6.24x performance enhancement using Intel optimized tensorflow, affinity thread settings optimization and numactl interleave policy

Hasanul Karim

Although GPUs are considered to be better in training deep learning algorithms, they often come with high price tags and not accessible for everyone. As CPUs are preferred choice for inference, this study aims to investigate if the training performance of CPU can be improved by using optimization techniques such as optimizing the affinity settings of the environment, inter and intra thread parallelization and memory/cpu binding policy control using numactl for NUMA enabled CPUs. It was shown that on an AWS instance with Intel Xeon CPU with 18 physical cores with 2 Numa nodes, the performance could be improved 6.24x compared to not optimized environment.

Configuration:

A single node runtime behavior on an Intel® Xeon® CPU E5-2666 v3 @ 2.90GHz with 18 physical cores with 2 threads per core and 60 GB of RAM* was investigated. This node was accessed via a c4.8xLarge AWS instance with 2 Numa Nodes available. (Detailed CPU configuration can be found in the supplementary section)

tf_cnn_benchmarks: High performance benchmarks was used for the test. **ResNet-50** model with synthetic data and batch size of 128 and 30 batches was trained with different settings as detailed in the experimental details section below.

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

Experimental details and results:

The baseline and top 2 configurations are discussed here. The MKL and MKLDNN were set to verbose in order to output when MKL is used for computation by *export MKL_VERBOSE=1; export MKLDNN_VERBOSE=1* through the bash script.

Setup 1: Default environment/tensorflow settings: A conda environtment "tf" was created and tensorflow 1.13.1 was installed via pip, python 3.6 was installed. The bash script activated the tf environment before running the first part of the code. The training was run with the following settings:

tf_cnn_benchmarks.py --device=CPU --data_name=imagenet --batch_size=128 --num_batches=30 --model=resnet50

--data_format=NHWC

This setup trained 30 batches with an average of 5.48 images/sec. htop was used to monitor the load distribution on the CPU cores and threads during the process and it showed significant amount of "red" threads which represents idle or waiting threads as can be seen from Fig1.

Setup 2: Intel optimized tensorflow with optimized environment1: intel tensorflow 1.13.1 (which includes mkl support) was installed in a new environment named "IDP" and was activated before running the second part from the bash script. The environment settings used is below:

KMP_AFFINITY=granularity=fine,verbose,compact,1,0; KMP_BLOCKTIME=0; KMP_SETTINGS=1; OMP_NUM_THREADS=18; OMP_PROC_BIND=true

KMP_AFFINITY settings used was recommended by tensorflow benchmark guidelines. OMP_NUM_THREADS were set to the physical core of the CPU. Num_inter_threads were set to 2. The training was run with the following settings:

tf_cnn_benchmarks.py --device=CPU --data_name=imagenet --batch_size=128 --num_batches=30 --model=resnet50

