Accelerating TensorFlow on Modern Intel Architectures

Elmoustapha Ould-Ahmed-Vall, Mahmoud Abuzaina, Md Faijul Amin, Jayaram Bobba, Roman S Dubtsov, Evarist M Fomenko, Mukesh Gangadhar, Niranjan Hasabnis, Jing Huang, Deepthi Karkada, Young Jin Kim, Srihari Makineni, Dmitri Mishura, Karthik Raman, AG Ramesh, Vivek V Rane, Michael Riera, Dmitry Sergeev, Vamsi Sripathi, Bhavani Subramanian, Lakshay Tokas, Antonio C Valles
Intel Corporation

ABSTRACT

TensorFlow is a popular machine learning and deep learning framework that enables data scientists to address a wide-range problems such as image classification, speech recognition, to name a few. TensorFlow also operates at large scale and in heterogeneous environment and allows users to train neural network models on a variety of devices such as multicore CPUs, general purpose GPUs, and custom ASICs known as TPUs.

Although TensorFlow supports multicore CPUs, our evaluation of default CPU backend in TensorFlow revealed that it offers sub-optimal performance on server-class Intel® Xeon® and Xeon Phi™ platforms. We address this sub-optimal performance issue in this paper by developing optimizations for Intel platforms. Our optimizations offer significant gain, up to the order of 86X, over default CPU backend on a set of commonly used neural network models. All of our optimizations are upstreamed to TensorFlow repository and are available for users.

CCS CONCEPTS

• **Computing methodologies** → **Artificial intelligence**; **Vector / streaming algorithms**; *Machine learning*;

KEYWORDS

Deep learning, Intel architecture, Optimization

1 INTRODUCTION

Machine learning has seen phenomenal growth in last few years and it is being applied to a variety of problems in different fields. This success can be attributed to availability of large datasets and the easy access to powerful computational resources (thanks to Moore's law) for processing these datasets.

AIM'17, September 2017, Portland, OR, USA

TensorFlow is a popular software framework for developing deep learning models that can be trained on large datasets. TensorFlow supports both large-scale training and inference: models can be trained on a large distributed cluster of different devices such as CPUs, GPUs, and TPUs, and inference can be done on as small a device as a mobile phone [11].

Although, TensorFlow can be used to train deep learning models on CPU devices, our evaluation highlighted that default CPU backend (developed using Eigen [4] template library for linear algebra) for TensorFlow does not deliver best performance on Intel® Xeon® and Xeon Phi™ platforms. We addressed this issue by analyzing performance of the default CPU backend in TensorFlow and by developing optimizations to exploit features of Intel® Xeon® and Xeon Phi™ platforms. This paper describes various challenges that we encountered and the solutions that were developed to address them. We evaluated our optimizations on a set of common neural network models. In comparison to existing CPU backend in TensorFlow, our optimizations deliver significant gains up to the order of 70x. All of the optimizations that we developed are already upstreamed to TensorFlow repository [10] and readily available for users to try out.

This paper is organized as follows. Section 2 covers necessary background to understand the proposed optimizations. Section 3 actually details out all the optimizations that are part of TensorFlow code base now. Evaluation results on Intel® Xeon® and Xeon Phi™ platforms are then presented in Section 4. Section 5 briefly mentions future work, and Section 6 concludes the paper.

2 BACKGROUND

TensorFlow. The background provided here is not meant to be a comprehensive description of TensorFlow — one may refer to references such as [5, 10] for detailed description of TensorFlow. Rather this description is supposed to be a short summary of TensorFlow concepts that are necessary to understand the optimizations discussed in the next section.

TensorFlow provides a simple dataflow-based programming abstraction by offering a high-level scripting interface (typically Python) that allows users (typically data scientists) to construct data-flow graphs and experiment with them easily. A neural network model is represented as a directed data-flow graph in which graph nodes represent individual mathematical operators (called *operations*) such as matrix multiplication, convolution, etc, and the edges between the

nodes represent data-flow between the nodes. To be precise, edges carry multi-dimensional arrays called *tensors*.

