WHITE PAPER

Artificial Intelligence
Deep Learning
Inference
Linux
TensorFlow
Anaconda
Python

How to maximize inference performance with TensorFlow on Intel® platforms

Tristan Ladrech
Intel Corporation

Stanislas Odinot
Intel Corporation

Marc Gaucheron
Intel Corporation

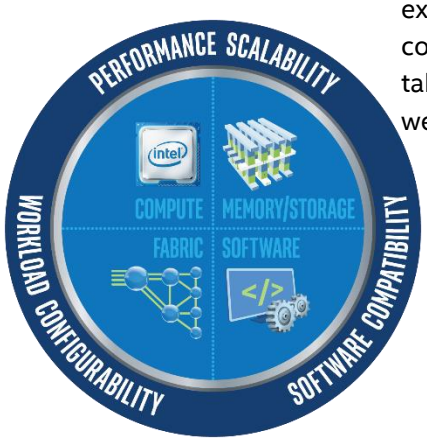
Ludovic Sauge
Intel Corporation

Deep learning is a fast-growing discipline and a lively ecosystem at many levels. New algorithms, neural networks topologies and software stacks are created daily. As in other new domains, feature development is the priority. Later, when performance is required, setting up more hardware for acceleration is frequently the easiest way to address the problem. Achieving performance to keep pace with this rapid innovation is exciting but raises certain concerns for data scientists.

Even in the specific field of deep learning inference processing, identifying the right hardware for batch image classification or natural language processing, for example, can be challenging. Learning how to squeeze performance out of this hardware adds even more complexity and is often a lengthy process. Lastly, although these difficulties can be overcome, when the solution is applied to a large-scale inferencing project, the total cost can quickly become prohibitive, even in the cloud.

To help resolve most of these problems, this white paper offers data scientists a closer look at a computing resource widely available at various price points: the CPU (central processing unit). Since well-known deep learning frameworks like TensorFlow have become available, tremendous progress has been made to optimize their use on CPUs. The latest Intel processors also bring new AI capabilities to accelerate inferencing processing specifically.

By following a few simple steps described in this document a data scientist can expect significant levels of performance on CPUs with very little impact on source code or accuracy, but huge advantages in cost savings and scaling. In this paper we take the inference on a ResNet-50 and a synthetic dataset as an example, as it is a well-known topology.



Notices & Disclaimers

Intel technologies may require enabled hardware, software or service activation. Your costs and results may vary.

No product or component can be absolutely secure.

Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. For more complete information about performance and benchmark results, visit http://www.intel.com/benchmarks .

Intel Advanced Vector Extensions (Intel AVX) provides higher throughput to certain processor operations. Due to varying processor power characteristics, utilizing AVX instructions may cause a) some parts to operate at less than the rated frequency and b) some parts with Intel Turbo Boost Technology 2.0 to not achieve any or maximum turbo frequencies. Performance varies depending on hardware, software, and system configuration and you can learn more at http://www.intel.com/go/turbo.

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

Table of Contents

Section 1. What to expect

Introduction

Maximizing inference performance with TensorFlow on Intel platforms requires the following steps, each step successively adding more performance:

1. Select the best CPU
2. Set up an optimized version of TensorFlow
3. Make your Python code "aware" of your processor and server specifications
4. Optimize your model (pruning, constant folding, and batch normalization)
5. Run multiple inference instances in parallel to maximize your CPU performance
6. Quantize your model to achieve even greater model performance.

In this section (**Section 1**) of the whitepaper, we briefly present the level of performance you should expect when doing inference on ResNet-50 and a synthetic dataset with Python code. If you are convinced of the possibility of rapid performance improvement, we recommend that you review certain basic principles about your CPU and what you need to know on TensorFlow in **Section 2.**

You can also jump directly into **Section 3** where you can try out a short procedure to achieve high levels of performance on any CPU. **Section 3** is basically a tutorial with the goal of enabling you to experiment quickly, before diving into deeper understanding. Each step of the tutorial has its own dedicated section with further information. Moreover, some optimizations like multi-instancing and quantization are too complex for quick experimentation: in these cases, refer to the dedicated sections.

Lastly, if these experimentations motivate you to increase your knowledge of optimizing inference workloads to run on CPUs, we encourage you to read all the other sections.

Please note that this tutorial uses ResNet-50 as an example to show you how to apply these optimizations.

Optimizing inference on Intel processors

By following the recommendations laid out in this whitepaper you can expect a significant performance boost for your inference processing workload. This is illustrated below with ResNet-50 using a synthetic ImageNet dataset. The performance indicated in the column on the right is achievable by using 2nd Generation Intel® Xeon® Scalable Processors, which support Intel® DL Boost technology. Yet even on older generation CPUs, there are still possibilities for much greater performance from software.

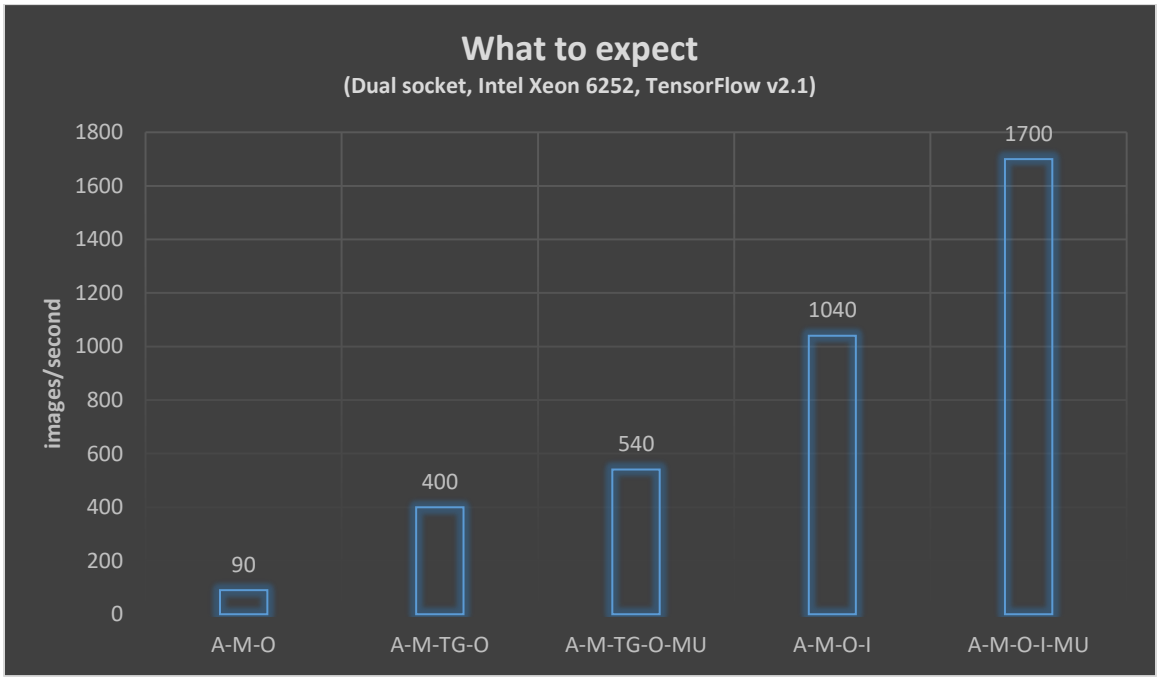


*Figure 1 - What you can expect from the different optimizations. **A**: AVX-512 activated, **M**: oneDNN (formerly MKL-DNN) enabled, **TG**: Transform Graph optimized model, **O**: intra-threads and inter-threads optimized, **I**: INT8 quantized, **MU**: Multiple instances launched*

Configuration 1 (90 img/s): this is what you can expect running inference on a synthetic dataset in commonly occurring conditions:

- oneDNN (formerly MKL-DNN) enabled in TensorFlow 2.1 (see **Section 2** for details)
- AVX-512 activated in processors used
- Optimized thread allocations (see **Section 5** for details)

Configuration 2 (400 img/s): this is Configuration 1 with **neural network optimization** (see **Section 6**).

Configuration 3 (540 img/s): this is Configuration 1+2 and **multi-instancing,** detailed in **Section 7**.

Configuration 4 (1040 img/s): this is Configuration 2 with neural network **quantization** (see **Section 8**).

Configuration 5 (1700 img/s): this is Configuration 4 with multi-instancing.

Section 2. What you need to know

TensorFlow: CPU optimized version

Intel has been collaborating for several years with Google to optimize TensorFlow performance for deep learning training and inference on Intel Xeon processors using oneAPI Deep Neural Network Library or oneDNN[1] (formerly MKL-DNN). To improve performance, it is critical in most instances to install TensorFlow with oneDNN embedded. In the past this was a non-trivial operation but is now relatively easy to do using the many methods explained in this Intel webpage[2]. In this whitepaper, we used Anaconda, for its convenience and the way it enables multiple Python and TensorFlow versions to coexist.

Processors

As we already stated, keeping track of all the hardware available to run inference and new versions of this hardware can be complex. However, this task is easier with processors, as we can narrow down key specifications for processors to essentially the following five: how many processors there are in the server and the number of cores, the frequency, the register size and the specialized instruction set per processor. Performance typically improves as these specifications increase or broaden, on condition that data scientists follow the guidelines below.

1. CPU sockets

In datacenters and whether your code runs in a virtual machine (VM) or directly on a physical system, it is important to know that most servers use two processors. The processors are connected, for example, via the Intel Ultra Path Interconnect (UPI), while each processor has its own dedicated memory. This specific topology needs to be handled carefully if you plan to optimize your Python code because by default this code will often run randomly on either CPU (CPU0 or CPU1 in the diagram below) with or without the dedicated memory for that processor. This can have a significant negative impact on performance.

As a data scientist, one of the best techniques to boost your inference processing performance is to run your code twice, each time on a different CPU while forcing that CPU to use its own dedicated memory: see how in **Section 7**. To get even more performance, you can work at the CPU core level which is also explained in detail in **Section 7** of this document.

[1] https://github.com/oneapi-src/oneDNN
[2] https://software.intel.com/content/www/us/en/develop/articles/intel-optimization-for-tensorflow-installation-guide.html

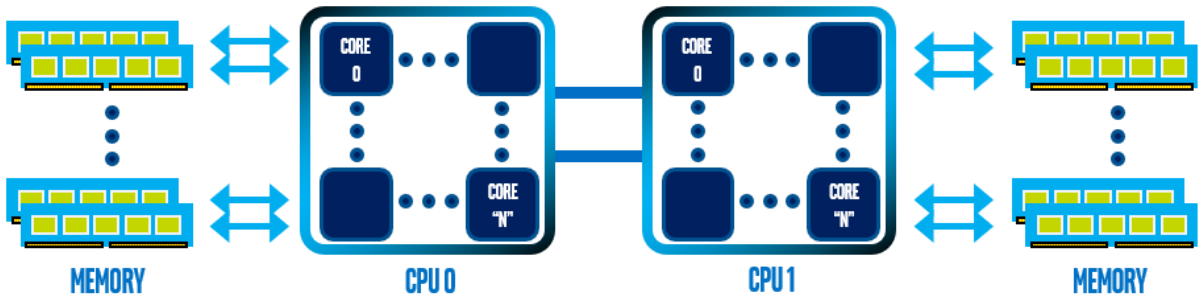


Figure 2 - In the cloud or datacenter, a common hardware configuration is two processors, each on a socket and physically linked together to get more performance and/or memory capacity per server. However, extracting maximum performance from a two-socket server requires tuning that is explained later in this document.

