PROFILING AND CHARACTERIZATION OF DEEP LEARNING MODEL INFERENCE ON CPU

BY

YANLI QIAN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Adviser:

Professor Wen-Mei Hwu

ABSTRACT

With the rapid growth of deep learning models and higher expectations for their accuracy and throughput in real-world applications, the demand for profiling and characterizing model inference on different hardware/software stacks is significantly increased. As the model inference characterization on GPU has already been extensively studied, it is worth exploring how performance-enhancing libraries like Intel MKL-DNN help to boost the performance on Intel CPU. We develope a profiling mechanism to capture the MKL-DNN operation calls and formulate the tracing timeline with spans on the server. Through profiling and characterization that give insights into Intel MKL-DNN, we evaluate and demonstrate that the optimization techniques including blocked memory layout, layers fusion, and low precision operation used in deep learning model inference have accelerated the performance on the Intel CPU.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I would like to first thank my thesis advisor, Professor Wen-Mei Hwu, for his guidance and support during my master's study. His wisdom and passion for research and teaching have encouraged me to work hard and dive deep during my graduate research and study. I'm also grateful for what I have learned from the applied parallel programming courses he taught, which helped me a lot during my research development.

I would also like to thank the members of Professor Hwu's research group who get involved in this project: Cheng Li and Abdual Dakkak. It is a pleasure to work together with them and they give me a lot of advice and support.

I want to thank my friends and colleagues during my study here at the University of Illinois at Urbana-Champaign, as well as my friends in other parts of the world, for their encouragement during my graduate study and job hunting.

Finally, I want to say thanks to my family. No words can express my gratitude and love to them.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Background

Machine learning (ML) as a technique of artificial intelligence grants machines the capability to learn from big data and make accurate predictions without human intervention. Deep learning (DL) is a subfield of ML, which is more advanced as it can extract the features from raw data through deep neural network architecture. DL models have been widely applied across various domains to solve real-world problems like image classification, machine translation, object detection, segmentation, etc. To make DL model computation more reliable, robust, and scalable for large datasets running on distributed heterogeneous hardware, the DL frameworks are developed as an interface or library that allows developers to build DL models much more efficiently [1]. Popular frameworks like TensorFlow, PyTorch, and Apache MXNet serve as efficient tools to reduce the computational complexity of DL. Given a DL model, it is crucial to optimize the model so that it achieves efficient runtime performance under a constrained hardware resource. This optimization introduces an increasing demand for understanding and characterizing the DL model using a profiling mechanism. In the past few years, many deep neural networks have been accelerated by many-core processors such as modern graphical processing units (GPUs), which feature high computational throughput and large memory access bandwidth [2]. In partic-

ular, Nvidia supports a parallel computational architecture called CUDA that helps to accelerate simple processing operations in parallel computing methodologies [3]. A central processing unit (CPU) also has the capability to perform fast complex computations such as matrix multiplication or vector operation through auxiliary math kernel libraries. There are trade-offs between using GPU or CPU for a DL model computation in terms of reliability, speed, memory cost, and latency [4]. It is always preferable to have a high throughput, low latency, and low hardware cost for a DL application, so it is critical to use a profilier to understand these performance metrics and reveal the possible optimization for a DL model. As we already have many GPU-based DL profiling tools provided by Nvidia CUDA (e.g., nvproof, CUPTI), it becomes an interesting topic to profile and characterize the DL model performance on CPU.

1.2 Architecture of Deep Learning Frameworks

The rapid proliferation of deep neural networks (DNN) in the past few decades has become the focus of many researchers and developers in AI communities. The collection of DNN architectures is diversified with the surging of neural networks such as AlexNet [5], ResNet [6], VGG [7], GoogleNet [8], etc. The development and innovations of these neural networks are intended to provide better feature detection and higher accuracy in prediction tasks [9], which results in architectural divergences between DNN models. First and foremost is the difference in the number of layers. Starting with AlexNet in 2012, which only has 8 layers in total [5], ResNet has evolved from 18 layers to 152 layers in 2016 [6]. These changes in layer numbers indicate that deep network architectures usually produce better results on complex

learning problems compared to shallow architecture. Moreover, DNNs are distinguished from each other in terms of filter dimensions, the number of parameters, padding/stride size, etc., but the commonality is that they all demand acceleration and optimization on both hardware and software systems [9].

DNN training consists of a feed-forward propagation step using the training samples and a backward propagation that adjusts the network parameters. It is a compute-intensive process due to the massive number of input data and training epochs. The goal of training is to get a converged, well-trained model. There are two types of learning processes in training: supervised and non-supervised. In supervised learning, the labeled ground truth is provided throughout the training process. In non-supervised learning, DNNs can infer and extract the features from input data without labeled responses.

Most DNN structures use a cascade of layers to model in a high-level abstraction. In general, these layers are nonlinear processing units and are connected successively, so the output from the previous layer is fed into the next layer as the input [9]. Common layers are Convolution, ReLu, Pooling, and Fully-Connected.

1.3 Machine Learning Pipeline

A machine learning pipeline is mostly used for automating machine learning workflows. Running an ML algorithm typically involves a sequence of tasks, including data pre-processing, feature engineering (extraction), model fitting, prediction, and validation stage [10]. Figure 1.1 explains the end-to-end ML pipeline block. It can be considered as a high-level logical flow of how an ML project is operated. An ML pipeline manages and monitors the entire

workflow, while an ML model is only a small part of pipeline infrastructure [11].

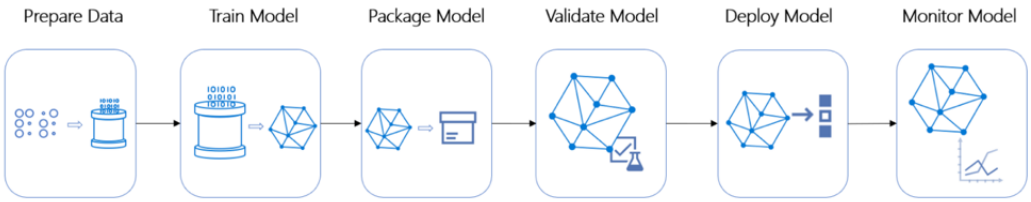


Figure 1.1: Machine learning pipeline. The figure is adapted from [11].

There are various motivations for creating and using an ML pipeline, but the most important thing is to have a better structure and to segregate different parts of the ML stage.

1.4 Math Kernel Library for Deep Neural Network

Math Kernel Library for Deep Neural Network (MKL-DNN) [12] is an open-source performance library for deep learning application developed by Intel and is intended for improving the application performance on Intel CPU architecture. It is also known as the Deep Neural Network Library (DNNL) and oneAPI Deep Neural Network Library (oneDNN). Traditionally, the training processes of deep learning frameworks like TensorFlow, Caffe, and MXNet are accelerated by massive parallel computing on GPUs with CUDA, as well as software frameworks like cuBLAS and cuDNN. CPU can conduct training and inference as well, but it is not commonly used in deep learning due to its limited arithmetic throughput, memory bandwidth, and thread-parallelism. Intel's recent augmen-tation of CPU hardware, as well as the toolkits and libraries like MKL-DNN, [Eliminate the two extra hyphens.]nprovement of CPU-based deep learning ap-plications.

MKL-DNN takes advantage of single-instruction multiple-data (SIMD) instructions through vectorization [13]. It utilizes the hardware cache and instruction sets effectively. Moreover, the library supports multithreading by exploiting multiple cores in modern Intel CPU architecture. Having more available cores to work in parallel can remarkably boost the performance of deep learning applications

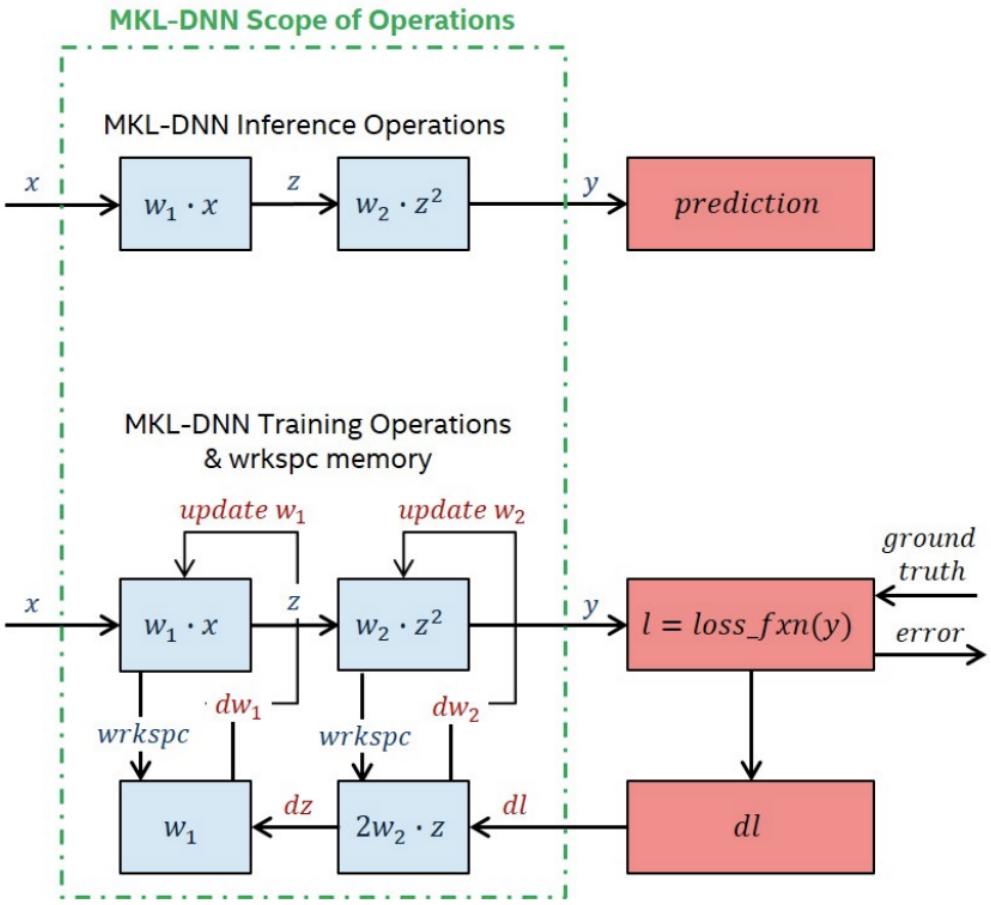


Figure 1.2: MKL-DNN scope of design. The figure is adapted from [14].

MKL-DNN, as the CPU-based deep learning optimization, has achieved a speed-up over the under-optimized design. Figure 1.2 shows the MKL-DNN scope of design. It includes highly vectorized threads as building blocks for implementing convolutional neural networks with C and C++ interface

[15]. Currently, MKL-DNN implements the optimized operators, which are also called primitives, that are commonly used in convolutional neural networks and recurrent neural networks. The most frequently used primitives are Convolution, Batch_normalization, Inner Product, Eltwise_Relu, Pooling, and Reorder.

