WHITE PAPER

Accelerating Memory-Bound Machine Learning Models on Intel® Xeon® Processors

Authors

Mattson Thieme (Intel),
Wei Wang (Intel),
Martin Kraus (Siemens Healthineers),
Prashant Shah (Intel)

Abstract

Machine learning models must retrieve and process data from memory during training. Given the type, size, and shape of that data, as well as the compute hardware and software stack, the training time will be gated by either the computation speed (compute-bound) or the data retrieval speed (memory-bound). This article outlines behavior symptomatic of memory-bound deep learning applications, and suggests optimizations which may accelerate training times up to 100X in similar settings.

Introduction

The terms compute-bound and memory-bound indicate which of those two actions - computation or data retrieval - demand the largest portion of the application's runtime, and thus on which action that runtime is gated. Figure 1 provides example signatures for each state.

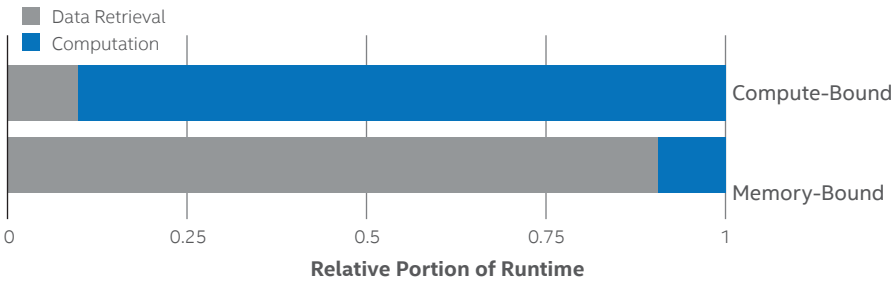


Figure 1. The runtime of compute-bound applications is mostly dependent on the computation, whereas the runtime of memory-bound applications is mostly dependent on the data retrieval speed.

Most optimization techniques assume that the application is compute-bound, but optimizing the compute portion of a memory-bound application will have little to no effect on runtime. Standard recommendations may in fact substantially degrade performance in memory-bound settings, leaving the data scientist unsure about next steps.

The model used in this study was a meta-learning topology with long short-term memory (LSTM) units and FC layers. As we will see in the coming sections, the FC layers play a critical role in the model being memory-bound. However, architectural details of this example topology are less relevant to the present discussion, and as such will not be reviewed.

Table of Contents

Configuration

The model was written in Intel® Optimization for TensorFlow* 1.9[1], which can be installed via the following command:

```
conda install -c anaconda tensorflow
```

We investigate single-node runtime behavior on an Intel® Xeon® Platinum 8124M processor @ 3.00GHz with 36 physical cores and 75 GB of RAM. This node was accessed via a c5.18xlarge AWS* instance running the Deep Learning AMI v10.0[2] with Intel® Hyper-Threading Technology (Intel® HT Technology) and Intel SpeedStep® technology enabled.

***Note:** the size of the RAM was not a constraint in this application.

Problem Signature

This section details training behavior symptomatic of memory-bound models. Optimizations for, and explanations of, such behavior will be discussed in the following section.

1. Counterintuitive response to OMP_NUM_THREADS

The first indicator that a model may not be compute-bound is its runtime with respect to the OMP_NUM_THREADS environment variable. The OMP_NUM_THREADS variable dictates how many threads a Python* program may use at runtime. By default, each thread will be assigned its own physical core, and in most compute-bound scenarios, increasing OMP_NUM_THREADS up to the number of physical cores will allow the program to take full advantage of the entire machine for parallel processing. If increasing OMP_NUM_THREADS (again, only up to the number of physical cores on the machine) results in lower performance, the application may be biased toward memory-bound operations.

2. Congested cores

The recommended settings[3] for OMP_NUM_THREADS is the number of physical cores on the machine. However, in memory-bound scenarios, such a setting may yield a great deal of idle thread activity across the cores due to threads waiting for data. Figure 2 shows core utilization while training the model with OMP_NUM_THREADS=36. Red bars indicate waiting, or idle, threads.