Execution of a neural network model is mapped to executing operators in the data-flow graph by considering their constraints. Constraints consist of simple data-flow constraints (inputs of an operation need to be available to execute it) and control-flow constraints (a user may explicitly add control-flow constraint between operations to enforce sequential execution.) When an operator is ready for execution, TensorFlow runtime calls the *kernel* for that operator on the device that it has assigned to the operator. Kernel, in simple words, is the boilerplate code for performing the function of the operator. A single operation may have multiple registered kernels with specialized implementations for a particular device or data type. Kernels for all TensorFlow operators for CPU device are implemented using Eigen [4] open-source library for linear algebra.

3 OPTIMIZATIONS

Execution cost of a data-flow graph in TensorFlow boils down to execution cost of individual operations in the graph and the their execution order. Essentially, the optimizations that can be applied to reduce this cost can be logically broken down into two main categories: (1) operation-level optimizations to reduce the execution cost of each and every operation, and (2) graph-level optimizations to reduce the execution cost of the graph of operations. We will discuss about these categories below.

3.1 Operation-level optimizations

Reducing execution cost of an individual operation on Intel architectures demands exploiting underlying architecture features to their full potential. Modern Intel® Xeon® and Xeon Phi™ processors offer AVX2 and AVX512 SIMD instructions respectively among other features. Operator-level optimizations exploit such features to efficiently perform individual operations.

In operation-level optimizations, we map these operations into APIs provided by Intel-optimized Math Kernel Library [8] and Math Kernel Library for Deep Neural Networks [7] (MKL-DNN). These libraries contain several architecture-specific algorithmic optimizations. These optimizations are generic and agnostic of a specific architecture, and hence are applicable to a broad spectrum of multi-core and many-core chips.

Intel MKL-DNN, in particular, contains optimized implementations for a number of important deep learning primitives that are building blocks of different deep learning models. In addition to matrix multiplication and convolution, these building blocks include: direct batched convolution, inner product, pooling (min, max and avg), normalization (local response normalization across channels (LRN), batch normalization), activation (ReLU) and data manipulation primitives (multi-dimensional tranposition (conversion), split, concat, sum and scale). Each of these primitives are optimized using techniques (such as cache blocking, data layout, vectorization, threading, and register blocking) discussed in detail in the reference [6]. So we will not discuss these techniques here, but would point interested reader to the reference.

Ideally, one would like all of TensorFlow operations to be mapped to MKL-DNN APIs, but unfortunately, MKL-DNN optimizes a commonly used subset of the TensorFlow operations. For uncommon operations that are not yet optimized using MKL-DNN, we optimize them directly by leveraging some of the key techniques mentioned above. As an example, we optimized TensorFlow `Slice` operation by exploiting inherent parallelism in the operation.

TensorFlow kernels are implemented inside C++ files that reside in `tensorflow/core/kernels`. Our operations-level optimizations also follow same design and are implemented in their own kernels that are registered for CPU device type. For instance, our Conv2D operator is implemented in `tensorflow/core/kernels/mkl_conv_ops.cc`. Steps on how to build TensorFlow to leverage our optimizations can be found online [9].

3.2 Graph-level optimizations

An important limitation of operation-level optimizations is that they are unaware of the operations that are surrounding (*context* in a conceptual sense) the operation being optimized. Lack of awareness of the context restricts effectiveness of these optimizations to individual operations. Graph-level optimizations precisely address this limitation. Graph-level optimizations analyze the graph of operations and may modify the graph to ensure efficient execution of overall graph.

Layout conversion optimization. In the operation-level optimization section, we briefly mentioned that MKL-DNN implements data layout optimization. Data layout optimization requires input tensors to be laid out in a fashion that enables efficient execution of an operation. For instance, convolution is typically implemented using nested loops where data is accessed in a strided manner in the innermost loop. It is beneficial, in such a case, to lay out data so that the access in the innermost loop is continuous. This aids both a better utilization of cache lines (and hence bandwidth), while improving prefetcher performance. We loosely call such layout *MKL-DNN layout*.