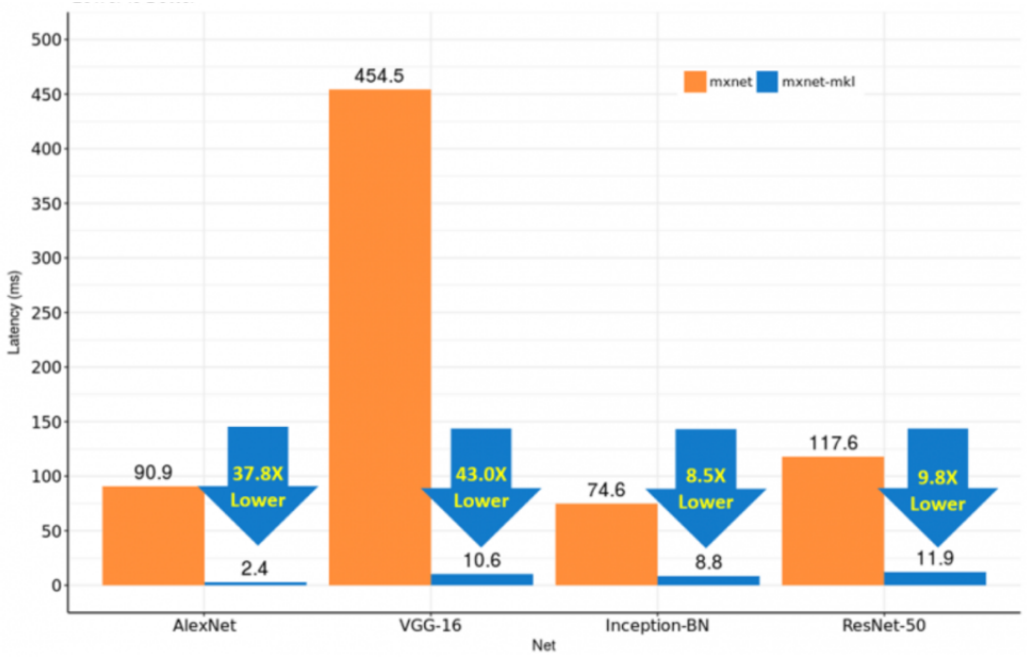


Figure 1.3: MXNet latency improvements with Intel MKL-DNN. The figure is adapted from [16].

MKL-DNN generates optimal code for some functions at runtime based on their input parameters and instruction sets that are supported by the system using just-in-time compilation (JIT). It chooses the optimal computation algorithm under the current CPU dynamic status. Currently, there are many existing frameworks like MXNet supporting Intel-Architecture CPU-based training and inference with MKL-DNN integrated to accelerate the neural network operator on multiple operating systems (e.g., Linux, macOS, and Windows). In the v1.2.0 MXNet release, Intel MKL-DNN is applied for the performance improvement in terms of latency optimization, throughput

improvement, and batch scalability [17].

The figure 1.3 shows that MXNet with MKL-DNN integration achieves an at-most 43x lower latency compared to the MXNet without any optimization when batch size is small [18]. More experiments have demonstrated performance improvements while using MKL-DNN. The result [17] presented by Google at TensorFlow summit 2018 shows that 3x performance enhancement and scaling efficiency can be achieved by using Intel MKL-DNN.

1.5 OpenTracing

OpenTracing [19] is an API library for distributed tracing. Distributed tracing is a method for monitoring applications that are built with microservices architectures, and it helps debug and understand the performance of the software architecture [19].

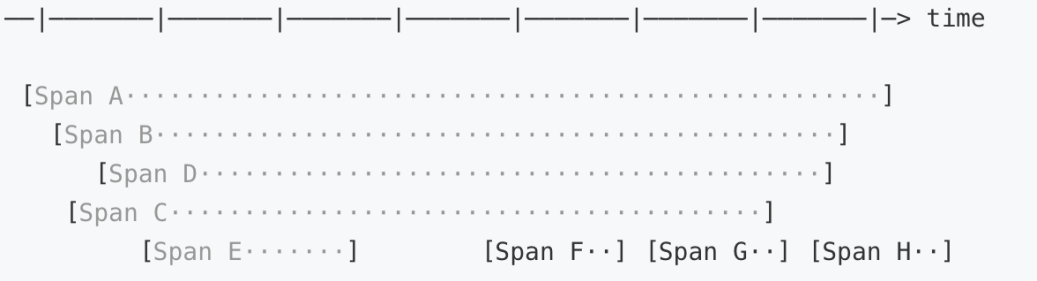


Figure 1.4: A single trace with multiple spans

A trace in OpenTracing is a set of spans ordered in a particular sequence. Figure 1.4 is a simple example for a single trace. A span contains the following metadata: an operation name, a start timestamp, a finish timestamp, span tags, span logs, a span context, and span references. The relationship between two different spans can be ChildOf or FollowsFrom. For a ChildOf relationship, a parent span can have multiple child spans, and they execute

in either parallel or sequential. For a FollowFrom relationship, the child span follows the parent span at any timing position.

OpenTracing has reduced the programmer's workload for instrumenting applications for dis-tributed tracing. To instrument an application via OpenTracing API, it is necessary to have an OpenTracing-compatible tracer correctly deployed and listening for incoming span requests [19]. In the current release version, there are two distributed tracing systems: Zipkin and Jaeger. Both systems are powerful tools for tracking requests, and which one to use in the real-world application largely depends on the development demand. While Zipkin has existed for a more extended period, Jaeger has seen a wider adoption in programming language coverage and OpenTracing instrumentation libraries [20].

CHAPTER 2

RELATED WORK

2.1 MLModelScope

MLModelScope [21] is a platform that helps to facilitate the experimentation with and evaluation of machine learning models. It is designed as a tool to measure and evaluate ML models with real-world artificial intelligence workflow across the hardware/software stack. It serves as an integrated platform, which makes it easier for users to deploy their models/frameworks, evaluate the model performance and accuracy, and analyze the results. The MLMoelScope component for across-stack profiling is named XSP [22], which provides an insightful view of ML model execution and leverages distributed tracing to aggregate profiling results from different sources. Despite the profiling overhead at hardware/software stack, XSP can accurately capture the latencies at all levels with a leveled and iterative measurement approach.

Figure 2.1 depicts the six profiling trace levels of MLModelScope: application, model, framework, layer, library, and hardware. By selecting different trace levels, it allows user to profile the corresponding tracing span. Its scalable across-stack profiling scheme helps to coordinate and combine the tracing span from different profilers into a single trace [21]. Currently, MLModelScope is focused on ML model performance evaluation, specifically on the GPU platform.

The paper by Li, et al [22] illustrates the model-, layer-, and GPU kernel-level profiling

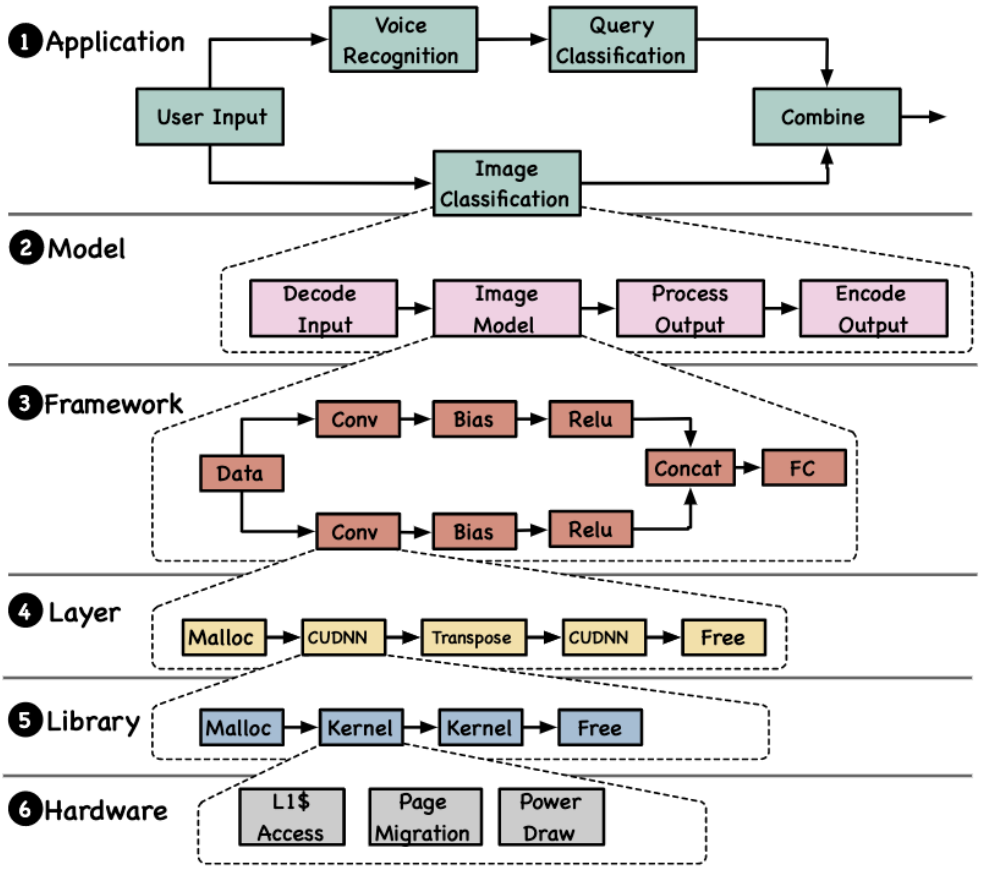


Figure 2.1: The overview of MLModelScope. The figure is adapted from [21].

levels on GPUs.

Model-level profiling measures the execution steps within the model inference. The XSP design makes use of tracing APIs to mark the beginning and ending position within an inference job. For example, in a model prediction using the TensorFlow framework with TF_SessionRun call, the tracing APIs can be put around this call to record the tracing span.

Layer-level profiling aims to profile the layer granularity [22]. An CNN ML model consists of multiple layers, and each layer contains information like operation name (e.g., convolution, batch normalization, softmax), input/ output size, kernel size, etc. Layer-level profiling deploys the existing ML framework

profiling options. TensorFlow RunOptions setting allows users to choose the desired TraceLevel variable, and the result returned from a session run can be collected and published onto the tracing server.

GPU-kernel level profiling is achieved by NVIDIA CUPTI [23] library, which can capture the CUDA API calls, GPU activities, and metrics. Since GPU kernels are always launched asynchronously in ML frameworks or libraries, the XSP design creates two spans in the timeline [22]. One is the launch span that captures the CUDA API calls, and the other is the execution span that captures the execution duration. These two spans are correlated by CUPTI and then aggregated during the profiling analysis.

