Optimizing Neural Network Training through TensorFlow Profile Analysis in a Shared Memory System

Fabrício Gomes Vilasbôas[1], **Calebe de Paula Bianchini**[1 2], **Rodrigo Pasti**[1], **Leandro Nunes de Castro**[1 2]

[1]Natural Computing and Machine Learning Laboratory - LCoN
Mackenzie Presbyterian University

[2]School of Computing and Informatics - FCI & PPGEEC
Mackenzie Presbyterian University

Abstract. *On the one hand, Deep Neural Networks have emerged as a powerful tool for solving complex problems in image and text analysis. On the other, they are sophisticated learning machines that require deep programming and math skills to be understood and implemented. Therefore, most researchers employ toolboxes and frameworks to design and implement such architectures. This paper performs an execution analysis of TensorFlow, one of the most used deep network frameworks available, on a shared memory system. To do so, we chose a text classification problem based on tweets sentiment analysis. The focus of this work is to identify the best environment configuration for training neural networks on a shared memory system. We set five different configurations using environment variables to modify the TensorFlow execution behavior. The results on an Intel Xeon Platinum 8000 processors series show that the default environment configuration of the TensorFlow can increase the speed up to* 5.8*. But, fine-tuning this environment can improve the speedup at least* 37%*.*

1. Introduction

Deep Neural Networks (DNN) has been showing a great differential in the solution of complex problems, mainly text, image, and video analysis, and have positioned themselves as state-of-the-art for human language processing [1]. One of the major advantages of deep networks lies in the flexibility of building architectures with multidimensional tensors, allowing them to be customized for the most diverse types of applications. This flexibility also allows problems to be solved in different ways and with a great capacity for generalization and accuracy, even in problems with high numbers of dimensions and quantities of data [1] [2].

The training of Neural Networks in large amounts of data, and with a large number of layers and weights, requires large processing and memory capacities, which most of the time brings as prerequisite the use of multiprocessor servers in shared and distributed memory systems. Therefore, it is possible to affirm that the effective training of Neural Networks only occurs through the effective employment of High-Performance Computing (HPC). This work focuses on the analysis of the execution profile to identify the best environment configuration for solving a text classification problem performed in a shared memory system using specific mathematical libraries.

One of the most used libraries for the development of neural network algorithms is TensorFlow [3]. This library offers several high-level methods that help to reduce the development complexity and time. TensorFlow was implemented in C and has APIs for C++, Go, Python, Java, Swift and JavaScript languages. Our focus here is on the Python API. This specific API uses the Numpy [4] module for manipulating data structures and for some matrix operations. This has greatly facilitated the diffusion of this library since Python has been used on a large scale for the development of data analysis solutions.

The user sets up a series of parameters when compiling TensorFlow, defining what will be the computational architecture to be used. It can be compiled for distributed memory systems using the MPI library [5]- [6], for shared memory systems using the OpenMP standard [7], for processing in GPUs using CUDA [8] or for a hybrid model which involves all of the above. Besides, the user can define the math operations library. The standard TensorFlow library is Eigen [9], but the user can define others, such as OpenBlas [10], LAPACK [11] or MKL [12]. Although it is possible to change the algorithms of this DNN library, such as in [22] that presents a custom implementation of a DNN library, or such as in [23] that evaluates the I/O performance of a DNN library using MPI, or such as [24] that evaluates a DNN execution over the Titan supercomputer, on this paper we performed a standard DNN library profile analysis compiled for a shared memory system using Eigen and MKL mathematical libraries. So, TensorFlow operation can be presented in three levels, the first level being closer to the user and the third closer to the machine: Level 1 API for Python; Level 2 methods in C; and Level 3 routines of the mathematical operations libraries.

DNN are explored in this paper for the classification of texts. Convolutional networks were used with feedforward layers, allowing the learning of input patterns with a greater amount of aggregated information [13]- [14]. For example, with vector representations for words and sentences, such as those generated by Word2Vector [13], it is possible to represent texts as $n \times m$ tensors, where n is the word vector dimension and m is the number of words in the text. It is also possible to generate p different representations, thus composing different learning channels, which generates three-dimensional tensors for each sample.

This paper brings a profile analysis of the TensorFlow library on a shared memory system for a deep neural network training task using different mathematical libraries and modifying the behavior of the OpenMP runtime. Through the profile analysis, it is possible to discover more efficient ways of performing the computations involved in the network training process.

