



Training Deep Convolutional Neural Networks with Horovod* on Intel® High Performance Computing Architecture

Authors:

Mattson Thieme (Intel), **G Anthony Reina** (Intel), **Jim Miller** (GE), **Deepthi Karkada** (Intel), **Mirabela Rusu** (GE), **Prashant Shah** (Intel)

Table of Contents

Introduction	1
Data and Topology	1
BraTS Dataset	1
Benchmarking Metric	2
U-Net Architecture	3
Hardware	4
Single-Node Hardware	4
Multi-Node Hardware	5
Intel Omni-Path Architecture ..	5
Software Tools	5
TensorFlow + Horovod	5
Optimizations	6
Intel-Optimized TensorFlow ...	6
Transposed Convolution	7
Data Storage	8
Results	8
Discussion	9
Summary	10
Single and Multi-Node:	10
Multi-node only:	10
Acknowledgements	10
References	10

Introduction

This paper reviews a biomedical image segmentation project conducted in partnership with the AI team at General Electric's Global Research Center. We begin by detailing the network topology (U-Net) and the Brain Tumor Segmentation (BraTS) dataset¹ used to benchmark training performance. All training is performed on Intel Xeon® Platinum 8168 servers, and we outline both single and multi-node implementations. Leveraging Intel's® Math Kernel Library for Deep Neural Networks (MKL-DNN), we demonstrate a greater than 7X speedup in time-to-train on a single node and another 2X speedup in a multi-node environment. We conclude with a summary of best-known-methods for optimizing Convolutional Neural Network (CNN) topologies on Intel architecture.

Code and instructions for running this implementation of U-Net can be found at https://github.com/NervanaSystems/topologies/tree/master/distributed_unet/Horovod.

Data and Topology

BraTS Dataset

The BraTS dataset contains multi-contrast magnetic resonance (MR) scans from patients undergoing treatment for gliomal brain tumors (Menze, 2015). For this study, the 2016 collection from the BraTS dataset was divided into 31,000 2D image slices.² The BraTS image annotations contain four channels corresponding to various regions of the tumorous components: edema, non-enhancing solid core, necrotic/cystic core, and the enhancing core (Figure 1). The objective of this study was to identify the entire tumor volume, defined as the union of all four components (Figure 1). Each image is paired with a single ground truth mask (Figure 2, left). Of the 31,000 grayscale images, 24,800 were randomly assigned for training the model, while the remaining 6,200 were used for testing. As only ~30% of pixels in the original 240 x 240 pixel images contain brain matter, the images were cropped to 128 x 128 pixels about the center.

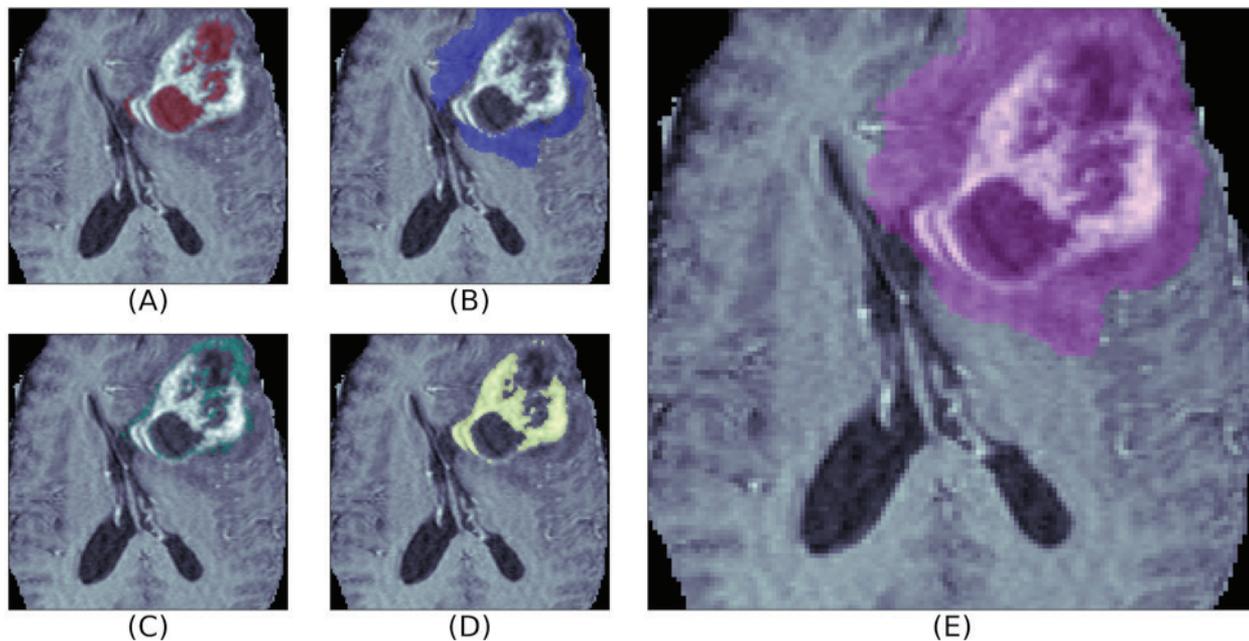


Figure 1. Glioma sub-regions. The original dataset contained 4 different segmentation masks (A-D): necrosis, edema, enhancing core, non-enhancing core. For this benchmark, the model was trained to predict the total mask (E).