Unfortunately, if MKL-DNN layout is different than the actual layout of the tensor (we call this layout *TensorFlow layout*), then we would need to convert the actual layout into MKL-DNN layout. We would also need to convert MKL-DNN layout on the output side of an operation into TensorFlow layout. To be precise, the example in Figure 1 specifies sample code for handling different layouts for 2D convolution. Lines 7 and 8 are the layout conversion calls for converting TensorFlow layout to MKL-DNN layout for input and filter respectively; line 12 is the layout conversion call for converting MKL-DNN layout back into TensorFlow layout.

Layout conversions are expensive data shuffling operations as they involve reordering of tensor dimensions and elements. A logical question then is: can we reduce these conversions, if not eliminate them completely? Data layout conversion

```
class MklConv2DOp : public OpKernel {
 void Compute(OpKernelContext* context) override {
  const Tensor& tf_input = context->input(0);
  const Tensor& tf_filter = context->input(1);
  Tensor* output = context->allocate_output(..);

  mkl_input = convert_to_mkldnnlayout(tf_input);
  mkl_filter = convert_to_mkldnnlayout(tf_filter);

  mkl_output = mkldnn_conv2d_fwd(mkl_input,
                                 mkl_filter, ..);
  *output = convert_to_tflayout(mkl_output);
 }
}
```

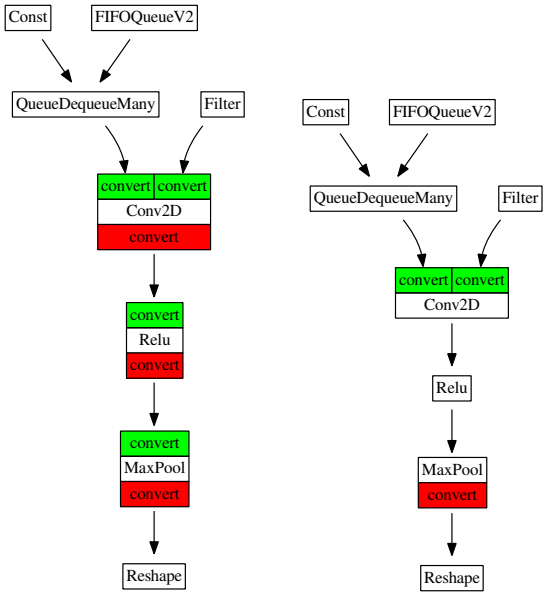
Figure 1: Code demonstrating layout conversions

optimization relies on an important observation to precisely answer this question.

Data layout conversion optimization analyzes the operation graph and eliminates layout conversions in an operation when the operations surrounding it are all MKL-DNN operations. To be precise, it identifies a largest subgraph of operations that are MKL-DNN operations and inserts input and output layout conversions only on all the entries and exits from the subgraph, getting rid of layout conversions on the input and output of each operation in the subgraph. Figure 2 represents the optimization conceptually. Figure 2a represents a typical operation graph with common neural network operators, such as convolution, activation, and pooling. The green boxes in the graph represent layout conversions for the inputs of an operation; the red boxes represent layout conversion for the output of an operation. Notice the optimized graph in Figure 2b, where the conversions are only inserted at the entry and exit of the subgraph of Conv2D, ReLU and Maxpool.

Data layout conversion optimization essentially performs lazy conversions — it relies on propagating tensors in MKL-DNN layout between operations until we need to convert them into TensorFlow layout. Technically, MKL-DNN layout is metadata for a tensor, and thus to propagate MKL-DNN layout, we represent it as a tensor. Thus, every operation that is optimized using MKL-DNN (we call such operations *MKL-DNN operations*) accepts $2 \times i$ inputs and generates $2 \times o$ outputs, where i is the number of inputs and o is the number of outputs of the corresponding TensorFlow operation. The layout conversion optimization is thus implemented in two parts: a graph rewrite pass that modifies the operation graph to enable layout propagation and modifications to operations themselves to accept additional inputs and generate additional outputs. The reason for using the approach of rewriting graph is to maintain transparency to the TensorFlow user — since the TensorFlow graph is rewritten transparently, the user can use same Python APIs provided by TensorFlow to write models for Intel CPUs.