To check in Linux whether your server is based on one or two processors, you can run the `lscpu` command:

```
shell$ lscpu | egrep 'Socket'
Socket(s):           1
```

2. Physical cores and logical cores

The more Intel cores you have, the better. Nowadays, nearly all CPUs are multi-core: they contain several processing units that can handle different tasks simultaneously. The physical cores allow applications to divide up work for simultaneous processing, called "threads" in software. Most Intel processors also include Intel® Hyper-Threading Technology, a hardware innovation that allows more than one thread to run on each core. Having more threads means that more work can be done in parallel. As of today, an Intel CPU offers two execution contexts per physical core. This means that one physical core works like two logical cores to handle different software threads.

Proper handling of threads in deep learning is critical to extract top performance. For this, data scientists have basically two options. The first is to use a library which allows multi-thread programming. The second, which applies reasonably well to inference workloads, is to simply run your Python code multiple times, a technique also used in this whitepaper.

To check how many physical and logical cores are running in a server or VM, and how they are organized in the operating system, we recommend using the `hwloc` package which includes several handy tools like `lstopo`.

```
shell$ sudo apt install hwloc
shell$ lstopo --no-caches --no-io --cpuset
```

For example, for a single socket with 18 cores, the physical and logical core organization is displayed like this:

```
Machine (187GB) cpuset=0x0000000f,0xffffffff
  Package L#0 cpuset=0x0000000f,0xffffffff
    Core L#0 cpuset=0x00040001
      PU L#0 (P#0) cpuset=0x00000001
      PU L#1 (P#18) cpuset=0x00040000
…
```

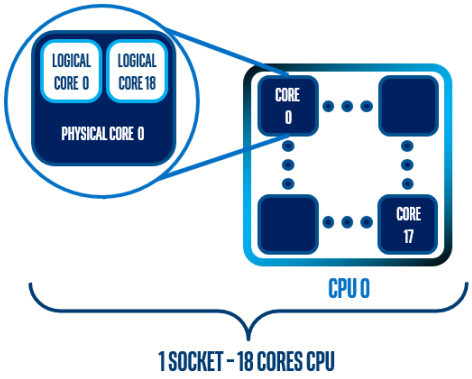


Figure 3 - Representation of cores numbering in a single socket

In a dual socket server with 24 dual cores, the first physical core number is 0 and the logical core numbers for this physical core number are 0 and 48.

```
Machine (1511GB total) cpuset=0xffffffff,0xffffffff,0xffffffff
  NUMANode L#0 (P#0 755GB) cpuset=0x000000ff,0xffff0000,0x00ffffff
    Package L#0 cpuset=0x000000ff,0xffff0000,0x00ffffff
      Core L#0 cpuset=0x00010000,0x00000001
        PU L#0 (P#0) cpuset=0x00000001
        PU L#1 (P#48) cpuset=0x00010000,0x0
…
```

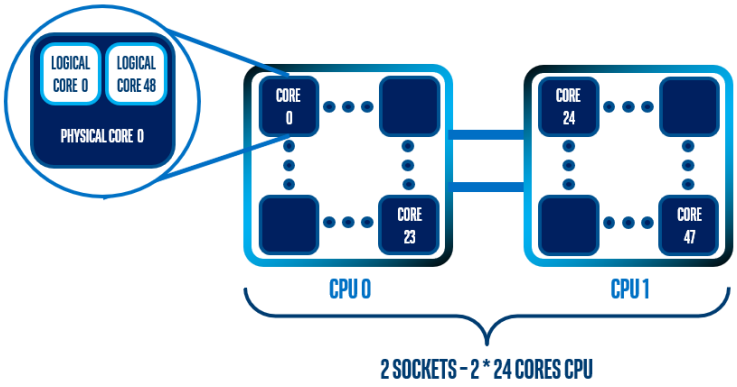


Figure 4 - Representation of cores numbering in a dual socket

3. Frequency

In general, **a higher clock speed means a faster CPU frequency**, and frequency measures the number of cycles your CPU executes per second. This frequency is usually expressed in GHz (gigahertz). A cycle is a period where billions of transistors within the processor switch on and off to execute instructions (low-level calculations like arithmetic). For example, a CPU with a clock speed of 3.2 GHz executes 3.2 billion cycles per second, assuming that multiple instructions are completed in a single clock cycle; in other cases, one instruction might be handled over multiple clock cycles.

4. Register size

The larger the register sizes are, the better. Depending on the generation of your CPU or the nature of your virtual machine (whether in the cloud or not), you will have access to different "n-bits" registers:

- 128 bits (SSE)
- 256 bits (AVX2)
- or 512 bits (AVX-512)

Between 2012 and 2017, most of the Intel Xeon E5 processors series (code name Sandy Bridge, Ivy Bridge, Haswell and Broadwell) included the AVX and later the AVX2 instruction set with 256 bit wide registers. With most commercial deep learning applications using 32 bits of floating-point precision (FP32), each core of these processors could process up to 8 x 32 bits.

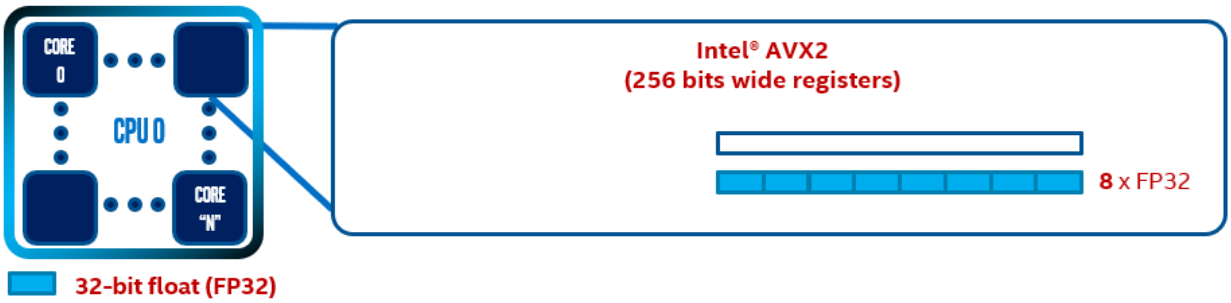


Figure 5 - Intel AVX2 representation

Launched late 2017, 1st Generation Intel Xeon Scalable Processors (code name Skylake) introduced the AVX-512 instruction set with registers twice as large (512 bits) as those of AVX2 (256 bits), allowing the processing of up to 16 x 32-bit values. Additionally, these processors have FMA (Fused Multiple Add operations, A=A*B+C). FMA is valuable for linear algebra-based machine learning (ML) algorithms, deep learning, and neural networks (dot product, matrix multiplication).

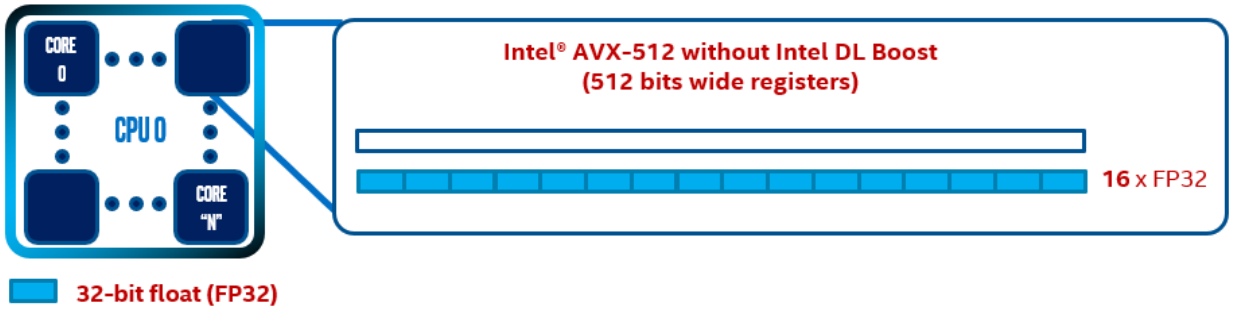


Figure 6 - Intel AVX-512 representation

5. Specialized instruction set

Available since 2019, 2nd generation Intel Xeon Scalable Processors (code name Cascade Lake) are currently the first processors to support **Intel DL Boost** technology. This new instruction set (AVX512_VNNI) enables 8-bit multiplies with 32-bit accumulates with 1 instruction. This allows 4 x more inputs compared to FP32 and a theoretical peak of 4 x more compute with only 1/4 of the memory requirements.

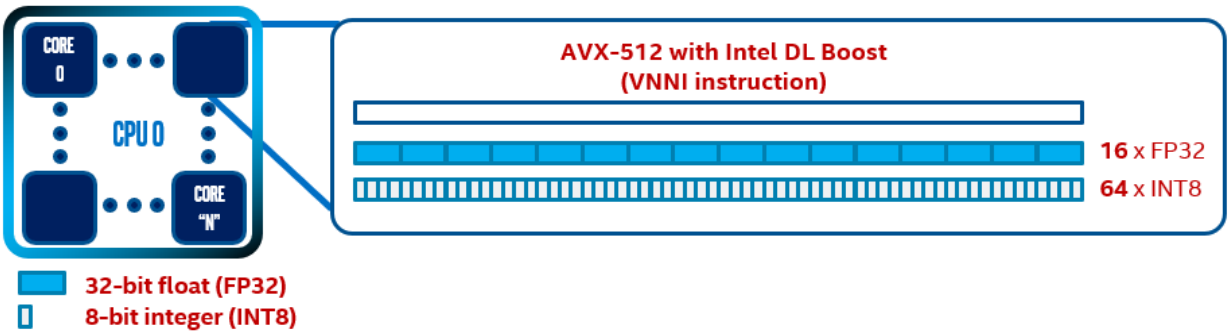


Figure 7 - Intel AVX-512 with Intel DL Boost representation

If you want to check your CPU capabilities quickly in Linux, run the following command:

```
shell$ grep -o 'avx[^ ]*' /proc/cpuinfo
avx
avx2
```

By using the Linux grep command above, you can quickly check how wide the registers are in your processor. The current VM displays "AVX" and "AVX2" which mean 128- and 256-bits wide registers.

To get details on the CPU model on which you are running your deep learning workload, execute this command in Linux:

```
shell$ cat /proc/cpuinfo | grep "model name" | head -1
```

If the output looks like the example below, enter the CPU model (6252 in this example) in the Intel CPU database at ark.intel.com to retrieve its specification.

```
model name : Intel(R) Xeon(R) Gold 6252 CPU @ 2.10GHz
```

If the output does not explicitly mention a processor model but looks like the example below, it means you are running in a virtual machine. You will need to consult your cloud service provider website or ask your server administrator to get details on the processor specification.

```
model name : Intel Core Processor (Haswell, no TSX)
```

Section 3. Quick start

By following each of the steps in this section, a data scientist can expect a significant performance gain from an optimized inference setup compared to a basic one. The level of this gain will depend on attention to certain details relating to the TensorFlow version, a few lines of code, and the underlying processor used for inference, as well as learning how to optimize a ResNet-50 neural network.

Assuming you are using any server or VM running a recent version of Ubuntu (18.04 was used for this tutorial) you can follow these steps to quickly see for yourself how to extract performance from any Intel processor.