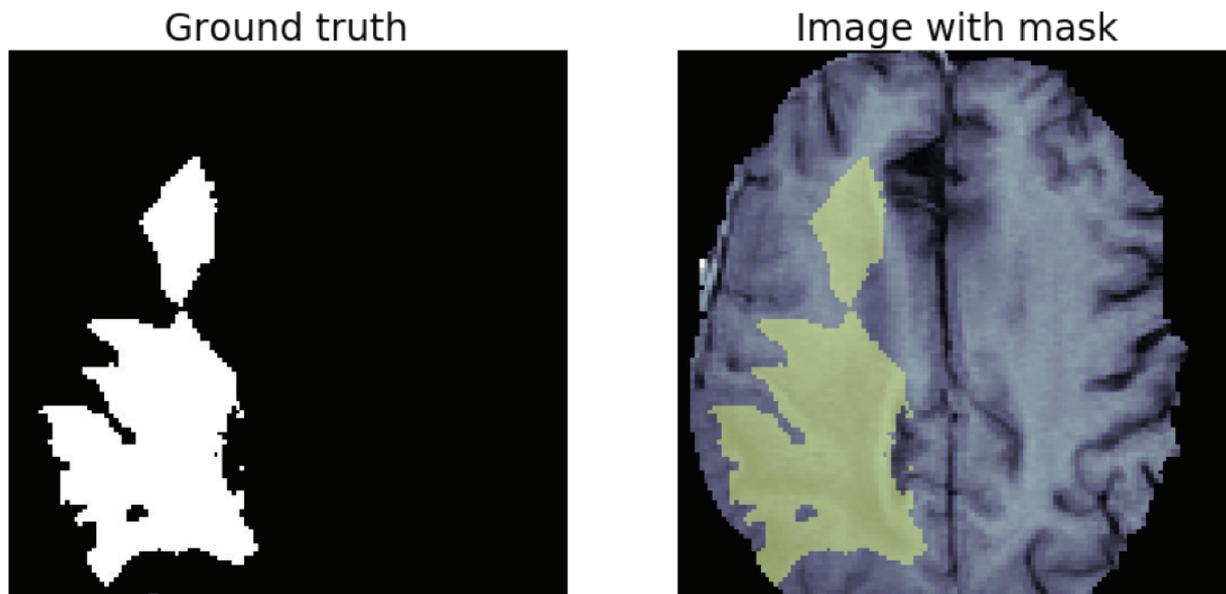


Figure 2. Test set example: ground truth mask (L) and MR image with ground truth overlay (R)

Benchmarking Metric

The standard accuracy metric on the BraTS dataset is the Dice coefficient: a similarity measure in the range [0,1] which reflects the intersection over union (IOU) of the predicted and ground truth masks. It is defined as follows

$$\text{Dice coefficient} = \frac{2 \cdot |P \cap T|}{|P| + |T|} \quad \text{where } P \text{ and } T \text{ are the prediction and ground truth masks, respectively}$$

Menze (2015) reported that the Rater-versus-Rater Dice coefficient for expert neuro-radiologists was 0.85 ± 0.08 (mean \pm std for the whole segmentation mask). Zhao reported that state-of-the-art models for this dataset have Dice coefficients of greater than or equal to 0.85 (2018). Optimizations outlined in the following sections enabled the model to match dice coefficients from current state-of-the-art segmentation models in both the single and multi-node cases.

U-Net Architecture

The U-Net topology used in this study is a deep CNN designed for semantic segmentation (Figure 3) based on Ronneberger's original design (2015). Since its introduction in 2015, U-Net has quickly become the de facto deep learning topology for biomedical image segmentation and has been instrumental in creating prediction models across a wide gamut of imaging modalities.

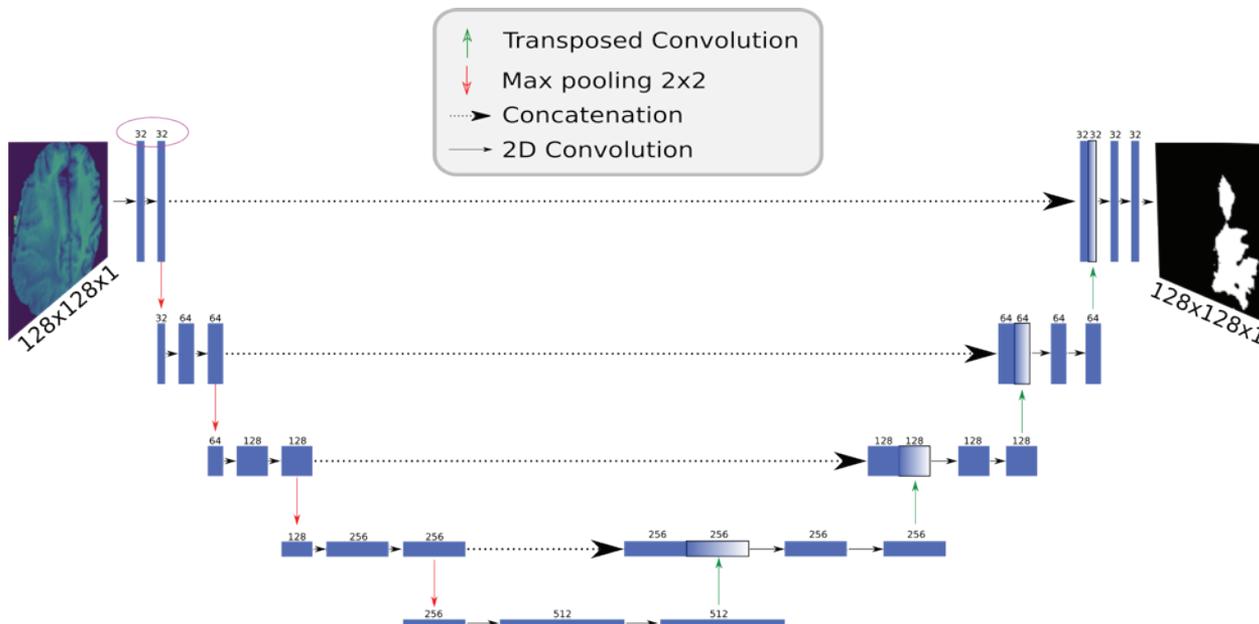


Figure 3. U-Net network diagram (Ronneberger, 2015). Numbers above each layer indicate the number of channels in that layer. Note that the channel count in the first layer (purple ellipse) differs from the original topology by a factor of 2.