The graph rewrite pass performs two tasks: (1) rewrites MKL-DNN operations to generate additional inputs and outputs, and (2) inserts operations that perform layout conversions as discussed earlier. The algorithm for the rewrite



(a) Graph with input and output conversions **(b) Graph with optimized conversions**

Figure 2: Layout conversion optimization example

pass is specified in Algorithm 1. The algorithm operates on a topologically sorted input graph. In the first loop, it rewrites every MKL-DNN operation to produce additional inputs and outputs. Second loop then processes the rewritten graph to insert *_TfToMkl* and *_MklToTf* layout conversion operations in the graph. If we apply this algorithm to the example graph from Figure 2a, we get the graph in Figure 3. Dotted arrows in the figure represent propagation of MKL-DNN layouts.

The graph rewrite algorithm for this optimization is implemented in `tensorflow/core/graph/mkl_layout_pass.cc` and `tensorflow/core/graph/mkl_tfconversion_pass.cc`. Layout conversion operator resides in `tensorflow/core/kernels/mkl_tfconv_op.cc`.

Fusion. Neural network models typically employ certain combination of operators. For instance, it is common in deep learning neural networks to add bias to the output of convolution. Fusion optimization is designed to detect such commonly used patterns and fuse the subgraphs matching those patterns into compact graph nodes. Currently, our fusion pass looks for convolution + bias pattern, and if found, merges them into a new graph node called *_MklConv2DWithBias*. Fusion optimization may enable getting rid of unnecessary data copy operations (output of convolution may need to be copied to the input of add operation); it also offers more opportunities for operation-level optimization as the fused operation has same visibility as that of the fused subgraph. Our fusion optimization is implemented as a part of generic graph rewrite pass and resides in `tensorflow/core/graph/mkl_layout_pass.cc`.

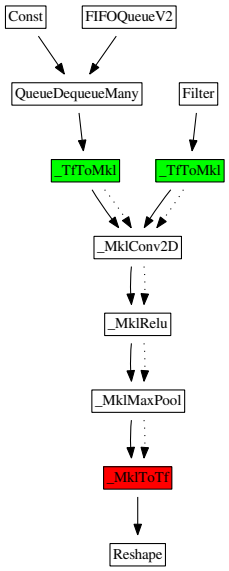


Figure 3: Output graph from graph rewrite pass

Memory allocation. Memory allocation optimization develops an efficient solution to handle frequent allocations and deallocations of large memory buffers. In deep learning training and inference cases, these large buffers are typically of the order of few hundred MBs. We observed that the default CPU allocator in TensorFlow was leading to frequent `mmap` and `munmap` calls and costly page clear operations for large allocations and deallocations. We addressed this problem by developing a custom pool allocator using existing pool allocator in TensorFlow. Our custom pool allocator ensures that deallocated buffers are pooled rather than releasing to OS immediately. Our allocator also exploits an interesting characteristic of deep learning training. In training case, multiple invocations of same operator across different minibatches mostly lead to buffer allocations of same size (since the input shape is same.) This characteristic aids the pool allocator in reusing large buffers and reduces fragmentation. Our CPU allocator is implemented in `tensorflow/core/common_runtime/mkl_cpu_allocator.h`.

Load balancing. The operator graph in TensorFlow, being a data-flow graph, offers excellent opportunities for exploiting parallelism among operators. For instance, for the graph in Figure 3 both *_TfToMkl* operators can be executed in parallel since they are not data- or control-dependent on each other. In addition to the parallelism between operators, the kernels themselves exploit underlying hardware parallelism using threads. For instance, *_MklConv2D* uses OpenMP threads to achieve work sharing and efficient execution. TensorFlow threading model thus has two parameters: `inter_op_parallelism_threads` specify the maximum number of operators that can be executed in parallel, and `intra_op_parallelism_threads` specify the maximum number of threads to use for executing individual operators. Both

