolchains and BLAS libraries. For fair comparison, we also evaluate the same libraries with respective compilers on x86 systems. The hardware and compilers are described in as follows:

4.1 Hardware

We make use of the Ookami cluster to experiment with A64FX processors and Stampede2 at TACC and clusters - Popeye and Rusty, at the Flatiron institute for the Intel CPUs. The CPU versions for all are mentioned in table 2. A64FX is an ARM based chip developed by Fujitsu with the ARM v8-A ISA and the first to implement Scalable Vector Extensions with a 512 bit implementation enabling vector length agnostic (VLA) programming. It contains 48 cores (A64FX on Fugaku may have 2/4 additional I/O cores on Fugaku) and runs at a steady 1.8GHz frequency. There are no stepping modes as seen on the x86 CPUs. The 48 cores are divided into 4 Core Memory Groups (CMGs) representing NUMA domains and have 8GB of High Bandwidth Memory per CMG, 32GB HBM2 on one node. These nodes contain 64KB L1 cache and 8MB shared L2 cache. Ookami has an Infiniband HDR100 interconnect with 200 gigabit switches and a high performance Lustre file system.

Table 2: CPU and Compilers

Cluster	CPU	Compiler	BLAS library	Compiler Flags
Ookami	Fujitsu A64FX	Fujitsu compiler v4.7[a]	SSL2	-Nclang -Kfast -Knolargepage -lpthread
Ookami	Fujitsu A64FX	ARM Compiler v22.0	ARMPL v2022.0.1	-O3 -mcpu=a64fx -mtune=a64fx -lpthread
Ookami	Fujitsu A64FX	GNU v11.2.0	OpenBLAS v0.3.19	-O3 -mcpu=a64fx -mtune=a64fx -lpthread
Rusty	Intel Xeon Gold 6148 (40 cores)	GNU v10.3.0	OpenBLAS v0.3.19	-O3 -lpthread -mtune=skylake-avx512
Rusty	Intel Xeon Gold 6148 (40 cores)	Intel Compiler v2022.0.1	MKL v2022.0.1	-O3 -lpthread -mtune=skylake-avx512
Popeye	Intel Xeon Platinum 8358 (64 cores)	Intel Compiler v2022.0.1	MKL v2022.0.1	-O3 -lpthread -mtune=icelake-server
Stampede2	Intel Xeon Platinum 8160 (48 cores)	Intel v19.1.1	MKL v19.1.1	-O3 -lpthread -mtune=skylake-avx512

[a] We use the Fujitsu Compiler in Clang mode and the LLVM OpenMP library implementation.

All x86 CPUs involved in evaluation support 512 bit extensions to Intel's 256 bit Advanced Vector Extensions SIMD instruction set for x86 ISA. The Skylake CPUs on Stampede2 have 48 cores, 24 cores per socket with 32KB L1i and L1d cache, 1MB L2 cache and 33MB L3 cache. This CPU has 4 stepping states with a max frequency of 3.7GHz. Hyperthreading is enabled with 2 threads per core. The interconnect is a 100Gb/sec Intel Omni-Path (OPA) network with a fat tree topology employing six core switches [18]. We use Intel's MPI implementation in our experiments on Stampede2. The Skylake CPUs on Rusty have 40 cores, 20 per socket with Hyperthreading disabled. They also have 4 stepping states with a max frequency of 3.7GHz and contain same L1 and L2 cache as the Skylake CPU on Stampede2 except the L3 cache is 28MB. Icelake CPUs were added for benchDNN performance runs because they support VNNI (Vector Neural Network Instructions) as an extension to AVX512 designed specifically for inference. The Icelake CPU has 64 cores, 32 per socket with Hyperthreading disabled. They have 6 stepping states with a max frequency of 3.4GHz and contain 48KB L1d cache, 32KB L1i cache, 1.28MB L2 cache and 48MB L3 cache.

Two different clusters are used for PyTorch benchmarks on x86 because of the missing latest Intel oneAPI compilers and the presence of Hyperthreading on Stampede2.

4.2 Compilers and BLAS libraries

We have access to the ARM, Cray, Fujitsu, GNU, LLVM and NVIDIA compilers on the Ookami cluster. NVIDIA's compiler is not used because it does not generate SVE code and defeats the purpose. Cray's compiler is also not used because we ran into compilation problems with Python where it stalled during compilation. LLVM versions 12 and 13 are available on the system but version 14 is the first where the vectorizer uses scalable auto-vectorization by default to generate SVE instructions on compatible targets. At the time of experimentation, LLVM version 14 was not available on the system.

We are left with 3 compilers on the ARM architecture - ARM, Fujitsu and GNU. The ARM and Fujitsu toolchains have vectorized math library support via the ARM Performance Libraries and SSL2, respectively. The GNU compilers suffer from this drawback that they do not have a vectorized math library implementation (at least for ARM architecture). During experimentation, OpenBLAS v0.3.19 had support for vectorized routines. Therefore, we make use of OpenBLAS with the GNU compilers. For all experimentation on the FX700 system, we use Open MPI version 4.1.2 except with the Fujitsu compiler which has it's own MPI implementation based on Open MPI v4.0.1. RDMA support in Open MPI is provided via UCX over Infiniband.