--data_format=NHWC --mkl=true --num_inter_threads=2 --num_intra_threads=18

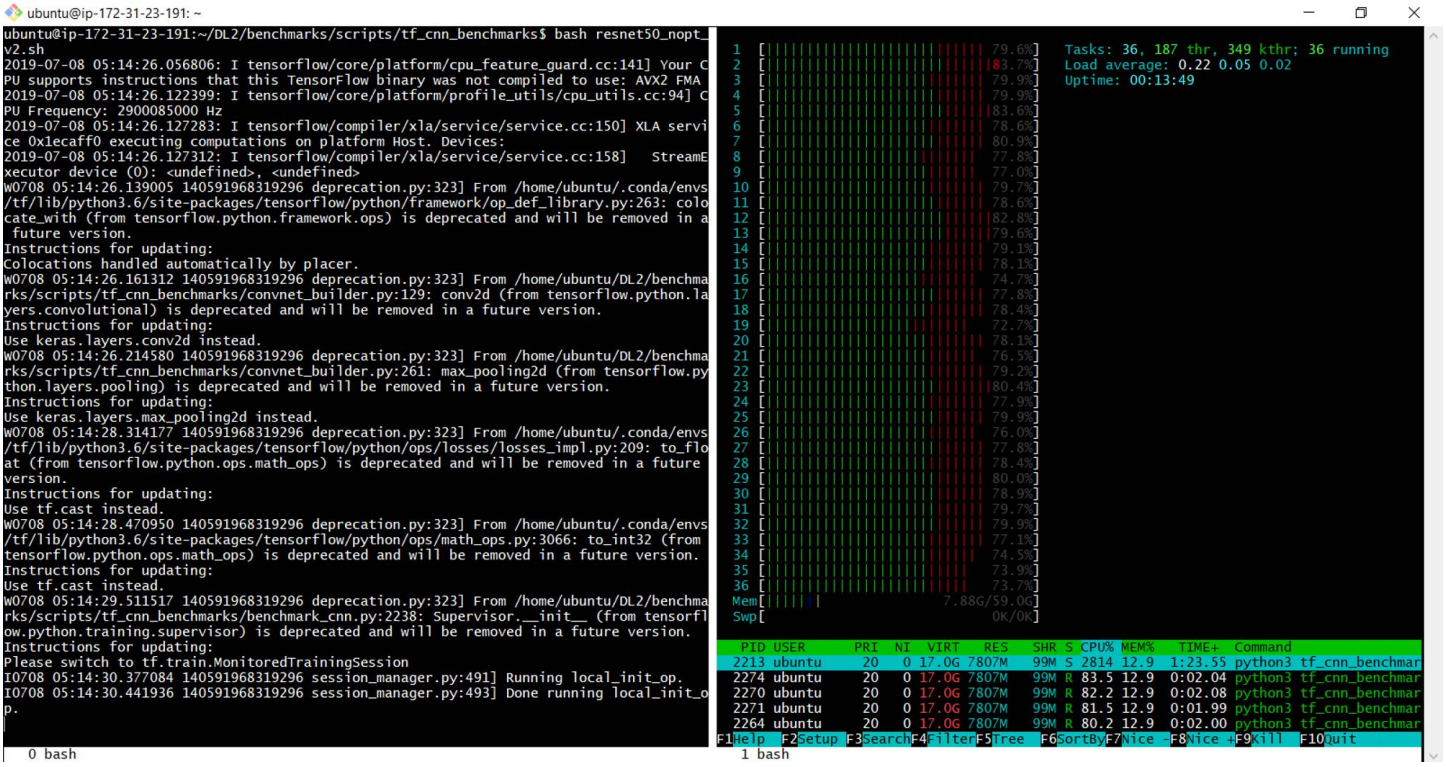

Fig1: *(setup1:no optimization) – htop (right) showing the load distribution on the threads of the CPU and on Memory, Red represents idle or 'waiting' thread, while training (left)*

Setup 2 returned an average of 33.62 images/sec, which is 6.14x performance increase compared to the un-optimized settings. It can be seen from Fig2. that the no of red threads are much less compared to setup 1 which means the affinity settings optimization are enabling a better load distribution and reducing the number of idle threads.