Data: G: Input operation graph
Result: G'': output graph with optimized layout conversions
$G_t \leftarrow topological\_sort(G)$;
$G' \leftarrow []$
`/* First task of graph rewrite pass. */`
for $\forall$ *operation* O *in* G_t **do**
  if $is\_mkldnn\_op(O)$ **then**
    $O'_{inputs} \leftarrow []$;
    for *every input* I *of* O **do**
      $O'_{inputs} \leftarrow O'_{inputs} \cup I$;
      `/* ` I_{mkl} `is extra input that carries MKL-DNN layout. */`
      $O'_{inputs} \leftarrow O'_{inputs} \cup I\_mkl$
    end
    $O' \leftarrow O'_{inputs}$;
    $G' \leftarrow G' \cup O'$;
    delete O;
  else
    $G' \leftarrow G' \cup O$;
  end
end
`/* Second task of graph rewrite pass. */`
$G'' \leftarrow []$;
for $\forall$ *edge* E *in* G' **do**
  if $\neg is\_mkldnn\_op(E_{src}) \wedge is\_mkldnn\_op(E_{dst})$ **then**
    `/* ` $\_TfToMkl$ `is TensorFlow layout to MKL-DNN layout conversion op. */`
    $T \leftarrow \_TfToMkl$;
    $T_{inputs} \leftarrow E_{src}$;
    $T_{outputs} \leftarrow E_{dst}$;
    $G'' \leftarrow G'' \cup T_{inputs}$;
    $G'' \leftarrow G'' \cup T_{outputs}$;
    delete E ;
  else if $is\_mkldnn\_op(E_{src}) \wedge \neg is\_mkldnn\_op(E_{dst})$ **then**
    `/* ` $\_MklToTf$ `is MKL-DNN layout to TensorFlow layout conversion op. */`
    $T \leftarrow \_MklToTf$;
    $T_{inputs} \leftarrow E_{src}$;
    $T_{outputs} \leftarrow E_{dst}$;
    $G'' \leftarrow G'' \cup T_{inputs}$;
    $G'' \leftarrow G'' \cup T_{outputs}$;
    delete E ;
  else
    $G'' \leftarrow G'' \cup E$;
end

Algorithm 1: Algorithm for graph rewrite pass for layout conversion optimization

the parameters together have multiplying effect on the total number of threads. Incorrect setting for these parameters can easily lead to over-subscription or under-subscription of the underlying hardware. We addressed this issue by carefully tuning commonly used deep learning models to various Intel ® Xeon ® and Xeon Phi ™ CPUs. We will discuss settings of these parameters in Evaluation section.

4 EVALUATION

We evaluated effect of our optimizations on Intel® Xeon® E5-2699 V4 processor [1] and Intel® Xeon Phi™ 7250 [2] processor running Red Hat Enterprise Linux Server (release 7.2) and 3.10.0-327.el7.x86_64 Linux kernel. TensorFlow-1.2 source code was compiled with GCC-6.3 and default bazel options[1] and the generated wheel was used as the baseline. We then applied our optimizations on the same source code, and used same compiler with `--config=mkl --copt=''-DEIGEN_USE_MKL_VML'' -c opt` bazel options to generate the optimized wheel. As mentioned earlier, steps on how to build TensorFlow to leverage our optimizations can be found online [9]. For our preliminary evaluation, we used three Convnet [3] benchmarks as these benchmarks are popular among deep learning community.

Hardware description. Intel® Xeon® E5-2699 V4 processors, code-named Broadwell, support the second generation of Intel Advanced Vector Extension (AVX2) instruction set architecture, which includes the 256-bit Fused Multiply-Add (FMA) instructions. Each processor core have two FMA pipelines which can deliver a maximum throughput of 32 single precision (32-bit) and 16 double precision (64-bit) floating point operations per clock cycle. The Intel® Xeon® E5-2699 V4 processors used in our evaluation studies has a total of 44 physical cores in a two socket configuration, with each socket containing 22 cores and provide a peak single precision floating-point throughput of about 3 Teraflops.

Intel® Xeon Phi™ processors code-named Knights Landing are the latest generation of Intel® Many Integrated Core (Intel® MIC) family of architecture. Intel® Xeon Phi™ 7250 processor [2] used in our evaluation studies has a total of 68 compute cores on a single die, with each core containing two 512-bit Vector Processing Units (VPU). The entire chip can deliver a peak throughput of about 6 Teraflops for single precision and 3 Teraflops for double precision floating-point computations. In addition to the high compute capabilities, Knights Landing also provides a 16 GB on package high bandwidth memory in the form of Multi-Channel DRAM (MCDRAM) which delivers significantly higher bandwidth than DDR4 memory. Depending on the memory foot-print of the running applications, MCDRAM can either be configured as a last-level cache for hot data structures (known as Cache mode) or a separate allocatable memory pool (known as Flat mode) or a combination of both (Hybrid mode).