Some recent works propose to make a profile analysis of neural networks training applications. [28] and [30] propose the profile analysis of Convolutional Neural Network (CNN), which is a class of deep neural networks, training it on GPUs platforms. Our paper differs from those on processing platforms. We propose to analyze a DNN library on a shared memory system based on CPU. One other paper is [29]. The author in this paper proposes the profile analysis of three different open-source neural networks frameworks: TensorFlow, Deep Learning4j and H2O. They use the *MNIST* handwritten digit database to perform their analysis on Intel Core i5-7200U, Intel Core i7-2700K, and NVIDIA Tesla K40. The conclusion for the paper is that the Intel Core i7-2700K is better for the neural network training using this data. Our paper proposes a deep analysis on a single CPU series, Intel Xeon Platinum 8000, and for a single framework, TensorFlow, aiming the best environment setting for the neural network training task.

The paper is organized as follows. Section 2 describes the deep neural network architecture used, and Section 4 brings the profile analysis performed, together with the research methodology and experimental results obtained. The paper is concluded in Section 5 with a general discussion and perspectives for future research.

2. The Deep Neural Network Used

In this paper, we used a Convolutional Neural Network (CNN) to perform sentiment analysis [16]- [17] from tweets. CNNs are used with feedforward layers, allowing the learning of input patterns with a greater amount of aggregated information [18]. The input data for CNN was generated by vector representations of words and sentences with Word2Vector. The texts were represented as $n \times m$ dimensional tensors, where n is the dimension of the word vector present in a text, and m is the number of word vectors. We generated p different arrays of representations, thus composing different learning channels with three-dimensional tensors ($n \times m \times p$) for each text (tweet). A CNN with 7 layers was used with the following architecture:

- **Convolutional layer:** This layer aims to reduce the dimensionality of the input tensors and to combine word vectors. A $2 \times n$ kernel was used, where two-by-two word combinations were generated during the kernel's stride. The number of kernel columns coincides with the number of dimensions of the kernel vector words, so the vectors are combined with the whole word;
- **Max Pooling Layer:** The most relevant features of the convolutional layer are extracted from the Max Pooling layer;
- **Feedfoward layers:** A total of 5 feedforward layers were used to complete the learning and progressively reduce the dimensionality of the tensors until the network output. The output layer has one neuron for each class.

The activation functions used were all hyperbolic tangents and the input data were normalized in the range $[-1, 1]$. The number n of tensor words is limited to 30 words.

3. TensorFlow Parallelism Levels and Environment Variables

In Section 1, we presented the TensorFlow library divided into three different levels. Just to remember: Level 1 refers to API for Python, Level 2 refers to methods in C, and Level 3 refers to routines of the mathematical operations libraries. Each level has a parallelism strategy. This section presents an overview of these strategies.

At the first level, the optimization is on the data locality. The Python API uses the *numpy* module [4] to allocate and manipulate data structures. This module allocates the data contiguously in memory aiming for better performance on data access. The second level uses the *OpenMP* standard to perform the data manipulation since the TensorFlow documentation does not give us details about the implementation. The third level of parallelism strategy depends on the mathematical library used. The Intel Math Kernel Library (MKL) uses *OpenMP* and low-level SIMD (Single Instruction Multiple Data) instructions. The Eigen library benefits from acceleration using heterogeneous hardware. In this paper, we performed the tests using the *OpenMP* version of the Eigen.

Besides, *OpenMP* is an API specification [25], it also defines the thread model, memory model and runtime behavior of the threads. For the latter one, it uses environment variables, which is a dynamic-named value that can affect the way running threads will behave on a computer. Hereby, we used four environment variables to control the *OpenMP* runtime and MKL behavior. These are the variables:

- *KMP_BLOCKTIME*: sets the time that a thread should wait before sleeping after completing the execution of a parallel region;
- *MKL_NUM_THREADS*: sets the number of threads to MKL execution;
- *MKL_DYNAMIC*: enables Intel MKL to dynamically change the number of threads.
- *OMP_NUM_THREADS*: sets the number of threads on OpenMP runtime;
- *OMP_NESTED*: enables multiple levels of thread generation;
- *OMP_MAX_ACTIVE_LEVELS*: sets the maximum number of nested parallel regions;