On the x86 platform, we have the GNU and Intel compilers. MKL is the default BLAS library of choice on Intel CPUs. We have access to 2 different sets of Intel x86 clusters. We perform single node and distributed training on Skylake CPUs on Stampede2 at TACC which has the Intel 19.1.1 compilers and benchDNN performance benchmarking (Section refsubsect:bench) on the Flatiron machines because they have the latest Intel compilers 2022.0.1. We also note that HyperThreading (HT) is enabled on the Stampede2 machines and this is not desirable for performance and we use `OMP_PLACES=cores` to disable it for the GNU OpenMP library `libgomp` and `KMP_HW_SUBSET=1T` with the rest. On x86, only the GNU compiler builds of PyTorch are used for single node & distributed training comparison because the Intel compiler failed to compile PyTorch. The intel compilers are successful in compiling oneDNN and those results have been included in the next section. The specific versions of compilers and libraries can be seen in 2.

5 BENCHMARK DATASETS AND MODELS

We use two benchmarks for our evaluation. These are specifically chosen to get an idea of how the ARM chip performs on a research code over the PACS dataset and a relatively standard benchmarking dataset, CIFAR-10.

5.1 Datasets

- PACS [26] - The acronym stands for Picture, Art painting, Cartoon and Sketch - a popular dataset in distribution

generalization problems, representing 4 different data domains with 7 classes (dog, elephant, giraffe, guitar, house, horse, person). There are 9991 images in total and we use Art painting, Cartoon and Photo for training & validation and Sketch for testing. This corresponds to 5444 images for training, 605 for validation and 3942 for testing. This dataset is used to evaluate single node performance for training and inference. We employ the following transformations to the images during training:
  - Resize
  - Random Horizontal Flip
  - Color Jittering
  - Random Gray Scaling
  - Normalization

  During inference, we only employ Resize and Normalization.
- CIFAR-10 [22] - This is a very popular dataset. It contains 60,000 images, with 50,000 for training and 10,000 for inference. There are 10 classes - airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. This dataset is used to evaluate distributed performance for training. Normalization is the only transformation applied to these images for training and inference.

5.2 Models

We evaluate the performance of 3 models in our experiments namely, `ResNet - 34` [16], `ResNext 50-32x4d` [40] and `Wide Resnet 101` [41]. As one may surmise, all of these models are based on residual connections. We chose models based on residual connections primarily because they can be converted to the MKLDNN block format and they represent state-of-the-art models in object recognition tasks. Other models from respective model families provided in `torchvision.models` were evaluated for their compatibility with MLKDNN block formats, and the aforementioned classes were the only ones that could work without significant changes from the user. To that end, the other models can still leverage PyTorch's JIT mode for getting performance during training.

Residual Networks are popular because they introduced the concept of skip connections to address problems with deeper networks such as vanishing gradients and outperformed all models in the ILSVRC challenge in 2015 [1] without adding additional complex computations in the network. The residual network family of models can be converted to the oneDNN block format without making any changes. In recent times, the transformer models for language processing and vision have also received the spotlight for state-of-the-art results and contain many more parameters (a magnitude higher), but we do not consider those models in these study due to computational resource constraints such as allotted computation time. From initial testing, we found out that the visual transformer cannot be converted to the block format. The three models also represent different computational complexity of deep learning models. ResNet34 is a 34 layered network, ResNext50-32x4d is a 50 layered network and Wide Resnet 101 is a 101 layer network. Each of these models have approximately 21.79, 25.02 and 126.88 million trainable parameters. The number of trainable parameters were generated by the `torch-summary` package.

6 EVALUATION

In this section we describe the evaluation strategy and the runtime environment flags set to achieve said performance. Evaluation is performed in a single node and inter-node (distributed) basis. We evaluate training and inference runtime performance of the models on the benchmark datasets. 8 OpenMP threads are set for benchDNN performance profiling. As mentioned in 4, the A64FX chip has a memory limitation of 32GB on a node. On the Ookami cluster, the OS resides in main memory, and the application must fit in approximately 27GB. Hence for intra node, we decided to fill the memory of a node until we ran into Out-of-memory errors and landed on 128 images in a batch for training. MKLDNN block formats use memory efficiently compared to the native implementation in PyTorch. For the Wide ResNet model, we ran the JIT version of the application with 64 images to avoid running into Out-of-memory errors. For inter-node weak scaling runs, we decided to go with 128 images as that was the maximum number of images (in the power of 2) that could be set such that each node received at least 1 shard of the dataset and can perform at least one local gradient calculation and backpropagation. All training runs on both clusters were performed at least 5 times and their average epoch time / throughput is depicted in the plots.