The model takes as input a single channel, 2D image and outputs an equivalently-sized, binary mask wherein each pixel is assigned a class label. The network architecture is an encoder-decoder, with a contracting path that captures context (via max pooling) and an expanding path which enables localization (via upsampling). Unlike the standard encoder-decoder topology, feature maps in the expanding path are concatenated with the corresponding feature map from the contracting path. Augmenting downstream feature maps with spatial information acquired using smaller receptive fields allows the network to consider features at various spatial scales.

For both single and multi-node training, the Adam optimizer (learning rate 0.0005) was used to minimize the negative log of the Dice coefficient. To increase numerical stability, a Laplace smoothing of 1 was added and the final loss function algebraically rearranged to replace the division with a log subtraction:

$$\text{loss} = \log(|P| + |T| + 1) - \log(2 \cdot |P \cap T| + 1)$$

Hardware

Single-Node Hardware

Single node computation was performed on one Intel® Xeon® Scalable server using the Keras 2.2.0 API with a TensorFlow 1.8 backend. The CPU was an Intel Xeon 8168 @ 2.70GHz having 48 physical cores with hyperthreading and SpeedStep™ enabled, and the OS was CentOS 7 (centos-release-7-3.1611.el7.centos.x86_64):

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                96
On-line CPU(s) list:   0-95
Thread(s) per core:    2
Core(s) per socket:    24
Socket(s):              2
NUMA node(s):          2
Vendor ID:              GenuineIntel
CPU family:             6
Model:                  85
Model name:             Intel(R) Xeon(R) Platinum 8168 CPU @ 2.70GHz
Stepping:               4
CPU MHz:                1691.613
CPU max MHz:           3700.0000
CPU min MHz:           1200.0000
BogoMIPS:               5400.00
Virtualization:         VT-x
L1d cache:              32K
L1i cache:              32K
L2 cache:               1024K
L3 cache:               33792K
NUMA node0 CPU(s):     0-23,48-71
NUMA node1 CPU(s):     24-47,72-95
```

The environment settings were as follows:

TF_CPP_MIN_LOG_LEVEL	2
Hyperthreading	Enabled
SpeedStep™	Enabled
KMP_BLOCKTIME	0
KMP_AFFINITY	granularity=thread,compact,1,0
OMP_NUM_THREADS	48
intra_op_threads	48
inter_op_threads	2
batch size	128

Detailed explanations of these settings can be found on the Intel® website^{3,4}.

Multi-Node Hardware

Multi-node training was performed on a 3-node Intel® Xeon® Scalable cluster using the Horovod distributed training framework and the Keras 2.2.0 API with a TensorFlow 1.8 backend. Each of the machines were identical to the single-node Intel® Xeon® Scalable detailed above, having 48 physical cores with hyperthreading and SpeedStep™ enabled. The OS was CentOS 7 (centos-release-7-4.1708.el7.centos.x86_64) and the environment settings were as follows:

TF_CPP_MIN_LOG_LEVEL	2
KMP_BLOCKTIME	0
Hyperthreading	Enabled
SpeedStep™	Enabled
KMP_AFFINITY	granularity=thread,compact,1,0
OMP_NUM_THREADS	48
intra_op_threads	48
inter_op_threads	2
batch size	128 per node

Intel Omni-Path Architecture

In the multi-node implementation, workers communicated over the Intel® Omni-Path Architecture (OPA) fabric. The host fabric adapter was an Intel® Omni-Path HFI Silicon 100 Series PCIe x16 (rev. 11). To exploit the increased bandwidth of OPA, the IP addresses used to define the cluster were changed from the standard IPv4/6 address to the OPA specific address. This typically involves a change in the second octet of the IP address (e.g. 11.22.33.44 to 11.100.33.44). Running opainfo tests the OPA fabric for functionality and speed.

Software Tools

TensorFlow + Horovod

Horovod is a distributed training framework developed by Uber® for TensorFlow, Keras, and PyTorch. It uses the message passing interface (MPI) to orchestrate single/multi-worker training in single/multi-node environments. Horovod uses an all-reduce architecture wherein one worker passes parameter updates to a neighboring worker in a ring topology. This is in contrast to the standard TensorFlow hierarchical architecture in which workers pass parameter updates to centralized parameter servers. Uber® has demonstrated that Horovod scales more efficiently than the TensorFlow parameter server architecture in distributed training scenarios (Goyal et al. 2018).

To run a Horovod process, all nodes must have the same libraries and scripts installed within the same absolute directory paths. We installed openMPI 3.0.0, Horovod 0.13.5, TensorFlow 1.8 with Intel® MKL-DNN, and Keras 2.2.0 onto all three Intel® Xeon® Scalable nodes.² We then cloned our GitHub repository⁵ into the same absolute directory path on all three nodes. On one of the nodes, we initiated distributed Horovod training by running the command below.