2.2 MLPerf

MLPerf [24] serves as a machine learning benchmark suite that covers a wide range of ML applications, including image classification, machine translation, reinforcement learning, recommendation, and object detection [25]. In particular, the MLPerf Inference has been developed to measure the performance model training and inference on ML hardware/software.

Table 2.1 indicates a set of tasks in the vision and language area and their corresponding reference models. The heavy task refers to the model that consumes substantial computational resources on large systems like data centers. In contrast, light task refers to the model that is not limited by the computational resources and requires low latency on some small devices [18]. The reference models use a 32-bit floating-point weight. In contrast to the training per-formance metrics, which place more emphasis on the clock time spending to reach a specific model quality, the inference benchmark requires the model qualities, including latency and throughput. All tasks are standardized on

Table 2.1: ML Tasks in MLPerf Inference v0.5 [18].

Area	Task	Reference Model	Data Set	Quality Target
Vision	Image classification (heavy)	ResNet-50 v1.5 25.6M parameters 7.8 GOPS / input	ImageNet (224x224)	99% of FP32 (76.456%) Top-1 accuracy
Vision	Image classification (light)	MobileNet-v1 224 4.2M parameters 1.138 GOPS / input	ImageNet (224x224)	98% of FP32 (71.676%) Top-1 accuracy
Vision	Object detection (heavy)	SSD-ResNet34 36.3M parameters 433 GOPS / input	COCO (1,200x1,200)	99% of FP32 (0.20 mAP)
Vision	Object detection (light)	SSD-MobileNet-v1 6.91M parameters 2.47 GOPS / input	COCO (300x300)	99% of FP32 (0.22 mAP)
Language	Machine translation	GNMT 210M parameters	WMT16 EN-DE	99% of FP32 (23.9 SacreBleu)

well-recognized datasets (e.g., ImageNet, Microsoft COCO, etc.).

According to the MLPerf Inference design, it has three basic concepts: SUT (system under test), sample, and query [18]. Sample can be an image or a sentence, while the query is a set of N samples of an inference run. The load generator is responsible for simulating queries, tracing the latency of a query, validating the accuracy of the result, and computing the final metrics. To test the representativeness of inference applications on different platforms and systems, MLPerf defines four scenarios: single-stream, multiple-streams, server, and offline.

In the single-stream scenario, the load generator sends the initial query and continues to send the query with batch size one (one sample in a query) after the previous one completes. It considers the 90th-percentile latency in a query stream as the performance metric for this scenario [18]. In the multiple-streams scenario, the load generator runs multiple times to test the maximum number of streams that a system can support, which is also the performance metrics for this scenario.

The server scenario evaluates the online request of data center services.

The load generator sends queries in the same manner as the Poisson distribution, and the performance metrics are the maximum possible queries-per-second of the system [18].

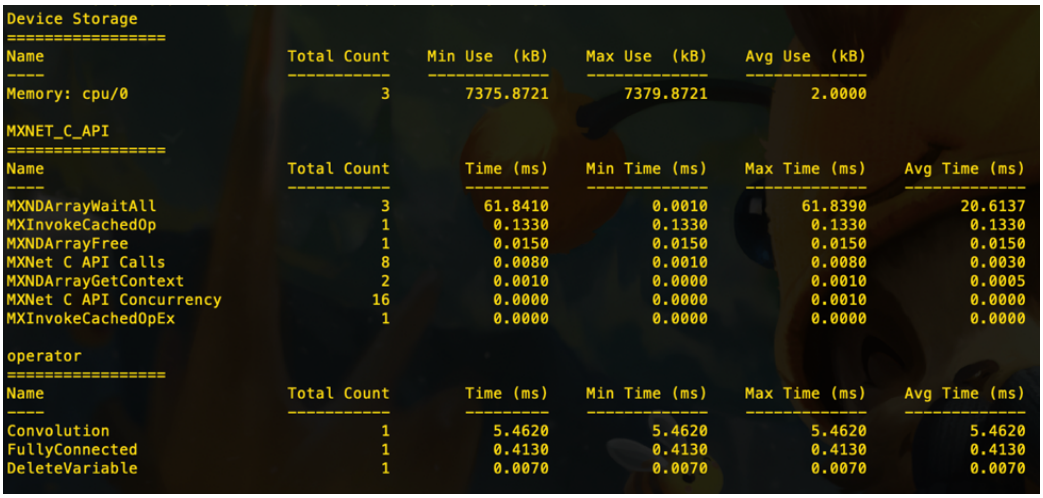
The offline scenario simulates the batch-processing situation in the real world. In this case, a large number of input data can arrive simultaneously, and the latency can be nonnegligible, so the performance metrics are measured as the throughput in samples-per-second.

MLPerf includes two divisions of competition/comparison of performance results among submissions [10]. The closed division only allows the designated ML model and calibration dataset to compare the hardware/ software performance fairly. Open division requires the same dataset but accepts any model with fewer restrictions that intends to advance innovation of ML [12].

2.3 Machine Learning Framework Profilers

Popular machine learning frameworks like TensorFlow and Apache MXNet have their built-in profilers for measuring the framework-level performance metrics.

TensorFlow has tf.Profiler module that can capture the layer information and metadata of a machine learning model layer. MXNet profiler [14] supports four types of profiling: symbolic operators, imperative operators, memory usage, and C API [26]. When ~~properly~~ configuring the profiler setting, the user can set the profiler state to "run" at the beginning of the opera-tions and disable it when all operations are complete. Another way to enable profiling is to turn on the global environment variable for automatic profiling. The profiling result shows the time spent on each operation and the count of operation calls. Figure 2.2 is a sample profiling output using MXNet profiler.



Device Storage

Name	Total Count	Min Use (kB)	Max Use (kB)	Avg Use (kB)
Memory: cpu/0	3	7375.8721	7379.8721	2.0000

MXNET_C_API

Name	Total Count	Time (ms)	Min Time (ms)	Max Time (ms)	Avg Time (ms)
MXNDArrayWaitAll	3	61.8410	0.0010	61.8390	20.6137
MXInvokeCachedOp	1	0.1330	0.1330	0.1330	0.1330
MXNDArrayFree	1	0.0150	0.0150	0.0150	0.0150
MXNet C API Calls	8	0.0080	0.0010	0.0080	0.0030
MXNDArrayGetContext	2	0.0010	0.0000	0.0010	0.0005
MXNet C API Concurrency	16	0.0000	0.0000	0.0010	0.0000
MXInvokeCachedOpEx	1	0.0000	0.0000	0.0000	0.0000

operator

Name	Total Count	Time (ms)	Min Time (ms)	Max Time (ms)	Avg Time (ms)
Convolution	1	5.4620	5.4620	5.4620	5.4620
FullyConnected	1	0.4130	0.4130	0.4130	0.4130
DeleteVariable	1	0.0070	0.0070	0.0070	0.0070

Figure 2.2: MXNet profiler output.

CHAPTER 3

PROFILER IMPLEMENTATION

3.1 Choose Profiling Tool for MKL-DNN

In deep learning, while training is about learning a new capability from existing data, ~~the~~ inference is more about applying this ~~well-~~trained capability to new data. In other words, the model inference step is ~~referring~~ to make a prediction based on a pre-trained machine learning algorithm [27]. Training is computed intensively mostly on GPUs due to their massively parallel~~izable~~ computing capability, but inference can perform on both CPUs and GPUs. Libraries like Intel MKL-DNN and Intel MKL that optimize deep learning applications on Intel processors have provided great benefits for CPU-based DL tasks. To achieve maximum performance, widely used DL frameworks like TensorFlow, Caffe, and MXNet that demand efficient utilization of computational resources have already been adapted to use MKL-DNN primitives for common types of neural network layers [27].

Profiling and characterizing DL model inference is a complicated task. Instead of manually insert timing code around the inference step, we want to find a more elastic and flexible profiling tool for CPU-based DL model inference.

Py-Spy. Py-Spy [28] is a sampling profiler for Python. A large portion of ML frameworks support the Python interface, and Python itself also has an extensive set of libraries for deep learning.

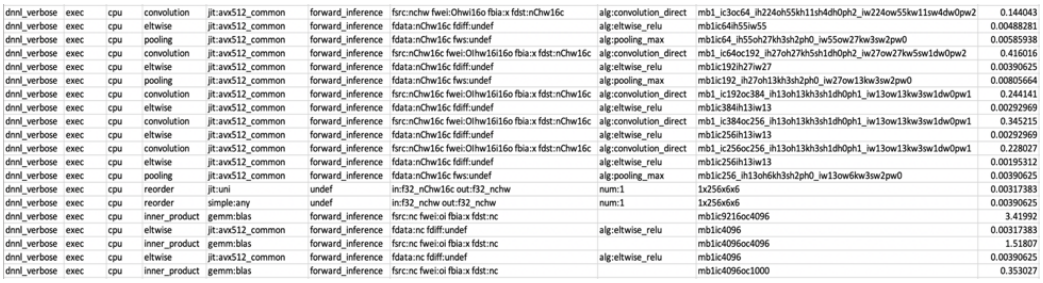
%Own	%Total	OwnTime	TotalTime	Function (filename:line)
0.00%	33.00%	0.000s	0.330s	get_model (gluoncv/model_zoo/model_zoo.py:264)
0.00%	33.00%	0.000s	0.330s	alexnet (gluoncv/model_zoo/alexnet.py:85)
0.00%	33.00%	0.000s	0.330s	<module> (gluon_forward.py:67)
0.00%	18.00%	0.000s	0.180s	get_model_file (gluoncv/model_zoo/model_store.py:188)
16.00%	16.00%	0.160s	0.160s	check_sha1 (gluoncv/utils/download.py:26)
12.00%	12.00%	0.120s	0.120s	load (mxnet/ndarray/utils.py:175)
0.00%	7.00%	0.000s	0.080s	_require_cuda_context (numba/cuda/cudadrv/devices.py:211)
0.00%	7.00%	0.000s	0.410s	<module> (gluon_forward.py:19)
0.00%	7.00%	0.000s	0.080s	push_context (numba/cuda/cudadrv/devices.py:130)
0.00%	7.00%	0.000s	0.080s	cuda_profiler_start (gluon_forward.py:21)
0.00%	7.00%	0.000s	0.080s	get_primary_context (numba/cuda/cudadrv/driver.py:475)
0.00%	7.00%	0.000s	0.080s	get_or_create_context (numba/cuda/cudadrv/devices.py:162)
7.00%	7.00%	0.080s	0.080s	safe_cuda_api_call (numba/cuda/cudadrv/driver.py:292)
0.00%	7.00%	0.000s	0.080s	_get_or_create_context (numba/cuda/cudadrv/devices.py:120)
0.00%	7.00%	0.000s	0.080s	get_context (numba/cuda/cudadrv/devices.py:194)
0.00%	7.00%	0.000s	0.070s	load_parameters (mxnet/gluon/block.py:394)
0.00%	6.00%	0.000s	0.060s	load_parameters (mxnet/gluon/block.py:384)
0.00%	6.00%	0.000s	0.060s	load (mxnet/gluon/parameter.py:959)
0.00%	3.00%	0.000s	0.030s	_call_cached_op (mxnet/gluon/block.py:805)
0.00%	3.00%	0.000s	0.030s	forward (mxnet/gluon/block.py:915)
0.00%	3.00%	0.000s	0.030s	<module> (gluon_forward.py:87)
0.00%	3.00%	0.000s	0.030s	forward_once (gluon_forward.py:80)
2.00%	2.00%	0.020s	0.020s	check_sha1 (gluoncv/utils/download.py:23)
2.00%	2.00%	0.020s	0.020s	__del__ (mxnet/_ctypes/ndarray.py:51)
0.00%	2.00%	0.000s	0.020s	load_parameters (mxnet/gluon/block.py:391)
0.00%	1.00%	0.000s	0.010s	_build_cache (mxnet/gluon/block.py:759)
0.00%	1.00%	0.000s	0.010s	_build_cache (mxnet/gluon/block.py:757)
0.00%	1.00%	0.000s	0.010s	_build_cache (mxnet/gluon/block.py:760)
0.00%	1.00%	0.000s	0.010s	_copyto (<string>:25)
1.00%	1.00%	0.010s	0.010s	_check_container_with_block (mxnet/gluon/block.py:231)
1.00%	1.00%	0.010s	0.010s	hybrid_forward (mxnet/gluon/nn/conv_layers.py:135)
0.00%	1.00%	0.000s	0.010s	_load_init (mxnet/gluon/parameter.py:306)
0.00%	1.00%	0.000s	0.010s	collect_params (mxnet/gluon/block.py:297)
1.00%	1.00%	0.010s	0.010s	__del__ (mxnet/_ctypes/ndarray.py:119)
0.00%	1.00%	0.000s	0.010s	hybrid_forward (gluoncv/model_zoo/alexnet.py:63)
0.00%	1.00%	0.000s	0.010s	collect_params (mxnet/gluon/block.py:305)