1. Installing prerequisite software (see Section 4 for more details)

To get the best performance, first set up an optimized software environment. This consists mainly of installing the right version of TensorFlow and Python. We installed Anaconda, used Python 3.7, and installed TensorFlow v2.1 through the Anaconda channel since that version is built using oneDNN (formerly MKL-DNN). For more information, go to **Section 4**.

You will also need to install Bazel to build tools allowing pre-trained neural network optimization, and `numactl` to "run processes with a specific NUMA scheduling or memory placement policy"[1]. More precisely, `numactl` will enable us to launch multiple instances efficiently to improve performance in **Section 7**. You can find installation information for the software described above as follows:

- TensorFlow here[2]
- Anaconda and Linux here[3]
- Bazel here[4]
- For `numactl`, see below

```
shell$ sudo apt install numactl --yes
```

2. Tune your TensorFlow code adding os lib + `ConfigProto` (see Section 5 for more details)

Our baseline requires the thread allocations. To perform better, TensorFlow must be aware of the physical cores in your server and how to parallelize the workflow. Add this information directly in your Python code using the os library.

- Find the number of physical cores:

```
shell$ lscpu | grep 'Core(s) per socket:'
```

[1] https://linux.die.net/man/8/numactl
[2] https://software.intel.com/content/www/us/en/develop/articles/intel-optimization-for-tensorflow-installation-guide.html
[3] https://docs.anaconda.com/anaconda/install/linux/
[4] https://docs.bazel.build/versions/master/install-ubuntu.html

- Add these lines to your Python code and put in your number for **`#number of physical cores`** (you can use the code in Appendix B as an example):

```
import os
os.environ["KMP_BLOCKTIME"] = "1"
os.environ["KMP_SETTINGS"] = "1"
os.environ["KMP_AFFINITY"]= "granularity=fine,verbose,compact,1,0"
if FLAGS.num_intra_threads > 0:
  os.environ["OMP_NUM_THREADS"]= #number of physical cores
```

- Add the following lines to your code to configure the session (you can use the code in Appendix B as an example):

```
configuration = tf.compat.v1.ConfigProto()
configuration.intra_op_parallelism_threads = #number of physical cores
configuration.inter_op_parallelism_threads = 1
# configuration.inter_op_parallelism_threads = 2 # for ResNet101
tf.compat.v1.Session(graph=model_graph, config= configuration)
```

3. Optimize your neural network (see Section 6 for more details)

- Clone a new TensorFlow folder in your working directory.
- Copy/paste your TensorFlow frozen graph in the "/tensorflow" folder. Our model was named "*resnet50_fp32_pretrained_model.pb*". Use your own model name in the following commands.
  If you want, you can download our model by running:

```
shell$ wget https://storage.googleapis.com/intel-optimized-
tensorflow/models/v1_6/resnet50_fp32_pretrained_model.pb
```

- Run:

```
shell$ bazel build tensorflow/tools/graph_transforms:summarize_graph
```

- Run:

```
shell$ bazel-bin/tensorflow/tools/graph_transforms/summarize_graph --
in_graph=resnet50_fp32_pretrained_model.pb
```

The command above will return information. The information that you will need has been underlined in the example output below:

```
Found 1 possible inputs: (name=input, type=float(1), shape=[?,224,224,3])
No variables spotted.
Found 1 possible outputs: (name=predict, op=Softmax)
```

- Run:

```
shell$ bazel build tensorflow/tools/graph_transforms:transform_graph
```

- Build your optimized model (replace *resnet50_fp32_pretrained_model.pb* by your model name):

```
shell$ bazel-bin/tensorflow/tools/graph_transforms/transform_graph \
--in_graph=resnet50_fp32_pretrained_model.pb \
--out_graph=optimized_resnet50_fp32_graph.pb \
--inputs='input' \--outputs='predict' \
--transforms='strip_unused_nodes remove_nodes(op=Identity,
op=CheckNumerics) fold_constants(ignore_errors=true) fold_old_batch_norms
fold_batch_norms'
```

- Check the TensorFlow Graph Tool for more information.

4. **Run two or more instances and check results (see Section 7 for more details)**

- As running multiple instances is more complex than the optimizations described so far, go to **III. How to launch multiple instances (summary)** for a quick tutorial. The first two parts of **Section 7** may also help you better understand and succeed. With multiple instances, we achieved up to 6X better performance compared to our baseline.

5. **Quantize your model to get the highest level of performance (see Section 8 for more details)**

Some CPUs like the latest 2nd Gen Intel Xeon Scalable Processors (Cascade Lake) support INT8 computations and can infer efficiently on a quantized model. If your processors have this capability, consider quantization of your model. By quantizing it, you will experience only a small loss in accuracy (0.07% in our case) but gain a substantial improvement in performance: **up to 3X in our case**[1]. Quantizing a model is not as easy as optimizing it, and there is no quick or generally applicable tutorial for quantization. Look at **Section 8** to learn more about quantization and for a specific tutorial showing how we quantized our model.

[1] Inference with dummy data. Throughput with optimized Resnet50 fp32 and multiple instances: 540 images/seconds. Throughput with optimized Resnet50 int8 and multiple instances: 1700 images/seconds with 0.07% of accuracy loss. Setup: Appendices C

Section 4. Set up your environment

In this section, you will learn how to setup your environment to achieve the best performance possible. To let data scientists take advantage of many Intel processors over a broad range of deep learning frameworks, including TensorFlow, Intel developed a library named Intel® MKL-DNN (Math Kernel Library for Deep Neural Networks), now part of a wider development initiative named oneDNN (GitHub here[1]). Optionally, oneDNN can be included while compiling TensorFlow. Many other ways to install TensorFlow with enhanced user-friendliness are also available: pip, prebuild container images, or Anaconda package manager, for example. Details of all options are given in this link[2]. In this whitepaper you will use Anaconda with Python 3.7 and install TensorFlow through the Anaconda channel, although others like the Intel channel could also be used. Both channels provide a TensorFlow version built using the oneDNN primitives, which usually leads to better performance.

> *The **conda** environment uses a powerful dependency resolver mechanism. Keeping the Python environment stable requires extreme caution when resolving any package dependency conflicts.*

First, we will start with the Linux package upgrade and installation:

```
# Update your packages
shell$ sudo apt-get update && sudo apt-get upgrade --yes
```

Utilities

To run multiple instances (**Section 7**) and precisely allocate the different workflows efficiently on each core of the CPU, you will need `numactl`. If you do not intend to do multi-instancing, you will not need it.

```
# numactl: installation
shell$ sudo apt install numactl --yes
```

Anaconda: download and install

In this white paper we decided to use Anaconda which offers the convenience of setting up a specific new environment without overlapping with the existing one.

```
# Anaconda: install required packages
shell$ sudo apt-get install libgl1-mesa-glx libegl1-mesa libxrandr2 libxrandr2
libxss1 libxcursor1 libxcomposite1 libasound2 libxi6 libxtst6 --yes

# Anaconda: download
```

[1] https://github.com/oneapi-src/oneDNN/
[2] https://software.intel.com/content/www/us/en/develop/articles/intel-optimization-for-tensorflow-installation-guide.html

```
shell$ wget https://repo.anaconda.com/archive/Anaconda3-2020.02-Linux-
x86_64.sh

# Anaconda: installation
shell$ bash ~/Anaconda3-2020.02-Linux-x86_64.sh
```

Now follow the Anaconda installation procedure. At the prompt press **ENTER**, read the **EULA**, type **yes** to accept the EULA, and confirm or change the installation path. Wait and enter **yes** to initialize Anaconda3.

> Important: close your shell and reopen it to let Anaconda modify your shell. Part of the convenience of conda is that you can create, export, list, remove, and update environments that have different versions of Python and/or packages installed in them. The default environment name will be `(base)` and the name will be changed in your shell depending on the environment you created and switched to. The example below uses `(base)`:

```
(base) user@hostname:~$
```

As we described above, we decided to use the environment functionality of Anaconda to avoid overlapping with the existing one. Create and activate an environment named "py-3.7-tf-2.1" using the conda tool.

```
# Anaconda: create and activate environment named py-3.7-tf-2.1
shell$ conda create --name py-3.7-tf-2.1 python=3.7 --yes
shell$ conda activate py-3.7-tf-2.1
```

Now your shell should start with `(py-3.7-tf-2.1) user@host`

TensorFlow

You can install TensorFlow either from the conda channel or the Intel® channel. From TensorFlow v1.9 onwards, Anaconda builds TensorFlow using Intel Math Kernel Library for Deep Neural Networks (Intel MKL-DNN, now renamed oneDNN) primitives to deliver maximum performance in your CPU[1]. The only difference between the two channels is that the update cadence is faster on the Intel® channel. According to your preference for the channel, run either:

```
# Anaconda: install Tensorflow from Anaconda channel
shell$ conda install tensorflow=2.1 --channel anaconda –yes
```

or

```
# Anaconda: install Tensorflow from Intel® channel
shell$ conda install tensorflow -c intel
```

Check your installation by importing the TensorFlow library and displaying its version.

[1] https://software.intel.com/content/www/us/en/develop/articles/intel-optimization-for-tensorflow-installation-guide.html

```
# Anaconda: check TensorFlow version
shell$ python -c "import tensorflow as tf; print(tf.__version__)"
# Should display -> 2.1.0
```

Section 5. Tune your TensorFlow code

After having ensured that you are using an optimized version of TensorFlow, the next step to accelerate inference is to make TensorFlow aware of the capabilities of the processor(s). This is straightforward and is done in two steps: first, you query the server for the CPU information; second, you add a few lines to your source code.

1. Enumerate your processor capabilities

The number of cores per socket inside your server is what matters. An easy way to collect this information is to run the following command to list all the physical cores in the server.

```
shell$ lscpu -b -p=Core,Socket | grep -v '^#' | sort -u | wc -l
```

2. Map your server capabilities into your TensorFlow* code

Import the Python `os` library and set following environment variables which will be used by Intel oneDNN to get optimal performance (this can be done inside the Linux shell using `export`). **Add the following lines to your Python code**. If you do not know where to paste them in your code, use our custom code (Appendix B) as an example.

```
import os
os.environ["KMP_BLOCKTIME"] = "1"
os.environ["KMP_SETTINGS"] = "1"
os.environ["KMP_AFFINITY"]= "granularity=fine,verbose,compact,1,0"
if FLAGS.num_intra_threads > 0:
  os.environ["OMP_NUM_THREADS"]= #number of physical cores
```

When you have done this, you will need to configure your TensorFlow session through `ConfigProto`, which sets the session parameters related to threads, core scaling and parallelism. **Add the following lines to your Python code**. If you do not know where to paste them in your code, use our custom code (Appendix B) as an example.

```
configuration = tf.compat.v1.ConfigProto()
configuration.intra_op_parallelism_threads = #number of physical cores
configuration.inter_op_parallelism_threads = 1
# configuration.inter_op_parallelism_threads = 2 # for ResNet101
tf.compat.v1.Session(graph=model_graph, config= configuration)
```

> **Note:**
> *Certain settings like KMP_BLOCKTIME or intra_op* / inter_op* can be optimized.*
> *If you want to better understand those specific settings to set up the Intel library and TensorFlow configuration, we recommend that you read this[1] Intel web page.*

[1] https://software.intel.com/content/www/us/en/develop/articles/maximize-tensorflow-performance-on-cpu-considerations-and-recommendations-for-inference.html

Section 6. Optimize your neural network

One of the most effective ways to achieve better performance while doing inference is to optimize your neural network in advance. This optimization is possible because although training computations in a neural network may require many nodes, some of these nodes are not involved in the output calculations when doing inference. Therefore, you can remove them. Others can be simplified or combined with other nodes.