4. TensorFlow Profile Analysis

To analyze the TensorFlow execution profile, we created five different environments. The environments were created using the Anaconda, which is self-denominated as "a package manager, an environment manager, a Python/R data science distribution, and a collection of over 1,500+ open source packages. Anaconda is free and easy to install, and it offers free community support." [21]. The environments description are:

- *serial*: default installation of Python, *intelpython3_full*, via Intel repository. To mimic a serial execution environment, we set *OMP_NUM_THREADS = 1*, *MKL_DYNAMIC = false* and *MKL_NUM_THREADS = 1*. ***This is the baseline environment***.
- *default*: default installation of Python, *intelpython3_full*, via Intel repository. This environment uses the MKL library. No environment variables was changed;
- *eigen*: default Python installation via *conda-forge* repository. This environment uses the Eigen library. No environment variables was changed. This Python installation uses GNU OpenMP runtime;
- *blocktime_0*: the *KMP_BLOCKTIME* variable was set to 0 within the *default* environment;
- *blocktime_30*: variable *KMP_BLOCKTIME* was set to 30 within the *default* environment. This configuration will be analyzed, because it is described as the best configuration for high performance [20];

The algorithm was executed 5 times in each environment and the results to be presented are the processing time average.

4.1. Dataset and Computational Environment

The dataset used was obtained by collecting several distinct themes in Tweeter for over 6 months. We are not particularly concerned with the accuracy, instead, the emphasis is given to composing a sufficiently large dataset to run the performance analyzes. The collected database consists of $223,050$ data objects extracted from tweets.

The purpose of these experiments is to achieve the best performance of a shared memory system using Intel Xeon Platinum 8000 Series. The chosen system was a single compute-node composed of two Intel Xeon 8160 @ 2.10 GHz processors, each with 24 physical cores (48 logical) and 33 MB cache memory, 190GB RAM, two Intel S3520 SERIES SSDs with 1.2 TB and 240 GB capacity and CentOS 7 operating system with kernel version 3.10.0-693.21.1.3l7.x86_64.

4.2. Results and Analysis: simple setup

This section presents the results and analysis of a simple setup. All five environment was used during this evaluation: *serial*, *default*, *eigen*, *blocktime_0* and *blocktime_30*. This evaluation is important due to the explanation about high performance DNN using the *KMP_BLOCKTIME* variable [20].

Fig. 1 shows the total execution time of each environment. The x axis presents the execution time and the y axis presents the environment.

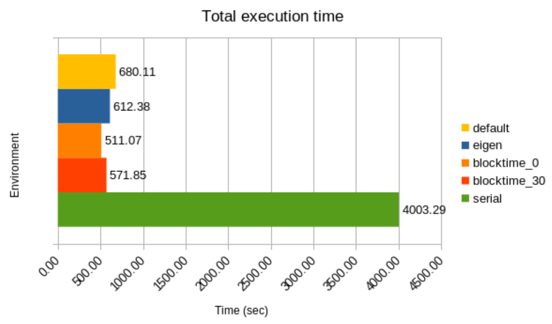


Figure 1. Total execution time in a simple setup

Looking at Fig. 1 we see that the execution time is: *serial*: 4003.29; *default*: 680.11; *eigen*: 612.38; *blocktime_0*: 511.07; *blocktime_30*: 571.85.

Fig. 2 shows that the highest speedup is 7.83 for the *blocktime_0* environment and the lowest performance is 5.89 for the *default* environment, using the *serial* environment as baseline.

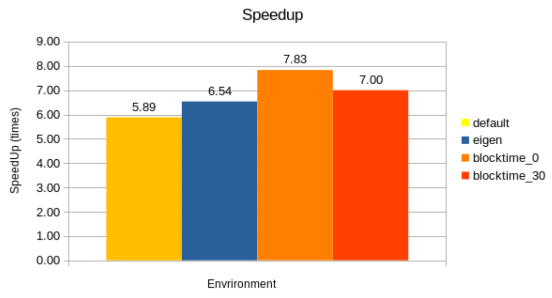
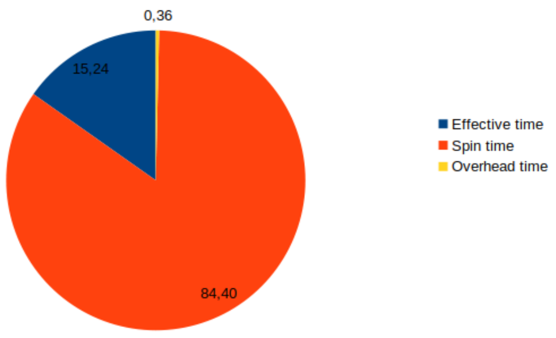