```
HOROVOD_FUSION_THRESHOLD=134217728 \           # Buffer size for Horovod all-reduce
mpirun \                                     # Command to run the job
-x HOROVOD_FUSION_THRESHOLD \                 # Use Horovod fusion buffer
-np <num_processes> \                         # Total number of workers
--hostfile <node_ips> \                       # List of IP addresses for nodes
--map-by socket \                             # Map workers by socket
-cpus-per-proc 24 \                           # Set to num cores per socket
--report-bindings \                           # Log core/node bindings for workers
--oversubscribe \                             # Allow more than 1 worker per core
bash exec_multiworker.sh <args>             # Job to run (plus args for job)
```

The `-np` flag specifies the total number of processes (workers) to initiate across all nodes. Horovod allows multiple workers to be placed on a single node. The batch size within each process remains the defined batch size-- Horovod does not automatically scale the batch size with the number of workers nor does it shard the dataset across the workers. The `--hostfile` flag points to a text file with a single IP address per line defining the nodes in the cluster. The `--map-by` flag designates what is considered a single resource - in this case, we used two-socket servers and declare each `socket` as a single resource. The `-cpus-per-proc` flag specifies how many cores each process on each resource should be allotted. The `--report-bindings` flag is optional and simply reports how the processes were bound with OpenMPI. It conveniently confirmed that the cores were bound correctly for a given socket and node. The `--oversubscribe` flag allows for mapping multiple workers per resource (if desired). `HOROVOD_FUSION_THRESHOLD` defines the buffer size (in bytes) for the all-reduce operations. If the all-reduce operations exceed this threshold, then Horovod performs the all-reduce immediately rather than waiting for the end of the epoch. This helps balance the communications versus operations speed tradeoff and is a hyperparameter that can be tuned based on the number of workers and number of operations in the topology (although the default setting is sufficient for most workloads). Finally, the MPI command calls the script to run within each process. In this case, a bash script was run which activated the virtual environment, set relevant environment variables, and then initiated the main python training script.

Within the main python script, a few minor code changes are required to distribute a model using Horovod:

1. Add the following import statements and initialize Horovod:

```
import tensorflow as tf
import keras as K
import horovod.keras as hvd
hvd.init()
```

2. Wrap the existing optimizer with Horovods' Distributed Optimizer:

```
opt = K.optimizers.Adam()
optimizer = hvd.DistributedOptimizer(opt)
```

3. Add callbacks for variable broadcasting and gradient aggregation:

```
callbacks = [ hvd.callbacks.BroadcastGlobalVariablesCallback(0),
              hvd.callbacks.MetricAverageCallback(),
              hvd.callbacks.LearningRateWarmupCallback( \
                warmup_epochs=args.num_warmups, verbose=1) ]
```

The `BroadcastGlobalVariablesCallback` is used to coordinate weight initializing across all workers at the start of training and whenever training resumes from a checkpoint. The `MetricAverageCallback` averages the gradients at the end of each epoch so that all workers begin a new epoch with the same weights (i.e. synchronous updates). The `LearningRateWarmup` callback is optional. It will gradually increase the learning rate for `num_warmups` epochs, at which point the Adam optimizer will take over and begin learning rate decay. Warmup can help convergence when using multiple workers because, as a natural consequence of using multiple workers, there is a proportional decrease in the number of weight updates per epoch (Goyal et al. 2018). Confer with the Uber Horovod website⁶ for more a detailed description of installation and configuration details.

Optimizations

Intel-Optimized TensorFlow

Intel optimized TensorFlow 1.8 contains Intel Math Kernel Library - Deep Neural Network (MKL-DNN) operations which are accessible without any changes to existing TensorFlow code.¹ MKL-DNN leverages the AVX-2 and AVX-512 SIMD vector instructions when creating primitives for the convolution, pooling, normalization, and inner product operations common to CNN topologies. Collaborating with Google®, Intel has integrated MKL-DNN into the TensorFlow main branch (beginning with r1.5), making MKL-DNN optimizations accessible via the standard conda install command (`conda install -c anaconda tensorflow`) or through building TensorFlow from source.

Within TensorFlow, `intra_op_parallelism` (intra-op) and `inter_op_parallelism` (inter-op) threads are critical hyperparameters for maximizing core utilization and achieving high throughput on Intel CPUs. These can be passed to TensorFlow session via `tf.config`. Intra-op threads are useful in mathematical operations which can be executed in parallel. For example, matrix multiplication (`tf.matmul`) and element wise summation (`tf.reduce_sum`) are operations that can be broken into smaller, parallel operations on separate CPU cores. Inter-op threads are sets of parallel operations which can run concurrently with little interdependence. For example, long or independent graph branches, data loading operations, and metric summary operations may benefit from increased inter-op threads. Intel recommends setting intra-op threads to the number of physical cores on the machine, and inter-op threads to the number of disjoint branches within the computation graph (usually in the single digits). However, both should be optimized on a case by case basis.

Transposed Convolution

The expanding path in the U-Net topology (Figure 3) can be implemented using any number of upsampling methods. Ultimately, upsampling transforms a low resolution image into a higher resolution image. In the original topology, a nearest neighbors algorithm (`UpSample2D`) was used to perform the upsampling operations. An alternative strategy, using [transposed convolution](#) (Zeiler and Fergus, 2013; Dumoulin, 2018) in place of the nearest neighbor approach, was explored in this project. See Figure 4 for a visual representation of both upsampling strategies.

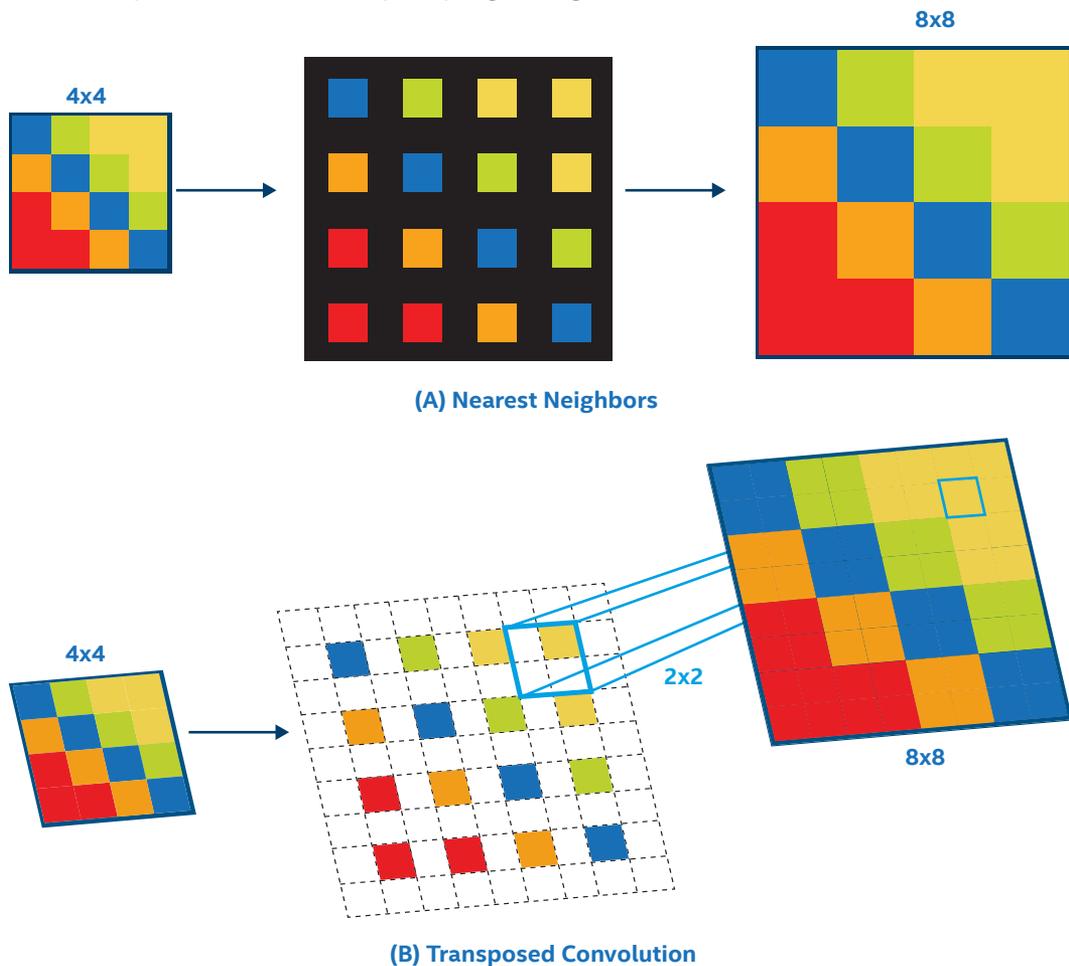


Figure 4: “Upsampling” with two different algorithms. (A) *Nearest Neighbors Algorithm*. Black areas are filled with copies of the center pixel. This produces a “pixelated” effect. More complex algorithms (e.g. bilinear interpolation and Lanczos resampling) use weighted combinations of the surrounding pixels to fill the black areas and provide a smoother transformation. (B) *Transposed convolution*. There are multiple ways to use transposed convolution to upsample an image. A 2x2 transposed convolutional filter with a 2x2 stride was chosen to achieve the effect of doubling the pixel dimensions. This is also referred to as a fractionally strided convolution (cf. Dumoulin, 2018 for more details on transposed convolution). The essential difference between transposed convolution and traditional upsampling are the convolutional kernels in the transposed convolution operation; these kernels contain learnable parameters which allow the operation to capture non-linear mappings.