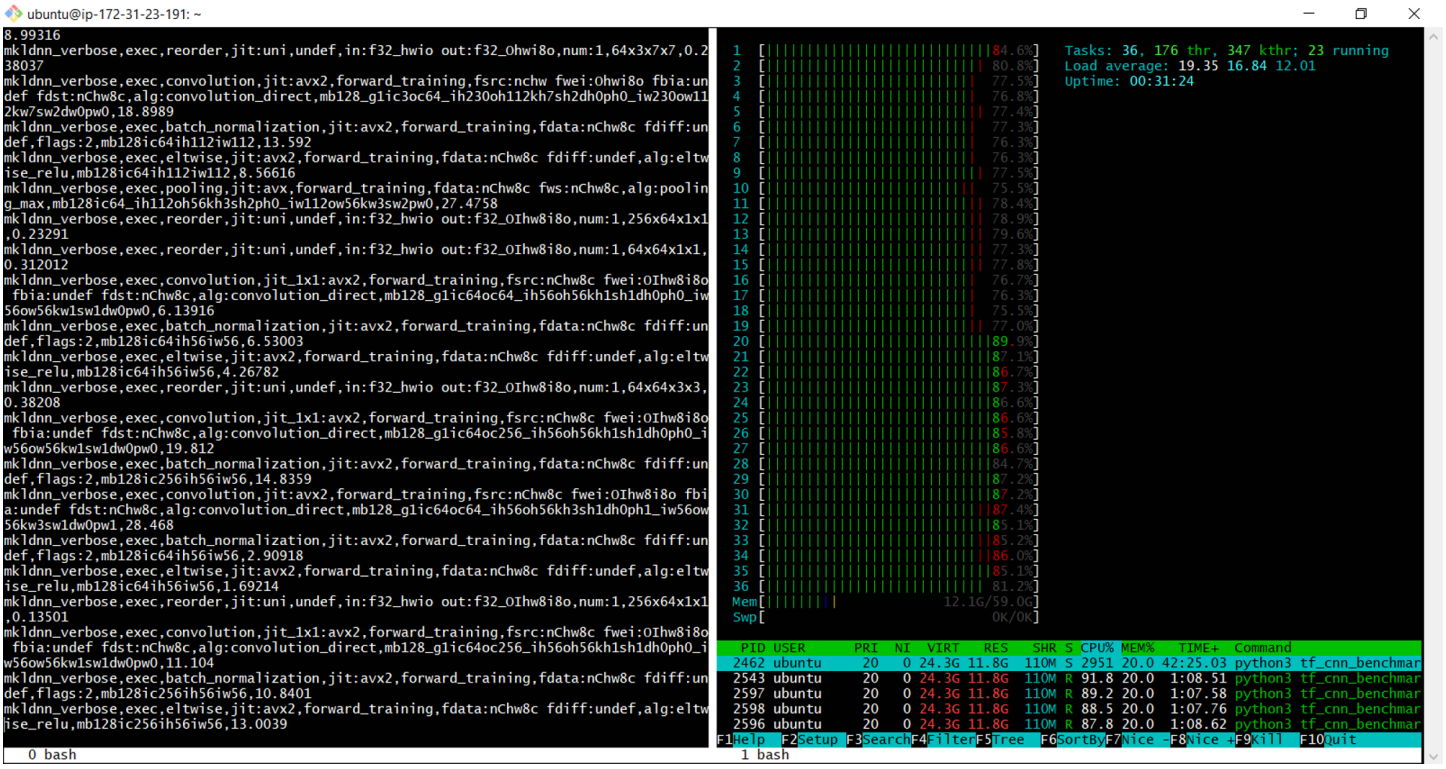

Fig2: *(setup2: intel optimized tensorflow and affinity settings optimized) - htop showing the load distribution on the threads of the CPU and on Memory, Red represents idle or 'waiting' thread. MKLDNN_VERBOSE is displaying the operation performed (left)*

Setup 3: Optimizing NUMA scheduling policy: The CPU used for this project had 2 NUMA nodes available as can be seen from "***lscpu***" command on the terminal:

NUMA node0 CPU(s): 0-8, 18-26
NUMA node1 CPU(s): 9-17, 27-35

NUMACTL gives the ability to control the NUMA scheduling policy and memory placement policy, concretely which cores to use to run the tasks and where to allocate data. It can be installed by, "***$ Sudo apt install numactl***". The default policy can be seen by "***numactl –s***" and the HW configuration including memory size, free memory and node distances can be seen by "***numactl –H"*** commands as shown below for this specific CPU:

numactl -s
policy: default
preferred node: current
physcpubind: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
cpubind: 0 1
nodebind: 0 1
membind: 0 1

numactl -H
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 18 19 20 21 22 23 24 25 26
node 0 size: 30143 MB
node 0 free: 29281 MB
node 1 cpus: 9 10 11 12 13 14 15 16 17 27 28 29 30 31 32 33 34 35
node 1 size: 30226 MB
node 1 free: 29980 MB
node distances:
node 0 1
0: 10 20
1: 20 10