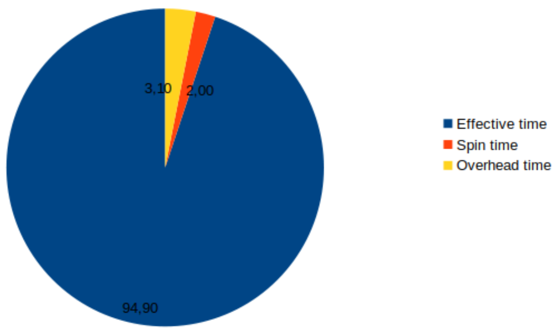
Figure 2. Speedup using a simple setup, as explained in [20]

When analyzing the *default*, *blocktime_0* and *blocktime_30* environments, which have the same parallelism technique, we observe the importance of setting the environment. When we set *KMP_BLOCKTIME=0*, there is a reduction of 169.04 seconds; when we set *KMP_BLOCKTIME=30* there is a reduction of 108.26 seconds. This indicates a reduction in the total execution time of approximately 15.92% for the *blocktime_30* environment and approximately 24.85% for the *blocktime_0* environment in relation to the *default* environment. This variation in runtime is explained by the OpenMP runtime thread management policy implemented by Intel.

According to this explanation [26], by default, each thread remains active for 200ms in spinlock after the completion of its task. As a result, the computational feature remains locked, preventing other tasks from being scaled for processing that resource. This behavior directly impacts the execution of MKL, since the scheduling of its tasks is dynamic. Therefore, the runtime has to wait for the release of the resource to stagger another task block for processing. To understand this behavior, we analyzed two environments at VTune, which is an Intel toolkit to profile execution. The profiling results are shown in Fig. 3. The main metrics presented are the effective time, spin time, and overhead time. Effective time is the CPU time spent in the user code, which does not include spin and overhead time. Spin time is the wait time during which the CPU is busy [27]. This often occurs when a synchronization API causes the CPU to poll while the software thread is waiting. Some spin time may be preferable to the alternative of increased thread context switches. Too much spin time, however, can reflect the lost opportunity for productive work. The overhead time is CPU time spent on the overhead of known synchronization and threading libraries, such as system synchronization APIs (e.g. system calls and OpenMP directives).



(a) *default* environment



(b) *blocktime_0* environment

Figure 3. Spin time analysis using Intel VTune Amplifier profiling tool.

Looking at Fig. 3 we see that the effective time increases 79.65%, the spin time decreases 82.29% and the overhead time increases 2.74%. This result indicates that when we define *KMP_BLOCKTIME=0* there is a 79.65% increase in the algorithm efficiency.

When we compare the total execution time of the *default* and *eigen* environments, we see that the *eigen* environment executes the algorithm 67.73 seconds faster. When we compare the total execution time of the *eigen* and *blocktime_0* environments, we see that the latter executes the algorithm 101.30 seconds faster. In this case, we are evaluating different OpenMP runtimes. The *default* and *blocktime_0* environments run over the

OpenMP runtime implemented by Intel and the *eigen* environment runs over the OpenMP runtime implemented by GNU.

In order not to do an unfair analysis, it is necessary to define under what aspects our analysis is being done. Within the GNU runtime, there is a behavior policy of the threads lifecycle that resembles behavior within the Intel runtime. This behavior is defined by the *OMP_WAIT_POLICY* variable, which defines how the threads will behave after the completion of their task. Thus, when this variable is defined as *OMP_WAIT_POLICY=active*, the runtime keeps the thread running in a loop without operations (spinlock) while the parallel region is active while keeping the computational resource locked. When this variable is set to *OMP_WAIT_POLICY=passive*, the runtime "deactivates" the execution of the thread even though there is an execution of the parallel region to which it belongs, releasing the computational resource to other threads. For this, we can associate the *OMP_WAIT_POLICY* variable behavior of the GNU runtime with the variable *KMP_BLOCKTIME* of the Intel runtime, having the following relation: *OMP_WAIT_POLICY=active* is equivalent to *KMP_BLOCKTIME=infinite* and *OMP_WAIT_POLICY=passive* is equivalent to *KMP_BLOCKTIME=0*. Therefore, we can say that the MKL routines have a better computational performance comparing to the Eigen library routines when considering the operations involving the execution of this neural network architecture for this database.

