Towards a Malleable Tensorflow Implementation

Leandro Ariel Libutti[1](✉), Francisco D. Igual[2], Luis Piñuel[2], Laura De Giusti[1], and Marcelo Naiouf[1]

[1] Instituto de Investigación en Informática LIDI (III-LIDI) Facultad de Informática, UNLP-CIC, La Plata, Argentina
{llibutti,ldgiusti,mnaiouf}@lidi.info.unlp.edu.ar

[2] Departamento de Arquitectura de Computadores y Automática, Universidad Complutense de Madrid, Madrid, Spain
{figual,lpinuel}@ucm.es

Abstract. The TensorFlow framework was designed since its inception to provide multi-thread capabilities, extended with hardware accelerator support to leverage the potential of modern architectures. The amount of parallelism in current versions of the framework can be selected at multiple levels (*intra- and inter-paralellism*) under demand. However, this selection is fixed, and cannot vary during the execution of training/inference sessions. This heavily restricts the flexibility and elasticity of the framework, especially in scenarios in which multiple TensorFlow instances co-exist in a parallel architecture. In this work, we propose the necessary modifications within TensorFlow to support dynamic selection of threads, in order to provide transparent malleability to the infrastructure. Experimental results show that this approach is effective in the variation of parallelism, and paves the road towards future co-scheduling techniques for multi-TensorFlow scenarios.

Keywords: TensorFlow · Malleability · Containers · Resource management · Co-scheduling

1 Introduction and Motivation

The exponential growth in the interest of Machine Learning in the last decade is directly related to three fundamental advances, namely: *(i)* the development of better algorithms with direct applications in many fields of science and engineering; *(ii)* the availability of massive amounts of data and the feasibility of efficiently storing and analyzing it; and *(iii)* the appearance of novel hardware architectures, typically parallel and/or homogeneous, that allow a proper exploitation of both new algorithms on large datasets in an affordable time.

Actually, the application of High Performance Computing techniques and architectures has renewed the interest on the application of Machine Learning on a plethora of problems, including applications to image recognition, segmentation, speech recognition, natural language processing or language translation,

© Springer Nature Switzerland AG 2020
E. Rucci et al. (Eds.): JCC-BD&ET 2020, CCIS 1291, pp. 30–40, 2020.
https://doi.org/10.1007/978-3-030-61218-4_3

among many others. Together with new computing architectures, the evolution of specific-purpose software frameworks has also contributed to the democratization of Machine Learning, hiding many of the underlying details in order to attain high-performance implementations. Many of these frameworks consider parallelism in general, and heterogeneity exploitation in particular, as a nuclear feature. Tensorflow [1], Caffe [2], Keras [3] or PyTorch [4], to name a few, offer optimized versions targeting specific hardware architectures, hiding many of the details to the final user while keeping performance for both training and inference near the peak of the underlying systems.

Tensorflow is, currently, one of the most extended frameworks targeting both training and inference. Its design is based on a dataflow-like execution model, in which users build an execution graph in which nodes represent operations (typically mathematical transformations), and edges between them denote dataflow between operations in terms of multi-dimensional arrays (that is, *tensors*). The amount of concurrency among operations is dictated by data dependences, while internally, each operation can be further parallelized in order to boost performance. Regarding parallelism, Tensorflow allows a static, a priori selection of two different levels of parallelism, namely: *inter-node parallelism*, denoting the amount of operations that can be executed in parallel at a given execution point respecting data dependences; and *intra-node parallelism*, that determine the amount of internal parallelism per operation. This double degree of parallelism is, however, static and must be selected by the user or the runtime software prior to the launch of a graph session; in other words, the parallelism in Tensorflow is *rigid* and cannot be reconfigured while an operation is running, opposed to a *malleable* nature other software packages.

Experiments have been carried out on the *intra-node parallelism* and *inter-node parallelism* parameters, seeking the definition of the most optimal values for certain Machine Learning algorithms, running in cpu backend [5]. The quest of *malleability* has been previously explored in other fields, mainly in the linear algebra arena [6–8], with promising results in terms of flexibility, resource usage and performance. Applied to Machine Learning in general, and Tensorflow in particular, a fully malleable TensorFlow implementation would allow a dynamic reconfiguration of the amount and nature of the effective parallelism while a training session (for example) is on the fly.