Table 3: benchDNN Problems

Index	Problem
1	mb256ic3ih224oc64oh112kh7sh2ph3n
2	mb256ic64ih56oc256oh56kh1ph0n
3	mb256ic64ih56oc64oh56kh1ph0n
4	mb256ic64ih56oc64oh56kh3ph1n
5	mb256ic256ih56oc64oh56kh1ph0n
6	mb256ic128ih28oc128oh28kh3ph1n
7	mb256ic128ih28oc512oh28kh1ph0n
8	mb256ic512ih28oc128oh28kh1ph0n
9	mb256ic256ih14oc256oh14kh3ph1n
10	mb256ic256ih14oc1024oh14kh1ph0n
11	mb256ic1024ih14oc256oh14kh1ph0n
12	mb256ic512ih7oc512oh7kh3ph1n
13	mb256ic512ih7oc2048oh7kh1ph0n
14	mb256ic2048ih7oc512oh7kh1ph0n
15	mb256ic256ih56oc128oh56kh1ph0n
16	mb256ic128ih56oc128oh28kh3sh2ph1n
17	mb256ic512ih28oc256oh28kh1ph0n
18	mb256ic256ih28oc256oh14kh3sh2ph1n
19	mb256ic1024ih14oc512oh14kh1ph0n
20	mb256ic512ih14oc512oh7kh3sh2ph1n

6.1 benchDNN

benchDNN is a robust harness for testing and performance analysis of primitives provided by oneDNN. We make use of benchDNN to run performance tests on the convolution primitive with different problems keeping it consistent with [21]. Although they compare performance of integer convolutions, we run performance tests on 32 bit floating point convolutions in forward and backward mode

Table 4: benchDNN execution

Execution Command	OMP_NUM_THREADS=8 numactl –membind=0 –cpunodebind=0 ./benchdnn –conv –mode=p –fix-times-per-prb=5 –reset –dir=dir –cfg=cfg (problem)
Direction	FWD_B, BWD_WB, FWD_I
Configuration	f32, u8s8s8, s8s8s8

because these are essential for understanding the compute time taken by PyTorch on x86 and ARM64 during training. For inference, we run tests in the forward mode on Skylake and Icelake machines - the latter to explore the VNNI extension to the AVX512 vector instruction set. The results of these runs can be seen in figure 1.

FWD_B stands for forward mode with bias, BWD_WB for backward mode with bias and FWD_I for forward mode inference. For A64FX inference mode the configuration is s8s8s8 instead of u8s8s8 as explained in [21]. u8 and s8 represent unsigned and signed integers and f32 is 32 bit floating point. f32 (or f32f32f32) represents, source weights and destination respecively. Similarly for u8s8s8 and s8s8s8. More details about datatypes can be found in benchDNN's documentation. We consider the same problems as in [21] and elaborate on the directions and configurations. And these problems are run with similar settings i.e., with 8 OpenMP threads, convolution driver, performance mode, etc. The `(problem)` can be found in Table 3.

As we can see from Figure 1, the Xeon Scalable processor has an advantage over A64FX in forward (FWD_B) and backward (BWD_WB) mode. There is a larger gap in backward pass performance between the ARM and x86 chip. The forward and backward modes are run with 32 bit floating point format because they are the commonly used training data type. However, in integer convolutions, we see that A64FX has comparable performance or significantly outperforms the Skylake CPU in many problems. The Icelake CPU outperforms all in the integer configuration due to the VNNI instruction `vpdpbusd` which combines 3 vector instructions `vpmaddubsw`, `vpmaddwd` and `vpaddd` into 1. We note that these instructions are primarily for low precision operations and have no influence on training modes.

6.2 Single Node

The purpose of evaluating single node performance is to test various runtime environment flags to speedup computation, and see how A64FX performs compared to Skylake. These flags can then be directly applied in a distributed setting and the focus can be shifted to the communication bottlenecks. For single node runs on the PACS dataset, we set the batch size to 128 and convert inputs to block format during training. The execution is done in eager mode and hence we build and use TCMalloc for fast memory allocation. We take advantage of intra-node parallelization via the OpenMP Environment Variables. The following settings are applied – `OMP_NUM_THREADS=46`, `OMP_PROC_BIND=close`, `OMP_PLACES=cores` and `OMP_STACKSIZE=8M`. For `libomp`, we can alternatively use the KMP_* variables, for example, `KMP_AFFINITY=granularity=fine,compact,1,0` and `KMP_BLOCKTIME=0` and for `libgomp`, we can use `GOMP_CPU_AFFNITY=0-45`. With the Fujitsu compiler, it is recommended to set `XOS_MMM_L_HPAGE_TYPE=none` to disable huge pages.

One can note that we use 46 threads instead of 48. This is done so that 2 cores are left idle for system or I/O operations. The A64FX chip in FX1000 in Fugaku has additional 2/4 cores that perform system or I/O operations, missing in the A64FX processor in the FX700 system.