4.3. Results and Analysis: fine tuning

This section presents the results and analysis of a fine-tuning setup on Intel's Python distribution. We analyze the total execution time setting up the environment variables *KMP_BLOCKTIME*, *OMP_NUM_THREADS*, *MKL_NUM_THREADS*, *MKL_DYNAMIC* and *OMP_NESTED*.

Fig. 4 presents the total execution time. The magenta bars present the results for *OMP_NUM_THREADS = 1, 12, 24, 48, 96*, *MKL_NUM_THREADS = 1* and *MKL_DYNAMIC = false*; the blue bars present the results for *OMP_NUM_THREADS = MKL_NUM_THREADS = 1, 12, 24, 48, 96* and *MKL_DYNAMIC = false*; the orange bars present the results for *OMP_NUM_THREADS = 1, 12, 24, 48, 96*, *KMP_NUM_THREADS = 96* and *MKL_DYNAMIC = false*; yellow bars present the results for the *default* environment as mentioned on previous section. For all experiments, the *KMP_BLOCKTIME* is equal to 0, *OMP_NESTED* is equal to $true$ and *OMP_MAX_ACTIVE_LEVELS* is equal to 2 less than the *default* environment that *KMP_BLOCKTIME* is equal to 200 and *OMP_NESTED* is equal to $false$.

Looking at Fig. 4, we conclude that the best execution time is 495.18 seconds for *OMP_NUM_THREAD = KMP_NUM_THREADS = 1* and the worst execution time is 4003.29 seconds for *OMP_NUM_THREAD = KMP_NUM_THREADS = 96*.

Analyzing Fig. 4 we see that when we increase only *OMP_NUM_THREADS* the variation on the total execution time is not significant. However, when we increase both *OMP_NUM_THREADS* and *MKL_NUM_THREADS* there is a significant variation. This behavior can be explained by the mathematical operations that a neural network training involves. This observation can be corroborated by the result of *OMP_NUM_THREADS = 1* and the *MKL_NUM_THREADS = 96*. There was a decrease of 1.92 times on execution time. Therefore, we can conclude that the increase in the number of threads for the mathematical library is more important than increasing the number of threads for the

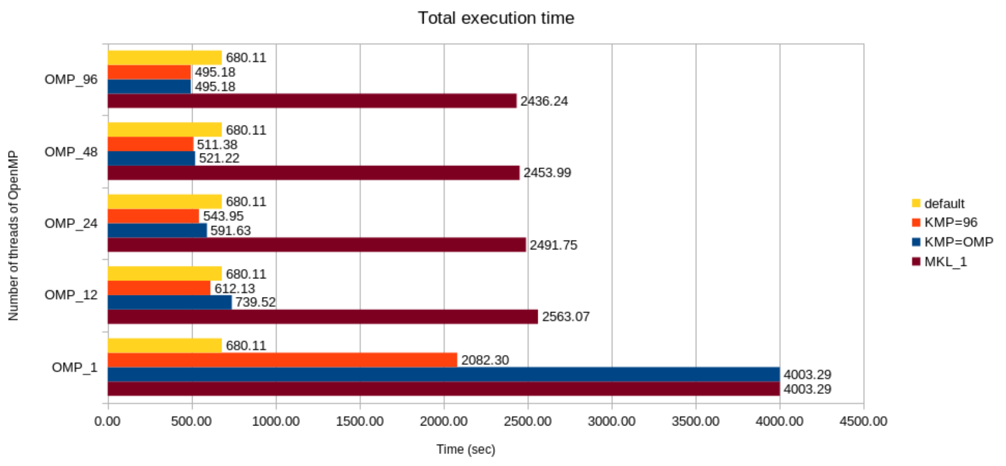


Figure 4. Total execution time after fine tunning

TensorFlow operations.

Comparing the total execution time of the *default* environment with the others fine settings, we see that from the *OMP_NUM_THREADS = 12* and *KMP_NUM_TREADS = 96* we have gain in performance. Fig. 5 presents these gains in performance considering *KMP_NUM_THREAD = 96* for all cases.