This static selection in the degree and type of parallelism allows a proper exploitation of the underlying hardware by deciding appropriate values for each parameter depending on the available resources and operation types. However it is merely static. In scenarios in which multiple TensorFlow instances arise at any temporal point (e.g. multiple training sessions sharing a common platform), a flexible and dynamic resource management scheme becomes mandatory; that is, considering a graph in which inter-parallelism is decided a priori, for example, the emergence of a second training session needs a re-configuration of the degree of parallelism in order to properly divide the underlying computing resources. This feature is, as of today, not possible within TensorFlow.

Our final target is a common scenario in which individual TensorFlow instances are confined inside a container, which is a typical setup on common cloud services [9]; on shared-resources scenarios, reducing the amount of cores per container would encompass oversubscription situations provided the TensorFlow instance within is not informed consequently. Our goal, hence, is to inform the internal TensorFlow instance to reduce/increase the amount of parallelism according to the reduce/increased amount of resources assigned to the container.

In this paper, we provide the necessary mechanisms and modifications in TensorFlow to allow *malleability*, that is, dynamic variation of the number of threads at any point of the execution. Our approach is general enough to reduce or expand the level of inter- and intra-parallelism within the framework from an external entity (e.g. a co-scheduler system software) with no impact for the user. As of our knowledge, this is the first effort to introduce thread malleability in the framework, and paves the road towards the development of co-scheduling schemes that allow an efficient sharing of computing resources in architectures shared by multiple TensorFlow instances. As far as we are aware, this is the first effort towards malleability integration within TensorFlow.

The rest of the paper is structured as follows. Section 2 describes the internal infrastucture of TensorFlow in terms of multithreading support, with special interest in the deployment of threadpools and queues to support this functionality. Section 3 introduces and deeply describes the necessary modifications within the framework to support malleability. Section 4 reports execution traces for the modified malleable TensorFlow implementation. Finally, Sect. 5 closes the paper with a number of conclusions and future research lines opened by this fundamental modification in the framework.

2 Threading Model in Tensorflow

2.1 Execution Components

The computation within Tensorflow is defined by means of a graph composed by an arbitrary number of *compute nodes*. Each compute node features zero or more inputs and outputs, and represents an instance of a kernel operation defined in the framework, such as a general matrix-matrix multiplication (MatMul). The values that flow across nodes (input and output values) are called *tensors*, data structures of arbitrary dimensions, where the element type is specified at graph construction time. Additionally, nodes can present dependences that must be satisfied before the execution of the next node.

Tensorflow defines a client who is responsible for communicating with one or more workers. Each worker controls a set of devices identified by type and name. Each device is responsible for handling the execution of *ready nodes*, that is, compute nodes whose input dependences have been satisfied.

Hence, the compute nodes of the graph are executed in an order dictated by their input dependences, following a so-called *dataflow* execution model. Once they are fulfilled, the node becomes eligible for execution and it is added to a *ready node queue* belonging to a worker, from which it is extracted, scheduled

and finally executed. Upon finishing its execution, the number of dependencies of the nodes that are linked to it is decreased by one. The ready node queue is scheduled in an unspecified order, delegating each node to an available computing *worker* for execution (itself or another worker).

To handle the execution of the graph nodes, the Tensorflow framework defines an *executor* entity in charge of planning and dispatching elements in the task queues of each *thread* of a deployed *threadpool*. Therefore, multiple threads may be scheduling the tasks of the ready node queue within the executor. Each thread analyzes whether it can run the nodes by checking various decision criteria. If they are met, it pushes the node in its own Q_{inline}, which contains all the nodes that thread can execute. Otherwise, if one of the decision criteria is not met, it delegates the node to be executed in another thread.