Figure 3.1: Py-Spy profiling sample

Figure 3.1 shows an AlexNet inference Py-Spy profiling sample. It discloses application-level performance metrics. The Py-Spy profiling result shows the CPU time and the percentage for each function call. It can be used to trace the Python function calls, but it has no access to capture specific MKL-DNN primitives.

MKL-DNN Verbose. Currently, Intel MKL-DNN has limited support for existing profiling tools since it has no annotated object code ~~generated at run-time~~ [14]; thus, other profilers cannot correctly attribute measurements to sections of the code. It has its own mode called "verbose" that turns out to be very useful for collecting information about the MKL-DNN primitives, including what computational functions are

called, what parameters are passed, and the execution time of these functions. To activate the verbose mode, we can control either the `DNNL_VERBOSE` variable or the built-in function `dnnl_set_verbose`. The verbose mode should take precedence over any environmental variable while running a DL model inference.



dnnl_verbose	exec	cpu	convolution	jit:avx512_common	forward_inference	fsrc:nchw fwei:Ohwi16o fbia:x fdst:nChw16c	alg:convolution_direct	mb1_ic3oc64_ih224oh55kh11sh4dh0ph2_iw224ow55kw11sw4dw0pw2	0.144043
dnnl_verbose	exec	cpu	eltwise	jit:avx512_common	forward_inference	fdata:nChw16c fdiff:undef	alg:eltwise_relu	mb1ic64ih55iw55	0.00488281
dnnl_verbose	exec	cpu	pooling	jit:avx512_common	forward_inference	fdata:nChw16c fws:undef	alg:pooling_max	mb1ic64_ih55oh27kh3sh2ph0_iw55ow27kw3sw2pw0	0.00585938
dnnl_verbose	exec	cpu	convolution	jit:avx512_common	forward_inference	fsrc:nChw16c fwei:OIhw16i16o fbia:x fdst:nChw16c	alg:convolution_direct	mb1_ic64oc192_ih27oh27kh5sh1dh0ph2_iw27ow27kw5sw1dw0pw2	0.416016
dnnl_verbose	exec	cpu	eltwise	jit:avx512_common	forward_inference	fdata:nChw16c fdiff:undef	alg:eltwise_relu	mb1ic192ih27iw27	0.00390625
dnnl_verbose	exec	cpu	pooling	jit:avx512_common	forward_inference	fdata:nChw16c fws:undef	alg:pooling_max	mb1ic192_ih27oh13kh3sh2ph0_iw27ow13kw3sw2pw0	0.00805664
dnnl_verbose	exec	cpu	convolution	jit:avx512_common	forward_inference	fsrc:nChw16c fwei:OIhw16i16o fbia:x fdst:nChw16c	alg:convolution_direct	mb1_ic192oc384_ih13oh13kh3sh1dh0ph1_iw13ow13kw3sw1dw0pw1	0.244141
dnnl_verbose	exec	cpu	eltwise	jit:avx512_common	forward_inference	fdata:nChw16c fdiff:undef	alg:eltwise_relu	mb1ic384ih13iw13	0.00292969
dnnl_verbose	exec	cpu	convolution	jit:avx512_common	forward_inference	fsrc:nChw16c fwei:OIhw16i16o fbia:x fdst:nChw16c	alg:convolution_direct	mb1_ic384oc256_ih13oh13kh3sh1dh0ph1_iw13ow13kw3sw1dw0pw1	0.345215
dnnl_verbose	exec	cpu	eltwise	jit:avx512_common	forward_inference	fdata:nChw16c fdiff:undef	alg:eltwise_relu	mb1ic256ih13iw13	0.00292969
dnnl_verbose	exec	cpu	convolution	jit:avx512_common	forward_inference	fsrc:nChw16c fwei:OIhw16i16o fbia:x fdst:nChw16c	alg:convolution_direct	mb1_ic256oc256_ih13oh13kh3sh1dh0ph1_iw13ow13kw3sw1dw0pw1	0.228027
dnnl_verbose	exec	cpu	eltwise	jit:avx512_common	forward_inference	fdata:nChw16c fdiff:undef	alg:eltwise_relu	mb1ic256ih13iw13	0.00195312
dnnl_verbose	exec	cpu	pooling	jit:avx512_common	forward_inference	fdata:nChw16c fws:undef	alg:pooling_max	mb1ic256_ih13oh6kh3sh2ph0_iw13ow6kw3sw2pw0	0.00390625
dnnl_verbose	exec	cpu	reorder	jit:uni	undef	in:f32_nChw16c out:f32_nchw	num:1	1x256x6x6	0.00317383
dnnl_verbose	exec	cpu	reorder	simple:any	undef	in:f32_nchw out:f32_nchw	num:1	1x256x6x6	0.00390625
dnnl_verbose	exec	cpu	inner_product	gemm:blas	forward_inference	fsrc:nc fwei:oi fbia:x fdst:nc		mb1ic9216oc4096	3.41992
dnnl_verbose	exec	cpu	eltwise	jit:avx512_common	forward_inference	fdata:nc fdiff:undef	alg:eltwise_relu	mb1ic4096	0.00317383
dnnl_verbose	exec	cpu	inner_product	gemm:blas	forward_inference	fsrc:nc fwei:oi fbia:x fdst:nc		mb1ic4096oc4096	1.51807
dnnl_verbose	exec	cpu	eltwise	jit:avx512_common	forward_inference	fdata:nc fdiff:undef	alg:eltwise_relu	mb1ic4096	0.00390625
dnnl_verbose	exec	cpu	inner_product	gemm:blas	forward_inference	fsrc:nc fwei:oi fbia:x fdst:nc		mb1ic4096oc1000	0.353027

Figure 3.2: AlexNet inference task verbose output

Figure 3.2 shows a sample inference profiling using verbose mode for AlexNet from Apache MXNet Gluon model zoo [29]. Each ~~subsequent~~ line in the figure gives the infor-mation of an MKL-DNN primitive call, containing marker string, operation, engine kind, primitive name, propagation kind, input/output data format, auxiliary information, description, and execution time in millisecond. For convolution primitive, the problem description is dumped in BenchDNN [30] format. Other primitives like reorder, sum, and concatenate simply describes the logical dimensions.

3.2 Data Parsing and Processing

Verbose mode outputs a list of comma-separated lines that describe the MKL-DNN primitive calls. It gives a framework library level view of a model inference step, as it records the details of primitives that are used for deep neural network computation. Since verbose output does not distinguish the layer of

a model, we need to process the data to reproduce the layer information, e.g., the execution time spent on each layer of DL model, for a holistic view of the DL model inference stage. A framework can perform model optimization at runtime, so the measured layers may differ from those statically defined in the model graph [6]. We also notice that MKL-DNN has its mechanism to determine the best fit memory layout at the runtime, so there are certain operations for memory format reorder operations that happened during the inference job.

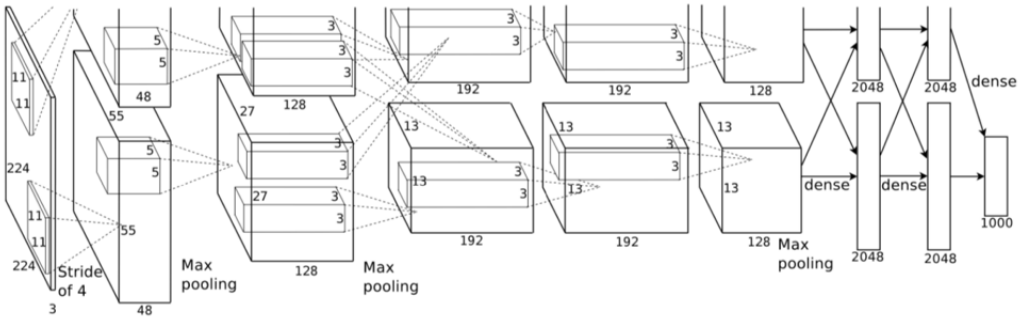


Figure 3.3: AlexNet architecture. The figure is adapted from [29].