Our training runs on the PACS dataset in Figure 2 represent a application research environment where costly transforms, optimization algorithms and learning rate scheduling are used for getting high prediction accuracy. And there we see the benefits of using the oneDNN block formats and their advantage over jit scripting provided by PyTorch. Epoch times are lower on Skylake nodes and we can see this from our evaluation of the convolution primitive with benchDNN. With the ResNet-34 and Wide ResNet models, we can see similar performance between the ARM v22.0, Fujitsu v4.7 and GCC v11.2 compilers. However, the ARM compiler outperforms the Fujitsu and GCC compilers in training. The performance of the Fujitsu compiler is surprising, and requires further investigation. The Skylake CPUs outperform the ARM chip, but the gap reduces as the model size increases (from approximately 77.1% faster with ResNet-34 to approximately 20 % faster with Wide ResNet).

Inference results are slightly different in Figure 3. A64FX gives competitive and in one case better performance than the Skylake CPU. ARM v22.0 compiled PyTorch on Ookami has the highest throughput of about 104 images per second and Fujitsu v4.7 compiled PyTorch on A64FX is second with about 98 images per second. GCC compiled PyTorch on Skylake has surprisingly lower throughput with the ResNet-34 model, and requires further investigation. JIT scripted results are laid out for a reader. If a custom model with new operators is not supported by oneDNN block formats, one can still take advantage of JIT scripting or tracing functionality provided by PyTorch, though at reduced performance.

6.3 Distributed

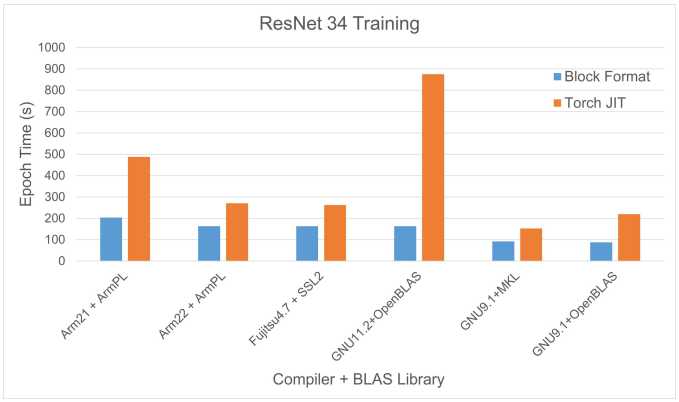
We use easily available sample code for distributed training with horovod on the CIFAR-10 dataset. Some modifications are made to it so that the oneDNN block format can be used and we add a user defined argument to pass either of the three models we experiment with. No other changes are made to the code. Distributed training with horovod represents a hybrid MPI+OpenMP application. We use mpirun to launch multiple copies of the program instead of horovodrun to control process and thread binding on a node. The run command looks like `mpirun –map-by ppr:1:node:pe=46 -x var . . . python3 . . .` where -x var represents the shared environment variables. The same OpenMP / KMP variables are passed to all nodes.

Figure 1: benchDNN Performance Evaluation: The first three columns in all groups represent A64FX and the remaining are Skylake 6148, except the last figure where black column represents Icelake 8358. *on A64FX, s8s8s8 configuration is used instead of u8s8s8 for inference.

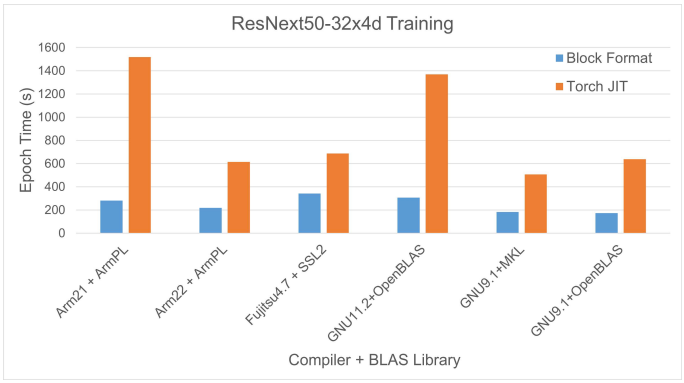
We also test horovod with RDMA by disabling MPI threads. However, we do not see significant performance gains and in some cases the program stalls. Therefore, we go ahead with the default TCP for MPI communication in Horovod. However, we shall see that this impacts performance.

Figure 4 shows our results for inter-node runs. Each node corresponds to 1 rank. ARM(22) compiled PyTorch is faster than Fujitsu compiled PyTorch in all cases even though Fujitsu's MPI is highly optimized and has low overheads compared to Open MPI compiled with the ARM compilers. We see much higher throughput with all three models when comparing the ARM and Fujitsu compiler builds. At a maximum of 128 nodes, the ARM compiler has 17.4% and 77.7% higher throughput with the ResNet-34 and Wide ResNet 101 models respectively. The applications scale very well on the Stampede2 cluster with Intel's OPA interconnect with approximately 56.06% with ResNet-34, 104.4 % with ResNext 50-32x4d and 27.02% higher throughput at 128 nodes than the best respective model throughput we recorded on Ookami.