2.2 Thread Behavior

Each thread in the thread pool features two main procedures to *(i)* schedule and *(ii)* execute the nodes of the graph. Specificaly, each thread in the pool features two different task queues, namely Q_{ready}, containing nodes ready to be scheduled, and Q_{inline}, containing nodes ready to be executed.

Node Scheduling Stage. The scheduling of ready nodes is performed by means of the following steps:

(STEP 1) Check if Q_{ready} is not empty. If it is not empty, proceed to the following step. Otherwise, finish scheduling.
(STEP 2) Get the next node N_{next} in Q_{ready}.
(STEP 3) If N_{next} is not an *expensive task* (that is, expected execution time is small) queue in Q_{inline} for its execution. Back to step 1.
(STEP 4) If N_{next} is expensive and the current thread already has a node of the same type ready, assign the node execution to another thread. Back to step 1.

Figure 1 illustrates the per-thread *scheduling* process.

Node Execution Stage. Second, the node execution procedure proceeds as follows:

(STEP 1) Check if Q_{inline} is not empty. If it is not empty, perform node planning. Otherwise, it waits for new nodes to be sent to it or for graph execution to finish.
(STEP 2) Get the next node N_{next} in Q_{inline}.
(STEP 3) Verify that it meets the conditions for execution.
(STEP 4) Run the node using the required kernel implementation.
(STEP 5) Decrement the dependencies of the nodes that depend on the execution of the current node.
(STEP 6) Check for new nodes to schedule. In case it is fulfilled, call the scheduling procedure.
(STEP 7) Back to step 1.

Figure 2 shows the node *execution* procedure.

2.3 Multi-level Parallelism: Intra- and Inter-parallelism

Tensorflow exposes and exploits two independent levels of parallelism, namely *intra-* and *inter*-parallelism. Both can be exploited in conjunction and under user request.

Intra-parallelism controls the number of threads to be used for the execution of a kernel operation (MatMul, ConCat, etc.). Obviously, the underlying implementation of the kernel –task– must support parallelization to leverage different degrees of intra-parallelism. On the contrary, inter-parallelism controls the amount of independent kernel operations that can be concurrently executed, leveraging the strategies depicted in the previous section.

Intra- and inter-parallelism, hence, can exploit per-task and per-graph parallelism, provided it is available. The framework provides simple mechanisms to determine each level of parallelism through its high-level API; the selected values, however, are valid across the complete task graph, which make Tensorflow a *rigid* piece of software from the threading control perspective.

Regarding implementation details, Tensorflow delegates the handling of inter-parallelism (also referred as *non-blocking* parallelism in the literature) to the implementation of the *ThreadPool* in the Eigen library [10] leveraging its flexibility and efficiency.

Each thread in the *threadpool* features a third queue of tasks (Q_{eigen}) that ultimately includes tasks to be executed by the corresponding thread; under situations without assigned tasks, work stealing between Q_{eigen} queues is inplace to improve thread occupancy. Under situations in which there are no available tasks in Q_{eigen}, the thread spins, and in case of being the only thread awake without tasks, stalls for a certain time waiting for the arrival of new jobs; after that time the thread falls asleep waiting for another thread to wake it up for work.

3 Malleability Integration in Tensorflow

The integration of malleability in the threadpool associated with non-blocking tasks (that is, inter-parallelism), requires a number of modifications both in the Tensorflow core and the management of the internal threadpool in the Eigen framework.

3.1 Required Modifications in the Eigen Threadpool

The Eigen library responsible for managing the threadpool does not allow a dynamic control of the number of active threads at any arbitrary moment. Therefore, extra information including *status* information is required on a per-thread basis in order to activate/deactivate the normal behavior of the thread exposed in the previous section, effectively stopping the processing of Q_{eigen}.