3. Operations performed on matrix vectors

Passing the following commands in the runtime environment forces Intel® Math Kernel Library (Intel® MKL) to print debug messages in real time:

```
export MKLDNN_VERBOSE=1

export MKL_VERBOSE=1
```

These debug messages include information about data types and tensor sizes being processed. With these environment variables set, information about each operation is printed to stdout and will look something like this:

```
MKL_VERBOSE SGEMM(N,N,1570,1,512,0x7fefcbffe168,0
x7fef32f92fc0,1570,0x7fef49fbf240,512,0x7fefcbffe170,0
x7fef49fe1040,1570) 104.38us CNR:OFF Dyn:1 FastMM:1
TID:0  NThr:1
```

Bolded values represent the shape of the tensor being processed and the runtime for that particular operation. This message indicates that TensorFlow is processing a batch of 512, 1570x1 dimensional tensors. When one of the tensors' dimensions is 1, it is called a matrix-vector, and processing data in the form of matrix-vectors takes far longer than processing that same data in the form of a regular matrix. This is because each matrix-vector must be loaded and processed individually, whereas, in a standard matrix, all the columns can be loaded at once and processed together (standard matrices can be thought of as a series of matrix-vectors zipped together). This is where the FC layers come in to play, as they operate on matrix-vectors. If the topology has a large number of FC layers, or the Intel MKL debug messages reveal large numbers of operations on matrix-vectors, one should be wary of the application being memory-bound.

Optimizations

Given the signature described above, we assume the model is memory-bound, and it follows that the idle threads observed in Figure 2 may be the result of spawning too many threads at the outset of the training run. Thus, our first action was to reduce the value of OMP_NUM_THREADS to reduce the number of threads spawned by the application - ideally alleviating the slowdown attributable to core congestion. Indeed, this is what happened.

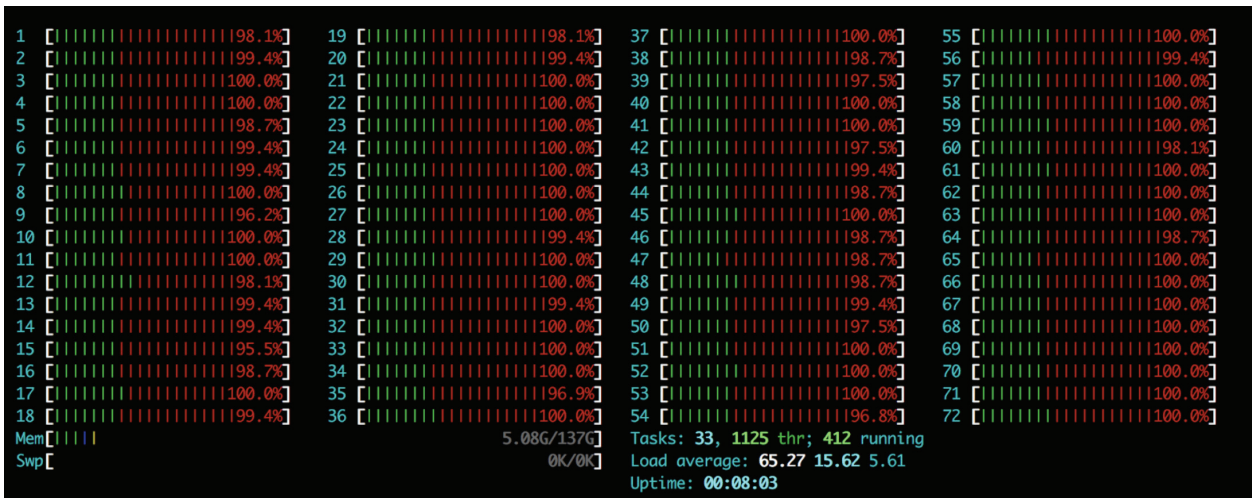


Figure 2. Core utilization while training with OMP_NUM_THREADS=36. Red bars indicate idle or "waiting" threads.