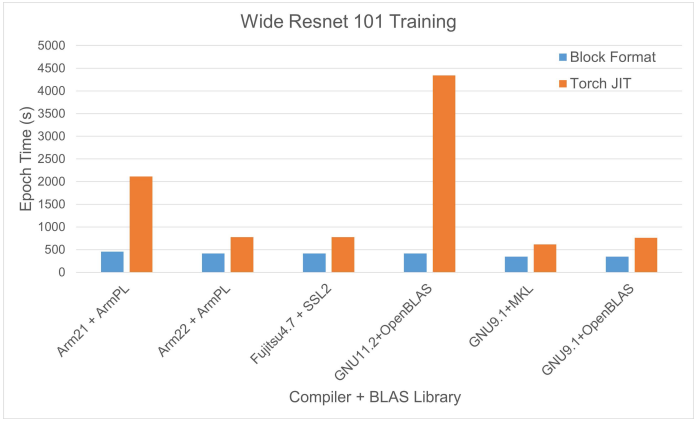
We make note of the scaling discrepancies in our experiments here. The GCC and Fujitsu compiled PyTorch builds fail to scale well on A64FX. There is a communication bottleneck and debugging or profiling that is challenging due to the limited memory on A64FX nodes. Using PyTorch's Kineto profiler was our initial choice. However, due to limited memory on the node, we are unsuccessful in leveraging this profiling tool. We also make use of Horovod's Timeline feature, and see that the communication bottlenecks are quite obtrusive negatively impact performance. In GCC's case, some operations are not optimized at all. For instance, we see that the



(a) ResNet34



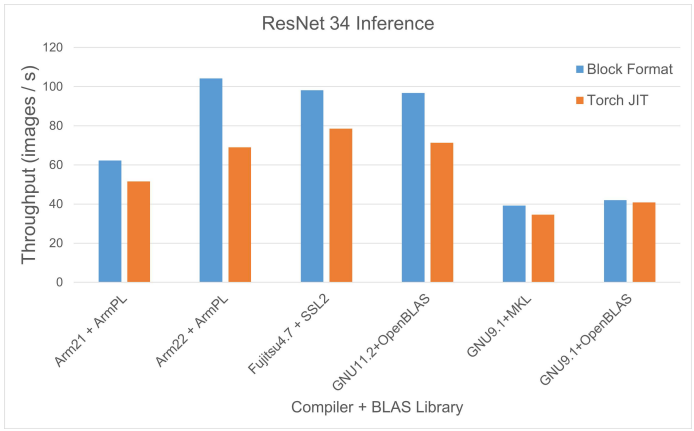
(b) ResNext50-32x4d



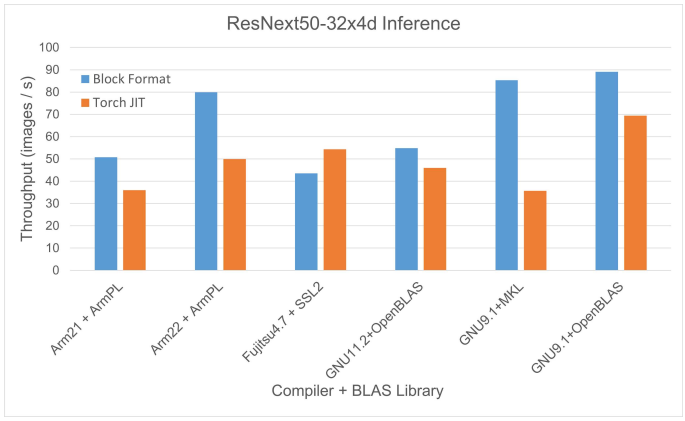
(c) Wide ResNet 101

Figure 2: Intra-node training processing time: The first four groups of columns represent A64FX and the last two represent Skylake 8160.

`memcpy` operations are slower than the corresponding Fujitsu- and ARM-compiled builds and this is because of the absence of an SVE optimized memcpy operation in glibc. Horovod does not make use of Infiniband on the Ookami cluster and partly explains the poor scaling compared to x86 runs.