In addition, the *wait* operation performed by the threads also requires modifications. In our modified version, each thread begins the process of waiting

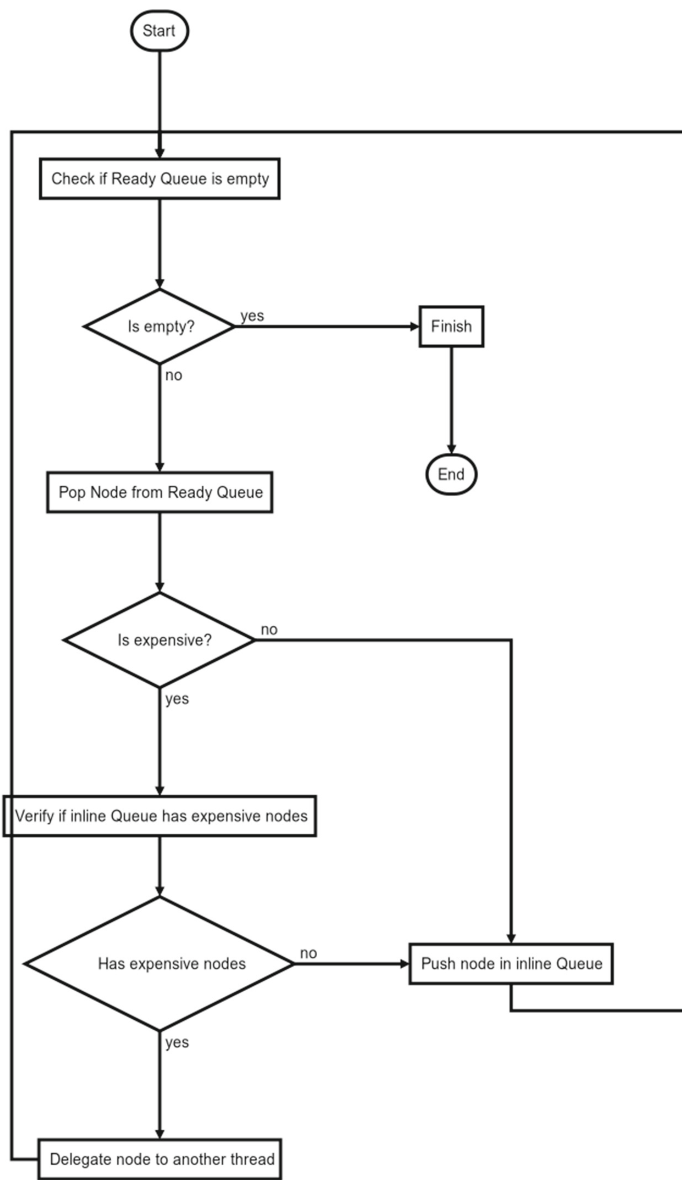


Fig. 1. Thread task scheduler.

for work by evaluating whether it should remain active or not depending on its current *state* (which can be modified externally in an asynchronous fashion).

In case the state is active, the corresponding thread analyzes if it is possible to continue executing nodes that are in its Q_{eigen} or in the queue of another thread. If the thread is inactive, it analyzes if it should wake up to another

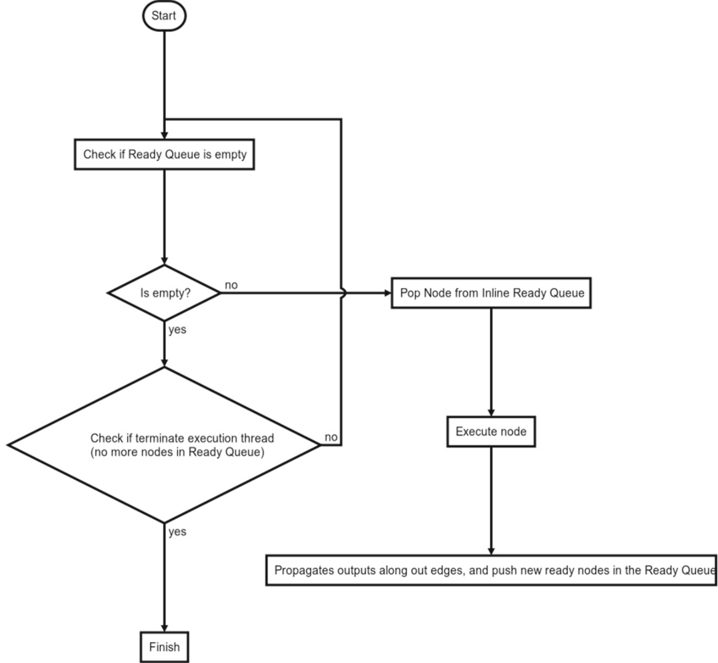


Fig. 2. Execution task procedure.