Result. Figure 4 and Figure 5 contain raw performance numbers (in terms of images per second) obtained from Convnet benchmarks for Alexnet, VGG, and GoogLeNet-v1 models. We also obtained performance numbers for different batch sizes (since the benchmark scripts accept batch size as an input argument.) X-axis in all the graphs represent different batch sizes, while Y-axis represents performance score (images/sec) reported by the benchmark.

[1] `bazel build -c opt //tensorflow/tools/pip_package:build_pip_package` command was used for compilation.

Overall, our optimizations deliver significant performance improvement over baseline. The best result obtained on Intel® Xeon® processor was 20X improvement in Alexnet model for inference, while on Intel® Xeon Phi™ we obtained 86X improvement in the same topology. Overall, our optimizations delivered improvements from 5X up to 86X.

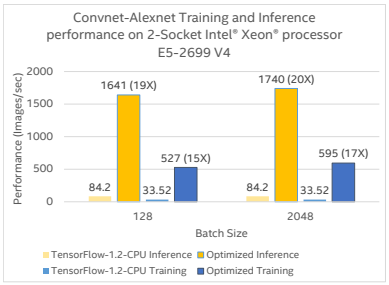
Recommendations for best performance. Our optimization exercise revealed several important parameters in TensorFlow that contribute to its performance on a given model. We briefly discuss these parameters and also provide guidelines on setting them for best performance on Intel architecture.

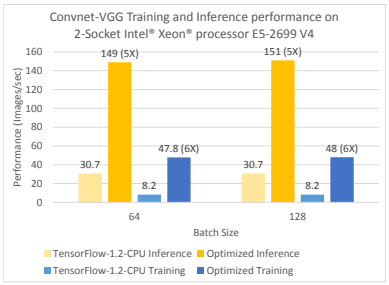
- **Inter-op-** and **intra-op-parallelism-threads.** We suggest that TensorFlow users experiment with the intra-op-parallelism-threads and inter-op-parallelism-threads parameters for optimal setting for each model and CPU platform. Incorrect setting of these parameters lead to over- and/or under-subscription on Intel architecture.
- **Batch size.** Batch size is another important parameter that impacts both the available parallelism to utilize all the cores as well as working set size and memory performance in general.
- **OMP_NUM_THREADS.** Since MKL-DNN library uses OpenMP library for threading, specifying appropriate value of OMP_NUM_THREADS is critical. Maximum performance requires using all the available cores efficiently. This setting is especially important for performance on Intel® Xeon Phi™ processors since it controls the level of hyperthreading (1 to 4).
- **KMP_BLOCKTIME.** Since TensorFlow threading model creates a large number of threads, wait time (in milliseconds) for threads may be important factor on certain models and CPU platforms. KMP_BLOCKTIME is an OpenMP environment variable that allows users to specify thread wait time in milliseconds.

The specific settings of these parameters we used for the evaluation are listed in Figure 6 and Figure 7.

5 FUTURE WORK

Although, we have upstreamed all of our optimizations, it does not represent the end of our optimization efforts by any means. Newer deep learning models may demand further optimizations. Moreover, Intel® Xeon® processor E5 v4 (codename Broadwell) and Intel® Xeon Phi™ processor 7250 (codename Knights Landing) based platforms lay the foundation for next generation Intel products, such as Intel® Xeon® (codename Skylake) and Intel® Xeon Phi™ (codename Knights Mill). Although, MKL-DNN library already supports Skylake processors, we may need some additional work to support these platforms in TensorFlow CPU backend. In addition, multinode setup of TensorFlow models on Intel architecture will demand further optimizations and tunings.