We use the inference task for AlexNet as an example. Figure 3.3 elaborates the architecture of AlexNet, as proposed in [29] by Krizhevsky. To extract the layer in accordance with the given model architecture, the ~~curial~~ attributes in the verbose output are the primitive kinds and their descriptions. The size of the input, output, and kernel can be obtained from the description string that is separated by "_". We make use of the regular expression in the Golang "regexp" package to derive the corresponding dimensions from the verbose output in key-value pairs and collect these pairs to match the statistical model layer information. Then we can reconstruct each layer of the model in sequence and get the output as shown in Figure 3.4~~:~~. Note that we have annotated the layer information in the last column of the table.

crucial?

dnnl_verbose	exec	cpu	convolution	jit:avx512_common	forward_inference	fsrc:nchw fwei:Ohwi16o fbia:x fdst:nChw16c	alg:convolution_direct	mb1_ic3oc64_ih224oh55kh11sh4dh0ph2_iw224ow55kw11sw4dw0pw2	0.144043	conv1
dnnl_verbose	exec	cpu	eltwise	jit:avx512_common	forward_inference	fdata:nChw16c fdiff:undef	alg:eltwise_relu	mb1ic64ih55iw55	0.00488281	conv1
dnnl_verbose	exec	cpu	pooling	jit:avx512_common	forward_inference	fdata:nChw16c fws:undef	alg:pooling_max	mb1ic64_ih55oh27kh3sh2ph0_iw55ow27kw3sw2pw0	0.00585938	pooling1
dnnl_verbose	exec	cpu	convolution	jit:avx512_common	forward_inference	fsrc:nChw16c fwei:OIhw16i16o fbia:x fdst:nChw16c	alg:convolution_direct	mb1_ic64oc192_ih27oh27kh5sh1dh0ph2_iw27ow27kw5sw1dw0pw2	0.416016	conv2
dnnl_verbose	exec	cpu	eltwise	jit:avx512_common	forward_inference	fdata:nChw16c fdiff:undef	alg:eltwise_relu	mb1ic192ih27iw27	0.00390625	conv2
dnnl_verbose	exec	cpu	pooling	jit:avx512_common	forward_inference	fdata:nChw16c fws:undef	alg:pooling_max	mb1ic192_ih27oh13kh3sh2ph0_iw27ow13kw3sw2pw0	0.00805664	pooling2
dnnl_verbose	exec	cpu	convolution	jit:avx512_common	forward_inference	fsrc:nChw16c fwei:OIhw16i16o fbia:x fdst:nChw16c	alg:convolution_direct	mb1_ic192oc384_ih13oh13kh3sh1dh0ph1_iw13ow13kw3sw1dw0pw1	0.244141	conv3
dnnl_verbose	exec	cpu	eltwise	jit:avx512_common	forward_inference	fdata:nChw16c fdiff:undef	alg:eltwise_relu	mb1ic384ih13iw13	0.00292969	conv3
dnnl_verbose	exec	cpu	convolution	jit:avx512_common	forward_inference	fsrc:nChw16c fwei:OIhw16i16o fbia:x fdst:nChw16c	alg:convolution_direct	mb1_ic384oc256_ih13oh13kh3sh1dh0ph1_iw13ow13kw3sw1dw0pw1	0.345215	conv4
dnnl_verbose	exec	cpu	eltwise	jit:avx512_common	forward_inference	fdata:nChw16c fdiff:undef	alg:eltwise_relu	mb1ic256ih13iw13	0.00292969	conv4
dnnl_verbose	exec	cpu	convolution	jit:avx512_common	forward_inference	fsrc:nChw16c fwei:OIhw16i16o fbia:x fdst:nChw16c	alg:convolution_direct	mb1_ic256oc256_ih13oh13kh3sh1dh0ph1_iw13ow13kw3sw1dw0pw1	0.228027	conv5
dnnl_verbose	exec	cpu	eltwise	jit:avx512_common	forward_inference	fdata:nChw16c fdiff:undef	alg:eltwise_relu	mb1ic256ih13iw13	0.00195312	conv5
dnnl_verbose	exec	cpu	pooling	jit:avx512_common	forward_inference	fdata:nChw16c fws:undef	alg:pooling_max	mb1ic256_ih13oh6kh3sh2ph0_iw13ow6kw3sw2pw0	0.00390625	pooling3
dnnl_verbose	exec	cpu	reorder	jit:uni	undef	in:f32_nChw16c out:f32_nchw	num:1	1x256x6x6	0.00317383	
dnnl_verbose	exec	cpu	reorder	simple:any	undef	in:f32_nchw out:f32_nchw	num:1	1x256x6x6	0.00390625	
dnnl_verbose	exec	cpu	inner_product	gemm:blas	forward_inference	fsrc:nc fwei:oi fbia:x fdst:nc		mb1ic9216oc4096	3.41992	fc1
dnnl_verbose	exec	cpu	eltwise	jit:avx512_common	forward_inference	fdata:nc fdiff:undef	alg:eltwise_relu	mb1ic4096	0.00317383	fc1
dnnl_verbose	exec	cpu	inner_product	gemm:blas	forward_inference	fsrc:nc fwei:oi fbia:x fdst:nc		mb1ic4096oc4096	1.51807	fc2
dnnl_verbose	exec	cpu	eltwise	jit:avx512_common	forward_inference	fdata:nc fdiff:undef	alg:eltwise_relu	mb1ic4096	0.00390625	fc2
dnnl_verbose	exec	cpu	inner_product	gemm:blas	forward_inference	fsrc:nc fwei:oi fbia:x fdst:nc		mb1ic4096oc1000	0.353027	fc3

Figure 3.4: AlexNet layer reconstruction output.

3.3 Publish to Tracing Server

To incorporate the pre-processed verbose data for different levels of profiling (e.g., framework level, library level), we leverage the distributed tracing system Jaeger [20], as discussed in Chapter 1. A span in Jaeger is referred to as a piece of timed operation or work. Each span contains unique information regarding the operation, which can be determined and customized by the developer. We collect the start and end timestamps, as well as the duration of primitive operations. The key-value pair tags, operation name, and other auxiliary information are described in the user-defined annotation. At the beginning of the model inference, we initiate a root span as the parent reference for all following spans. We then build the child span and establish necessary parent-child relations for some cases (e.g., make library-level spans the children of layer-level spans) during the span creation, which depends on what level of profiling we want to perform.

We create and publish the spans to either local or remote tracing server using a tracer. Then the tracing server aggregates the spans as published by different tracers into a single tracing timeline. First, we have to define a TraceEvent struct type that contains a collection of fields: operation name, timestamp, execution time, in/out data format, and a set of parameter key-value pairs. Each line of verbose output is assigned as a TraceEvent, and

then grouped as a list of TraceEvent for publishing.

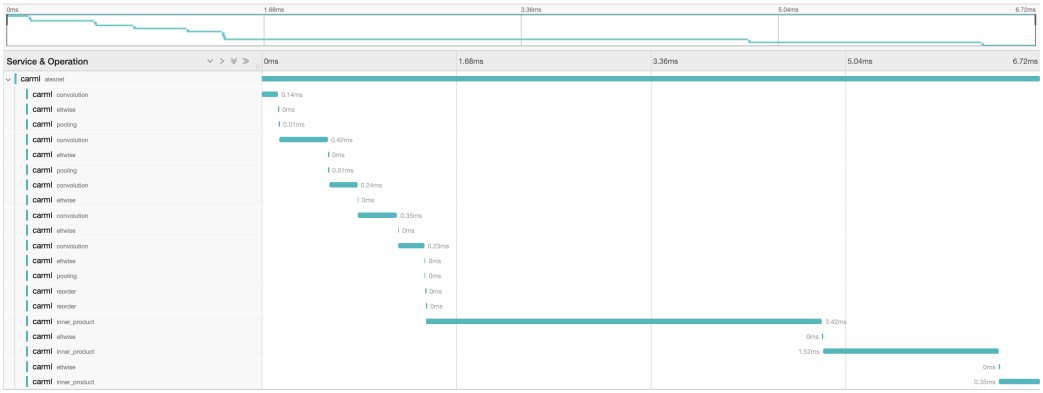


Figure 3.5: AlexNet inference output published to the tracing server.

Figure 3.5 shows a sample library-level profiling trace that represents one iteration of forward inference with `batch_size = 1` for AlexNet. The profiling result reveals what MKL-DNN primitives are called and the timestamp for each call in the inference endurance.

CHAPTER 4

PERFORMANCE CHARACTERIZATION OF CPU BASED DL INFERENCE

4.1 Evaluation on DL Models and Their Variants

4.1.1 Hyper-parameters tuning

In this section, we perform experiments on MobileNet1.0 [31] and evaluate how the performances of the model and its variants are altered by tuning the hyperparameters through profiling. MobileNet is a lightweight convolutional neural network primarily built for mobile or embedded device vision applications [32]. It takes advantage of deep-wise convolution to reduce the number of parameters and computational complexity. MobileNet introduces a hyper-parameter α, named the width multiplier, within 0 and 1. By multiplying α to channel (c), we can reduce the total number of channels to make the original model narrower. We apply α equals 0.25, 0.5, 0.75, and 1.0 to get four variant MobileNet models in our test and profile the corresponding inference performance in Figure 4.1. The only difference between variants is the number of channels. We can see that the more channel we have in a model inference, the higher the latency it induces. On the other hand, having more channels also results in higher accuracy. Therefore, it is essential to find an optimal number of channels to balance between latency

Another interesting observation is that for variants with α (channel number = 8 and 24), the reorder primitive is ca

What does "deep-wise" mean?

Do you have the corresponding accuracy numebrs for each alpha value? If so, you should show the numbers side-by-side with the latency figure.

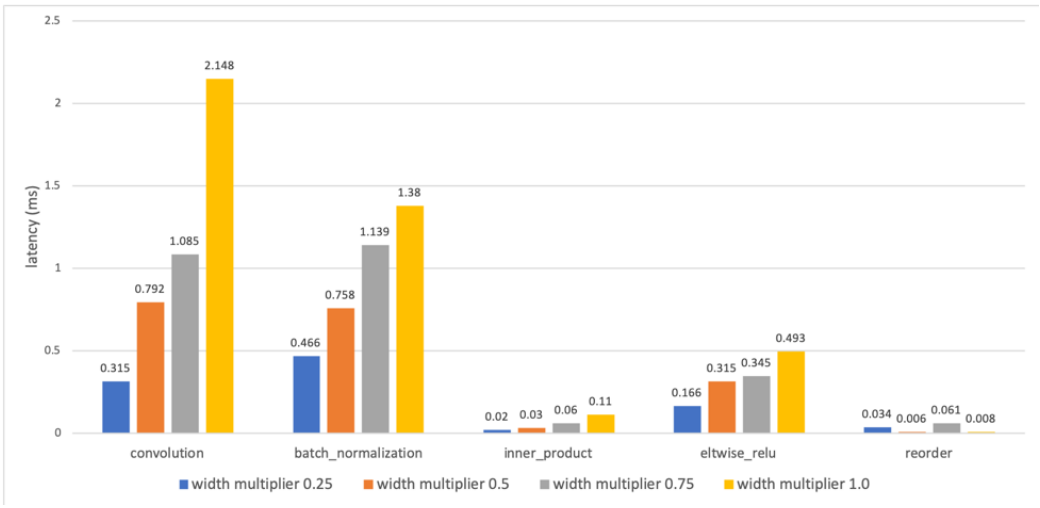


Figure 4.1: MobileNet variants latency of MKL-DNN primitives

model inference while the other two cases call it only once. This behavior is correlated to the MKL-DNN memory layout issue. MKL-DNN requires the use of blocked memory layout, which means concatenating input a fixed length of a block. The existing popular memory layouts recognition are `nhwc` (batch_size, height, width, channel), `nchw`, and `chwn`. If nchw is blocked by 16 (denote as `nChw16c`), then 16 channels will be treated as a single input and dumped into system memory at one time. MKL-DNN allows channels to be blocked by 8 or 16. Therefore, if channels are a multiple of 8 or 16, MKL-DNN will automatically detect and perform the reorder primitive to copy data between different memory formats.