thread (in case the queue of another thread is not empty and the other threads are asleep) and if it should not fall asleep because it is the only active thread. Finally, if tasks are not available (in a proprietary or alien queue), it waits for another thread to submit work.

3.2 Required Modifications in the Tensorflow Core

The *executor* entity defined in the core of Tensorflow is in charge of scheduling and execution of the graph, as stated in the previous Section, and therefore, it also requires modifications in order to support malleability.

So far, the thread status of the threadpool cannot be controlled from the executor. Actually, the only possible communication allows delegating the execution nodes to another thread. To add this type of extra control, the executor receives information from the thread pool of non-blocking tasks with the possibility of consulting the status of the threads and modifying the number of active threads.

In our modified TensorFlow version, if the executor receives a change in the number of active non-blocking threads from an external entity, it invokes a method of the threadpool so that the number of active threads is increased or decreased (executing nodes).

In addition, each thread keeps information regarding its activation state (*active* or *inactive* thread). This allows checking whether the thread can run or delegate new nodes to another thread. All these changes are made in the node queue scheduling procedure explained in the previous section. Figure 3 depicts the main steps performed by the new thread task scheduler.

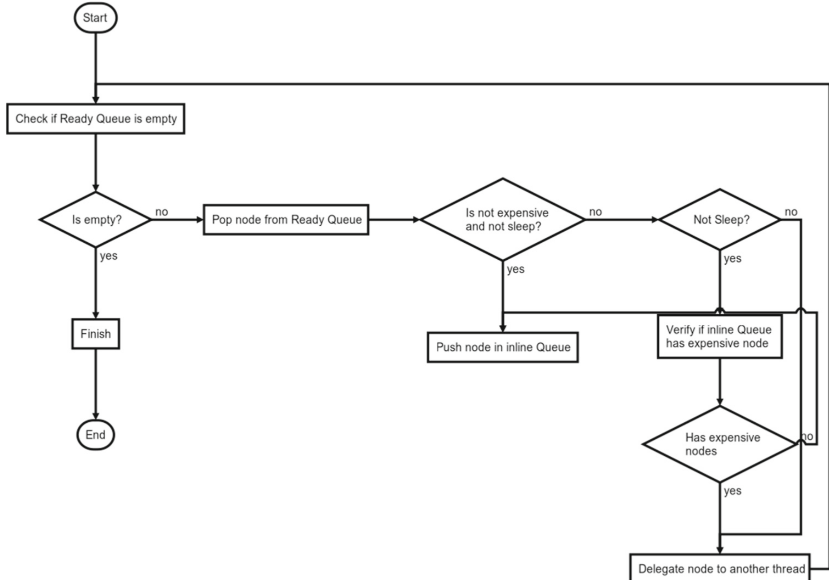


Fig. 3. New thread task scheduler.