Binding memory to node 1 by "***numactl –p***" or "***numactl –m***" resulted in reduced performance, both resulting in ~23.6 images/sec. reducing # of CPU threads or cores/nodes resulted in linearly decreasing performance which indicates this particular algorithm is more CPU bound than memory bound and should be scalable by increasing CPU cores. Htop can be used to verify the result of using different CPU binding as shown in Fig3.

It was observed that for this algorithm the memory interleave policy on all nodes returned the best performance and in fact showed better performance than setup2. "***numactl –i all***", allows the memory to be allocated using round robin on nodes. When memory cannot be allocated on the current interleave target fall back to other nodes as mentioned in https://linux.die.net/man/8/numactl page. This settings along with setup 2 settings performed 34.18 images/sec, which is 6.24x increase in performance from the setup1. As can be seen in Fig4, the task distribution on cores was pretty evenly distributed with very few idle threads resulting in a better performance. The training was run with the following settings:

numactl –i all tf_cnn_benchmarks.py --device=CPU --data_name=imagenet --batch_size=128 --num_batches=30 --model=resnet50 --data_format=NHWC --mkl=true --num_inter_threads=2 --num_intra_threads=18

Fig3: *Achieving CPU core distribution using numactl, (a) bind to node 0 only, using "numactl –C 0" and (b) Use only 1 thread from all available cores using "numactl –C 0,2,4….etc" note, both these settings resulted in lowering in performance for this algorithm, but might be useful for memory bound problems*