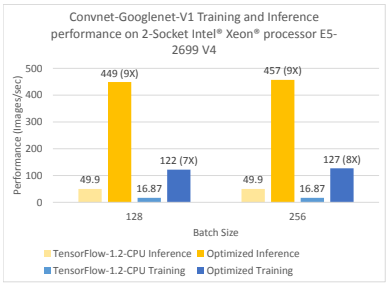
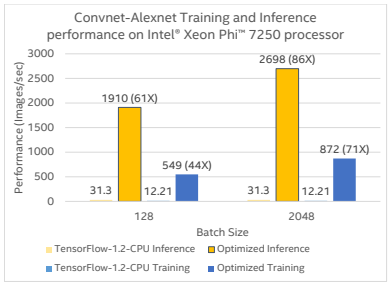
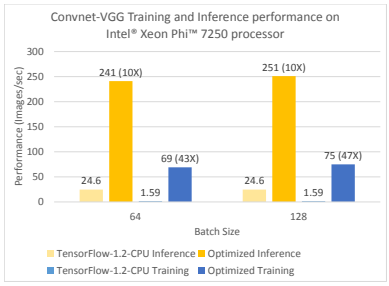


Figure 4: Optimized TensorFlow performance on Intel® Xeon® processor





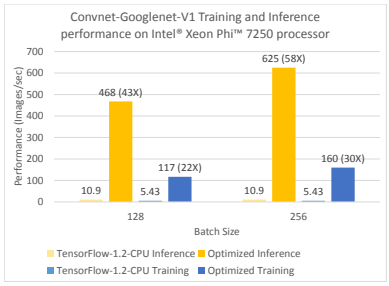


Figure 5: Optimized TensorFlow performance on Intel® Xeon Phi™ processor

Model	Data Format	inter_op	intra_op	KMP_BLOCK TIME
Alexnet	NCHW	1	44	30
GoogLeNet-V1	NCHW	2	44	1
VGG	NCHW	1	44	1

Figure 6: Settings for various parameters used during evaluation on Intel® Xeon® processor

Model	Data Format	inter_op	intra_op	KMP_BLOCK TIME	OMP NUM_THREADS
Alexnet	NCHW	1	136	30	136
GoogLeNet-V1	NCHW	2	68	inf	68
VGG	NCHW	1	136	1	136

Figure 7: Settings for various parameters used during evaluation on Intel® Xeon Phi™ processor

6 CONCLUSION

In this paper, we presented our preliminary experimental results obtained while optimizing TensorFlow deep learning framework on Intel® Xeon® and Intel® Xeon Phi™ processors. Our evaluation results demonstrate effect of our optimizations on default CPU backend in TensorFlow and deliver between 6X and 86X improvement. All of our optimizations are part of TensorFlow open-source code now. This report does not mark the end of our optimization exercise; we will be working on optimizing TensorFlow (including for future Intel architectures) and will be upstreaming our optimizations along the way.

REFERENCES

[1] 2016. *Intel(R) Xeon(R) Processor E5-2699 v4*. http://ark.intel.com/products/91317/Intel-Xeon-Processor-E5-2699-v4-55M-Cache-2_20-GHz.

[2] 2016. *Intel(R) Xeon(R) Processor E5-2699 v4*. http://ieeexplore.ieee.org/document/7477467/.

[3] 2017. *Easy benchmarking of all publicly accessible implementations of convnets*. https://github.com/soumith/convnet-benchmarks.

[4] 2017. *Eigen*. http://eigen.tuxfamily.org.

[5] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). http://tensorflow.org/ Software available from tensorflow.org.

[6] Intel Corporation. 2015. *Myth Busted: General Purpose CPUs Can't Tackle Deep Neural Network Training*. https://itpeernetwork.intel.com/.

[7] Intel Corporation. 2016. *Intel(R) Math Kernel Library for Deep Neural Network*. https://github.com/01org/mkl-dnn/releases.

[8] Intel Corporation. 2017. *Intel(R) Math Kernel Library*. https://software.intel.com/en-us/mkl.

[9] Intel Corporation. 2017. *TensorFlow Optimizations on Modern Intel(R) Architecture*. https://software.intel.com/en-us/articles/tensorflow-optimizations-on-modern-intel-architecture.

[10] Google Inc. 2016. *Computation using data flow graphs for scalable machine learning*. https://github.com/tensorflow/tensorflow.

[11] Google Inc. 2016. *TensorFlow Mobile*. https://www.tensorflow.org/mobile.