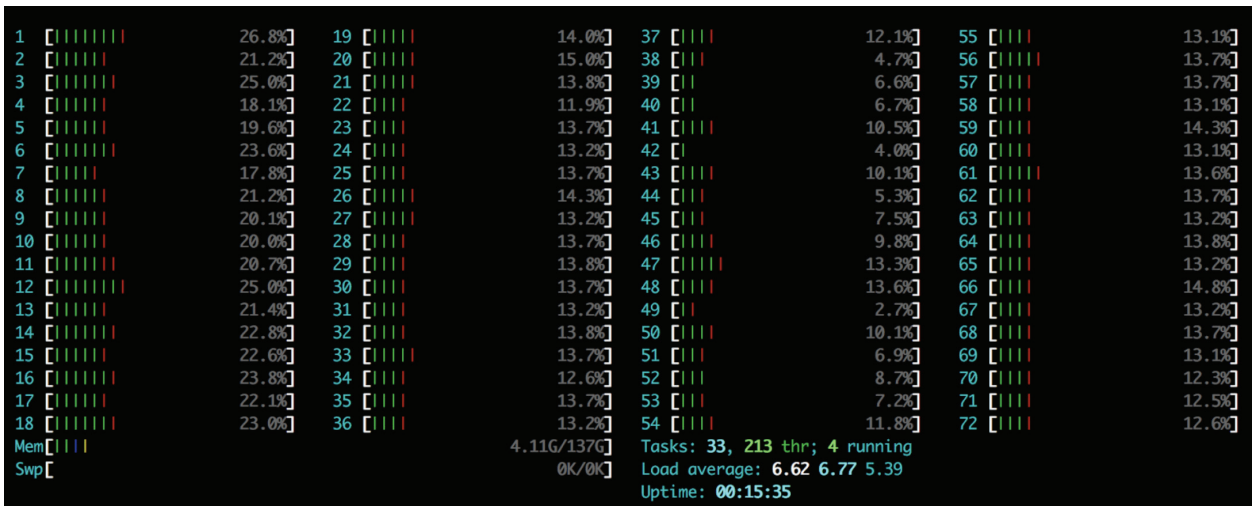


Figure 3. Core utilization during training after reducing OMP_NUM_THREADS to 1.

Figure 3 shows the core utilization after reducing OMP_NUM_THREADS from the number of physical cores (36) to 1. Note that idle threads have almost completely disappeared. Within TensorFlow, intra and inter-op threads were both set to the number of physical cores (default). This can be accomplished by setting intra_op_parallelism_threads and inter_op_parallelism_threads = 0 in tf.ConfigProto()and allows TensorFlow to configure thread pools for parallelization[3]. This is why activity is seen across all the cores as opposed to just one - TensorFlow is spawning its own thread pools from the single Python thread.

We then lock execution to only one socket using the NUMA facility via the following command:

```
numactl --cpunodebind=0 --membind=0 python main.py
```

The effect of this NUMA command is twofold:

1. NUMA locks the program execution to a single socket with --cpunodebind=0, and
2. NUMA enforces the policy that threads access only memory local to the socket on which they are running with --membind=0, which further reduces the effective memory latency.

Figure 4 shows how NUMA constrains the execution to a single socket:

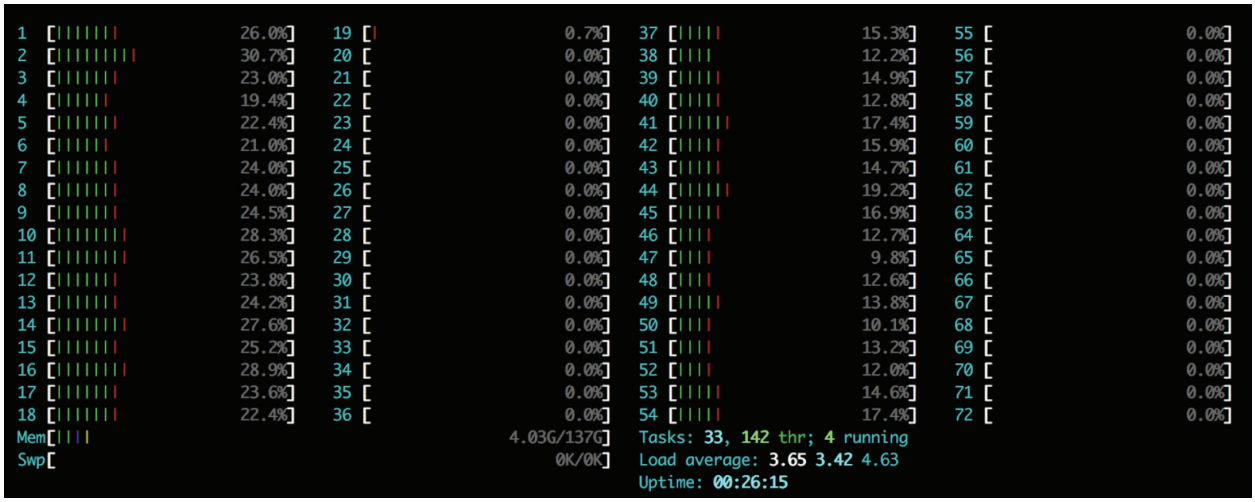


Figure 4. Core utilization after reducing OMP_NUM_THREADS to 1 and locking execution to only the physical cores on socket 0.