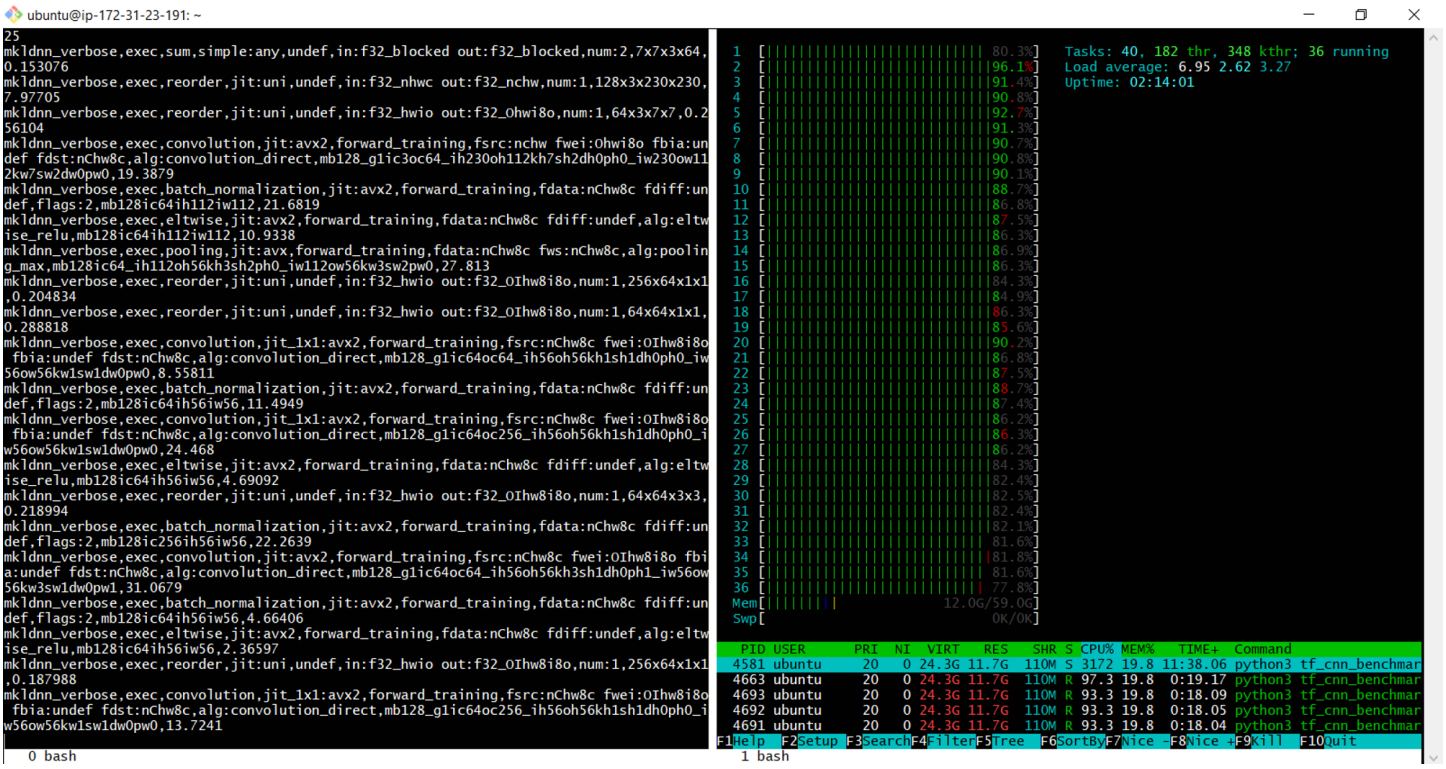

Fig4: *(setup3: intel optimized tensorflow and affinity settings optimized with numactl interleave policy) - htop showing the load distribution on the threads of the CPU and on Memory, Red represents idle or 'waiting' thread, MKLDNN_VERBOSE is displaying the operation performed (left)*