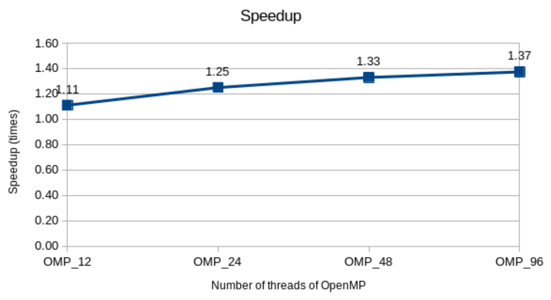


Figure 5. Speedup after fine tuning

Looking at Fig. 5 we see a minimum gain of 1.11 times for *OMP_NUM_THEADS = 12* and a maximum gain of 1.37 times for *OMP_NUM_THREADS = 96*.

In a complementary result visualization, Table 1 presents the total time reduction in seconds (second column) and percents (third column). This table presents a minimum time reduction of 67.10 seconds and a maximum time reduction of 184.93 seconds, representing, respectively, 10 and 27 percents of time reduction.

By the results presented in this section, we conclude that worth it the time spent in fine-tuning.

5. Conclusions and Future Trends

The application of Deep Neural Networks (DNN) is highly dependent on the use of frameworks to facilitate the implementation and deploy of algorithms. Also, such architectures are very computationally intensive and the use of parallel processing may be the only

OMP_NUM_THREADS	Time reduction (seconds)	Time reduction (%)
12	67.98	10
24	136.15	20.02
48	168.73	24.81
96	184.93	27.19

Table 1. Total execution time reduction

feasible alternative in some cases. Therefore, this paper brought a profile analysis of Tensorflow on five different computing environments in a text classification application. The results obtained lead to a better understanding of the execution patterns of a convolutional neural network and, thus, allow us to obtain more efficient forms of training such networks.

The results show that *default* environment had a time reduction of 83.01%, *eigen* environment had a time reduction of 84.70%, *blocktime_30* environment had a time reduction of 85.72% and *blocktime_0* had a time reduction of 87.23%. Therefore, the most efficient environment was the *blocktime_0* considering this shared memory processing system and this dataset.

Those results shows that the fine tuning effort worth for the Intel Xeon Platinum 8000 Series. We have improved the performance in 1.37 times when use *blocktime_0* environment and we set *MKL_DYNAMIC = false*, *OMP_NUM_THREAD = KMP_NUM_THREADS = 96*.

As future investigations, it is possible to highlight that, one of the most important features of the Skylake architecture is the Intel Advanced Vectors Extensions 512 (AVX-512). This set of instructions and registers allows the processors to operate over vectors of 512 bits at a single time. As future work, we propose to investigate the influence of the AVX-512 on the MKL, OpenBLAS/LAPACK, and Eigen mathematical libraries.

References

[1] Schmidhuber, J. (2014). Deep Learning in Neural Networks: An Overview, Neural Networks, 61(2015),pp. 85-117.

[2] Kalchbrenner, N.; Grefenstette, E.; Blunsom, P. (2014). A Convolutional Neural Network for Modelling Sentences, arXiv.org.

[3] Abadi, Martín, et al.(2016),Tensorflow: A System for Large-Scale Machine Learning, Proc. of the 12th USENIX Symposium on Operating Systems Design andImplementation (OSDI'16), pp. 265-283.

[4] Walt, Stéfan van der, S. Chris Colbert, and Gael Varoquaux. (2011), The NumPy Array:A Structure for Efficient Numerical Computation,Computing in Science & Engineering,13(2), pp.22-30.

[5] Gropp, William D., et al. (1999), Using MPI: portable parallel programming with the message-passing interface. 2ndEdition, MIT Press, 1999.

[6] Vishnu, Abhinav, Charles Siegel, and Jeffrey Daily. (2016), Distributed TensorFlow with MPI,arXiv preprint arXiv:1603.02339 (2016).

[7] Chapman, Barbara, Gabriele Jost, and Ruud Van Der Pas. (2008), UsingOpenMP: Portable Shared Memory Parallel Programming. MIT Press.

[8] Kirk, David B., and W. Hwu Wen-Mei. (2008), Programming Massively Parallel Processors: A Hands-On Approach,Morgan Kaufmann.

[9] Cummins,Chris,et al. (2017), Deep Learning for Compilers,Institute for Computing Systems Architecture.