By optimizing a ResNet-50[1], its inference performance was improved by **up to 4X** [2] without affecting its accuracy. The performance increased from 90 images/seconds to 400 images/seconds, which is clearly a substantial improvement.

We wrote a tutorial on how to do this for the specific case of the ResNet-50 using the TensorFlow tool Graph Transform. You should also be able to reuse that tutorial for other models.

1. Go to your TensorFlow folder (clone a new one in advance if you prefer)

Although you could do all the following operations in your current TensorFlow folder, we recommend cloning a new one in your work directory to avoid any unfortunate mistakes. We used TensorFlow 2.1 for this tutorial. Ensure that you use this version for your own cloned version of TensorFlow.

2. Copy/paste your model in your "tensorflow" folder

Your model graph must be a frozen graph '.pb' (protobuf) file. Instructions on how to freeze a graph are readily available from different sources on the web. To check that you are in the right folder, make sure it also contains a "WORKSPACE" file. In our case, the model file is:

resnet50_fp32_pretrained_model.pb

If you want, you can download it by running:

```
$ wget https://storage.googleapis.com/intel-optimized-
tensorflow/models/v1_6/resnet50_fp32_pretrained_model.pb
```

3. If you have not already done so, install Bazel from the official Bazel web page

While running the following command, you may need to install another version of Bazel, so just follow the recommendations. See the official web page here[3].

4. Determine the input and output of your graph

While in the "tensorflow" folder, run:

```
$ bazel build tensorflow/tools/graph_transforms:summarize_graph
```

[1] https://storage.googleapis.com/intel-optimized-tensorflow/models/v1_6/resnet50_fp32_pretrained_model.pb
[2] Inference with dummy data. Throughput with basic Resnet50 fp32: 90 images/seconds
Throughput with optimized Resnet50 fp32: 400 images/seconds. Both accuracies are identical.
Setup: Appendices C
[3] https://docs.bazel.build/versions/master/install-ubuntu.html

Once "summarize_graph" has been built, you can determine the input and output of your graph by running:

```
$ bazel-bin/tensorflow/tools/graph_transforms/summarize_graph --
in_graph=resnet50_fp32_pretrained_model.pb
```

Replace "resnet50_fp32_pretrained_model.pb" by your model name. The output of the command above should look like the following:

```
Found 1 possible inputs: (name=input, type=float(1), shape=[?,224,224,3])
No variables spotted.
Found 1 possible outputs: (name=predict, op=Softmax)
Found 25612214 (25.61M) const parameters, 0 (0) variable parameters, and 0
control_edges
Op types used: 272 Const, 267 Identity, 53 Conv2D, 53 FusedBatchNorm, 49 Relu,
16 AddN, 2 Mul, 1 Add, 1 AvgPool, 1 MatMul, 1 MaxPool, 1 Pad, 1 Placeholder, 1
Reshape, 1 Softmax, 1 Sub
To use with tensorflow/tools/benchmark:benchmark_model try these arguments:
bazel run tensorflow/tools/benchmark:benchmark_model -- --
graph=resnet50_fp32_pretrained_model.pb --show_flops --input_layer=input --
input_layer_type=float --input_layer_shape=-1,224,224,3 --output_layer=predict
```

We have highlighted the inputs and outputs of our model. In our case, input=***input*** and output=***predict***. Make sure to remember these outputs (yours can be different than ours).

5. Optimize your model

Now you have all the information required to build your optimized model. First, build "transform_graph" with Bazel:

```
$ bazel build tensorflow/tools/graph_transforms:transform_graph
```

Next, you can build your optimized model:

```
$ bazel-bin/tensorflow/tools/graph_transforms/transform_graph \
      --in_graph=resnet50_fp32_pretrained_model.pb \
      --out_graph=optimized_resnet50_fp32_graph.pb \
      --inputs='input' \--outputs='predict' \
      --transforms='strip_unused_nodes remove_nodes(op=Identity,
op=CheckNumerics) fold_constants(ignore_errors=true) fold_old_batch_norms
fold_batch_norms'
```

The argument `in-graph` refers to the model to optimize, and `out-graph` refers to the optimized model that transform_ graph will create. The arguments `inputs` and `outputs` correspond to the inputs and outputs determined at the previous step. Finally, `transforms` enumerates the optimizations applied to the neural network. You can find the list of all the possible transformations on the GitHub TensorFlow repository in the Graph Transform section[1].

[1] https://github.com/tensorflow/tensorflow/blob/master/tensorflow/tools/graph_transforms/README.md

> **Note**
> `strip_unused_nodes` *and* `remove_nodes` *are pruning operations. They remove all nodes not used in the inference calculations.*
> `fold_constants`, `fold_old_batch_norms`, *and* `fold_batch_norms` *are mathematical optimizations.*
> *You can add more optimization if you want but check the accuracy of the resulting model.*

6. Outputs of the optimization

This optimization improved inference performance on ResNet-50[1] by **up to almost 4.5X** [2]. In addition, the accuracies of the optimized and basic models are identical. Therefore, this optimization should be done since it enhances the performance of your code without any drawbacks.

[1] https://storage.googleapis.com/intel-optimized-tensorflow/models/v1_6/resnet50_fp32_pretrained_model.pb
[2] Inference with dummy data. Throughput with basic Resnet50 fp32: 90 images/seconds
Throughput with optimized Resnet50 fp32: 400 images/seconds. Both accuracies are identical.
Setup: Appendices C

Section 7. Execute your code multiple times

I. **Why** launch multiple instances

When the following optimization is combined with that of the previous section, overall inference performance on ResNet-50[1] can be improved by **up to 6X** [2]. Performance after the following optimization showed a significant increase from 400 images/seconds to 540 images/seconds. Moreover, there is no negative impact on accuracy.

Inferencing on a dataset is a perfectly parallelizable process because the final output does not depend on the way the data are processed as long as each piece of data or datum is processed by the whole algorithm. However, inference computations are often not as parallelizable (keep in mind the difference between inferencing on a dataset and running inference computations on a datum). Therefore, Python code running TensorFlow may not be able to take full advantage of multi-core processors. The question is therefore, how can we parallelize the process of inferencing without unduly parallelizing the computations to maximize the potential advantage of multiple cores?

The answer is to run multiple instances of TensorFlow and correctly allocate cores to each instance. Running multiple instances simply means launching your code multiple times with each instance inferencing on a delimited part of the dataset. Thus, each instance has its own data stream, and therefore represents an independent inference stream.

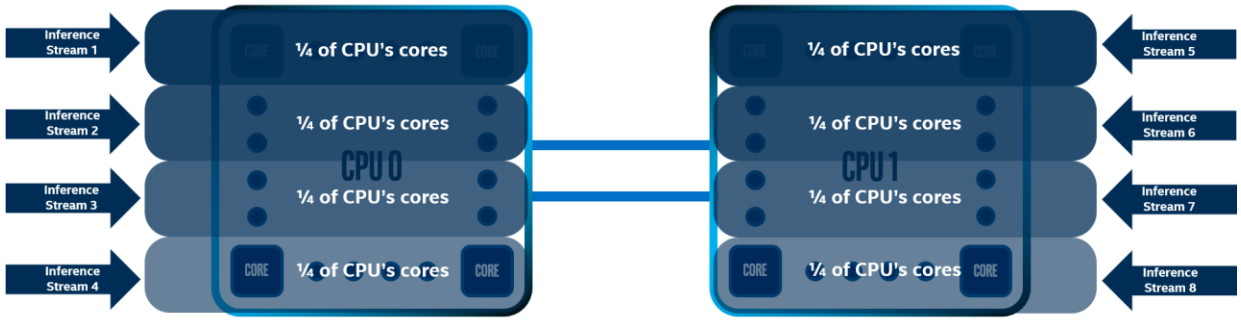


Figure 8- 8 inference streams on 2 sockets each containing an Intel Xeon Scalable Processor

However, as we have pointed out previously, the allocation of cores must follow certain rules to be fully efficient:

1. As you can see in Figure 8, cores allocated to each instance must be on the same processor/socket. There are several reasons for this, including optimization of the memory allocation to each instance or avoidance of the latency between the cores of different sockets during computations.
2. Memory must be strictly allocated to its socket.

[1] https://storage.googleapis.com/intel-optimized-tensorflow/models/v1_6/resnet50_fp32_pretrained_model.pb
[2] Inference with dummy data. Throughput with single instance, optimized Resnet50 fp32: 400 images/seconds. Throughput with ideal number of instances, optimized Resnet50 fp32: 540 images/seconds. Both accuracies are identical. Setup: Appendix C

3. A mapping of the abilities of parallelization of your program must be done to determine the best configuration of cores/instances. An example of code to test this is provided (Appendix A and Appendix B)

To reinforce these points, look at the following figures from our experimentations on ResNet-50[1]:

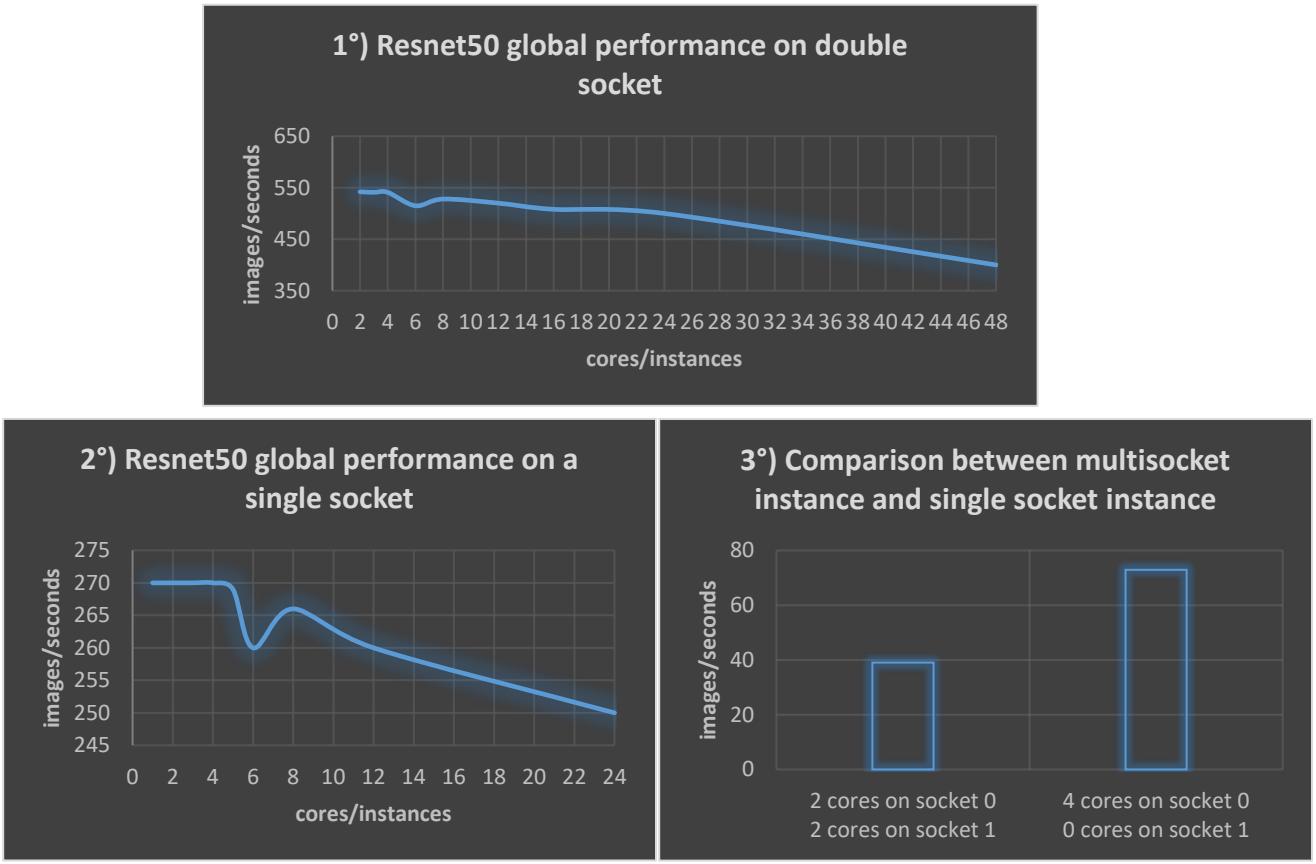


Figure 9 Key measures in multi-instances

It is important to explain the figures in these graphs. The calculations are run on two 24-core processors[2]. `numactl` enables us to either run our program on a single socket or on both. When multiple instances are run, the code provided returns the performance of each instance and since they are running simultaneously, the global performance is the sum of each single instance performance. As the first and second graphs show, the process of inferencing is highly parallelizable since the throughput on a double socket is exactly twice that of a single socket. Therefore, this method is perfectly scalable. However, the computations that were run during inference on our ResNet-50 are not so parallelizable; it is difficult for an instance to take advantage of more than 4 allocated cores. We can use the formula below to find the number of instances to run to achieve the greatest efficiency:

(Number of cores) / (Ideal number of cores per instance) = (Number of instances to run)

[1] Inference with dummy data on optimized ResNet-50 fp32. Accuracies are identical. Setup: Appendix C
[2] Intel® Xeon® Gold 6252

However, the cores must not be allocated randomly to each instance. As we indicated earlier, there are certain rules to follow to be as efficient as possible.

II. How to launch multiple instances (detailed)

1. Map your processor

To map your processor, run the following command:

```
shell$ lscpu
```

From all the data that is then displayed, note the following specifically:

```
CPU(s):              96
On-line CPU(s) list: 0-95
NUMA node0 CPU(s):   0-23,48-71
NUMA node1 CPU(s):   24-47,72-95
```

Ensure you have more than 1 core to work on. If this is not the case, there is no point in running multiple instances.

It is important to highlight that a bi-socket server (two CPUs) is used in this example (the most complex case). Each CPU has 24 physical cores that can each be split into 2 logical cores, which means 48 logical cores for each CPU. Consequently, the system detects the 96 logical cores of the two CPUs. The system shows that these cores are numbered between 0 and 95, but most importantly it shows which core belongs to which CPU. Thus, the cores numbered between 0 and 23, and between 48 and 71 belong to the CPU of the socket 0. The others belong to the CPU of the socket 1. You can therefore deduce that logical core 0 and 48 result from physical core 0, logical core 1 and 49 result from physical core 1, and so on. The same logic applies to the socket 1 CPU cores. The following picture shows how this numbering applies.

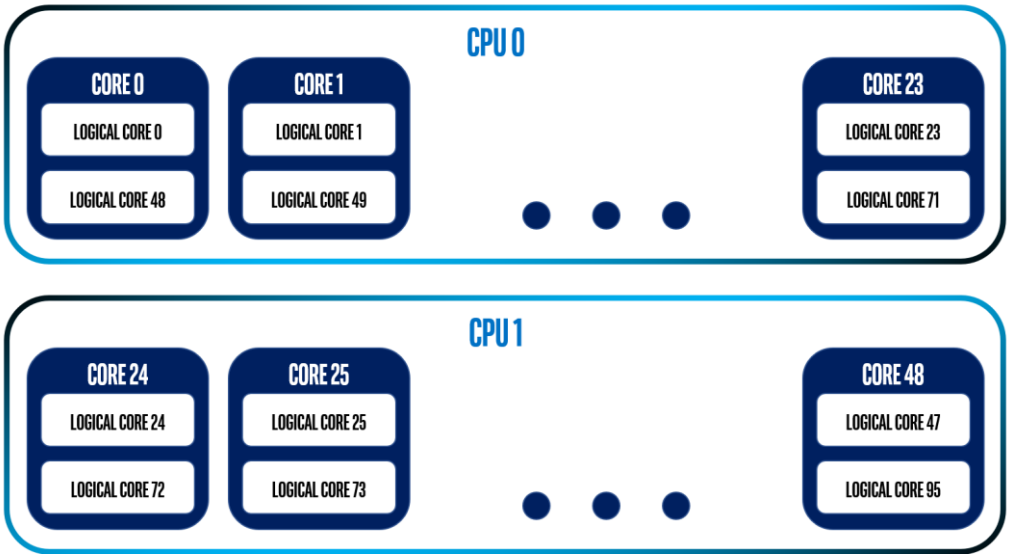


Figure 10 Representation of the cores and logical cores

2. Launch multiple instances

• Enabling your Python code to set environment variables dynamically

In this step, you will learn how to launch multiple instances to increase performance. First, you need to modify your program to enable core allocation. More specifically, your code must be able to dynamically allocate the cores to each instance. However, while the modifications (see below) may look complex, they are in fact quite simple.

Earlier, you were instructed to add the following lines to your Python code. Now, the highlighted part must be changed:

```
import os
os.environ["KMP_BLOCKTIME"] = "1"
os.environ["KMP_SETTINGS"] = "1"
os.environ["KMP_AFFINITY"]= "granularity=fine,verbose,compact,1,0"
if FLAGS.num_intra_threads > 0:
  os.environ["OMP_NUM_THREADS"]=  # number of physical cores
```

Delete these highlighted lines and replace them with the lines below:

```
arg_parser = ArgumentParser(description='Parse args')

arg_parser.add_argument('-k', "--kmp-affinity", help='Specify the kmp-affinity
to tune performances', dest="kmp_affinity",
default="granularity=fine,verbose,compact,1,0")

arg_parser.add_argument('-c', "--physical-core", help='Specify the number of
physical core', dest="physical_core", default=48, type=check_positive_number)

args = arg_parser.parse_args()

os.environ["KMP_AFFINITY"] = args.kmp_affinity
os.environ["OMP_NUM_THREADS"] = str(args.physical_core)
```

If you already parse arguments in your Python code, be sure to integrate these lines in your parser. If you need further information about where to add them, example custom code is provided in **Appendix B**. These lines enable you to set up certain environment variables dynamically when you launch your Python code in the command line or via a Bash file. The variables to set up are the `KMP_AFFINITY` environment variable and `OMP_NUM_THREAD`. You can also use your own approach here if you prefer. As you can see, the `KMP_AFFINITY` default setting is the same as the previous one.

• Launching multiple instances via a Bash file

Create a Bash file to facilitate the mapping of the abilities of parallelization of your program. By creating a Bash file, you will not need to copy and paste lines in your command line each time you want to launch multiple instances.

Create a Bash file and open it:

```
$ touch multisocket.sh
$ vim multisocket.sh
```

Add the shebang at the beginning of the file (yours might be different than ours) and copy-paste these lines:

```
#!/bin/bash

# For reference
# numactl    -N = --cpunodebind
#            -m = --membind
# python     -k = --kmp-affinity
#            -c = --physical-core

# For 4 instances
nohup unbuffer numactl -N 0 -m 0 python eval_image_classifier_inference.py -k
"granularity=thread,proclist=[0-11,48-59],explicit,verbose,compact,1,0" -c 12
&
nohup unbuffer numactl -N 0 -m 0 python eval_image_classifier_inference.py -k
"granularity=thread,proclist=[12-23,60-71],explicit,verbose,compact,1,0" -c 12
&
nohup unbuffer numactl -N 1 -m 1 python eval_image_classifier_inference.py -k
"granularity=thread,proclist=[24-35,72-83],explicit,verbose,compact,1,0" -c 12
&
nohup unbuffer numactl -N 1 -m 1 python eval_image_classifier_inference.py -k
"granularity=thread,proclist=[36-47,84-95],explicit,verbose,compact,1,0" -c 12
&
```

The explanations for these commands are as follows:

- First, replace `eval_image_classifier_inference.py` by the name of your own Python code file.
- `nohup unbuffer` enables you to run multiple commands at once and save their output in a nohup.out file.
- `numactl -N 0 -m 0` enables you to constrain your code to use specific components.
  - `-N 0` constrains your Python code to run on the CPU of the socket 0.
  - `-m 0` then constrains your code to use the memory of the socket 0: this must be specified. Otherwise, if the cores of the socket 0 start to work with the memory of the socket 1, it could lead to a non-negligible increase in the latency as shown in Figure 9.

Concerning the arguments of the Python code, you added a few lines that enable you to set up certain environment variables in the command line. The settings `granularity=thread` and `explicit,verbose,compact,1,0` are recommended for running TensorFlow. Note also the addition of `proclist=[0-11,48-59]` and its variants (the values change at each command line) which refer to the logical cores allocated to each specific instance. The values of this setting are critical to achieve better performance. As you can see, in each case the range of logical cores to use is associated with its

"sibling" range of logical cores. Thus, the range "0-11" is associated with the range "48-59", and so on. The goal is to have the logical cores of each range on the same socket.

Finally, `-c 12` refers to the number of physical cores to use to run the instance. As determined earlier, the best setting with ResNet-50 was to use 4 cores per instance. You can see what such commands might look like in Appendix A.

Note: you must adapt `proclist=[0-11,48-59]`, `-c 12,` and `-N 0 -m 0` to your own setup!

Activate your conda environment (**Section 4**) if you have not already done so and then launch multiple instances:

```
$ conda activate py-3.7-tf-2.1
$ ./multisocket.sh
```

3. Map your performance

Mapping your code performance consists of finding the ideal number of cores per instance. As an example, to map the performance of a ResNet-50[1] with the codes provided in Appendix A and Appendix B, do the following:

- Follow the tutorial in **Section 9 - With provided code.**
- Launch your code for a given number of instances.
- Check your results by opening `nohup.out`. The performance of each instance is measured in images/seconds. Since all the instances were running simultaneously, you can calculate the overall performance simply by summing up the performance of each instance.
- Change the number of instances by editing the Bash file and iterate to find the ideal number of cores per instance.

4. Apply this method to a real case

When you apply this method, it is important to use a separate data stream for each instance. If this is not done, each instance will have the same data stream and will do inference on the same data. Consequently, your performance will be far worse than a basic launch of your program since you are asking each instance to process the same data.