Did you mean "into a fixed-length block"?

MobileNet-0.25 and MobileNet-0.75 start with channel numbers 8 and 24, so `nChw8c` serves as the primary memory layout in the computation of the first few layers. As the channel number becomes a multiple of 16 (e.g., $c = 48$), MKL-DNN evokes the reorder primitive immediately to switch memory format from `nChw8c` to `nChw16c`. `nChw16c` is preferable to `nChw8c` because it is supported and accelerated by AVX-512 while `nChw8c` only uses AVX2. AVX-512 instruction set supports 512-bit width register and higher computa-

tional capability. MobileNet-0.5 and MobileNet-1.0 whose starting channel numbers are already multiples of 16 do not require reordering during the inference. The above explains why only the numbers of reorder operations differ among the four variants of MobileNet.

4.1.2 Neural Network Depth Tuning

In this section, we evaluate the pre-trained ResNet from the MXNet Gluon model zoo with different network depths and perform insightful profiling for all its five variants. The architecture of 18-layer, 34-layer, 50-layer, 101-layer, and 152-layer ResNet is shown in Table 4.1. In He et al. [6] divide the entire... uthor divides the entire network into 5 components, so each layer name represents a section of network architecture but not a single layer.

Table 4.1: ResNet Architecture. The table is adapted from [6].

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

We run the model inference on the Intel i9-7900x processor and use the profiler as introduced in Chapter 3 to characterize the latency in each layer for the five models. We only consider the latency of the convolutional layer and fully-connected layer and do not take the activation or pooling layer

into account. Figure 4.2 shows the latency comparison between different layers for the five models. From conv1 to conv_5x, the latency is computed by accumulating the execution time of MKL-DNN convolution primitives. The fully-connected layer involves three MKL-DNN primitive calls: sum, eltwise_relu, and inner_product, so the latency is the ~~add-on~~ sum of all three operations time.

We observe that for models whose layer structures are the same, they have similar latency in that layer. The channel of 50-layer conv5_x is 8 times larger than conv2_x, but the latency shown in Figure 4.2 is only 1.1 times longer. One possible explanation for this is the hardware architecture support on Intel CPUs. This is intuitive assuming that the CPU can accommodate up to 8-wide SIMD vectors. The layout allows up to 8 channels to be consecutive to each other for SIMD execution.

4.2 Model Quan MKL-DNN

4.2.1 Low precisio U

We compare the perfo m Apache MXNet on the Intel Core i9-7900X processor with 20-core for three configurations: the default MXNet model, the MXNet 32-bit floating-point (FP32) model built with MKL-DNN, and MXNet INT8 model built with MKL-DNN. Figures 4.3 and 4.4 show the MXNet inference speed-up with `batch_size = 1` and `batch_size = 128` accordingly under the above three configurations. We perform experiments on six pre-trained deep neural network models. The default MXNet native build serves as the baseline. Compared to the result between the baseline and MXNet optimized with MKL-DNN, it is obvious that both inference throughput and latency performance are greatly improved.

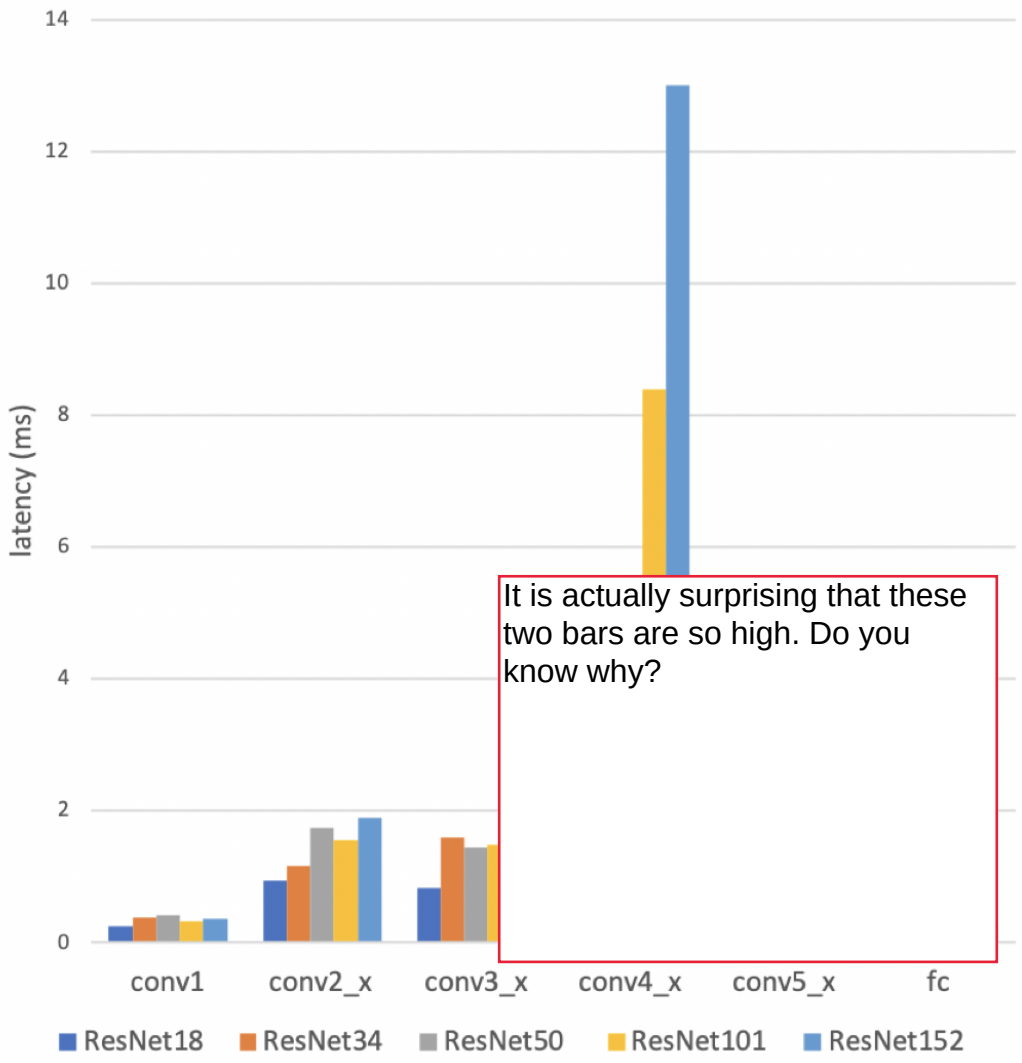


Figure 4.2: ResNet and its variants' execution time on layers

To enhance the performance metrics and reduce the deployment cost for inference, Apache MXNet has integrated the MKL-DNN backend into its default build. It is also commended for its quantization approaches that "allow inference to be carried out using integer-only arithmetic" [33]. In general, most of the DNN models are in FP32 precision mode, but the quantized inference takes advantage of lower precision (e.g., INT8). The paper [33] indicates that lower precision arithmetic helps to speed up the computation and save more memory storage and bandwidth. In Figures 4.3 and 4.4, the

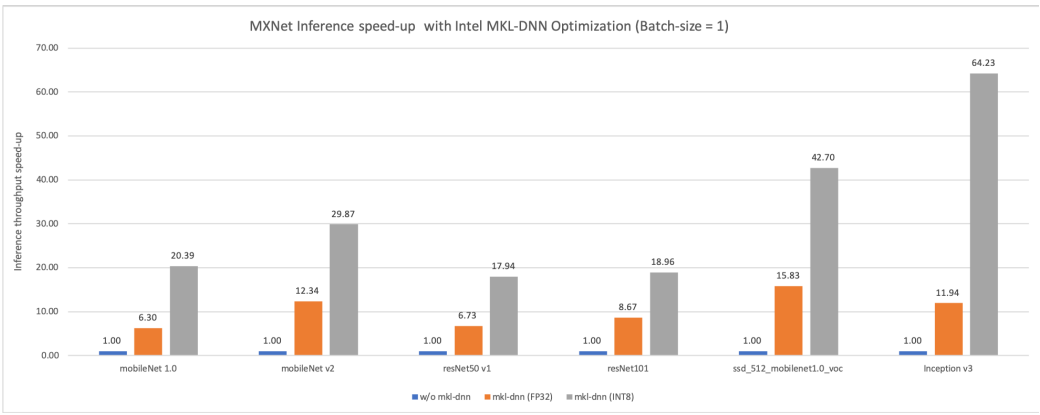


Figure 4.3: MXNet model inference speed-up with `batch_size = 1`.

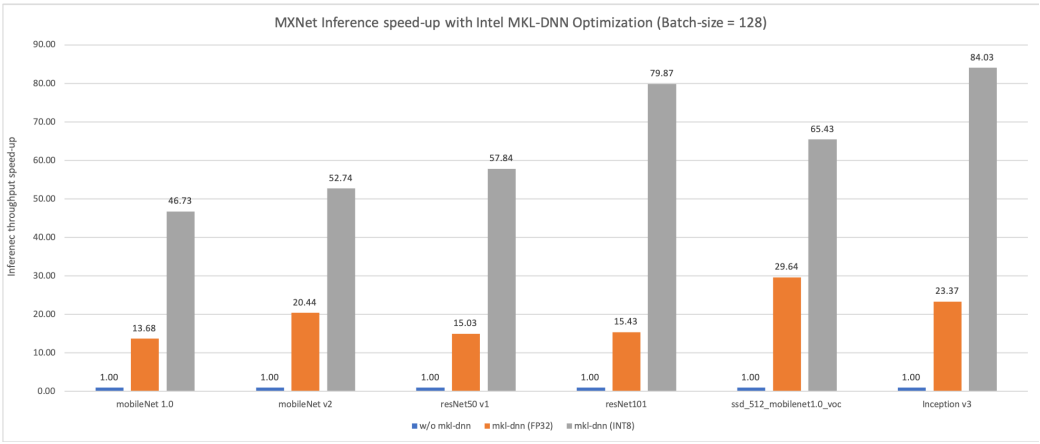


Figure 4.4: MXNet model inference speed-up with `batch_size = 128`.

second and third columns in each cluster show the acceleration between using FP32 and INT8. We can conclude that using INT8 low-precision mode results in more significant performance improvement.

Figure 4.5 shows the pipeline for deriving an INT8 inference from a given pre-trained FP32 model. Quantization with calibration is an offline stage that collects statistical thresholds information and uses symmetric quantization for each layer to determine the scaling factors [34]. The following INT8 inference stage then makes use of the calibrated model for inference with higher throughput but lower accuracy due to the decrease in precision.