Finally, we can apply multi-worker training techniques to spawn a second process on the remaining socket - utilizing every core on the machine for maximum throughput. The Horovod* communication library synchronizes gradients and handles communication between the two processes[4]. See our recent article on Multi-Node Scaling of TensorFlow with Horovod for details on installing and implementing multi-worker training[5].

Results

While the core utilization in Figures 3 and 4 appear lower than that in Figure 2, removing idle threads congesting the cores allowed the remaining threads to execute more optimally. In conjunction with the NUMA commands locking execution to a single socket and giving threads access to only local memory resulted in a 100X speedup in training time[§]. Figure 5 details runtimes per epoch rounded to the nearest second.

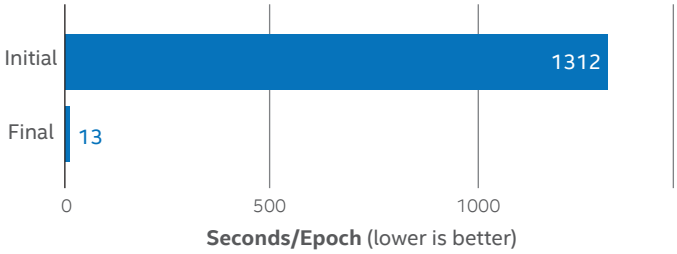


Figure 5. Runtimes before and after reducing OMP_NUM_THREADS to 1 and locking execution to the physical cores. Values shown are rounded to the nearest second.[§]

Summary

In this article, we've reviewed training behavior symptomatic of memory-bound deep learning models and proposed optimizations for resolving slowdowns in such scenarios. If your model presents a similar signature, namely:

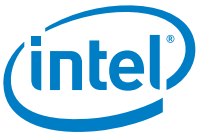
- Training time increases when increasing OMP_NUM_THREADS (up to the number of physical cores).
- Utilities for visualizing core utilization (such as htop) reveal significant core congestion from idle threads (red bars).
- Exporting MKL_VERBOSE and MKLDNN_VERBOSE indicates that operations are being performed on matrix-vectors, as opposed to standard matrices.

Then we recommend:

- Reducing the value of OMP_NUM_THREADS to prevent core congestion.
- Locking execution to a single socket using the numactl utility:

```
numactl --cpunodebind=0 --membind=0 python main.py
```

For more information on Intel's entire suite of machine learning developer tools, and to get started using the Intel Optimization for TensorFlow, please see https://www.intel.com/content/www/us/en/analytics/machine-learning/overview.html.



[1] https://software.intel.com/en-us/articles/tensorflow-optimizations-on-modern-intel-architecture

[2] Instructions for launching an Intel Optimized Deep Learning instance on AWS can be found at: https://aws.amazon.com/machine-learning/amis/

[3] https://www.tensorflow.org/performance/performance_guide#optimizing_for_cpu

[4] An overview of the Horovod training library can be found at: https://eng.uber.com/horovod/

[5] Installation and implementation details for TensorFlow with Horovod can be found at: https://ai.intel.com/using-intel-xeon-for-multi-node-scaling-of-tensorflow-with-horovod/

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. For more complete information visit http://www.intel.com/benchmarks.

[§] Configurations: Performance results are based on testing as of September 30, 2018 and may not reflect all publicly available security updates.

Single Node Testing by Intel as of September 30, 2018:

1-node, Intel Xeon Platinum 8168 processor @ 3.00GHz, Total memory 75 GB, HyperThreading: Enable, SpeedStep: Enable, OS: CentOS 7, Kernel: centos-release-7-3.1611.el7.centos.x86_64

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

 1018/EOH/MESH/PDF 338397-001US