Unlike transposed convolution operations, `UpSample2D` operations have no learnable parameters, which may lead one to believe that `UpSample2D` would run faster than transposed convolution. However, our previous work has shown that optimized primitives in Intel’s MKL-DNN enable the transposed convolution operation to run up to 17x faster than `UpSample2D` on tensors of equivalent dimension (Thieme 2018).

Data Storage

To avoid bottlenecks in network traffic while performing multi-node computation, copies of the entire dataset are stored locally on all workers. A Python generator is used to feed random batches into the training framework. All training examples are randomly shuffled after each epoch.

Results

Models trained in single and multi-node environments both achieved state of the art segmentation accuracy on the test set (Test Dice ≥ 0.85). Figure 5 shows an example of predicted vs. ground truth masks. Good agreement is observed between models built with the UpSampling2D vs. transposed convolution operations in the expanding path. A histogram of pairwise differences between UpSampling2D and transposed convolution shows no statistical difference between the trained models.

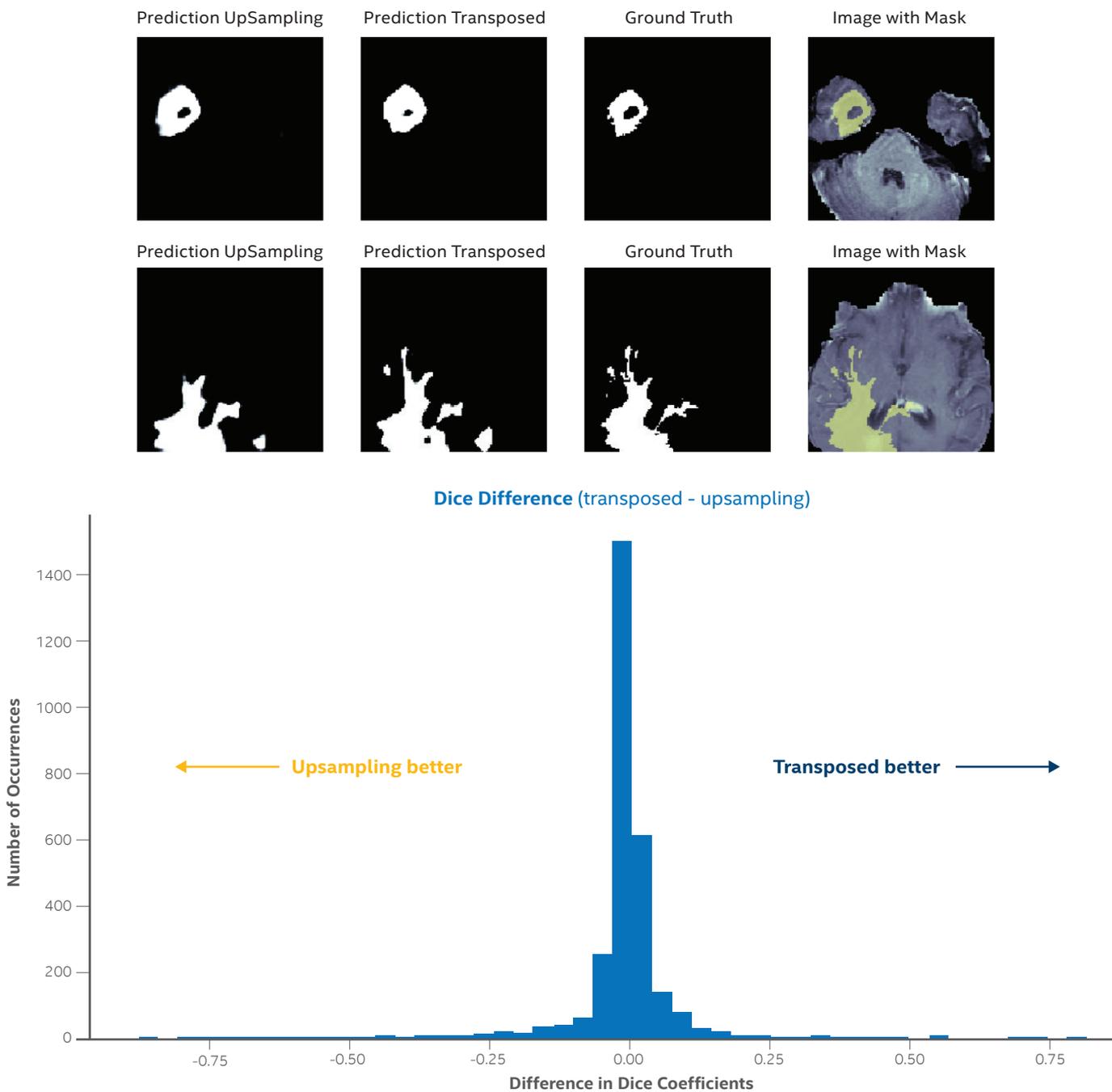


Figure 5. (TOP) Both nearest neighbor and transposed convolution methods can produce similar increases in image size (“upsampling”). Predicted tumor segmentation masks from U-Net models trained with UpSample2D were compared with transposed convolution (2x2 filter with 2x2 stride). (BOTTOM) A histogram of the pairwise difference between Dice scores of the two models on the test dataset. A Kolmogorov-Smirnoff two sample test shows no significant difference between the two distributions.

On average, Intel MKL-DNN optimizations reduced the training time by greater than 7X on a single node. Approximately 25% of this gain in speed is attributable to switching from the UpSampling2D to the transposed convolution operation in the topology. An additional 2X improvement in average time to train was gained using Horovod to distribute training across 12 workers on three Intel® Xeon® Scalable nodes (Figure 6).

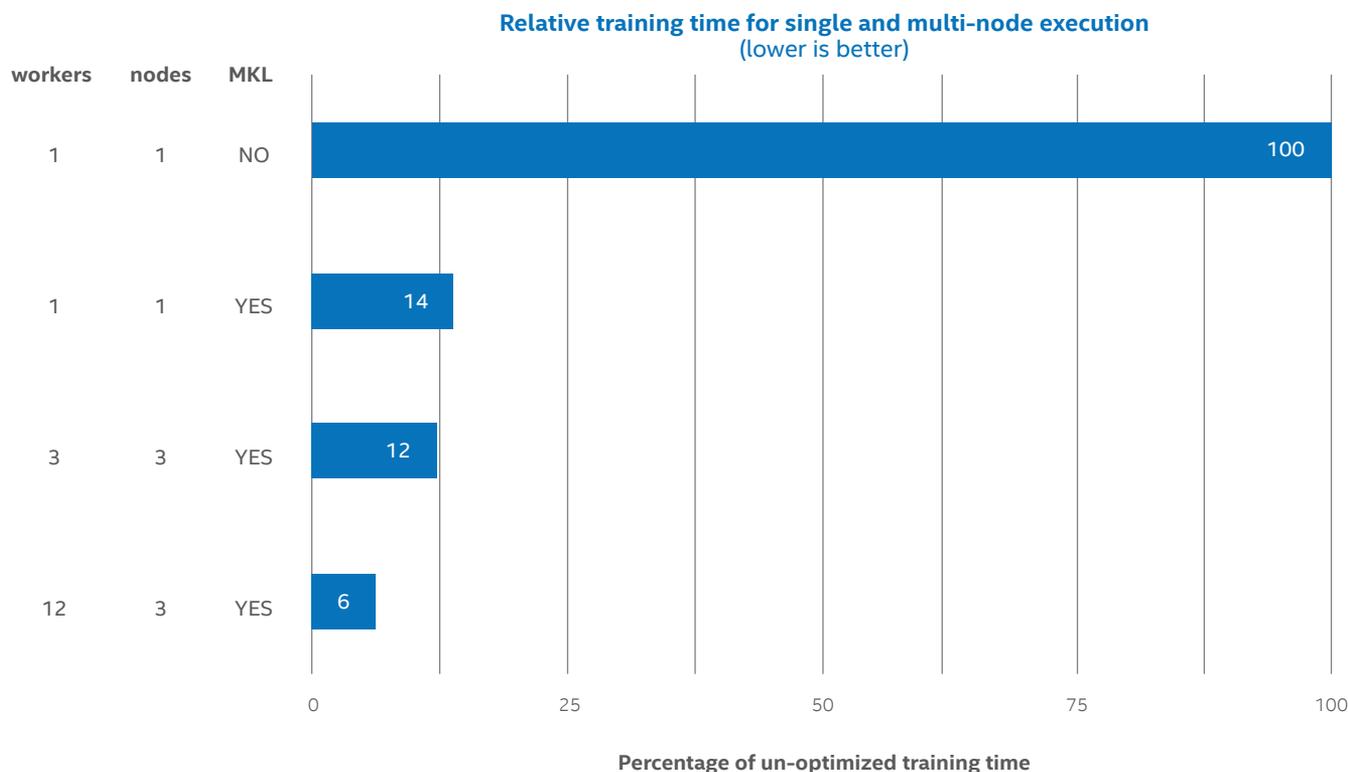


Figure 6. Average training times (as percentage of un-optimized training times) for various configurations of single and multi-node execution to reach a test Dice of > 0.85. “Workers” indicates the total number of workers across all nodes.

Discussion

Leveraging optimizations in Intel MKL-DNN enables substantial speedups when training deep learning models on CPUs and requires no code changes to existing train scripts. These speedups may be realized in both single- and multi-node environments.

We stress that, due to the inherently stochastic nature of gradient descent, there is a variance in time to train to state of the art in each configuration. Hyperparameters such as learning rate, optimizer choice, sharding, and simply the random initialization of model weights affected convergence curves. As an example— in the worst-case scenario— if the weights in a single node trial were randomly initialized more optimally than the weights in a multi-node trial, then the single node model will effectively be given a head start. In short, although we report the mean time to train to state of the art, the high variance over the 10 repeated trials means that, on any individual run, single node training may converge faster to state of the art than multi-node (Figure 6). Nevertheless, we saw substantial increases in average training speed in distributed settings, and believe that the full potential of distributed training may be better observed with larger datasets and more complex models than the one outlined here.