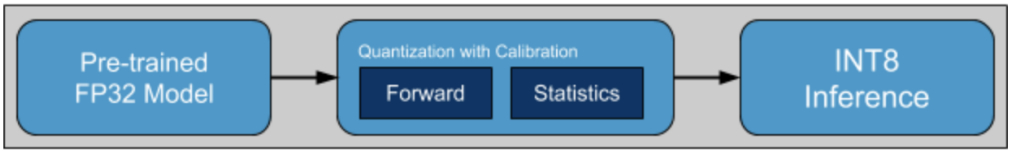


Figure 4.5: Model quantization pipeline. The figure is adapted from [34].

4.2.2 Layer Fusion

We take advantage of our profiling schema, as discussed in Chapter 3, to provide a detailed analysis of Apache MXNet ResNet50.v1 quantization. We perform both FP32 and INT8 ResNet50.v1 inference with `batch_size = 1` on the same Intel-CPU Architecture. We call the FP32 model the "baseline model" and the INT8 model the "quantized model". As shown in Figure 4.3 and 4.4, the quantized model achieves a 3x or even 4x speed-up over the baseline model with `batch_size = 1` and `batch_size = 128`.

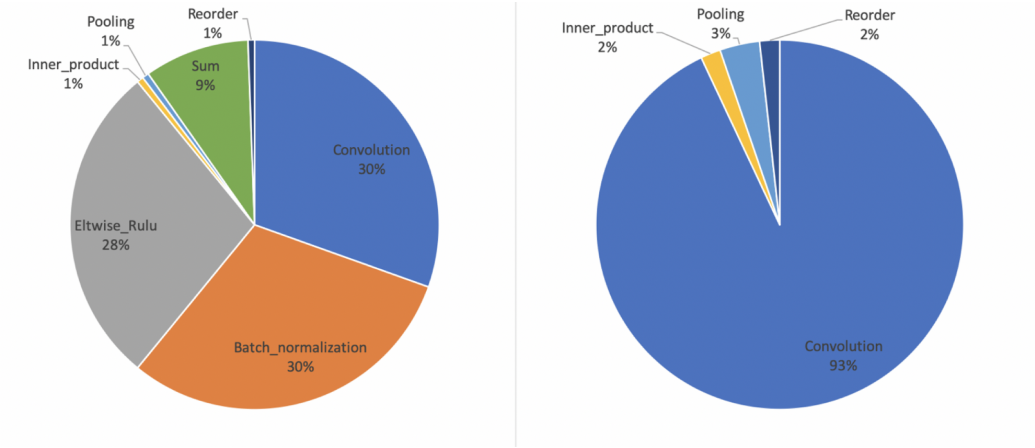


Figure 4.6: Left: MKL-DNN primitive calls distribution for ResNet50.v1 (FP32). Right: MKL-DNN primitive calls distribution for ResNet50.v1 (INT8).

Since the pre-trained ResNet50.v1 from MXNet Gluon model zoo is already optimized with the MKL-DNN, Figure 4.6 shows the distribution of the number of MKL-DNN primitive calls in both baseline and quantized

models. Reorder does not count as a strict forward inference in propagation, and it mostly appears in the warmup stage for inference. It is mainly used for transforming data between different memory formats without changing the mathematical tensor in MKL-DNN [35]. The tracing spans visualize the sequence of the primitive calls in the ResNet50 v1 inference step and reorder always happens before convolution. THe reason for this is that, in MKL-DNN, some compute-intensive operations like convolution, inner product, and LSTM require particular memory format to achieve a performance boost.

The

We notice that the number of convolution primitives in the baseline has an approximately x3 i

This is not clear to me. It looks to me that the number of convolutions did not decrease but the number of other layers decreased so the percentage o fthe convolution becomes 3x. Please check the data. If it is indeed because the number of convolutions increased by 3X, I would like to understand why.

havior is the decrea

eltwise_relu, and su

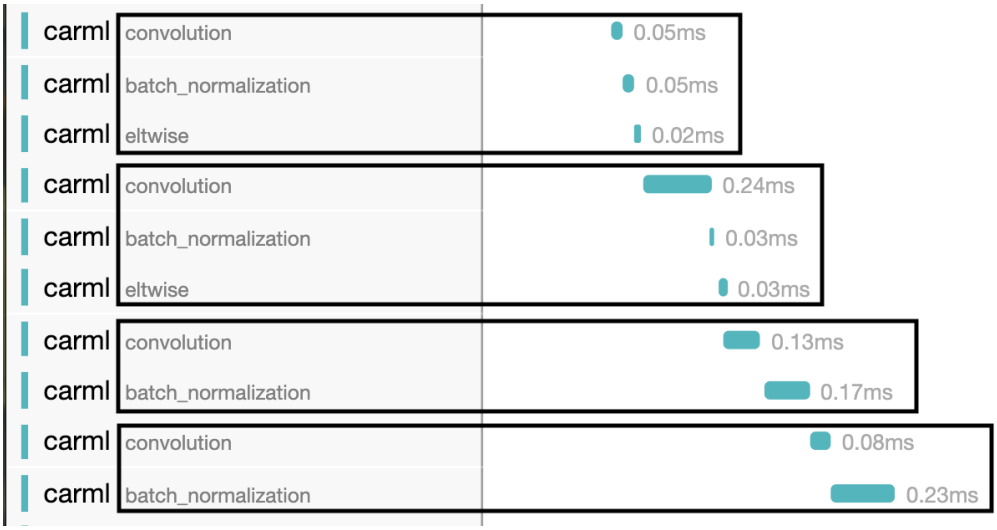


Figure 4.7: Residual block for ResNet50.v1 (FP32).

Layer fusion in the quantized model is to combine the primitive operations like convolution, batch_normalization, and eltwise_relu into a single quan-

Figure 4.8: Residual block for ResNet50.v1 (INT8).

tized convolution layer. It is powered by the MKL-DNN that allows several operations running in a single execution. The benefit of layer fusion is to reduce the library call overhead. Instead of calling three operations and having data read/write in t only requires one-time preparation for a library call, s I/O operations. Figure 4.7 shows a residual block witl tion-batch_normalization-relu operations, and Figure 4 sidual block with four corresponding fused convolution

The bigger benefit is usually the reduced memory bandwidth consumption since the data is no longer written out to memory and read back by the next layer.

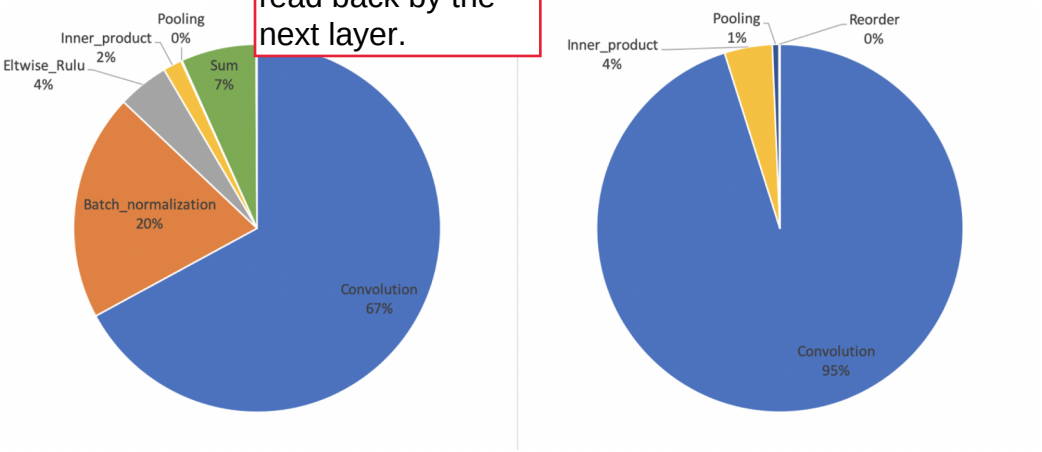


Figure 4.9: Left: MKL-DNN primitive calls execution time distribution for ResNet50.v1 (FP32). Right: MKL-DNN primitive calls execution time distribution for ResNet50.v1 (INT8).

We observe that for each quantized convolution layer, the execution time for a single primitive operation is less than the accumulated time for three

separate primitives. This observation validates the performance improvement powered by layer fusion and quantization. Figure 4.9 shows the latency distribution aggregated by layer type in baseline and quantized model.

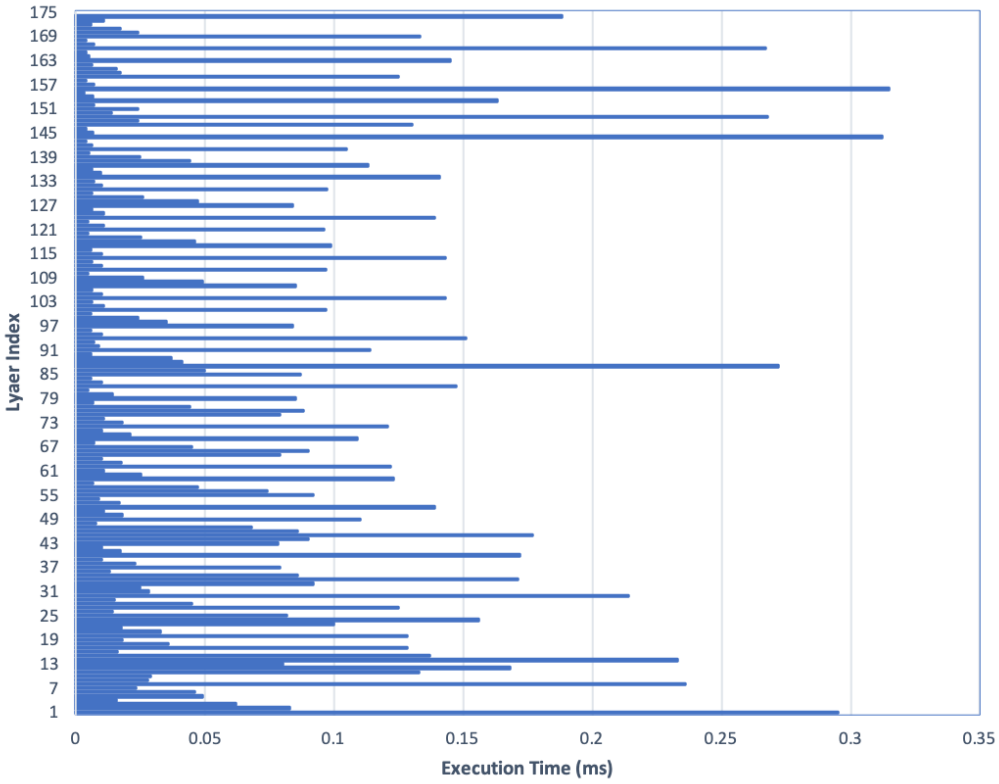


Figure 4.10: Execution time for ResNet50.v1 (FP32) layer primitives

Convolution makes up a majority of latency in both models. Especially in the INT8 precision model, it takes up to 95% of total latency. The time complexity of convolution is determined by the input/output channel number and the kernel size. Figure 4.10 shows the execution time for each layer primitive in the FP32 precision model. According to [6], which discusses the proposed structure of ResNet, the last three blocks (each block consists of three convolution layers, and activation or pooling layers if necessary) have larger channel numbers. It is reflected in Figure 4.10 that the layer with a larger index is more time-consuming than the layer with a lower index.