[10] Zhang, X., Q. Wang, and W. Saar. (2017), OpenBLAS: An optimized BLAS library, http://www.openblas.net/.

[11] Anderson, Edward, et al. (1990), LAPACK: A Portable Linear Algebra Library for High-Performance Computers,Proceedings of the 1990 ACM/IEEE Conference on Supercomputing,IEEE Computer Society Press, pp. 2-11.

[12] Wang, Endong, et al. (2014), Intel Math Kernel Library, High-Performance Computing on the Intel Xeon Phi. Springer, Cham, pp.167-188.

[13] Mikolov, T.; Chen, K.; Corrado, G.; Dean, J., Efficient Estimation of Word Representations in Vector Space, arXiv preprint arXiv:1301.3781

[14] Jurafsky, D.; Martin, J. H. (2009). Speech and Language Processing, Prentice Hall, 2nd Edition

[15] Hager, Georg, and Gerhard Wellein. (2010), Introduction to High Performance Computing For Scientists And Engineers. CRC Press

[16] Pang, B.;Lee, L.(2008), Opinion Mining and Sentiment Analysis, Foundations and Trendsin Information Retrieval,2(1–2), pp 1-135. http://dx.doi.org/10.1561/1500000011

[17] Severyn, A.;Moschitti, A. (2015), Twitter Sentiment Analysis with Deep Convolutional Neural Networks, Proc. of the 38th Int.ACM SIGIR Conf.on Research and Development in Information Retrieval, pp. 959-962.

[18] Lai, S.; Xu, L.; Liu, K.; Zhao, J. (2015). Recurrent Convolutional Neural Networks for Text Classification, Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, pp. 2267-2273

[19] Snyder, Lawrence. (1998), A Taxonomy of Synchronous Parallel Machines, Technical Report,Washington Univ.Seattle,Dept of Computer Science.

[20] TensorFlow. Perfomance. Acessed at Jul/19. https://www.tensorflow.org/guide/performance/overview#tunin

[21] Anaconda. Anaconda Distribution. Accessed at Jul/19. https://docs.anaconda.com/anaconda/

[22] Thorsten Kurth, Jian Zhang, Nadathur Satish, Evan Racah, Ioannis Mitliagkas, Md. Mostofa Ali Patwary, Tareq Malas, Narayanan Sundaram, Wahid Bhimji, Mikhail Smorkalov, Jack Deslippe, Mikhail Shiryaev, Srinivas Sridharan, Prabhat, and Pradeep Dubey. 2017. Deep learning at 15PF: supervised and semi-supervised classification for scientific data. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17). ACM, New York, NY, USA, Article 7, 11 pages. DOI: https://doi.org/10.1145/3126908.3126916

[23] Ammar Ahmad Awan, Karthik Vadambacheri Manian, Ching-Hsiang Chu, Hari Subramoni, Dhabaleswar K. Panda, Optimized large-message broadcast for deep learning workloads: MPI, MPI+NCCL, or NCCL2?, Parallel Computing, Volume 85, 2019, Pages 141-152, ISSN 0167-8191, https://doi.org/10.1016/j.parco.2019.03.005.

[24] Durbha SS, Kurte KR, Bhangale U. Semantics and high performance computing driven approaches for enhanced exploitation of earth observation (EO) data: state of the art. Proc Natl Acad Sci India Sect A Phys Sci. 2017;87(4):519-539. https://doi.org/10.1007/s40010-017-0432-z

[25] OpenMP Architecture Review Board. OpenMP Application Programming Interface. v5.0. Accessed at Jul/19. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[26] Intel. Supported Environment Variables. Accessed at Jul/19. https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-supported-environment-variables

[27] Maurice Herlihy and Nir Shavit. The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc. 2008. San Francisco, CA, USA.

[28] Kim, Heehoon, et al. Performance analysis of CNN frameworks for GPUs. 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, 2017.

[29] Kochura, Yuriy, et al. Performance analysis of open source machine learning frameworks for various parameters in single-threaded and multi-threaded modes. Conference on Computer Science and Information Technologies. Springer, Cham, 2017.

[30] S. A. Mojumder et al., Profiling DNN Workloads on a Volta-based DGX-1 System. 2018 IEEE International Symposium on Workload Characterization (IISWC), Raleigh, NC, 2018, pp. 122-133. doi: 10.1109/IISWC.2018.8573521