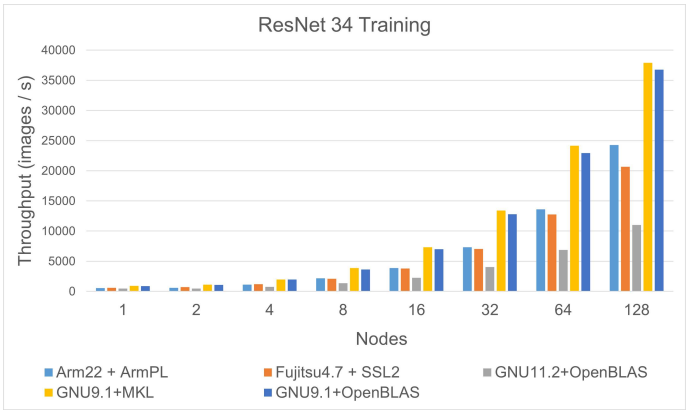
(a) ResNet34



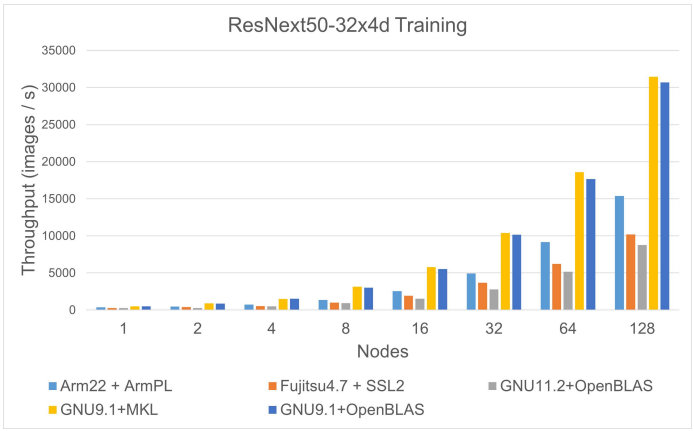
(b) ResNext50-32x4d

(c) Wide ResNet 101

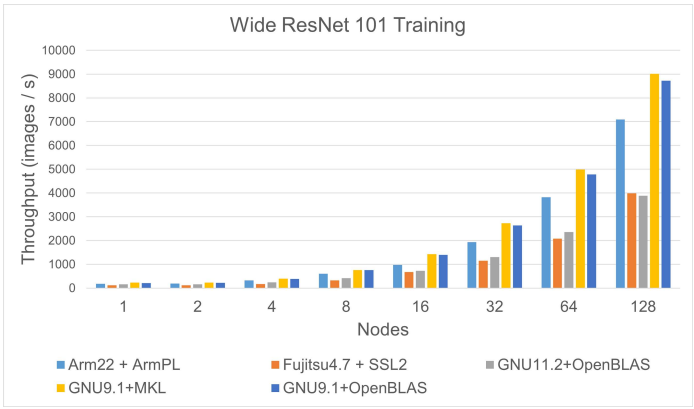
Figure 3: Intra-node inference throughput: The first four groups of columns represent A64FX and the last two represent Skylake 8160.



(a) ResNet34



(b) ResNext50-32x4d



(c) Wide ResNet 101

Figure 4: Distributed training throughput: The first three columns in every group represent A64FX and the last two represent Skylake 8160.

6.4 ARM compiler improvement w.r.t. benchmark application execution time

We see significant performance improvements with the ARM compiler v22.0 over v21.0 and take a moment to showcase this via the single node runs for the same dataset, PyTorch application and environment variables set at run time. The single node training and inference figures 2 and 3 respectively, show us this improvement. For training, the epoch times reduce and show a 20.12%, 22.39% and 9.2% improvement in training time with MKLDNN block format. During inference, the throughput increases from 59 images to 81 images per second - a 37.2% improvement. The following tables 5 and 6 tell us the reduction in training time per epoch and inference time by that percentage with the MKLDNN block and default memory formats.

7 OBSERVATIONS

- ARM toolchain delivers high throughput when scaled but tuning is required.
- Transforms applied to images increase the memory usage of the application and one must be careful while running their application on A64FX. One may tune batch size and transforms as needed.
- The new ARM compiler (22.0) has significant improvements compared to the previous major version (21.0).
- GCC- and Fujitsu-compiled PyTorch do not scale as expected and this warrants a thorough investigation.
- 512 bit SVE optimized kernels perform comparably with AVX512 optimized kernels in benchDNN.
- As seen in the benchdnn results, backward passes are significantly slower for some problems on A64FX compared to Skylake.
- The VNNI instructions of the extended AVX512 ISA show significant reduction in processing time.

8 CONCLUSION

In this work we did not compare performance of PyTorch's native DistributedDataParallel framework and there will be a follow up work to see which distributed framework approach performs better. Intel maintains its own extension which has hardware specific optimization, graph optimization for PyTorch which are upstreamed from time to time, but we do not make use of that distribution for evaluation. PyTorch's just-in-time functionality is another promising direction to accelerate research without having to wait for optimized oneDNN primitives and PyTorch's implementation of the same to catch up. This work did not study model convergence which is imperative. Achieving high throughput at the expense of model divergence is costly and undesirable. Weak scaling should be performed cautiously.

The A64FX processor can be used to scale machine learning workloads and building a tuned environment plays a crucial role in it. A more thorough set of experiments with more applications, perhaps from the MLPerf Training suite, and the latest PyTorch, oneDNN and torchvision versions can give us a better picture of which compiler/build performs better. Performance of the application depends on the compiler used to build PyTorch and this is evident from the performance differences of the ARM compiler

Table 5: Training Improvement with the ARM compiler 22.0 vs 21.0

Memory Format	ResNet-34	ResNext-50-32x4d	Wide ResNet 101
MKLDNN block format	20.12%	22.39%	9.2%
Torch JIT	44.46%	59.51%	63.29%

Table 6: Inference Improvement with the ARM compiler 22.0 vs 21.0

Memory Format	ResNet-34	ResNext-50-32x4d	Wide ResNet 101
MKLDNN block format	40.28%	36.41%	17.23%
Torch JIT	25.23%	28.06%	7.4%

itself. Currently, we see that communication is not well hidden by computation, and that is hindering scaling applications on the Ookami cluster. While developers may focus more on improving this overlap for GPU operations, attention to CPU operations is also necessary. Overall the ARM compilers performs significantly well on the Ookami cluster and in a few cases deliver comparable scaled performance for training and inference and in turn supports the place of ARM infrastructure in HPC and AI.