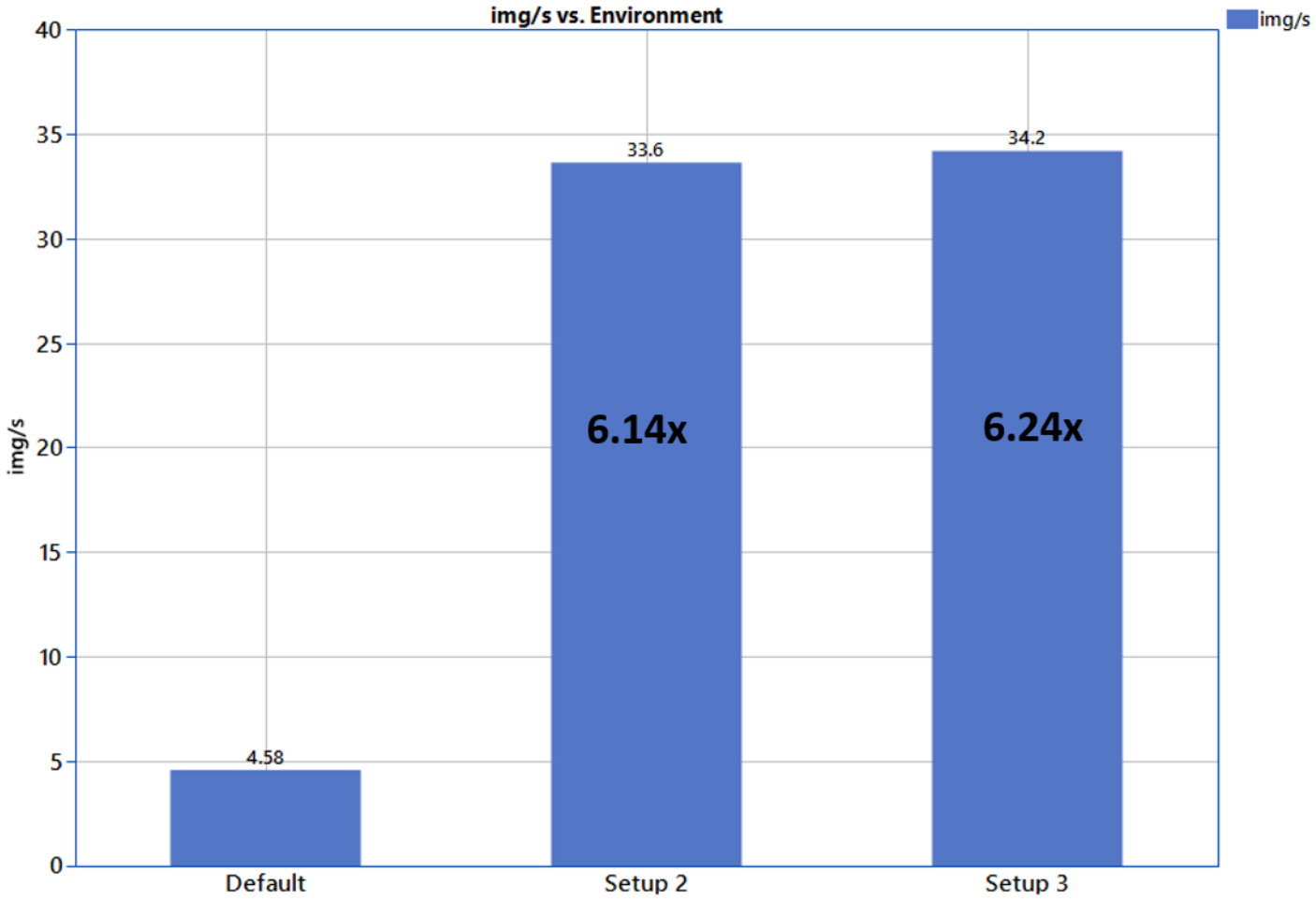

Fig4: *performance comparison of the three setups discussed in this article*

Table 1: Summary with tensorflow run time and affinity thread settings and corresponding training and inference time

Setup	Optimization settings	Training speed (images /s)	Performance improved
1	Default – no optimization	4.58	baseline
2	***Affinity settings:*** KMP_AFFINITY=*granularity=fine,verbose,compact,1,0* KMP_BLOCKTIME=*0* KMP_SETTINGS=*1* OMP_NUM_THREADS=*18* OMP_PROC_BIND=*true* ***tf_cnn_benchmarks.py flags:*** *--device=CPU* *--data_name=imagenet* *--batch_size=128* *--num_batches=30* *--model=resnet50* *--data_format=NHWC* *--mkl=true* *--num_inter_threads=2*	33.62	6.14x

	--num_intra_threads=18		
3	***Affinity settings:*** KMP_AFFINITY=*granularity=fine,verbose,compact,1,0* KMP_BLOCKTIME=*0* KMP_SETTINGS=*1* OMP_NUM_THREADS=*18* OMP_PROC_BIND=*true* ***Numactl settings:*** *numactl –i all* ***tf_cnn_benchmarks.py flags:*** *--device=CPU* *--data_name=imagenet* *--batch_size=128* *--num_batches=30* *--model=resnet50* *--data_format=NHWC* *--mkl=true* *--num_inter_threads=2* *--num_intra_threads=18*	34.18	6.24x

Conclusions

It was seen that the training performance on a CPU can be improved by upto 6.24x by optimizing the environment settings, tensorflow inter and intra thread parallelization settings and numactl memory allocation settings. Different numactl settings can be used to decide whether a problem is memory bound or CPU bound. For a CPU bound problem, using interleave memory policy worked best, but for memory bound problems, local or memory binding on nearest node and reducing no of cores to use might work better as illustrated in this whitepaper from intel, Accelerating memory-bound machine-learning models on intel xeon processors. Although in tensorflow benchmark website it's mentioned that NHWC data format might not be optimized yet, there was not significant difference observed while using either data format. Note this project scope is only for training, it might be a different case for inference. KMP_BLOCKTIME of 0 vs 1 didn't make much difference in performance and hence was not discussed in the experimental section. The CPU used had 9 sockets. The performance gain/loss by binding to specific socket(s) was not tested in this review and is subject to future work.