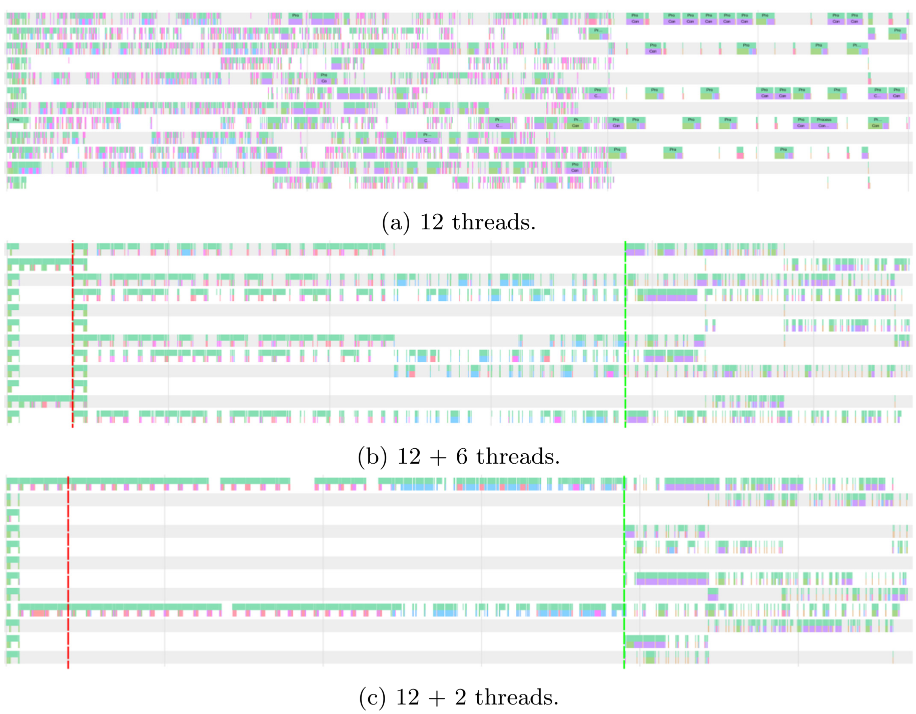
4 Experimental Results

Figure 4 report some experimental results obtained on a real malleable TensorFlow implementation modified to integrate the modifications described in Sect. 3. These results were extracted on a system based on an Intel Core i7-8750H processor featuring 6 physical cores (12 logical cores via HyperThreading technology), running at 2.2 GHz of nominal frequency. The system features 32 GBytes of DDR4 RAM memory. From the software perspective, TensorFlow version 2.0.0 was used as our baseline implementation, running on an Ubuntu 18.04 OS.

The traces report execution timelines (one horizontal line per worker thread) for a ResNet56 model defined through Keras, trained through 5 epochs, with 20 steps per epoch. Input dataset images were defined for a dimension 32×32 and 3 channels. The number of classes is fixed to 10, with a batch size 128.

We report three different scenarios for the aforementioned training process, namely:

- Figure 4a is a typical TensorFlow implementation with 12 worker threads from the beginning to the end of the execution.
- Figure 4b corresponds to a modified, malleable TensorFlow implementation in which two different thread count changes are performed: the first one limits the number of threads to 6 (reducing from the original 12 worker threads), at the point marked with a vertical red line. Afterwards, worker threads are again restored to 12, at the point marked with a vertical green line. It is observed that before the red line, only two threads run. This occurs because the threads they are running do not delegate operations to others. Operations are delegated when it is expensive and there are other light operations to execute. After the green line, thread number 10 does not execute operations as explained above. It is observed that between the green and red lines, thread 1 becomes inactive and thread 9 begins to run. This is because thread 1 had no more nodes to run and competes with the other threads to get a new one. In this case, thread 9 got new nodes, leaving thread 1 idle.
- Figure 4c shows a similar situation, but reducing the number of active threads to 2 instead of 6, and restoring to full parallelism afterwards. After the green line, threads 3 and 6 are not activated as explained in the previous trace.



(a) 12 threads.

(b) 12 + 6 threads.

(c) 12 + 2 threads.

Fig. 4. Execution traces for three different threading scenarios. (Color figure online)

Although still general, these results demonstrate the ability of our modified TensorFlow version to seamlessly achieve malleability, and paves the road towards the integration or this malleable version with an application co-scheduler that orchestrates, under demand, the assigned resources to independent TensorFlow implementations.

5 Conclusions and Future Work

In this paper, we have introduced and described the main modifications that are required to transform a fixed-parallelism TensorFlow implementation into a *malleable* implementation, in which the degree of parallelism can be dynamically selected and varied (reduced or increased) while the application is running.

This functionality is not present nowadays in the default TensorFlow distribution, and can pave the road towards flexibility and elasticity in shared-resources scenarios (e.g. cloud servers running multiple TensorFlow instances).