ACKNOWLEDGMENTS

The authors would like to thank Stony Brook Research Computing and Cyberinfrastructure, and the Institute for Advanced Computational Science at Stony Brook University (https://iacs.stonybrook.edu/) for access to the innovative high-performance Ookami computing system, which was made possible by a $5M National Science Foundation grant (#1927880). The software was tested, in part, on facilities run by the Scientific Computing Core of the Flatiron Institute Stony Brook (https://www.simonsfoundation.org/flatiron/scientific-computing-core/). The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper. URL: https://www.tacc.utexas.edu/. The authors would like to thank the Ookami project and the community for their support. The authors also thank Géraud Krawezik for providing access to compute resources and helpful discussions in setting up the environment at the Flatiron Institute.

REFERENCES

[1] 2015. *Imagenet Large Scale Visual Recognition Challenge 2015.* https://image-net.org/challenges/LSVRC/2015/results

[2] 2019. *AdaSum with Horovod.* https://horovod.readthedocs.io/en/stable/adasum_user_guide_include.html

[3] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[4] Ammar Ahmad Awan, Hari Subramoni, and Dhabaleswar K. Panda. 2017. An In-Depth Performance Characterization of CPU- and GPU-Based DNN Training on Modern Architectures. In *Proceedings of the Machine Learning on HPC Environments* (Denver, CO, USA) *(MLHPC'17)*. Association for Computing Machinery, New York, NY, USA, Article 8, 8 pages. https://doi.org/10.1145/3146347.3146356

[5] Md Abdullah Shahneous Bari, Barbara Chapman, Anthony Curtis, Robert J. Harrison, Eva Siegmann, Nikolay A. Simakov, and Matthew D. Jones. 2021. A64FX performance: experience on Ookami. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. 711–718. https://doi.org/10.1109/Cluster48925.2021.00106

[6] Robert Bird, Nigel Tan, Scott V Luedtke, Stephen Lien Harrell, Michela Taufer, and Brian Albright. 2021. VPIC 2.0: Next generation particle-in-cell simulations. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2021), 952–963.

[7] 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).

[8] 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.

[9] Manuel F. Dolz, Héctor Martínez, Pedro Alonso, and Enrique S. Quintana-Ortí. 2022. Convolution Operators for Deep Learning Inference on the Fujitsu A64FX Processor. In *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 1–10. https://doi.org/10.1109/SBAC-PAD55451.2022.00027

[10] Steven Farrell, Murali Emani, Jacob Balma, Lukas Drescher, Aleksandr Drozd, Andreas Fink, Geoffrey Fox, David Kanter, Thorsten Kurth, Peter Mattson, Dawei Mu, Amit Ruhela, Kento Sato, Koichi Shirahata, Tsuguchika Tabaru, Aristeidis Tsaris, Jan Balewski, Ben Cumming, Takumi Danjo, Jens Domke, Takaaki Fukai, Naoto Fukumoto, Tatsuya Fukushi, Balazs Gerofi, Takumi Honda, Toshiyuki Imamura, Akihiko Kasagi, Kentaro Kawakami, Shuhei Kudo, Akiyoshi Kuroda, Maxime Martinasso, Satoshi Matsuoka, Henrique Mendonza, Kazuki Minami, Prabhat Ram, Takashi Sawada, Mallikarjun Shankar, Tom St. John, Akihiro Tabuchi, Venkatram Vishwanath, Mohamed Wahib, Masafumi Yamazaki, and Junqi Yin. 2021. MLPerf™ HPC: A Holistic Benchmark Suite for Scientific Machine Learning on HPC Systems. In *2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*. 33–45. https://doi.org/10.1109/MLHPC54614.2021.00009

[11] Catherine Feldman, Benjamin Michalowicz, Eva Siegmann, Tony Curtis, Alan Calder, and Robert Harrison. 2022. Experiences with Porting the FLASH Code to Ookami, an HPE Apollo 80 A64FX Platform. In *International Conference on High Performance Computing in Asia-Pacific Region Workshops* (Virtual Event, Japan) *(HPCAsia 2022 Workshop)*. Association for Computing Machinery, New York, NY, USA, 72–77. https://doi.org/10.1145/3503470.3503478

[12] Fujitsu Global. 2019. *Fujitsu Processor A64FX.* www.fujitsu.com/global/products/computing/servers/supercomputer/a64fx

[13] Fujitsu Global. 2021. *Fujitsu PyTorch fork.* https://github.com/fujitsu/pytorch

[14] 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).

[15] Philipp Grete, Joshua C Dolence, Jonah M Miller, Joshua Brown, Ben Ryan, Andrew Gaspar, Forrest Glines, Sriram Swaminarayan, Jonas Lippuner, Clell J Solomon, et al. 2022. Parthenon–a performance portable block-structured adaptive mesh refinement framework. *arXiv preprint arXiv:2202.12309* (2022).