To make multiple separate data streams you can proceed in the following way:

1. Divide your dataset in as many subsets as the number of instances you want to run. Note that each subset needs to contain unique data not shared by other subset to avoid processing the same data multiple times. Dividing the dataset in unique subsets can be done easily with a Python code.
2. Enable your Python code to take the subset to process as an argument. This can be done by adding an argument to your parser. It might look like this:

```
arg_parser.add_argument('-d', "--dataset", help='Specify the dataset to
process', dest="dataset", default="")
```

[1] https://storage.googleapis.com/intel-optimized-tensorflow/models/v1_6/resnet50_fp32_pretrained_model.pb

3. Change the Bash file (called "multisocket.sh" in our tutorials) and add the argument of the subset. Each instance must process a different subset than all the others. It might look like this:

```
#!/bin/bash

#For 12 instances

nohup unbuffer numactl -N 0 -m 0 python eval_image_classifier_inference.py -k
"granularity=thread,proclist=[0-3,48-51],explicit,verbose,compact,1,0" -c 4 -d
subset1/ &

nohup unbuffer numactl -N 0 -m 0 python eval_image_classifier_inference.py -k
"granularity=thread,proclist=[4-7,52-55],explicit,verbose,compact,1,0" -c 4 -d
subset2/ &
.
.
.
```

This concludes the section on how to launch multi-instances to process a dataset more quickly.

III. **How** to launch multiple instances (summary)

- Map your processor(s) by running

```
shell$ lscpu
```

- Your command will return information. The information that matters is the following:

```
CPU(s):              96
On-line CPU(s) list: 0-95
NUMA node0 CPU(s):   0-23,48-71
NUMA node1 CPU(s):   24-47,72-95
```

Ensure you have more than 1 core to work on. If this is not the case, there is no point in running multiple instances.

- Previously, in **Section 5** you were instructed to add the following lines to your Python code. Delete the highlighted part:

```
import os
os.environ["KMP_BLOCKTIME"] = "1"
os.environ["KMP_SETTINGS"] = "1"
os.environ["KMP_AFFINITY"]= "granularity=fine,verbose,compact,1,0"
if FLAGS.num_intra_threads > 0:
  os.environ["OMP_NUM_THREADS"]= # number of physical cores
```

and replace it by the following lines (you can reorganize your code if you prefer):

```
arg_parser = ArgumentParser(description='Parse args')
```

```
arg_parser.add_argument('-k', "--kmp-affinity", help='Specify the kmp-
affinity to tune performances', dest="kmp_affinity",
default="granularity=fine,verbose,compact,1,0")

arg_parser.add_argument('-c', "--physical-core", help='Specify the number
of physical core', dest="physical_core", default=48,
type=check_positive_number)

args = arg_parser.parse_args()

os.environ["KMP_AFFINITY"] = args.kmp_affinity
os.environ["OMP_NUM_THREADS"] = str(args.physical_core)
```

- To launch two instances (without creating a Bash file here), use one of the following approaches.
    - For a double socket, run the following (adapt the commands to your setup):

```
nohup unbuffer numactl -N 0 -m 0 python eval_image_classifier_inference.py
-k "granularity=thread,proclist=[0-23,48-71],explicit,verbose,compact,1,0"
-c 24 &
nohup unbuffer numactl -N 1 -m 1 python eval_image_classifier_inference.py
-k "granularity=thread,proclist=[24-47,72-95],explicit,verbose,compact,1,0"
-c 24 &
```

    - For a single socket, run the following (adapt the commands to your setup):

```
nohup unbuffer python eval_image_classifier_inference.py -k
"granularity=thread,proclist=[0-11,48-59],explicit,verbose,compact,1,0" -c
12 &
nohup unbuffer python eval_image_classifier_inference.py -k
"granularity=thread,proclist=[12-23,60-71],explicit,verbose,compact,1,0" -c
12 &
```

- In the lines above you must adapt the `proclist` argument to your own configuration (brought to light by running `lscpu`). If you need to better understand the underlying logic of the arguments, refer to first two parts of this section for more details. Also, change the Python code name to the name of your own Python code.
- Now that you know how to launch multiple instances, you can map your model performance to determine the best number of cores per instance. Note that if you are working on a CPU with more than 8 cores, 2 instances is rarely the best number of instances.
- You should also look at:

  I. How to launch multiple instances (detailed) - 4.Apply this method to a real case to learn how to process real data with this method.

Section 8. Quantization

A further way to increase your inference performance is to infer on a quantized model. Quantization is the process of reducing the precision of the numerical format used in the neural network computations. In deep learning, it is generally done by converting FP32 models into INT8 or BF16 models. Quantization allows you not only to shrink the size of your models, but also to boost their performance. It also has been shown that accuracy is almost unaffected by a properly done quantization. There are multiple ways of quantizing a model. You can either perform a post-training quantization or a quantization aware training. Both methods have pros and cons. However, since we are not discussing training in this paper, we will assume a choice of post-training quantization.

There are many post-training quantization techniques. TensorFlow offers the following ones:

- Post-training float16 quantization
- Post-training int8 dynamic range quantization
- Post-training int8 quantization

To increase performance when inferencing, your processors must support hardware acceleration for INT8 like the latest Intel Xeon SP processor (code name Cascade Lake) does. Be sure to check your hardware for this support.

Quantization is not a fully automated process yet. No tool currently available can guarantee to fully efficiently quantize any existing neural network. However, this paper gives you a non-exhaustive state of the art of the different ways to quantize your neural network to help you quantize your model effectively. Note that some tools are designed to produce models specially optimized for specific types of devices, like TensorFlow Lite for mobile devices. Therefore, it is important to define your final use case before choosing a tool to quantize your model.

In this paper, the Intel quantization tool API was used to quantize a ResNet-50[1] and we therefore include a tutorial showing how to do it. The goal of our approach in this paper is to show you how powerful the efficient quantization of your model can be.

Quantize with TensorFlow tools

For quantization with TensorFlow, most resources currently focus on TensorFlow Lite. However, the TensorFlow Lite tool produces a ".tflite" model that is optimized for mobile devices rather than for x86 CPUs. During the recent years, TensorFlow has released many other converting tools that keep models in the ".pb" format. The tool maintained in the last version of TensorFlow (currently 2.1) is the Transform Graph tool. Even though that tool is now starting to be deprecated for quantization, we will present it as it could suit your need.

Quantize using TensorFlow Lite

TensorFlow Lite is an open source deep learning framework specially designed for on-device inference. It converts ".pb" models into ".tflite" models while optimizing them.

If your goal is to do inference on a server, you should not quantize with TensorFlow Lite. On a server with x86 CPUs, the resulting ".tflite" file will give poorer performance than a classic ".pb" file.

[1] https://storage.googleapis.com/intel-optimized-tensorflow/models/v1_6/resnet50_fp32_pretrained_model.pb

Moreover, TFLite does not currently take advantage of numerous cores. On the other hand, if your goal is to do on-device inference, TensorFlow Lite will be the better choice. TensorFlow Lite takes energy efficiency into account and outperforms ".pb" files on mobile devices. They are many tutorials explaining how to quantize a model and as the tool is well designed, it requires very few programming skills.

Quantize with TensorFlow Transform Graph

Previously, we showed you how to optimize your model by using Transform Graph. It is also possible to quantize your model with this tool. The section **Eight-bit Calculations**[1] shows explicitly how to quantize with Transform Graph. However, this tool starts to be deprecated and, as underlined under the command line of the section **Eight-bit Calculations**[1], "*Only a subset of ops are supported, and on many platforms the quantized code may actually be slower than the float equivalents, but this is a way of increasing performance substantially when all the circumstances are right.*"

The tool is nonetheless easy to use, and if it corresponds to your use case, you can try it. In the right circumstances with a good fit for your model, it can save you time. Otherwise, other options exist.

Quantize with Intel tools

Intel has also developed tools to quantize models for TensorFlow. These tools are specifically designed to take full advantage of Intel hardware, and therefore we recommend their use to achieve the best performance. In this paper, we describe the **Intel quantization tool API**[2] and **OpenVINO**. The **Intel quantization API** was used to quantize the ResNet-50[3] model, and therefore a detailed tutorial is included for that tool. OpenVINO is also an effective tool that meets many different needs.

Quantize using the Intel quantization tool API

This tool is under active development. Its goals are to:

> - Unify the quantization tools calling entry
> - Remove the Tensorflow source build dependency
> - Transparent the model quantization process
> - Reduce the quantization steps
> - Seamlessly adapt to inference with Python script.
>
> *Quotation 1 https://github.com/IntelAI/tools/tree/master/api*

[1] https://github.com/tensorflow/tensorflow/blob/master/tensorflow/tools/graph_transforms/README.md#eight-bit-calculations

[2] https://github.com/IntelAI/tools/tree/master/api

[3] https://storage.googleapis.com/intel-optimized-tensorflow/models/v1_6/resnet50_fp32_pretrained_model.pb

It is also advised to use the TensorFlow MKLDNN (oneDNN) enabled version since:

> The Quantization Programming APIs are specified for Intel Optimizations for TensorFlow based on the MKLDNN enabled build. This APIs call the TensorFlow Python models as extension, and provide some special fusion rules, such as fold_convolutionwithbias_mul, fold_subdivmul_batch_norms, fuse_quantized_conv_and_requantize, mkl_fuse_pad_and_conv, rerange_quantized_concat etc.

When you install TensorFlow from the official Anaconda command or link, the MKLDNN (oneDNN) enabled version is usually installed by default and without further notification. For deeper insight on the subject, please refer to **Section 4**.

Currently, this tool can quantize numerous models from the Model Zoo for Intel Architecture and multiple official classification models present on TensorFlow's list. The intention is that it will soon be able to quantize any model, including custom models.

In our case the improvement in performance was significant. With our setup (Appendix C), the throughput is 90 images/second with a classic FP32 ResNet-50. With our quantized ResNet-50, it is 1040 images/second. We even achieved **1700 images/second** by running multiple instances (**Section 7**). Inference performance was improved by **up to 18X**.

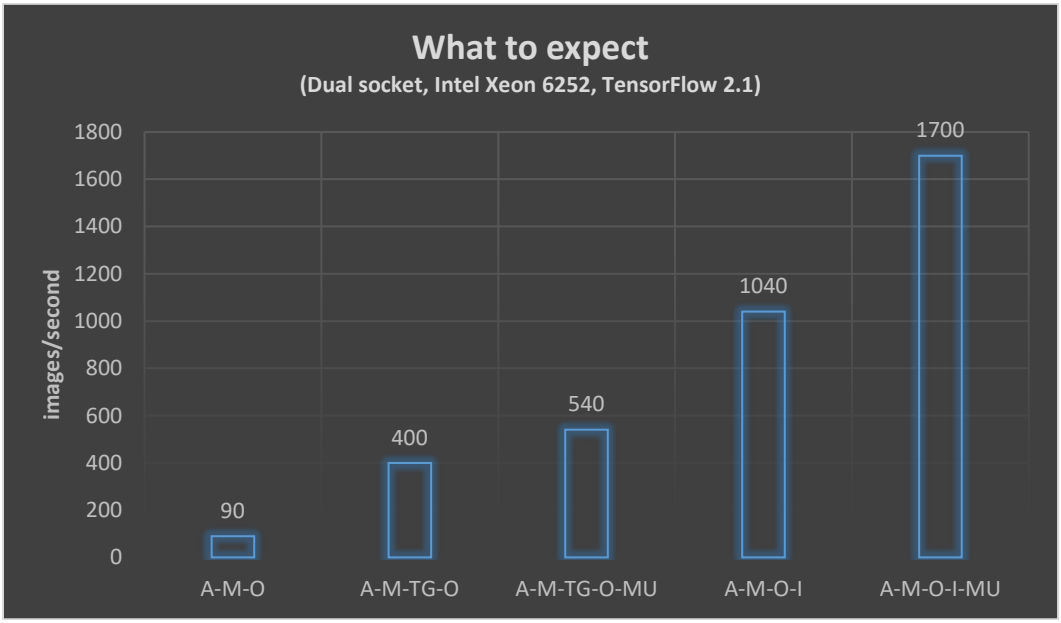


*Figure 11 - What you can expect from the different optimizations. **A**: AVX-512 activated, **M**: oneDNN (formerly MKL-DNN) enabled, **TG**: Transform Graph optimized model, **O**: intra-threads and inter-threads optimized, **I**: INT8 quantized, **MU**: Multiple instances launched*

Accuracy remains almost unaffected by quantization, with only 0.7% loss.