Finally, we experimented with sharding the dataset evenly across the workers. For a small number of workers (4 or less) we found that the model converged to state of the art. However, when the number of workers was increased to 12 (3 nodes with 4 workers per node), the model required more epochs to converge to state of the art and this negated the increased throughput gained from adding the additional workers. We hypothesize that this effect was due to the shard size becoming so small that the shard no longer accurately reflected the statistical distribution of the larger dataset. In essence, each shard was no longer an identical and independent distribution (IID) of the original dataset. In such a degenerate case, each worker would train on a skewed sampling of the original dataset and report biased gradient updates during aggregation at the end of the epoch. Consider, for example, the case where a worker trains on a shard which contained samples without a tumor present in the image. The gradients reported by this worker may actually cause the model to diverge from the global minimum during that epoch. Ultimately, we opted to not shard, but instead allowed each worker to train on the entire dataset. In this way, the total time per epoch remained constant, but the convergence to state of the art improved on average with the additional workers. Additional work will be needed to fully understand the effects of data sharding on deep learning.

Summary

Optimizations outlined in this paper may significantly decrease training times for single and multi-node CNN models on Intel Architecture. Consider the following best-known-methods when running DL workloads on Intel CPUs:

Single and Multi-Node:

- Install TensorFlow with MKL optimizations with `conda install -c anaconda tensorflow`. To test for the presence of MKL, run `export MKL_VERBOSE=1` which will trigger MKL to print debug statements whenever an operation is invoked.
- We favor the Anaconda distribution of Python which includes Intel-optimized versions of TensorFlow, Python, Numpy, Pandas, and scikit-learn.
- Set intra-op threads to the number of physical cores available to the process, and inter-op threads to the number of disjoint branches in the computation graph (usually in the single digits).
- When performing upsampling in convolutional layers, use the transposed convolution instead of `UpSampling2D`.

Multi-node only:

- When communicating between Intel Xeon nodes in a cluster, make use of the larger communication bandwidth available on the Intel OPA fabric.
- Keep copies of the entire dataset on each worker node to reduce network traffic.
- Use a warm up strategy to compensate for the proportionally fewer weight updates when training with multiple workers.
- Experiment with multiple workers per node to find optimal distributed execution settings for your topology and hardware.
- Sharding may not help as the number of workers increases because each worker may not be working with a statistically-representative subset of the data.

Acknowledgements

The authors would like to thank the GE Global Research AI team for their invaluable assistance over the course of the project.

References

- Bakas S, Akbari H, Sotiras A, Bilello M, Rozycki M, Kirby JS, Freymann JB, Farahani K, Davatzikos C. "Advancing The Cancer Genome Atlas glioma MRI collections with expert segmentation labels and radiomic features", *Nature Scientific Data*, 4:170117 (2017) DOI: 10.1038/sdata.2017.117
- Bakas S, Akbari H, Sotiras A, Bilello M, Rozycki M, Kirby J, Freymann J, Farahani K, Davatzikos C. "Segmentation Labels and Radiomic Features for the Pre-operative Scans of the TCGA-GBM collection", *The Cancer Imaging Archive*, 2017. DOI: 10.7937/K9/TCIA.2017.KLXWJJ1Q
- Bakas S, Akbari H, Sotiras A, Bilello M, Rozycki M, Kirby J, Freymann J, Farahani K, Davatzikos C. "Segmentation Labels and Radiomic Features for the Pre-operative Scans of the TCGA-LGG collection", *The Cancer Imaging Archive*, 2017. DOI: 10.7937/K9/TCIA.2017.GJQ7R0EF