Figure 4.11 shows the correlated INT8 model inference execution time on different layers. The average latency on each layer is 50% less than the average layer latency in the FP32 model. Along with the reduced number of primitives calls in each layer, the quantized model overall performance is better than that of the baseline model.

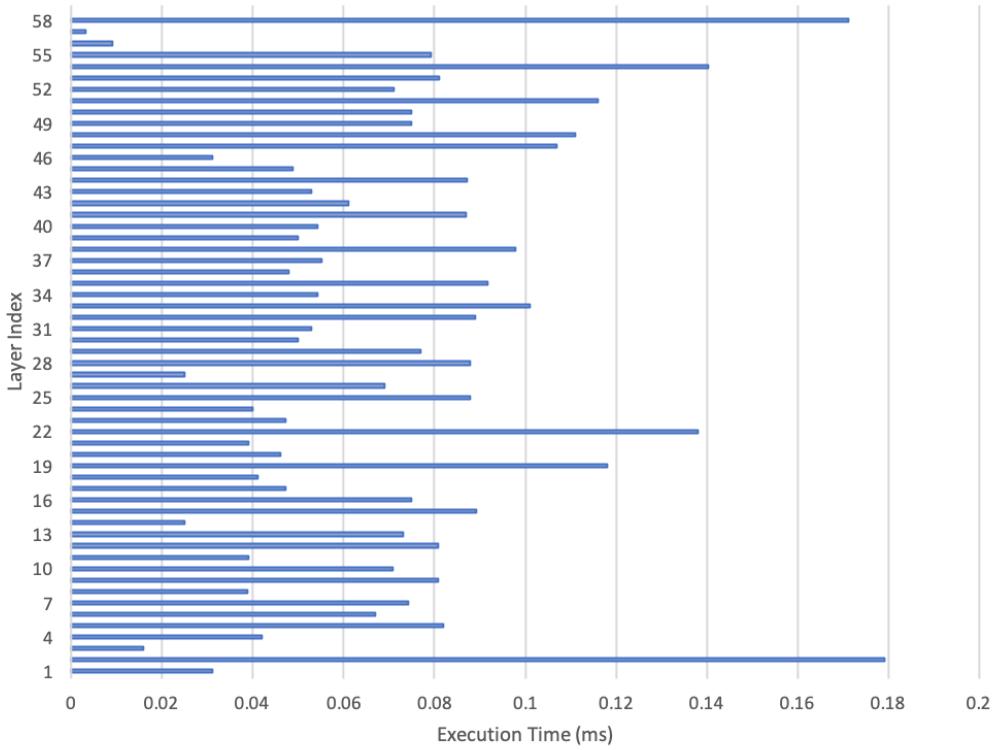


Figure 4.11: Execution time for ResNet50.v1 (INT8) layer primitives

CHAPTER 5

CONCLUSION

5.1 Accomplishment

In conclusion, the Intel MKL-DNN is a powerful library for machine learning and deep learning acceleration on Intel CPU architecture. For any deep neural network application, it is worthwhile to characterize its performance, which helps developers to explore more possible optimization on the model in the future. There are helpful tools to get the application or model level profiling, but it is also beneficial to learn the behavior of the framework libraries.

We study how the MKL-DNN library helps to boost the performance of the DNN inference stage. We design the profiler to capture the MKL-DNN library API calls together with all necessary auxiliary information (e.g., latency, input/output data format, memory layout, etc.), and we parse and analyze these data to get a holistic library- and layer-level view. We then warp and convert these statistics/metadata into spans, and publish to tracer server as tracing events for better data visualization. We also perform experiments to evaluate a variety of DNN models from Apache MXNet leveraging our profiler as well as other toolkits. By analyzing the profiling results and the corresponding model implementation, we have seen how MKL-DNN helps to improve the ML model inference performance in terms of blocked memory layout, layer fusion, and low-precision quantization.

5.2 Lessons Learned

Profiling CPU-based DL model inference is an interesting field to dive into deeply. GPU has already been widely adopted in deep learning. Its performance on deep neural networks training and inference stages is studied extensively by researchers as well. Therefore, it is worthwhile to explore CPU and its performance libraries that help to make the DL computation more efficient.

I have learned how to design and implement a profiler that helps to visualize and gain insight into a model inference stage. It is crucial to have a general idea of what one's profiler should be able to show at the first stage. A good profiler is user-friendly and should be intuitive enough for better visualization.

By implementing a profiling schema and characterizing the inference of DNN models, I gained a deep understanding of the Intel MKL-DNN library. The information I get from profiling results allows me to explore the mechanism behind MKL-DNN. I learned the optimization techniques used in MKL-DNN through conducting extensive research and experiments, and the profiling outputs validate my predictions as well. Through this research study, I found that there are many commonalities between optimization techniques on both CPU and GPU. I also gained insights into optimization in future work.

REFERENCES

[1] "Top 5 deep learning frameworks, applications, and comparisons." [Online]. Available: https://www.analyticsvidhya.com/blog/2019/03/deep-learning-frameworks-comparison/

[2] C. Li, Y. Yang, M. Feng, S. Chakradhar, and H. Zhou, "Optimizing memory efficiency for deep convolutional neural networks on GPUs," *SC16*, 2016.

[3] C. Li, Y. Yang, M. Feng, S. Chakradhar, and H. Zhou, "Optimizing memory efficiency for deep convolutional neural networks on GPUs," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 633–644.

[4] G. Baltazar, "CPU vs GPU in machine learning," 2018. [Online]. Available: https://blogs.oracle.com/datascience/cpu-vs-gpu-in-machine-learning

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, p. 84–90, May 2017.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[7] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CVPR*, April 2015.

[8] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014. [Online]. Available: http://arxiv.org/abs/1409.4842

[9] A. A. Awan, H. Subramoni, and D. 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*, ser. MLHPC'17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3146347.3146356

[10] S. Koen, "Architecting a machine learning pipeline," 2019. [Online]. Available: https://towardsdatascience.com/architecting-a-machine-learning-pipeline-a847f094d1c7

[11] "What are ML pipelines - Azure machine learning." [Online]. Available: https://docs.microsoft.com/en-us/azure/machine-learning/concept-ml-pipelines

[12] "oneDNN." [Online]. Available: https://github.com/oneapi-src/oneDNN

[13] "Effectively train and execute machine learning and deep learning projects on CPUs." [Online]. Available: https://techdecoded.intel.io/resources/effectively-train-and-execute-machine-learning-and-deep-learning-projects-on-cpus/gs.4dwt2d

[14] "Intel math kernel library for deep neural networks (intel mkl-dnn)." [Online]. Available: https://www.alcf.anl.gov/sites/default/files/2019-08/Greeneltch_Argonne_mkldnn%20201905_Final.pdf

[15] H. Lupesko, "MXNet boosts CPU performance with MKL-DNN," Jan 2019. [Online]. Available: https://medium.com/apache-mxnet/mxnet-boosts-cpu-performance-with-mkl-dnn-b4b7c8400f98

[16] "Apache MXNet v1.2.0 optimized with intel math kernel library for deep neural networks." [Online]. Available: https://www.intel.com/content/www/us/en/artificial-intelligence/posts/apache-mxnet-v1-2-0-optimized-with-intel-math-kernel-library-for-deep-neural-networks-intel-mkl-dnn.html

[17] "Accelerating deep learning on CPU with Intel MKL-DNN." [Online]. Available: https://medium.com/apache-mxnet/accelerating-deep-learning-on-cpu-with-intel-mkl-dnn-a9b294fb0b9

[18] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, and T. S. John, "MLPerf inference benchmark," 2019. [Online]. Available: https://arxiv.org/pdf/1911.02549.pdf

[19] "Opentracing." [Online]. Available: https://opentracing.io/

[20] "Jaeger: open source, end-to-end distributed tracing." [Online]. Available: https://www.jaegertracing.io/

[21] A. Dakkak, C. Li, J. Xiong, and W.-M. Hwu, "MLModelScope: Evaluate and measure ML models within AI pipelines," 11 2018.

[22] C. Li, A. Dakkak, J. Xiong, W. Wei, L. Xu, and W.-m. Hwu, "XSP: Across-stack profiling and analysis of machine learning models on gpus," 2019.

[23] "CUDA profiling tools interface," 2018. [Online]. Available: https://developer.nvidia.com/cuda-profiling-tools-interface

[24] "MLPerf inference." [Online]. Available: https://mlperf.org/inference-overview/overview

[25] P. Mattson, V. J. Reddi, C. Cheng, C. Coleman, G. Diamos, D. Kanter, P. Micikevicius, D. Patterson, G. Schmuelling, H. Tang, and et al., "MLPerf: An industry standard benchmark suite for machine learning performance," *IEEE Micro*, vol. 40, no. 2, p. 8–16, Jan 2020.

[26] "Profiling MXNet models." [Online]. Available: mxnet.apache.org/api/python/docs/tutorials/performance/backend

[27] "Training versus inference," Feb 2019. [Online]. Available: https://blogs.gartner.com/paul-debeasi/2019/02/14/training-versus-inference/

[28] "Py-Spy." [Online]. Available: https://pypi.org/project/py-spy/0.1.0/

[29] "MXNet Gluon model zoo." [Online]. Available: https://gluon-cv.mxnet.io/model_zoo/index.html

[30] "Benchdnn." [Online]. Available: https://github.com/oneapi-src/oneDNN/tree/master/tests/benchdnn

[31] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[32] S. H. Tsang, "Review: MobileNetV1 - depthwise separable convolution (light weight model)," April 2019. [Online]. Available: https://towardsdatascience.com/review-mobilenetv1-depthwise-separable-convolution-light-weight-model-a382df364b69

[33] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. G. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," *CoRR*, vol. abs/1712.05877, 2017. [Online]. Available: http://arxiv.org/abs/1712.05877

[34] P. Zhao, X. Chen, Z. Qin, and J. Y. Ye, "Model quantization for production-level neural network inference," Nov 2019. [Online]. Available: https://medium.com/apache-mxnet/model-quantization-for-production-level-neural-network-inference-f54462ebba05

[35] "Reorder API reference." [Online]. Available: https://oneapi-src.github.io/oneDNN/dev$_g uide_r eorder.html$