Our work, however, is still a fundamental step towards more advanced functionality proposed as future work, among which we can name:

1. *Integration with a co-scheduler.* A malleable library/framework infrastructure only makes real sense when combined with a higher-level resource manager (or co-scheduler), that leverages malleability of the underlying malleability (in this case within TensorFlow) and dynamically modifies the amount of resources assigned to them in a co-ordinated fashion. We are working in this type of resource orchestrator to support efficient co-existence of TensorFlow instances in the same machine.
2. *Creation of a malleability API.* As of today, the malleability is internally selected on specific execution points as proof of concepts. Its management, however, must be transparent and externally selectable, on demand. For that to happen, an ad-hoc API to select the number of active/inactive threads will become mandatory, together with an infrastructure to support thread variation by means of OS signal reception. Both functionalities are in our roadmap.
3. *Management through containers.* Containers allow a dynamic reduction of resources in terms of number of cores, amount of memory and external devices, among others. However, externally reducing the number of assigned cores without a proper reduction of internal software threads derives in a non-acceptable oversubscription effect. As TensorFlow training/inference processes are usually confined within Docker containers, it is mandatory to support malleability in the framework. The interaction between per-container resource management and mallebility in TensorFlow is thus a primary goal of our research.
4. *Intra-task malleability.* The introduced techniques only affect *inter-* parallelism. Malleability within nodes/tasks (*intra-*) is also of interest for us to create a completely malleable parallelism. For that to happen, malleable underlying libraries are mandatory (e.g. malleable BLIS [7] for BLAS tasks –e.g. MatMul for fully connected layers–).

5. *Heterogeneity support (use of GPUs).* The integration of worker threads associated with hardware accelerators –mainly GPUs–, and its dynamic activation/deactivation is also in our roadmap, so that graphics processors can also be assigned or unassigned to existing TensorFlow instances at runtime.
6. *Test with real-world problems.* Obviously, the evaluation of the overhead and benefits introduced by malleable TensorFlow implementations on real models and workloads is mandatory and will be of interest in the near future.

References

1. Abadi, M., et al.: TensorFlow: large-scale machine learning on heterogeneous systems (2015). Software available from tensorflow.org
2. Jia, Y., et al.: Caffe: convolutional architecture for fast feature embedding. In: Proceedings of the 22Nd ACM International Conference on Multimedia, MM 2014, New York, NY, USA, 2014, pp. 675–678. ACM (2014)
3. François Chollet et al. Keras (2015). https://keras.io
4. Paszke, A., et al.: Pytorch: an imperative style, high-performance deep learning library. In: Wallach, H., Larochelle, H., Beygelzimer, A., Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems 32, pp. 8024–8035. Curran Associates Inc. (2019)
5. Hasabnis, N.: Auto-tuning tensorflow threading model for CPU backend. In: 2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC), pp. 14–25. IEEE (2018)
6. Catalán, S., Herrero, J.R., Quintana-Ortí, E.S., Rodríguez-Sánchez, R., van de Geijn, R.A.: A case for malleable thread-level linear algebra libraries: the LU factorization with partial pivoting. IEEE Access **7**, 17617–17633 (2019)
7. Rodríguez-Sánchez, R., Igual, F., Quintana-Orti, E.S.: Integration and exploitation of intra-routine malleability in blis. J. Supercomput. 11 (2019)
8. Rey, A., Igual, F.D., Prieto-Matías, M.: Variable intra-task threading for power-constrained performance and energy optimization in DAG scheduling. J. Supercompu. **75**(3), 1717–1731 (2019). https://doi.org/10.1007/s11227-019-02760-6
9. Xu, P., Shi, S., Chu, X.: Performance evaluation of deep learning tools in docker containers. In: 3rd International Conference on Big Data Computing and Communications, BIGCOM 2017, Chengdu, China, August 10–11, 2017, pp. 395–403. IEEE Computer Society (2017)
10. Guennebaud, G., Jacob, B., et al.: Eigen v3 (2010). http://eigen.tuxfamily.org