- Chen J., Monga R., Bengio S. and R. Jozefowicz. "Revisiting Distributed Synchronous SGD." ICLR 2016. <https://openreview.net/pdf?id=D1VDZ5kMAu5jEJ1zfEWL>
- Drozdal M., Chartrand G., Vorontsov E. et al. "Learning Normalized Inputs for Iterative Estimation in Medical Image Segmentation." arXiv:1702.05174v1 [cs.CV] 16 Feb 2017
- Dumoulin V. and F. Visin. "A guide to convolution arithmetic for deep learning". 11 Jan 2018 (<https://arxiv.org/pdf/1603.07285.pdf>)
- Goyal P, Dollar P, Girshick R, et al. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. arXiv:1706.02677v2 [cs.CV] 30 Apr 2018 (<https://arxiv.org/pdf/1706.02677.pdf>)
- Menze BH, Jakab A, Bauer S, Kalpathy-Cramer J, et al. "The Multimodal Brain Tumor Image Segmentation Benchmark (BRATS)", *IEEE Transactions on Medical Imaging* 34(10), 1993-2024 (2015) DOI: 10.1109/TMI.2014.2377694
- Thieme M. and G. A. Reina. Biomedical Image Segmentation with U-Net. 23 Jan 2018 (<https://ai.intel.com/biomedical-image-segmentation-u-net/>)
- Ronneberger O., Fischer P. , and T. Brox. "U-Net Convolutional Networks for Biomedical Image Segmentation." arXiv:1505.04597v1 [cs.CV] 18 May 2015
- Zeiler M. and R. Fergus. Visualizing and Understanding Convolutional Networks. 28 Nov 2013. <https://arxiv.org/pdf/1311.2901v3.pdf>
- Zhao X., Wua Y., Song G., Li Z., Zhang Y., and Y. Fan. "A deep learning model integrating FCNNs and CRFs for brain tumor segmentation." *Medical Image Analysis* 43 (2018) 98–111.



¹ More information about the Brain Tumor Segmentation (BraTS) challenge can be found at <https://www.med.upenn.edu/sbia/brats2018.html>

² Instructions for installing Intel® Optimized TensorFlow™ can be found at: <https://software.intel.com/en-us/articles/intel-optimized-tensorflow-wheel-now-available>. Many of these changes have already been included in the main TensorFlow™ branch.

³ For an overview of the OpenMP thread affinity interface, see: <https://software.intel.com/en-us/articles/openmp-thread-affinity-control>

⁴ For an overview of the Intel® MKL-DNN optimizations used in TensorFlow™ <https://software.intel.com/en-us/articles/tensorflow-optimizations-on-modern-intel-architecture>

⁵ Code and execution instructions for this implementation of U-Net can be found at https://github.com/NervanaSystems/topologies/tree/master/distributed_unet/Horovod.

⁶ Overview of Horovod training library can be found at: <https://eng.uber.com/horovod/>

Portions of this work were supported by NIH National Institute of Biomedical Imaging and Bioengineering Neuroimaging Analysis Center P41EB015902.

Notices

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark* and MobileMark*, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

§ Configurations:

Single Node Testing by Intel as of September 7, 2018:

1-node, Intel® Xeon® Platinum 8168, Total memory 768 GB, HyperThreading: Enable, SpeedStep: Enable, OS: CentOS 7, Kernel: centos-release-7-3.1611.el7.centos.x86_64

Multi-Node Testing by Intel as of September 7, 2018:

3-node, Intel® Xeon® Platinum 8168, Total memory 768 GB, HyperThreading: Enable, SpeedStep: Enable, Network devices: Intel® Omni-Path Host Fabric Interface (Intel® HFI) Silicon 100 Series PCIe* x16 (rev. 11), OS: CentOS 7, Kernel: centos-release-7-3.1611.el7.centos.x86_64

Performance results are based on testing as of September 7, 2018 and may not reflect all publicly available security updates. See configuration disclosure for details.

No product can be absolutely secure.

For more information go to <http://www.intel.com/performance>.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. Check with your system manufacturer or retailer or learn more at intel.com.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

This sample source code is released under the Intel Sample Source Code License Agreement.

Intel, the Intel logo, Intel SpeedStep, and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.