ResNet-50 FP32 Accuracy		ResNet-50 INT8 Accuracy	
Top 1	Top 5	Top 1	Top 5
0.7430	0.9188	0.7374	0.9155

Now that you know how much you can gain by quantizing your neural network, we will show you how to quantize a FP32 ResNet-50[1]. First, create a directory "*api_quantization_intel*" to put all the new files in, and a new conda environment to protect your existing environment from modification:

```
shell$ mkdir api_quantization_intel
shell$ cd api_quantization_intel/
shell$ conda create --name py-3.7-tf-1.15-intel python=3.7
shell$ conda activate py-3.7-tf-1.15-intel
```

Install the right version of TensorFlow and the intel-quantization toolkit:

```
shell$ conda install tensorflow=1.15.0 -c intel
shell$ conda install -c intel intel-quantization
```

Check that you installed the right version of TensorFlow (1.15.0):

```
shell$ python -c "import tensorflow as tf; print(tf.__version__)"
shell$ python -c "import tensorflow;
print(tensorflow.pywrap_tensorflow.IsMklEnabled())"
```

Download the Model Zoo directory and check its version:

```
shell$ git clone https://github.com/IntelAI/models.git models && cd models
shell$ git checkout v1.5.0
```

Download the quantization tool repository and the FP32 ResNet-50 pretrained model:

```
shell$ cd ../
shell$ git clone https://github.com/IntelAI/tools.git  quantization && cd
quantization
shell$ cd api/models/resnet50
shell$ wget https://storage.googleapis.com/intel-optimized-
tensorflow/models/resnet50_fp32_pretrained_model.pb
shell$ cd ../../../
```

Finally, quantize your model:

```
shell$ python api/examples/quantize_model_zoo.py \
--model resnet50 \
--in_graph
/PATH_TO_QUANTIZATION_DIR/api_quantization_intel/quantization/api/models/resne
t50/resnet50_fp32_pretrained_model.pb \
--out_graph
/<path_to_quantization_dir>/api_quantization_intel/quantization/api/models/res
net50/api_quantized_resnet50.pb \
--data_location /<path_to_data_dir>/ \
--models_zoo_location
/<path_to_quantization_dir>/api_quantization_intel/models
```

[1] https://storage.googleapis.com/intel-optimized-tensorflow/models/v1_6/resnet50_fp32_pretrained_model.pb

Note the need for data in the quantization command. You will need to pre-process the 2012 ImageNet validation dataset. Several tutorials for this are available on the web.

Quantize with OpenVINO™

OpenVINO™ is a toolkit developed by Intel, focused on image processing deep learning algorithms. It is designed to ease deployment and improve deep learning inference at the edge. Unlike training that almost always takes place in data centers, inference may also be done on an embedded platform. OpenVINO™ is optimized for a wide list of supported hardware platforms for edge computing deployment. If this corresponds to your use-case, we suggest that you try it.

OpenVINO™ is a complete tool that on its own would take an entire white paper to discuss. OpenVINO™ is not only focused on inferencing with TensorFlow but can also be used with many other deep learning frameworks too, including Caffe, MXNet, ONNX, and Kaldi. The OpenVINO™ toolkit comes with a Deep Learning Model Optimizer, a Deep Learning Inference Engine, samples showing how to use the Inference Engine and a set of tools to facilitate your work on your models.

However, quantization is one of its most relevant features. OpenVINO™ lets you quantize a large number of models from supported frameworks. You can check the supported models here:

https://docs.openvinotoolkit.org/latest/_docs_IE_DG_Introduction.html#supported_models

Performance is outstanding in most cases which makes OpenVINO™ a "must-try" inference engine. For example, it achieves a performance of 971 images/second while inferencing on a Caffe ResNet-50 on a single Intel Xeon Gold 5218T[1]. Even though this result is not directly comparable to our results, considering that we are not working on the same framework, it is an example of the level of performance that OpenVINO™ can achieve. The OpenVINO™ official website also offers many quality tutorials that are easy to use, so we have not included any additional tutorials in this paper.

[1] https://docs.openvinotoolkit.org/latest/_docs_performance_benchmarks.html#xeon-gold

Section 9. Performance checking

This section will show you two different ways to run a benchmark on a ResNet-50 using TensorFlow and exploiting all the potential of your Intel hardware.

The first method is to use Model Zoo for Intel Architecture. We provide a quick tutorial on how to use the Model Zoo tool on a ResNet-50 in this paper. With basic understanding of this tool, you should be able to run a benchmark with any model that is supported by Model Zoo.

The second method is to check performance with the code provided in **Appendix A** and **Appendix B**, which is a simplified version of Model Zoo. Unlike the tutorial for Model Zoo, the tutorial for this second method will enable you to try multi-instance. You will have to modify the code to run the benchmark on another model. No dataset is needed in either tutorial since both tools generate dummy data.

With Model Zoo for Intel Architecture

Model Zoo for Intel Architecture enables you to test your model and see how it can benefit from optimization and quantization. However, you will not be able to try multi-instance. To do so, go to **With provided codes**.

Environment

There are multiple ways to install and use Model Zoo, e.g. via a container. If you prefer a different way than the one described below, please visit the Model Zoo website.

1. Prerequisites

We used Anaconda for this tutorial and therefore you should also use it to make sure that you get the results intended. Unlike the previous tutorials, some functions used to run the benchmark by Model Zoo are not available in TensorFlow 2. For this reason, you will install TensorFlow 1.15 (latest 1.xx TensorFlow) instead. However, the optimizations made on your model using TensorFlow 1.xx are also supported by TensorFlow 2.xx and you will be able to achieve similar performance with TensorFlow 2. If you do not already have Anaconda installed, refer to **Section 4**. Create and activate a conda environment, and install TensorFlow 1.15:

```
shell$ conda create --name py-3.7-tf-1.15 python=3.7
shell$ conda activate py-3.7-tf-1.15
shell$ conda install tensorflow=1.15 -c anaconda
```

2. Git clone Model Zoo and pre-trained ResNet-50 model

Now, clone the Model Zoo repository. If not already installed, install git. After cloning the Model Zoo repository, you will see a directory called “models”: this will be your new working directory. Rename it “work_dir” since it contains a directory also called “models”. Model Zoo is organized in a specific way and we recommend that you take the time to see how.

```
shell$ git clone https://github.com/IntelAI/models.git
shell$ cp -r models/ work_dir/
shell$ rm -r models/
```

Before running the command `rm -r models`, you should check if "work_dir/" contains the same directories and files as "models/".

You can choose any model that is supported by Model Zoo. For this tutorial, download into your "models" directory the following ResNet-50 FP32 pre-trained model:

```
shell$ cd work_dir
shell$ wget https://storage.googleapis.com/intel-optimized-
tensorflow/models/v1_5/resnet50_fp32_pretrained_model.pb
```

Before continuing, check your TensorFlow version, and check if MKL is enabled to run the inference:

```
shell$ python -c "import tensorflow as tf; print(tf.__version__)"
shell$ python -c "import tensorflow;
print(tensorflow.pywrap_tensorflow.IsMklEnabled())"
```

Running the benchmark

At this point, you should have all the files needed. You may have noticed that "work_dir" contains 4 directories and various files. The directories relevant to this tutorial are "~/work_dir /benchmarks/" and "~/work_dir /models/".

Go to the benchmark directory:

```
shell$ cd benchmarks
```

In that directory, there is a Python file called "launch_benchmark.py". To launch the benchmark, run:

```
shell$ python launch_benchmark.py \
       --in-graph ../resnet50_fp32_pretrained_model.pb \
       --model-name resnet50 \
       --framework tensorflow \
       --precision fp32 \
       --mode inference \
       --batch-size=128 \
       --num-inter-threads=2 \
       --num-intra-threads=2 \
       --warmup-steps=4 \
       --data-num-inter-threads=2 \
       --data-num-intra-threads=2
```

Going deeper with Model Zoo

Model Zoo comes with more than 20 different pre-trained neural networks that you can try:
https://www.intel.com/content/www/us/en/artificial-intelligence/posts/model-zoo-ia.html
https://github.com/IntelAI/models/tree/master/benchmarks

With provided codes

The custom code provided (Appendix B) is inspired by the Model Zoo code. However, it is more compact to make it easier to understand and it enables multi-instancing, letting you reproduce the conditions for the best performance. You may not be able to reproduce exactly the same levels of performance as the ones in this paper, as you would need exactly the same setup (Appendix C). However, you will be able to push your hardware to its limits in the same way that we did.

Create a working directory and activate the conda environment created in **Section 4**:

```
shell$ mkdir work_dir
shell$ cd work_dir/
shell$ conda activate py-3.7-tf-2.1
```

Create a Bash file and open it (called multisocket.sh in this tutorial):

```
shell$ touch multisocket.sh
shell$ vim multisocket.sh
```

Copy-paste the Bash code of Appendix A into the file. Adapt the code to your system; refer to **Section 7** if you do not know how to adapt it. Save and quit.

Create a Python file and open it (called "eval_image_classifier_inference.py" in this tutorial):

```
shell$ touch eval_image_classifier_inference.py
shell$ vim eval_image_classifier_inference.py
```

Copy-paste the Python code of Appendix B in the file. Replace the model name by yours (line 21, the `INPUT_GRAPH` argument). Save and quit. Then copy your model into "work_dir".

```
shell$ cp <PATH TO YOUR MODEL>/your_model.pb ./
```

Open a second terminal and run:

```
shell$ top
```

The second terminal will show you how busy your CPU is, and you will be able to see when your Python code finished executing. Now, in the first terminal, run the code:

```
shell$ ./multisocket.sh
```

Now, you should see a Python process consuming almost all your CPU capacity. Once the Python code has finished executing (it should disappear from the `top` table), check your results in the first terminal by running:

```
shell$ vim nohup.out
```

Scroll down, and at the end of the file, you should see multiple throughputs in images/second. They correspond to the throughputs of the different instances. Sum them to get your total throughput.