[16] 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.

[17] hpl-ai.org. 2022. *HPL AI Mixed Precision Benchmark.* https://hpl-ai.org/doc/results/

[18] https://portal.tacc.utexas.edu/. 2017. *Stampede2 User Guide.* https://portal.tacc.utexas.edu/user-guides/stampede2

[19] https://www.oneapi.io/open-source/. 2016. *oneAPI Deep Neural Network Library.* https://github.com/oneapi-src/oneDNN

[20] IACS. 2020. *Ookami Homepage.* https://www.stonybrook.edu/commcms/ookami/

[21] Kentaro KAWAKAMI, Kouji KURIHARA, Masafumi YAMAZAKI, Takumi HONDA, and Naoto FUKUMOTO. 2021. A Binary Translator to Accelerate Development of Deep Learning Processing Library for AArch64 CPU. *IEICE Transactions on Electronics* advpub (2021), 2021LHP0001. https://doi.org/10.1587/transele.2021LHP0001

[22] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[24] Nouamane Laanait, Joshua Romero, Junqi Yin, M Todd Young, Sean Treichler, Vitalii Starchenko, Albina Borisevich, Alex Sergeev, and Michael Matheson. 2019. Exascale deep learning for scientific inverse problems. *arXiv preprint arXiv:1909.11150* (2019).

[25] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324. https://doi.org/10.1109/5.726791

[26] Da Li, Yongxin Yang, Yi-Zhe Song, and Timothy M Hospedales. 2017. Deeper, broader and artier domain generalization. In *Proceedings of the IEEE international conference on computer vision*. 5542–5550.

[27] Amrita Mathuriya, Deborah Bard, Peter Mendygral, Lawrence Meadows, James Arnemann, Lei Shao, Siyu He, Tuomas Kärnä, Diana Moise, Simon J. Pennycook, Kristyn Maschhoff, Jason Sewall, Nalini Kumar, Shirley Ho, Michael F. Ringenburg, Prabhat, and Victor Lee. 2018. CosmoFlow: Using Deep Learning to Learn the Universe at Scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, Texas) *(SC '18)*. IEEE Press, Article 65, 11 pages. https://doi.org/10.1109/SC.2018.00068

[28] Benjamin Michalowicz, Eric Raut, Yan Kang, Tony Curtis, Barbara Chapman, and Dossay Oryspayev. 2021. Comparing OpenMP Implementations with Applications Across A64FX Platforms. In *OpenMP: Enabling Massive Node-Level Parallelism*, Simon McIntosh-Smith, Bronis R. de Supinski, and Jannis Klinkenberg (Eds.). Springer International Publishing, Cham, 127–141.

[29] Benjamin Michalowicz, Eric Raut, Yan Kang, Tony Curtis, Barbara Chapman, and Dossay Oryspayev. 2021. Comparing the behavior of OpenMP Implementations with various Applications on two different Fujitsu A64FX platforms. In *Practice and Experience in Advanced Research Computing*. 1–4.

[30] Sparsh Mittal and Shraiysh Vaishay. 2019. A survey of techniques for optimizing deep learning on GPUs. *Journal of Systems Architecture* 99 (2019), 101635.

[31] MLCommons. 2020. *ML Commons*. https://mlcommons.org/en/training-hpc-07/

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[33] RIKEN. 2021. *Fugaku Supercomputer*. https://www.r-ccs.riken.jp/en/fugaku

[34] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[35] Akihiro Tabuchi, Koichi Shirahata, Masafumi Yamazaki, Akihiko Kasagi, Takumi Honda, Kouji Kurihara, Kentaro Kawakami, Tsuguchika Tabaru, Naoto Fukumoto, Akiyoshi Kuroda, Takaaki Fukai, and Kento Sato. 2021. The 16,384-node Parallelism of 3D-CNN Training on An Arm CPU based Supercomputer. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 152–161. https://doi.org/10.1109/HiPC53243.2021.00029

[36] Seiya Tokui, Ryosuke Okuta, Takuya Akiba, Yusuke Niitani, Toru Ogawa, Shunta Saito, Shuji Suzuki, Kota Uenishi, Brian Vogel, and Hiroyuki Yamazaki Vincent. 2019. Chainer: A deep learning framework for accelerating the research cycle. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2002–2011.

[37] TOP500.org. 2020. *HPCG June 2022*. https://www.top500.org/lists/hpcg/2022/06/

[38] TOP500.org. 2020. *Top500 November 2021*. https://www.top500.org/lists/top500/2021/11/

[39] Alexander Ulanov, Andrey Simanovsky, and Manish Marwah. 2017. Modeling Scalability of Distributed Machine Learning. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 1249–1254. https://doi.org/10.1109/ICDE.2017.160

[40] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. 2017. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1492–1500.

[41] Sergey Zagoruyko and Nikos Komodakis. 2016. Wide residual networks. *arXiv preprint arXiv:1605.07146* (2016).