Section 10. **Resources**

Popular Deep Learning Frameworks: optimized version installation method
https://software.intel.com/en-us/frameworks/tensorflow

oneDNN GitHub
https://github.com/oneapi-src/oneDNN/

KMP setting details
https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-thread-affinity-interface-linux-and-windows

[Quantization]
Low-Precision 8-bit Integer Inference with OpenVINO
https://docs.openvinotoolkit.org/latest/_docs_IE_DG_Int8Inference.html

Accelerate Lower Numerical Precision Inference with Intel® Deep Learning Boost
https://software.intel.com/en-us/articles/accelerate-lower-numerical-precision-inference-with-intel-deep-learning-boost

Get Started with Intel® Deep Learning Boost and the Intel® Distribution of OpenVINO™ Toolkit
https://software.intel.com/en-us/articles/get-started-with-intel-deep-learning-boost-and-the-intel-distribution-of-openvino-toolkit

Section 11. **Table of figures**

Section 12. **Appendices**

Appendix A

Example of Bash script to launch multiple instances. This example is based on 2 CPUs, each with 24 physical cores (48 logical cores).

```
#!/bin/bash

#For 12 instances

nohup unbuffer numactl -N 0 -m 0 python eval_image_classifier_inference.py -k
"granularity=thread,proclist=[0-3,48-51],explicit,verbose,compact,1,0" -c 4 &

nohup unbuffer numactl -N 0 -m 0 python eval_image_classifier_inference.py -k
"granularity=thread,proclist=[4-7,52-55],explicit,verbose,compact,1,0" -c 4 &

nohup unbuffer numactl -N 0 -m 0 python eval_image_classifier_inference.py -k
"granularity=thread,proclist=[8-11,56-59],explicit,verbose,compact,1,0" -c 4 &

nohup unbuffer numactl -N 0 -m 0 python eval_image_classifier_inference.py -k
"granularity=thread,proclist=[12-15,60-63],explicit,verbose,compact,1,0" -c 4 &

nohup unbuffer numactl -N 0 -m 0 python eval_image_classifier_inference.py -k
"granularity=thread,proclist=[16-19,64-67],explicit,verbose,compact,1,0" -c 4 &

nohup unbuffer numactl -N 0 -m 0 python eval_image_classifier_inference.py -k
"granularity=thread,proclist=[20-23,68-71],explicit,verbose,compact,1,0" -c 4 &

nohup unbuffer numactl -N 1 -m 1 python eval_image_classifier_inference.py -k
"granularity=thread,proclist=[24-27,72-75],explicit,verbose,compact,1,0" -c 4 &

nohup unbuffer numactl -N 1 -m 1 python eval_image_classifier_inference.py -k
"granularity=thread,proclist=[28-31,76-79],explicit,verbose,compact,1,0" -c 4 &

nohup unbuffer numactl -N 1 -m 1 python eval_image_classifier_inference.py -k
"granularity=thread,proclist=[32-35,80-83],explicit,verbose,compact,1,0" -c 4 &

nohup unbuffer numactl -N 1 -m 1 python eval_image_classifier_inference.py -k
"granularity=thread,proclist=[36-39,84-87],explicit,verbose,compact,1,0" -c 4 &

nohup unbuffer numactl -N 1 -m 1 python eval_image_classifier_inference.py -k
"granularity=thread,proclist=[40-43,88-91],explicit,verbose,compact,1,0" -c 4 &

nohup unbuffer numactl -N 1 -m 1 python eval_image_classifier_inference.py -k
"granularity=thread,proclist=[44-47,92-95],explicit,verbose,compact,1,0" -c 4 &
```

Appendix B

A simplified version of Model Zoo for Intel Architecture to run inference on a ResNet-50 with dummy data.

```
import os
import time
import numpy as np
from argparse import ArgumentParser

import tensorflow as tf
from tensorflow.python.platform import gfile


##### Environment Variables #####
os.environ["KMP_BLOCKTIME"] = "1"
os.environ["KMP_SETTINGS"] = "1"

##### Inference Variables #####
RESNET_IMAGE_SIZE = 224
BATCH_SIZE = 512
NUM_INTER_THREADS = 1
DATA_NUM_INTER_THREADS = 1
STEPS = 50
WARM_UP_STEPS = 4
INPUT_GRAPH = 'api_quantized_resnet50.pb'



def check_positive_number(value):
        if value:
            value = int(value)
            if value == 0 or value < 0:
                raise ArgumentTypeError("{} is not valid.".format(value))
        return value


arg_parser = ArgumentParser(description='Parse args')

arg_parser.add_argument('-k', "--kmp-affinity", help='Specify the kmp-affinity to tune
performances', dest="kmp_affinity", default="granularity=fine,verbose,compact,1,0")

arg_parser.add_argument('-c', "--physical-core", help='Specify the number of physical
core', dest="physical_core", default=48, type=check_positive_number)

args = arg_parser.parse_args()

os.environ["KMP_AFFINITY"] = args.kmp_affinity
os.environ["OMP_NUM_THREADS"] = str(args.physical_core)
DATA_NUM_INTRA_THREADS = args.physical_core


class eval_classifier_optimized_graph:
    """Evaluate image classifier with optimized TensorFlow graph"""
```

```
################################################################################
############################## Running Inference ###############################
################################################################################

    def run(self):
        """run benchmark with optimized graph"""


        print("Run inference")

        data_config = tf.compat.v1.ConfigProto()
        data_config.intra_op_parallelism_threads = DATA_NUM_INTRA_THREADS
        data_config.inter_op_parallelism_threads = DATA_NUM_INTER_THREADS
        data_config.use_per_session_threads = 1

        infer_config = tf.compat.v1.ConfigProto()
        infer_config.intra_op_parallelism_threads = DATA_NUM_INTRA_THREADS
        infer_config.inter_op_parallelism_threads = NUM_INTER_THREADS
        infer_config.use_per_session_threads = 1

        data_graph = tf.Graph()
        with data_graph.as_default():
            print("Inference with dummy data.")
            input_shape = [BATCH_SIZE, RESNET_IMAGE_SIZE, RESNET_IMAGE_SIZE, 3]
            images = tf.random.uniform(input_shape, 0.0, 255.0, dtype=tf.float32,
name='synthetic_images')

        infer_graph = tf.Graph()
        with infer_graph.as_default():
            graph_def = tf.compat.v1.GraphDef()
            with tf.io.gfile.GFile(INPUT_GRAPH, 'rb') as input_file:
                input_graph_content = input_file.read()
                graph_def.ParseFromString(input_graph_content)

            tf.import_graph_def(graph_def, name='')

        # Definite input and output Tensors for detection_graph
        input_tensor = infer_graph.get_tensor_by_name('input:0')
        output_tensor = infer_graph.get_tensor_by_name('predict:0')

        data_sess = tf.compat.v1.Session(graph=data_graph, config=data_config)
        infer_sess = tf.compat.v1.Session(graph=infer_graph, config=infer_config)

        num_processed_images = 0
        num_remaining_images = (BATCH_SIZE * STEPS)


################################################################################
################################## Performance #################################
################################################################################

        iteration = 0
        warm_up_iteration = WARM_UP_STEPS
        total_run = STEPS
        total_time = 0

        while num_remaining_images >= BATCH_SIZE and iteration < total_run:
```

```
            iteration += 1
            tf_filenames = None
            np_labels = None
            data_load_start = time.time()
            image_np = data_sess.run(images)

            data_load_time = time.time() - data_load_start

            num_processed_images += BATCH_SIZE
            num_remaining_images -= BATCH_SIZE

            start_time = time.time()
            predictions = infer_sess.run(output_tensor, feed_dict={input_tensor:
image_np})
            time_consume = time.time() - start_time


            print('Iteration %d: %.6f sec' % (iteration, time_consume))
            if iteration > warm_up_iteration:
                total_time += time_consume

        time_average = total_time / (iteration - warm_up_iteration)
        print('Average time: %.6f sec' % (time_average))

        print('Batch size = %d' % BATCH_SIZE)
        if (BATCH_SIZE == 1):
            print('Latency: %.3f ms' % (time_average * 1000))
        print('Throughput: %.3f images/sec' % (BATCH_SIZE / time_average))


if __name__ == "__main__":
    evaluate_opt_graph = eval_classifier_optimized_graph()
    evaluate_opt_graph.run()
```

Appendix C

2.9x inference performance for image classification-ResNet-50

Config1: Tested by Intel as of 6/29/2020. 2 socket Intel Xeon Gold 6252 Processor, 24 cores HT On Turbo ON Total Memory 1536 GB (12 slots/ 64GB/ 2666 Hz), BIOS: SE5C620.86B.02.01.0010.010620200716 (0x400002c), Ubuntu 18.04.4 LTS 5.3.0-53-generic, Deep Learning Framework: TensorFlow https://github.com/tensorflow/tensorflow/releases/tag/v2.1.0, ResNet50: https://storage.googleapis.com/intel-optimized-tensorflow/models/v1_6/resnet50_fp32_pretrained_model.pb, BS=512, synthetic data, 1 instance/2 sockets, Datatype: FP32, Model Optimized : No (**Section 6**)

Config2: Tested by Intel as of 6/29/2020. 2 socket Intel Xeon Gold 6252 Processor, 24 cores HT On Turbo ON Total Memory 1536 GB (12 slots/ 64GB/ 2666 Hz), BIOS: SE5C620.86B.02.01.0010.010620200716 (0x400002c), Ubuntu 18.04.4 LTS 5.3.0-53-generic, Deep Learning Framework: TensorFlow https://github.com/tensorflow/tensorflow/releases/tag/v2.1.0, ResNet50: https://storage.googleapis.com/intel-optimized-tensorflow/models/v1_6/resnet50_fp32_pretrained_model.pb, BS=512, synthetic data, 1 instance/2 sockets, Datatype: FP32, Model Optimized : Yes (**Section 6**)

Config3: Tested by Intel as of 6/29/2020. 2 socket Intel Xeon Gold 6252 Processor, 24 cores HT On Turbo ON Total Memory 1536 GB (12 slots/ 64GB/ 2666 Hz), BIOS: SE5C620.86B.02.01.0010.010620200716 (0x400002c), Ubuntu 18.04.4 LTS 5.3.0-53-generic, Deep Learning Framework: TensorFlow https://github.com/tensorflow/tensorflow/releases/tag/v2.1.0, ResNet50: https://storage.googleapis.com/intel-optimized-tensorflow/models/v1_6/resnet50_fp32_pretrained_model.pb, BS=512, synthetic data, 12 instances/2 sockets (**Section 7**), Datatype: FP32, Model Optimized : Yes (**Section 6**)

Config4: Tested by Intel as of 6/29/2020. 2 socket Intel Xeon Gold 6252 Processor, 24 cores HT On Turbo ON Total Memory 1536 GB (12 slots/ 64GB/ 2666 Hz), BIOS: SE5C620.86B.02.01.0010.010620200716 (0x400002c), Ubuntu 18.04.4 LTS 5.3.0-53-generic, Deep Learning Framework: TensorFlow https://github.com/tensorflow/tensorflow/releases/tag/v2.1.0, ResNet50: https://storage.googleapis.com/intel-optimized-tensorflow/models/v1_6/resnet50_fp32_pretrained_model.pb, BS=512, synthetic data, 1 instance/2 sockets, Datatype: INT8 (**Section 8**)

Config5: Tested by Intel as of 6/29/2020. 2 socket Intel Xeon Gold 6252 Processor, 24 cores HT On Turbo ON Total Memory 1536 GB (12 slots/ 64GB/ 2666 Hz), BIOS: SE5C620.86B.02.01.0010.010620200716 (0x400002c), Ubuntu 18.04.4 LTS 5.3.0-53-generic, Deep Learning Framework: TensorFlow https://github.com/tensorflow/tensorflow/releases/tag/v2.1.0, ResNet50: https://storage.googleapis.com/intel-optimized-tensorflow/models/v1_6/resnet50_fp32_pretrained_model.pb, BS=512, synthetic data, 12 instances/2 sockets (**Section 7**), Datatype: INT8 (**Section 8**)

END OF DOCUMENT